Gate Matrix Layout Techniques

By

A.S. Chakravarthy and P. Mazumder

CSE-TR-40-90

THE UNIVERSITY OF MICHIGAN
COMPUTER SCIENCE AND ENGINEERING DIVISION
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
Room 3402, EECS Building
Ann Arbor, Michigan 48109-2122
USA

Gate Matrix Layout Techniques

Anil S. Chakravarthy and P. Mazumder
Department of Electrical Engineering and Computer Science
Computing Research Laboratory
The University of Michigan
Ann Arbor, MI 48109

January 16, 1990

Abstract

The Gate Matrix Layout style was introduced in 1980 [2], and has since been used widely in IC design. Gate Matrix Layouts are characterized by simplicity of design and quick turnaround time. However, it has been found that straightforward implementation leads to very inefficient usage of chip area. In this survey, we discuss several algorithms that reduce the area required to implement the circuit. All these algorithms find a gate sequence that minimizes the area. These algorithms are grouped into five sections: algorithms that use the interval graph representation of the Gate Matrix Layout, algorithms that use min–net–cut to obtain a good gate sequence, algorithms that use probabilistic methods to minimize the area, algorithms that use properties of the Gate Matrix style to get a good gate sequence and algorithms that obtain a minimal layout corresponding to prefixed physical constraints. Also, we discuss algorithms that do net assignment such that the resulting assignment is realizable. Finally, we discuss algorithms that have been developed for the related problem of One–Dimensional Array Gate Assignment.

1 Introduction

The Gate Matrix style was introduced by Lopez and Law [2], and can be considered as an extension of the Weinberger style [1] for CMOS layouts. The Gate Matrix is composed of intersecting rows and columns. All transistors having a common input are placed on a common polysilicon line. Each polysilicon line is represented as a column. This column now serves a dual purpose. It is the gate of all the transistors that lie on it and it also acts as the common connection between these transistors. The columns of the Gate Matrix are equally spaced and parallel to each other.

The p-transistors of the CMOS circuit to be implemented using the Gate Matrix are placed in one 'half' of the Gate Matrix while the n-transistors are placed in the other half. There are three types of connections in the Gate Matrix as shown in Fig. 1(b): metal, polysilicon and diffusion. The rows are formed by connecting those transistors that are connected to each other either in series or in parallel. These connections are made using metal. Fig. 1(a) shows a CMOS circuit. Its implementation using the Gate Matrix style is shown in Fig. 1(b). In Fig. 2(a), we see the abstract representation of a Gate Matrix layout for a given netlist. A $\times$ denotes a transistor and a $\bullet$ denotes a contact. *Also, the connections to power and ground are not shown as they are assumed to be implemented in a second layer.*

Fig. 2(b) represents the same layout as Fig. 2(a) except that the arrangement of columns is different. This layout requires fewer rows. In general, it is possible to reduce the number of rows required to implement a circuit by choosing a suitable column sequence.

Our aim is to minimize the chip area. Given the circuit schematic, the number of columns, the width of each of these columns and the distance separating two adjacent columns are all fixed. Therefore, the total chip area is determined by the number of rows and the vertical spacing between any two rows. We have seen that the number of rows needed is lessened by taking a suitable gate sequence.

2 Formulation of the problem

Before we can formulate the problem, we need to know the concept of a *net*. Informally, we can think of a net as a group of transistors that are connected to each other either in series or in parallel, along with the *contacts* that this group has with output columns. More abstractly, a *net can be represented as a set of columns*, each of which acts either as an input to a member of the set or as an output of the structure represented by the net. Moreover, it is not possible to place one net over another in the same row. We can now define our problem:

- Input: A set of gates $G = \{g_1, g_2, \ldots, g_n\}$ and a set of nets $N = \{n_1, n_2, \ldots, n_m\}$.
- Output: A permutation of the gates $\pi\{G\}$ such that each gate is assigned a unique column and an assignment of every net to a (not necessarily unique) row *such that the number of rows (tracks) is minimized.*

In the next section, we show that this problem is a member of a very difficult class of problems known as NP-Complete problems. Section 3 discusses methods of obtaining a 'good' gate sequence. The worst-case performance of these algorithms is discussed in Section 4. The *min-net-cut, Simulated Annealing and Genetic Algorithm* methods are discussed in Sections 5 and 6. It is possible to improve the solutions we get from these algorithms by taking into account some properties native to the Gate Matrix style. Algorithms which use specific properties of the Gate Matrix style are presented in Section 7. Section 8 discusses algorithms that search for good net sequences rather than good gate sequences. Section 9 presents an algorithm that takes into account certain practical restrictions on the shape of the final layout. Section 10 discusses *net merging*, a useful post-optimization technique. After obtaining the gate sequence, we need to assign nets to the rows such that all diffusion connections between the rows become realizable. This, in itself, is a difficult problem and is discussed in Section 11. In the last section, Section 12, we present algorithms for a related problem known as the *one-dimensional array layout* problem.

3 Interval Graph formulation and the consecutive ones property

Ohtsuki et al. [3] gave the first graph-theoretic model for this problem using *interval graphs*. A graph $G = (V, E)$ is called an interval graph if there exists a set of finite closed intervals $\{I(v)\}_{v \in V}$ such that $(u, v) \in E$ implies and is implied by $I(u) \cap I(v) \neq \emptyset$. The set of intervals thus formed is called a *realization* of the graph G. Figs. 3(a) and 3(b) show an interval graph and its realization. A *clique* C of G is a subset of V such that $\forall v_i, v_j \in C, (v_i, v_j) \in E$. A clique C is a dominant clique if it is not a proper subset of any other clique. In Fig. 3(a), $\{n_1, n_2, n_4\}$ and $\{n_1, n_3, n_4\}$ are dominant cliques. In any graph, there are atmost $|V|$ dominant cliques, where $|V|$ is the size of the set V [12]. The *clique number* is the size of the largest dominant clique. In Fig. 3(a), the clique number is 3.

Also, given the set of gates and the set of nets, we can construct a *connection graph* $H = (V, E)$ as follows: The set of vertices V is identical to the set of nets N. Also, there is an edge in the connection graph between every pair of nets that have a common gate. Fig. 3(c) shows a set of nets and the set of gates associated with each net. Fig. 3(d) shows the connection graph for the nets given in 3(c).

The algorithms given in [3], [4], [5], [6], [7], [8] and [9] try to obtain a sequence of gates which minimizes the number of rows. Corresponding to every particular sequence, we can draw horizontal intervals corrresponding to the nets. This set of intervals defines an interval graph where the set of vertices is the same as the set of nets and the set of edges comprises of every pair of nets that cannot be placed in the same row *given this particular gate sequence*. So, all nets that form a clique in this interval graph have to be placed on different rows. Therefore, the clique number of the interval graph provides a lower bound on the number of rows that we will need if we adopt this gate sequence.

Notice that the connection graph is a subset of the interval graph. Also, the connection graph is independent of any gate sequence. If the connection graph is itself an interval graph, then we can obtain an optimum layout. If it is not, we need to *augment* the connection graph (i.e., add a set of edges) to obtain an interval graph. Fig. 3(d) represents a connection graph that is not an interval graph. If we add the edge (n_1, n_4) to this graph, we get the interval graph of Fig. 3(a). We can now reformulate our problem:

- Input: A connection graph.
- Output: A corresponding interval graph which has minimum clique number.

Kashiwabara and Fujisawa have shown this problem to be NP-Complete by relating it to the one-dimensional optimal assignment problem [11]. The related problem of finding a *minimum size augmentation* has also been shown to be NP-Complete [11]. We now discuss the algorithms that attempt to solve this problem.

3.1 Finding a minimal augmentation

Ohtsuki et al. [10] have given an algorithm to find a *minimal augmentation*. A minimal augmentation is defined as one which is not a proper subset of any other augmentation. A greedy method is described in [10]. We paraphrase it below:

Algorithm 1: Finding a minimal augmentation

- Any graph with less than four vertices is an interval graph (we can prove this by considering all possible combinations of vertices and edges and constructing intervals on the real line). At each step, add a vertex and a minimal set of edges such that the resultant graph is an interval graph. This augmentation is done $n - 3$ times (i.e., till all edges are included). The resultant augmentation is minimal because each of the constituent augmentations is minimal.

To obtain the gate sequence from the resultant graph, we need a special representation. Given a graph, we can construct a *vertex-versus-dominant clique (v.d.c.) matrix* $A = [a_{ij}]$ as follows:

$$a_{ij} = \begin{cases} 1, & \text{if vertex } i \in \text{dominant clique } j \\ 0, & \text{otherwise.} \end{cases}$$

Figs. 4(a) and 4(b) show the v.d.c. matrix A of a connection graph derived from the gate-sets in Table I. The matrix is said to have the *consecutive ones property* if all the ones in *each* row occur in consecutive positions. Fig. 4(c) shows a v.d.c. matrix B with the consecutive ones property[1]. As we will see, all the algorithms in this section use the consecutive ones property to obtain a 'good'

[1] For the moment, consider the ×'s as ones. Their significance will be explained presently.

layout. We now state an important theorem due to Fulkerson and Gross:

Theorem 1

- A graph is an interval graph iff there exists an ordering of dominant cliques such that the vertex–versus–dominant clique matrix has the consecutive ones property.

Such an ordering defines a *canonical realization*. For every net, the canonical realization defines an *interval*. In matrix B of Fig. 4(c), net 9 has an interval between positions 3 and 7, denoted $[3-7]$.. For every gate, the intersection of the intervals defined by the nets pertaining to it gives a set of positions at which the net can be placed. In Fig. 4(c), the position of g_7 is defined by $[6-7] \cap [5-7] \cap [3-7] = [6-7]$. So, gate g_7 can be placed either at relative positions 6 or 7. Those gates whose positions are defined by an interval of unit length, e.g., $[5-5]$, are placed first. Other gates are placed relative to these gates. It is proved in [3] that a gate sequence obtained in this fashion corresponds to the interval graph from which the gate sequence was derived.

It is possible to directly manipulate the v.d.c. matrix to obtain a solution. This idea leads us to our second algorithm due to Wing et al. [4].

3.2 Automated gate matrix layout

We can think of a straightforward method to make the v.d.c. matrix have the consecutive ones property. In every row, we fill in ones for zeros between the leftmost 1 and rightmost 1. For example, the v.d.c. matrix of Fig. 5(b) is made into the v.d.c. matrix of Fig. 5(c) by this method. Each *fill–in* is denoted by a $\times$. The matrix of Fig. 5(c) has the consecutive ones property. However, it is possible to reduce the maximum number of ones in any column by using more sophisticated methods. Hence, we can think of the result provided by this method as a sort of *upper bound* on the maximum column size. We know that the *lower bound* cannot be less than the maximum number of nets attached to a gate because all such nets have to be placed on different rows.

In the heuristic algorithm that we discuss next [4], the v.d.c. matrix with the consecutive ones property is constructed row by row, such that at any stage the partially constructed matrix has the consecutive ones property. The algorithm is described below:

Algorithm 2: Constructing the solution matrix row by row

- Input: The v.d.c. matrix A.
- Output: A matrix B derived by permuting the rows and columns of A, and having the consecutive ones property.

1. Use the straightforward method described above to check whether the upper bound generated is equal to the lower bound. If so, this is our optimal v.d.c. matrix.

2. Eliminate those rows of A which have only a single one. These rows correspond to nets connected to only one gate. These nets can always be placed later. Call the reduced matrix A'. We get the matrix of Fig. 5(b) by removing the rows 1, 3, 6, 13, 14, 17 and 18 from the matrix of Fig. 5(a).
3. We wish to keep the number of fill-ins to a minimum. We can acheive this by placing those rows which have ones in common next to each other as far as possible. To find out the permutation of the rows that will acheive this effect, we construct an *overlap graph* $O(V', E')$ such that V' = set of rows in A' and there is an edge between two vertices in V' if the two rows that they correspond to overlap (i.e., have atleast one gate in common). Fig. 5(e) shows the overlap graph of the matrix A'.
4. We now want a permutation of the rows. If we make a spanning tree out of $O(V', E')$ in a depth-first fashion, we can get a permutation of the rows such that rows that are connected to each other are likely to be adjacent in the permutation as well. By assuming vertex 2 as the root of the tree, we get the sequence [2, 8, 5, 4, 7, 9, 11, 10, 12, 15, 16].
5. Having obtained a permutation, we build a column sequence as follows:
   - Set B = the empty matrix.
   - For every row r_i in the permuted A' do
   - Try to place the row r_i in B such that all its ones are in sequence. If this causes a conflict with the order of columns imposed by previous rows, place the ones according to one of the following strategies:
     (a) *Using fill-ins:* If all the columns in which r_i has ones have been placed, then place the ones in their respective positions, and use a minimum number of fill-ins to produce the consecutive ones property in r_i. Row 10 was placed in the matrix B of Fig. 5(d) using this principle.
     (b) *Concatenate and fill strategy:* This is similar to strategy 1, except that this is used when all the columns in which r_i has ones have not been assigned positions. In this case, we place that column in the leftmost position and use fill-ins to produce the consecutive ones property in r_i. Row 12 was placed in B using this principle.
     (c) *Shift and insert strategy:* Insert the column between two placed columns so as to obtain the consecutive ones property in row r_i. To preserve the consecutive ones property of previously placed rows, it might be necessary to add some fill-ins in those rows. Row 15 was placed in B by inserting the column 6 between columns 2 and 5 and shifting rows 9 and 10.
   - Generally, for every row that has a conflict, all strategies are tried out and the one that yields the least maximum column size is chosen.

Fig. 5(d) shows the the v.d.c. matrix B. This matrix has the consecutive ones property. The algorithm that we have just discussed was the earliest algorithm developed for the gate matrix layout problem. It has many drawbacks, the major one being that the quality of the solution depends on the initial row permutation. Actually, the solution to the gate matrix layout problem is not related to the positions of the rows at all, and hence the permutation of the rows employed by the algorithm is superfluous. We now present an algorithm due to Wing and Huang [5] that builds the matrix B column by column using a greedy heuristic.

3.3 Building a minimal v.d.c. matrix using a greedy heuristic

The matrix A is reconstructed column by column such that at any stage, the partial matrix, denoted B, has the consecutive ones property. We give an outline of the algorithm below:

Algorithm 3: Constructing the solution column by column by a greedy method

- Step 1: Begin with the column that has the largest number of ones (i.e., the largest dominant clique). Let it be c_k. Denote the set of dominant cliques which have one or more vertices in common with c_k as $Adj(c_k)$.
- Step 2: Among all dominant cliques $c_i \in Adj(c_k)$, select one, which when placed next to column c_k in B, will result in the smallest increase of the size of the largest dominant clique already placed in B. If there are ties, choose the dominant clique having the most vertices in common with c_k. Now, denote this clique as the last clique to be placed c_k.
- Step 3: Repeat Step 2 with the column placed last as c_k. If $Adj(c_k)$ is empty, and all columns have not been placed yet, backtracking is necessary. Backtrack and find a dominant clique c_y such that $Adj(c_y)$ is not empty. Repeat Step 2 with c_y.
- Step 4: Stop when all dominant cliques have been placed in B.

In Figs. 4(a), 4(b) and 4(c), we find a connection graph, its dominant cliques $C_1, \ldots, C_7$, the v.d.c. matrix A and the reconstructed matrix B. Each $\times$ denotes a set of edges to be added to the connection graph to make it an interval graph. For example, the $\times$ at the intersection of vertex 2 and C_3 indicates that vertex 2 should be connected to all the existing members of C_3.

An extended version of this algorithm is given by Wing et al. [6]. This version places dominant cliques on both sides of the partially constructed matrix. Also, more rules are provided for breaking ties should they occur. Instead of starting with the largest dominant clique of the connection matrix, this algorithm is repeated with a number of different starting columns (i.e., dominant cliques) and the best of these solutions is chosen. In the paper, it is shown by example that the choice of the starting column makes a significant difference to the performance of the algorithm. The main

steps of the algorithm are given below:

Algorithm 4: Placing columns on both sides of the partial matrix

- Input: Matrix A.
- Output: Matrix B with the consecutive ones property.

Definitions : As we build B, we may need to convert some of the zeros of the partial B to ones so as to preserve the consecutive ones property. Each such conversion is called a *fill–in*. Define the rightmost column of B as c_r and the leftmost as c_l. Let *enlarge R(i) (enlarge L(i))* denote the increase in size of the largest dominant clique of B if the clique under consideration c_i is placed to the right of c_r (left of c_l). Let *com(i)* denote the number of vertices that c_i has in common with c_r.

- Step 1: Choose a starting column randomly and denote it c_r.
- Step 2: Repeat
  - Find $Adj(c_r)$.
    If $Adj(c_r)$is not empty,
    * Choose $c_i \in Adj(c_r)$ such that *enlarge R(i)* is minimum. If there be ties, use minimum fill-in(i) and maximum *com(i)*, in that order, as criteria to choose the next clique[2].
      If *enlarge R(i)* is greater than (lesser than) *enlarge L(i)*, place c_i to the right of c_r (to the left of c_l). Change c_r and c_l accordingly.
  - If $Adj(c_r)$ is empty,
    * If c_r is not the same as c_l,
      · Retract the rightmost clique and continue with the previous clique as c_r.
    * If c_r is the same as c_l,
      · Choose an unplaced clique which has the maximum size. Place it as c_r and continue.

  until all cliques are placed.

  Step 3:

- Iterate Steps 1 and 2 and at each stage retain that matrix B which has minimum clique number. If there is a tie here, retain the one with minimum number of total fill ins.

[2]Whenever we give more than one criterion in a particular order to resolve a tie, a later criterion applies only to those candidates which satisfy *all* of the earlier criteria.

One severe drawback of the two algorithms described above is that generation of all the dominant cliques of an arbitrary graph is in itself an NP–Complete problem. If the graph is known to be an interval graph its dominant cliques can be generated in polynomial time. In Li's algorithm [7], which we take up next, a *vertex–versus–dominant gate(v.d.g.) matrix* is generated and manipulated.

3.4 Generating a gate sequence from a v.d.g. matrix

Definitions: A *dominant gate* is defined as one whose set of associated nets is not a proper subset of the corresponding set of any other gate. In Table II, a gate–list is shown. The dominant gates have a D marked next to them.

For a connection graph $H = (V, E)$ with dominant gates $g_1, g_2, \ldots, g_p$, a v.d.g. matrix A is given by $[a_{ij}]$ where

$$a_{ij} = \begin{cases} 1, & \text{if vertex } i \in \text{dominant gate } j \\ 0, & \text{otherwise.} \end{cases}$$

A is an $n \times p$ binary matrix. Fig. 6(a) gives the v.d.g. matrix corresponding to Table II.

Finally, if for some row i in a v.d.g. matrix, $a_{ij} = 1$ and $a_{ik} = 0$ for $k \neq j$, we say that net i is *isolated* in column j. In Fig. 6(a), net 2 is isolated in column 1. If, for a particular ordering of dominant gates, $a_{ij} = 1$ and $a_{ik} = 0$ for all $k > j$, net i is said to *terminate* at column j. If A in Fig. 6(a) is considered an ordering, we can say that net 3 terminates in column 6. The algorithm is described below.

Algorithm 5: Constructing the columns of a v.d.g. matrix

- Input: A v.d.g. matrix A.
- Output: The output matrix B with the maximum column size minimized. B is obtained by permuting the rows of A and has the consecutive ones property.
- Step 1: Find all the dominant gates from the given net list. Construct a v.d.g. matrix A such that the rows correspond to the nets and the columns to the dominant gates. Let n and p be the number of rows and columns of the matrix.
- Step 2: For all columns in A do
  - a) mark g_i unplaced
  - b) count the number of ones in g_i and denote it x_i
  - c) count the number of isolated nets connected to g_i and denote it y_i.
- Step 3: Arbitrarily choose one column g_j as the leftmost one, mark it 'placed' and create a partially formed matrix $B = [g_j]$. Let $D = x_j$ (At the end of the algorithm D will hold the size of the largest dominant clique).

- Step 4: For each 'unplaced' column g_i do:
  - a) place g_i to the right of B
  - b) calculate the number of fill-ins(f_i) needed to make the extended B matrix have the consecutive ones property
  - c) compute the sum(s_i) of the numbers of fill-ins and ones in g_i, i.e., $s_i = x_i + f_i$
  - d) denote the number of terminated nets in g_i as z_i
  - e) remove g_i.
- Step 5: Choose an unplaced column g_k with the smallest s_k, and place it to the right of the partially formed B, i.e., $B = B||g_k$. In case of a tie, use the following criteria in the order shown:
  - a) Choose that column g_k with the largest number of terminated nets (z_k)
  - b) Choose that column g_k with the largest number of isolated nets (y_k)
  - c) Choose that column which has the least f_k when placed next to B

  Update D to $max(D, s_k)$.
- Step 6: If there still remain unplaced columns, go to Step 4. Otherwise, exit and D will be the number of tracks required.

In Fig. 6(b), B as constructed from A is shown. To illustrate the procedure, assume g_2 has been chosen as the first column. Table III shows the values calculated in step 4 for the remaining columns. g_3 is selected because it has the least s_i.

Li has also proposed two variants of the above algorithm, in both of which only Step 5 of the above algorithm is modified. We desribe them below:

- Variant 1: Instead of choosing a column with the smallest s_k as done above, this algorithm selects *all* unplaced columns g_k which have s_k less than or equal to that of D of the partial matrix B. This is done because none of these columns, when placed adjacent to B, can increase the value D of B. In case of a tie, resolve it using the following criteria:
  - a) Choose that g_k with the largest y_k
  - b) Choose that column which has the least f_k when placed next to B
- Variant 2: Choose that unplaced column g_k which has the least f_k when placed next to B. In case of ties, select that column with the largest z_k

So far, we have discussed algorithms that use *one* criterion to select a column and select that column which satisfies this criterion best (the criteria used to resolve ties are only secondary). Such algorithms are termed *greedy* and, in general, tend to produce only locally optimal solutions. Other algorithms [7,8] have been published which incorporate a set of criteria intended to produce globally optimal solutions. We discuss two such algorithms below.

3.5 Incorporating globally–oriented criteria into the column selection step

Xu et al. [8], in their algorithm, induce the consecutive ones property into a *vertex–versus–gate(v.g.) matrix*. The v.g. matrix can be directly obtained from the original description of the problem. The algorithm proceeds in two stages: In the first stage, the set of unplaced gates is divided into two disjoint subsets, the *selectable gate(s.g.) set* and the *unselectable gate(u.g.) set*. In the second, the algorithm chooses the 'best' column in the s.g. set using a weighted function of several factors like number of fill–ins, starting and terminating nets, etc. The skeleton of the algorithm is given below:

Algorithm 6: Using global connection criteria in the selection process

- a) Choose an initial column from A and concatenate it into the currently empty output matrix B.
- b) Partition the unplaced gates into the s.g. and u.g. sets.
- c) If s.g. $\neq \emptyset$,
  - then select the 'best' column[3] from it and concatenate it with matrix B.
  - else go to step e).
- d) If all columns of A have been put into B, then B is the output. Exit the algorithm.
- e) If s.g. $= \emptyset$ then go to a), otherwise go to b).

An initial column is chosen such that the nets attached to it have minimum incidence relations with other nets. To do this, we calculate the *column degree(dc)* and the *column incidence degree(dcin)* of every column. These terms reflect the *connectivity* of the nets attached to a particular column to other nets. We define these terms below:

- dv(i)= degree of vertex i in the given connection graph $H = (V, E)$.
- dc(j)= $\sum_{i=1}^{m} a_{ij} *$ dv(i)
  where m is the number of nets and a_{ij} is 1 or 0.

[3] In this section, we use the terms 'column' and 'gate' interchangeably.

- din(j)= $\sum_j$ dv(j); for all $j, (v_i, v_j) \in E$.
- dcin(j)= $\sum_{i=1}^{m} a_{ij}$* din(i)
  where a_{ij} is 0 or 1.

To choose the initial column, proceed through the following steps:

- a) Select the column with the smallest column degree. In case of a tie, resolve the tie using the following criteria in the order shown:
  - i) Select the column with the smallest column incidence degree.
  - ii) Select an initial column arbitrarily.

Fig. 7 shows the dv, din, dc and dcin entries for all nets and columns of Table IV. Going through steps a) and i) above we are left with {1,4, and 7} as possible starting columns. We choose 1 as the starting column arbitrarily.

To partition the set of unplaced gates into s.g. and u.g. sets, we adopt the following principle: If, in the connection graph $H = (V, E)$, some associated net of an unplaced gate is connected by an edge to an associated net of an already placed gate, the unplaced gate is put into the s.g. set. Otherwise, it is put into the u.g. set. To express this idea more formally, we need the following definitions:

- Define *cnet* to be the set of current nets (i.e., nets which are associated with atleast one of the placed gates).
- Define the set of nets which incident on the set of current nets as
  incnets= $\{v_i | (v_i, v_j) \in E$ and $v_j \in cnet$ and $v_i \notin cnet\}$
- Define $nset_i$ as the set of nets associated with gate g_i.

The set of unplaced columns, *upcol* is divided into s.g and u.g. We construct s.g. as follows:

- a) $g_i \in$ upcol is assigned to s.g. if $nset_i \subset incnets$.
- b) g_i is assigned to s.g. if $nset_i \cap cnet \neq \emptyset$.
- c) g_i is assigned to s.g. if s.g. $= \emptyset$ and $nset_i \cap incnets \neq \emptyset$ after steps a) and b).

Set u.g. = upcol - s.g.

We will continue the example of Fig. 7. Assuming that 1 has been chosen as the initial column, we calculate the following:
$cnet = \{5\}$; $incnets = 1, 4$. So, the s.g. set turns out to be {1,7,8}. Columns 7 and 8 are chosen because they contain net 5 in their netlists. Column 4 contains net 4 which overlaps with net 5.

Finally, we need a method to select the best column from s.g. to concatenate with B. Again, we need a few definitions before we can proceed:

- $nf(j)$ = number of fill-ins required to place column j next to B
- $ne(j)$ = number of terminating nets in column j when it is placed next to the partially constructed matrix B
- $ns(j)$ = number of starting nets in column j when it is placed next to the partially constructed matrix B
- $ncol(j)$ = number of ones in column j
- Define the *evaluation function* $\alpha(j)$ as:
  $\alpha(j) = k_1 * ncol(j) + k_2 * nf(j) - k_3 * ne(j)$
  where k_1, k_2, k_3 are *weights.*

To select the 'best' column, we go through the following steps:

- a) select the column with the smallest α. In case of a tie, use the following rules in the given order to resolve the tie:
  - i) Select the column with the smallest nf
  - ii) Select the column with the largest ne
  - iii) Select the column with the smallest ns
  - iv) Select an arbitrary column

To illustrate this method, we continue the example of Fig. 7. Take weights k_1, k_2 and k_3 all as 1.

- $ncol(4) = 1; nf(4) = 1; ne(4) = 0; ns(4) = 1; \alpha(4) = 2$
- $ncol(7) = 1; nf(7) = 0; ne(7) = 0; ns(7) = 0; \alpha(7) = 1$
- $ncol(8) = 3; nf(8) = 0; ne(8) = 0; ns(8) = 2; \alpha(8) = 3$

Since column 7 has the smallest α value, we choose it as the 'best' column.

Our last algorithm in this section is due to Liu and Liu [9]. This algorithm defines an upper bound on the size of the largest dominant clique that can be generated when B is built using the columns of A. Parameters are provided to adjust the size of this upper bound.

3.6 Generating B given a predefined upper bound on the size of the largest dominant clique

This algorithm uses two parameters to choose the next column to add to the partially constructed matrix B. Since these parameters are chosen at the beginning of the algorithm, it is claimed that the choice of the column at every step is based on global and not local criteria. As before, we need some definitions before we can proceed.

- Define a *personality matrix* A as
  $A = [a_{ij}]_{m \times n}$
  where
  $$a_{ij} = \begin{cases} 1, & \text{if net } i \text{ is connected to gate } j \\ 0, & \text{otherwise,} \end{cases}$$
  and number of nets is m and number of gates n. Notice that this matrix is the same as the v.g. matrix defined above.

- Define LB as the maximun number of ones in any column of A.

- Define $CS(i)$ as the number of ones in column i of A and $RS(j)$ as the number of ones in row j of A.

- Define the matrix B as a matrix corresponding to A and having the consecutive ones property.

- Define the *simplest personality matrix* A' corresponding to $A = [C_1 C_2 \ldots C_n]$ as follows:
  $A' = [C_1 \ldots C'_n]$ where $n' \leq n$ and the number of rows in A', m', is governed by $m' \leq m$. Also, for all $i, j \leq n'$, $C_i \not\subseteq C_j$ and for every C_i, there exists atleast one column $C_r (r \neq k)$ such that $C_k \bullet C_r \neq 0$. That is, A' contains all those columns of A which are *dominant* and which are *adjacent* to atleast one other column. This purges A of those columns which have no role in the construction of the matrix B. In Fig. 8, we show the simplest personality matrix A' corresponding to the personality matrix A. Columns 6 and 9 have been deleted.

- Define the *quasi-bound-difference* QBD as $QBD = UB - LB$ where UB is given by:
  $UB = \max\{CS(C_i) | C_i \in B\}$. Since it is not possible to know UB a priori, a quasi-bound-difference is set up at the beginning of the algorithm. This QBD controls the UB of the matrix B generated by the algorithm.

The algorithm constructs a matrix B' corresponding to A' rather than constructing one corresponding to A. In the paper, there is a theorem which states that constructing B' is equivalent to constructing B.

There is a second parameter c that varies between 1 and m'/LB. QBD and c act as dual checks on the column to be selected. Given a problem, a minimum upper bound UB_{min} ($UB_{min} \geq LB$) is postulated. Using UB_{min}, values for c and QBD are set. If a solution cannot be found for this

value of UB_{min}, the algorithm is repeated with a higher value.

Algorithm 7: Constructing column by column using upper bounds

- Step 1: Assume that the i^{th} column of B', $C_{i'_B}$, has been generated. Also, it is assumed that values were set for ϵ and QBD at the beginning of the algorithm. Among the unplaced columns, form a *candidate set* CA for the $(i+1)^{th}$ column. A column $C_j \in A'$ belongs to CA if it satisfies the following two conditons:
  - a) Fill-in$(C_{i_B}) + CS(C_{i_B}) + CS(C_j) - C_m * C_{i_B} \leq \epsilon * LB$.
  - b) Fill-in$(C_j) + CS(C_j) \leq LB + QBD$.
- Step 2: If CA is empty, then no solution exists. Enter another cycle with new values of ϵ and QBD. Otherwise, select one column in CA arbitrarily and continue.

In the paper, there are two improved variants of this algorithm. They vary in Step 2 only. We detail the two variants below:

- Variant 1, Step 2: If CA is not empty, select that column which has the most terminating nets associated with it. In case of a tie, choose that column which has the maximum size. If CA is empty, enter another cycle with new values for ϵ and QBD.
- Variant 2, Step 2: If CA is not empty, select that column which has the most terminating nets associated with it. In case of a tie, choose that column which has the minimum size. If CA is empty, enter another cycle with new values for ϵ and QBD.

In Fig. 8, we show the matrix B' constructed from the matrix A'. We illustrate one iteration of the algorithm. Assume that 4 has been chosen as the initial column arbitrarily. Let the values for ϵ and QBD be 2 and 1 respectively. We note that $LB = 4$ from inspection of A'.

- $C_j = 1$; Conditions a) and b) satisfied; Number of terminating nets =0
- $C_j = 2$; Conditions a) and b) satisfied; Number of terminating nets =0
- $C_j = 3$; Conditions a) and b) satisfied; Number of terminating nets =0
- $C_j = 5$; Conditions a) and b) satisfied; Number of terminating nets =1
- $C_j = 7$; Conditions a) and b) satisfied; Number of terminating nets =0
- $C_j = 8$; Conditions a) and b) satisfied; Number of terminating nets =0
- $C_j = 10$; Conditions a) and b) satisfied; Number of terminating nets =0

Since all columns satisfy conditions a) and b), we choose column 5 because it has the maximum number of terminating nets.

It is claimed that the above methods find nearly optimal results. It is also believed that the quasi–bound– difference method can, in general, convert many local optimization algorithms into nearly global optimization ones.

4 Worst–case performance of the greedy heuristics

We have seen several heuristic algorithms that solve the gate matrix problem. How well do the solutions produced by these algorithms relate to optimal solutions? To address the performance of these heuristic algorithms, we intoduce the concepts of *absolute approximation and relative approximation*. Let $H(I)$ and $OPT(I)$ denote the track requirements of the heuristic and optimal solutions, respectively, of a gate matrix instance I. H is an absolute approximation algorithm if, for every instance I, $H(I) \leq OPT(I) + k$, where k is some constant. H is called a relative approximation algorithm if $H(I) \leq r \bullet OPT(I) + k$, for some constants r and k. When such constants exist, they provide a bound on just how far from the optimum algorithm H may ever deviate. Deo et al. [13] have proved that no absolute approximation algorithms exist for the Gate Matrix problem. It has also been proved in the same paper that the 'greedy' algorithms discussed in the last section are not even relative approximate. In the following paragraphs, we describe the proofs briefly.

The Gate Matrix Layout problem has been shown to be NP- Complete [11]. Now, suppose that an absolute approximation algorithm AA exists. That is, given any Gate matrix instance I, AA uses atmost $OPT(I) + k$ rows where k is some fixed constant. Construct a gate matrix I' that has $k+1$ copies of I. In Fig. 9(a), we show a gate matrix I and a corresponding matrix I' where $k = 2$. Now, for the gate matrix I', the optimal solution takes $OPT(I') = (k+1)OPT(I)$ tracks. Since AA is an absolute approximation algorithm, it takes atmost $(k+1)OPT(I) + k$ tracks to solve I'. This means to say that atleast one of the identical copies took only $OPT(I)$ tracks. Since we did not put any restrictions on I, this means that AA can produce an optimal solution for any instance I. This cannot be true if AA is a polynomial- time algorithm because the Gate Matrix problem is NP- Complete[4].

It is also proved by example that the greedy algorithms of the last section cannot be even relative approximate. Consider the family of instances which have the following structure:

- $n_1 = \{g_1, g_3, g_5, \ldots, g_{n-1}\}$
- $n_2 = \{g_1, g_2\}$

[4]Unless a particular open question P=NP? has the answer true.

- $n_3 = \{g_3, g_4\}$

  $\vdots$

- $n_{n/2+1} = \{g_{n-1}, g_n\}$
- $n_{n/2+2} = \{g_2, g_4, \ldots, g_n\}$
- $n_{n/2+3} = \{g_2, g_4, \ldots, g_n\}$

A greedy heuristic which constructs the gate sequence by appending gates so as to minimize the increase in the size of the largest dominant clique will result in the matrix of Fig. 9(b)[5]. This matrix has $n/2+2$ tracks whereas a 4-track solution is possible[6]. Since $n/2+2$ is not bounded by 4 for any constants r and k, we conclude that the greedy algorithms are not relative approximate.

For small problems, the use of optimal algorithms might be feasible. Deo et al. have used *dynamic programming* [14] to give optimal solutions to the problem. We take up this algorithm next.

4.1 Dynamic programming formulation

What methods are available for obtaining an optimal solution to an arbitrary problem? The obvious method is to use exhaustive search, where we generate all possible gate sequences and retain that one which yields the best solution. This takes $O(n!)$ time, where n is the number of gates. Instead, a *dynamic programming* formulation has been proposed in [13], which takes $O(n^2 \bullet 2^n)$ time. It should be remembered that the gate matrix layout problem is NP–Complete, and that polynomial time algorithms that find optimal solutions are unlikely to be found. In Section 12, another optimal algorithm that is based on the branch–and–bound strategy is discussed.

Algorithm 8: Dynamic Programming

Given a set of gates $G = \{g_1, g_2, \ldots, g_n\}$, denote $c(G, g_i)$ as the minimum number of tracks required for the gate matrix layout if g_i is placed as the last gate. Then, the optimal solution could be found by taking the minimum of $c(G, g_i)$ over all gates $g_i \in G$. We note that if g_i is the only gate in G, then $c(\{g_i\}, g_i)$ =the number of nets that are connected to g_i. Hence, we can adopt a recursive formulation, whereby the size of the set G_j in $c(G_j, g_k)$ is progressively reduced till it has unit size.

[5]The following matrix has to be obtained regardless of the tiebreakers because the last two identical nets ensure that all even–numbered gates have three ones. So, all odd–numbered gates have to be placed before the even–numbered ones.

[6]Net n_1 on row 1; Nets $n_2, n_3, \ldots, n_{n/2+1}$ on row 2; and nets $n_{n/2+2}$ and $n_{n/2+3}$ on rows 3 and 4.

To calculate $c(G_j, g_k)$, it is not enough to find out the best solution among $c(G_j - \{g_k\}, g_j)$ over all g_j, where g_j is distinct from g_k. This is because gate g_k which is placed at the end might have the largest number of nets crossing it (i.e., have a column size greater than that of any of the columns in $G - \{g_k\}$). We can account for this by using two more functions. Let $f(G_j, g_i)$ denote the number of nets that cross g_i, do not terminate at g_i, and are not connected to g_i[7]. Let $r(G_j, g_i)$ denote the number of nets that cross g_i, do not terminate at g_i, and are connected to g_i. Now, we are in a position to define $c(G_j, g_k)$ recursively:

$c(G_j, g_k) = min_{g_i \in [G_j - \{g_k\}]}[max\{c(G_j - \{g_k\}, g_i),$
$f(G_j - \{g_k\}, g_i) + r(G_j - \{g_k\}, g_i)+$ the number of nets that begin at gate $g_k\}]$

i.e., the max checks to see if the last column g_k has the largest number of ones and the min takes the minimum value over all possible combinations. So, as mentioned before, this recursion is continued till we have G_j of unit size. It can be easily proved that this algorithm produces optimal solutions.

So far, we have discussed algorithms that do not take any properties of the Gate Matrix layout itself to generate a solution. We now discuss an algorithm due to Hwang et al. [15,16] that uses the permutability of series-connected transistors to improve the quality of the gate sequence generated.

5 A min-net-cut algorithm using the dynamic net-list property

Consider the example shown in Fig. 10(a). In net n_7, we have gate g bound to gates e and f. But, this is not necessary. Since gates e and d are in series, either of e or d can be connected to g. Similarly, either gate f or gate c could be connected to f. We can delay 'binding' these gates to the fixed gate g till we have decided upon a gate sequence. We could then represent net n_7 as $n_7 = \{g, n_8, n_6\}$. Such a net is called a *dynamic* net. If a net does not have any other net as a component, it is called a *fixed net*. If a net (e.g., n_7) contains any other net (e.g., n_8) as a component, it means that one and only one component of the internal net (i.e., n_8 in this case) is bound to the other components of the first net.

The algorithm uses the *min-net-cut* heuristic [17,18]. The min-net-cut algorithm does the following:

Algorithm 9: Min-net-cut

- It divides the set of gates S into two blocks S_L (the 'left' block) and S_R (the 'right' block) such that
  - the two blocks are roughly of the same size, and
  - the number of nets that are associated with gates in both blocks (called the *cutset*) is minimized.

[7] Though we do not have a gate sequence yet, we know the gates that exist on either side of g_i. Hence, we can calculate the number of nets that cross g_i.

- The procedure iteratively moves one gate at a time from one block to its complementary block so as to minimize the cutset of the final partitioning. The gate to be moved is chosen such that it will result in the greatest decrease in cutset size. This measure is quantified as the *gate-gain*, $Gain(i)$, which is the number of nets by which the cutset would reduce if the gate i were moved from its current block to its complementary block. To prevent the algorithm from becoming a greedy one, gates with negative gains are also moved occasionally. Also, to prevent the same set of gates from shuttling repeatedly back and forth, a gate is *locked* after a move, i.e., it is not available for further exchanges. Also, *pseudogates* are created in both S_L and S_R. The pseudogate in S_L represents all those gates that do not belong to the set S_L and lie to the left of the set S_L. This pseudogate is always locked. This prevents gates which should be placed to the left of S_L from being moved to the right. Similarly, a pseudogate for S_R is defined.

- The min-net-cut procedure is used recursively on blocks S_L and S_R till the final ordering of gates is obtained.

5.1 Computing the gain of each gate

The following general principle is followed to determine whether to increase or decrease the gain of a gate:

- For every net, if the net belongs to the cutset, and if the movement of gate i will result in the removal of gate i from the cutset, then the gain of gate i is increased by one[8].

- For every net, if the net does not belong to the cutset, and if the movement of the gate i will result in the net being added to the cutset, decrease the gain of gate i by one.

- Otherwise, there is no change to the gain of gate i due to this net.

Using the above general principle, we can arrive at the following rules for fixed nets.

- Rule 1: When the gate is the only gate of a net in the current block (*from-side*), increase its gain by one.

- Rule 2: When the net does not have any gates in the complementary block (*to-side*), decrease the gain of the gate by one.

However, when we are considering dynamic nets, we should recognize the role of bound gates. We give the corresponding rules below:

[8]The two rules stated here apply only to those nets which are associated with more than one gate. If a net is associated with only one gate, then the movement of the gate to the other block does not affect the cutset. Hence, such nets do not influence the gain of the gate.

- Rule 3: When the gate is the only bound gate of a net in the from–side, and there is some unbound gate of the net in the to–side, increase the gain of the gate by 1 (i.e., if the gate is moved to the to–side, the net can be completed using the gates in the to–side alone).

- Rule 4: When there are no bound gates of a net in the to–side, and there is some unbound gate [9] in the from–side, decrease the gains of all the bound gates in the from–side by one.

In Fig. 10(b), we illustrate the calculation of gate gains for simple examples containing fixed and dynamic nets.

Once the gate is moved[10], the gains have to be changed again for the next iteration. It is not necessary to again consider the effects of all the nets on the *free* (i.e., not locked) gates. The only gates affected are those that have common nets with the gate that was moved. The rules for updating the gains of these gates are the reverse of the rules stated above. The layout of the example in Fig. 10(a) is given in Fig. 19(c) in connection with the section on net merging.

The min–net–cut algorithm is called recursively on S_L and S_R till a gate ordering is obtained. The initial partitioning into two parts takes $O(N)$ time where N is the total number of transistors and gate–net contacts in the circuit. Since, we divide S into roughly two equal parts, the time complexity of the algorithm is given by the recurrence relation:

$$T(N) = cN + 2T(N/2).$$

This has the solution $T(N) = O(N \log N)$, which is the time complexity of this algorithm.

We have seen in the case of greedy algorithms that there can be unbounded deviation from the optimal solution. How do the solutions produced by the min–net–cut algorithm compare with the optimal solutions? We will first need to define the concept of an *ideal* min–net–cut. An ideal min–net–cut is one which will always produce a partition with a cutset of minimum size. It is proved in the paper that an ideal min–net–cut algorithm would be, in the worst case, $\log n$ approximate, i.e.,

$$A_{ideal}(I) \leq k * log\ n * OPT(I),$$

for all instances I, where $A_{ideal}(I)$ is the number of tracks required by the ideal min–net–cut algorithm. How is this true? Since our min–net–cut is ideal, the size of the cutset in the initial partition cannot be greater than the optimal solution (i.e., the required number of tracks)[11]. Also, we know that the number of tracks required by the ideal min–net–cut algorithm is given by:

[9] It is not entirely accurate to consider all the unbound gates together because they may belong to different fixed nets. This error is rectified in another paper [20]

[10] There is one other factor which decides the gate to be moved. It is possible in some cases to *merge* two nets into one. Net merging is discussed in a later section.

[11] To see this, take the optimal solution and place a partition such that the number of gates on either side is equal. Now, the number of nets that cross this partition line is atmost equal to the number of tracks in the optimal solution. Since this is our cutset, the size of the cutset is never greater than the required number of tracks.

Number of tracks$\leq OPT(I)$ +Max[Number of tracks required by S_L,
Number of tracks required by S_R].

From our earlier proof, we know that the algorithm will have to be called recursively $\log n$ times. Hence, the number of tracks required by the ideal min-net-cut algorithm is bounded by $\log n * OPT(I)$[12].

Min-net-cut is one algorithm that uses iterative improvement to get a good quality solution. There is another popular iterative technique called *Simulated Annealing* [19] that has been used on many other VLSI CAD problems as well. We now discuss an algorithm due to Leong et al. [21,22] that applies Simulated Annealing to the Gate Matrix layout. Another probabilistic paradigm, known as the Genetic Algorithm [23,24], is being used successfully on many VLSI CAD problems [25,26,27,28,29,30]. In the following section, we discuss algorithms that apply these general techniques to Gate Matrix Layout.

6 Probabilistic Algorithms

6.1 Simulated Annealing

Simulated Annealing is an optimization technique that uses an analogy with the problem of determining the lowest energy ground state of a physical system. To bring a fluid to a highly ordered low-energy state, it is necessary to cool it slowly, spending a long time at temperatures in the vicinity of the freezing point. At each temperature in this *annealing* process, slow cooling enables the system to reach equilibrium. In an optimization problem, the cost may be viewed as the energy and the temperature may be viewed as a control parameter which enables the attainment of a 'good' solution from some initial high-cost state. To apply the Simulated Annealing algorithm to a problem, we have to specify the following:

1. The *solution space*. This means that the problem has to be formulated to obtain a concise description of its configurations.
2. A method of obtaining the *neighboring* solutions of a solution. Since we improve the algorithm iteratively, it should be possible to go from any solution to any other solution by a sequence of steps.
3. The cost of a configuration.
4. An *annealing schedule*, i.e., a specification of the initial temperature, the rule for changing the temperatures, the duration of search at each temperature and the termination condition of the algorithm.

[12] Actually, this is a very loose estimate because we have assumed that the size of the cutset at every level (i.e., of recursion) is approximately equal to $OPT(I)$. It is conjectured by the authors that if we assume that the size of the cutset at every level is proportional to the number of gates, min-net-cut could turn out to be a relative approximation algorithm

For the gate matrix layout problem, the above requirements are met as follows:

Algorithm 10: Simulated Annealing

- A layout solution is specified by the following:
  - the gate permutation, and
  - the dynamic binding of the D-nets.

  So, the solution space here consists of all possible gate permutations along with all possible dynamic bindings.

- For the gate matrix layout, we can define two types of *moves* that take us from one solution to its neighbours.
  - M1: interchange the positions of two of the gates.
  - M2: interchange the binding of a D-net (i.e., represent the internal net in a dynamic net by some other transistor)

  When we make a move of type M1, the gate permutation is changed (perturbed), and this may affect the following:
  - Fixed nets that are connected to any of the two gates. The update of the wire length of such nets is straightforward.
  - The bindings of some D-nets. In this case, that binding which results in minimum net length is chosen. If this choice is not clear, then the selection of the binding is guided by the simulated annealing procedure.

  Also, we can go from any solution state to any other solution state by a sequence of M1 and M2 moves. In the Simulated Annealing algorithm, these moves are essentially made at random. However, we notice that move M1 leads to a more global rearrangement of nets as compared to move M2. Hence, the generation of moves is biased such that more moves of type M1 are attempted than moves of type M2.

- The cost function should reflect the fact that our primary aim is to minimize the number of rows. Let d_k denote the number of nets crossing gate g_k. Then, the minimum number of tracks required d is given by $d = max\{d_i\}$ for all $i = 1, 2, \ldots, n$ where n is the number of gates. Additionally, the cost function should include the total length of the nets D as a secondary factor. The algorithm chooses the following:

$$Cost = d^2 + \lambda D^2/n.$$

where λ is a constant chosen to reflect the relative importance of d and D. Hence, λ should be less than or equal to 1. In the algorithm, a value of $\lambda = 0.5$ has been used.

- The temperature schedule is of the form $T_{k+1} = r \bullet T_k$, for $k = 1, 2, \ldots$. Since we have to reduce the temperature during each iteration, a value of 0.8 for r has been selected. The initial temperature T_0 is chosen such that there should be a reasonable probability of acceptance for moves that increase the number of tracks required. The number of tracks required can never be greater than the number of nets m. So, in this implementation T_0 is of the form constant$\bullet m$ (i.e., $O(m)$).

It is very difficult to predict either the average or the worst case performance of the Simulated Annealing algorithm. However, the algorithm has given very good solutions on benchmark problems.

6.2 Genetic Algorithm

The Genetic Algorithm [34] models the process of evolution in obtaining answers to optimization problems. The key steps of the algorithm are as follows:

Algorithm 11: Genetic Algorithm

1. Each solution to the problem is represented as a string called a *chromosome*. We start from an initial random population consisting of several such solution strings.
2. The two operators, *reproduction and crossover*[13] are applied on the members of this *generation* (i.e., population). Reproduction allows particular chromosomes to be present in the next generation in proportion to their *fitness*. Crossover is an operator that takes two chromosomes, and returns their *offspring* which has substrings from both parents. An example of (simple) crossover is shown in Fig. 11(a). Although crossover occurs between random positions, it is very effective in producing better offspring because it acts on above–average chromosomes. Reproduction and crossover are carried out till all members of the new population have been generated.
3. Step 2 is repeated for a predetermined number of generations. The best answer ever found (i.e., the string with the highest fitness function) is output as the final answer.

For the Gate Matrix problem, the chromosome chosen in [30] is the gate sequence itself. However, the simple crossover shown in Fig. 11(a) will lead to illegitimate offspring as shown in Fig. 11(b). Hence,it is necessary to use the *permutation crossover* operators defined in [24]. There are three such operators: *Partially Matched Crossover (PMX)* , *Order Crossover (OX)* and *Cycle Crossover*

[13] In many Genetic Algorithm applications, use is made of several other operators known as mutation, inversion, etc.

(CX). OX chooses one section from the first parent to be transferred without modification to the offspring. The remaining elements are drawn from the second parent in the *order* of their occurence. Fig. 11(c) gives an example of such a crossover. PMX also chooses one section that is transferred in toto to the offspring. The rest of the offspring is obtained from the other parent by placing them in the same positions that they occupy in the parent. This is shown in Fig. 11(d). Cycle Crossover works by identifying a *cycle* within the first parent as follows: Suppose 4 is chosen as the crossover position. In Fig. 11(e), we have a 2 at position 4 in the first parent. Correspondingly, we have a 5 in the second parent. So, CX decides to take 5 from the first parent. This causes the corresponding number 1 also to be taken from the first parent. This causes 4 to be chosen from the first parent, which in turn causes 8 also to be selected. Now, we have a 2 corresponding to 8. Hence, we have completed a cycle. The other elements are chosen from the second parent. The final offspring is also shown in Fig. 11(e). In general, PMX and CX respect the *absolute* position of the genes within the parents, while OX respects the *relative* position of the genes. In the gate matrix layout, we do not place any particular restriction on the absolute position of the gates. As expected, preliminary results given in [41] have shown that OX works best on the gate matrix layout problem.

It is necessary to choose a cost function that reflects the quality of the chromosome. In particular, differences in fitness must be accentuated such that members with higher fitness are reproduced at a proportionately higher rate. To determine the quality of a layout, the primary factor is the number of tracks required. Given a gate permutation, the calculation of the number of tracks is straightforward. For each net, the leftmost and the rightmost gate are found by scanning. The *density* of these two gates as well as the gates in between, is increased by one. The wirelength required to place the net is equal to the difference between the rightmost and leftmost gate positions. After all the nets have been examined, the maximum gate density gives us the number of tracks required. The following cost function C is used:

$$C = k_1 * Numberoftracks + k_2 * wirelength,$$

where k_1 is much larger than k_2[14]. The fitness of a string is the inverse of its cost.

Preliminary results on benchmark problems as given in [30] are very encouraging. The Genetic Algorithm finds solutions comparable to that of existing algorithms in only a few generations. Shahookar and Mazumder [27] have obtained considerable improvement over the basic algorithm by the use of a meta-genetic parameter optimization procedure. This procedure is invoked to determine good values for the control parameters themselves. Typically, good values have been found for crossover probability, population size and mutation rate. It is planned to extend this approach to the gate matrix layout problem as well.

All the algorithms we have discussed so far have assumed that optimizing the n-part of the Gate matrix automatically optimizes the p-part. This is not necessarily true. In the next section, we describe two algorithms that show improvement by considering both halves together.

[14]Typically, k_1 is 25 and k_2 is 0.5.

7 Algorithms for minimizing both n and p–parts together

Consider the netlist given in Fig. 2. If we minimize this layout using Algorithm 2 which minimizes only the n–part, we get the layout of Fig. 2(a). However, if we minimize both parts together, we get the layout of Fig. 2(b) which has fewer rows than the layout of Fig. 2(a). The algorithms we discussed so far assumed that an optimum solution for the n–part (p–part) implied an optimal solution for both parts put together. But, from the above example, we know that this is not true. In this section, we discuss two algorithms due to Fujii et al. [31] and Chen and Hou [20] that minimize the total layout.

7.1 A heuristic for minimizing the total layout

In the earlier section, we defined each net by the set of gates that it was connected to. In this algorithm, the authors partition that set of gates into two parts, the set of input gates and the set of output gates. Also, this algorithm assumes that the circuit is totally composed of only NAND gates, NOR gates and inverters.

A directed graph called a *gate graph* is introduced to represent the ciruit information given in terms of the net–set and the gate–set. Define a gate graph $GG = (V, E)$ as follows:

- The set of vertices V of GG is identical to the set of gates G.
- There is an edge from gate g_i to gate g_j if there exists some net n_k for which g_j is an input gate and g_i is an output gate.
- For each such edge (g_i, g_j), attach n_k as a label.

Fig. 12 shows a net–list and its associated gate graph. Note that we have to specify both the input and output gates for each net.

Once we decide the assignment of the gates, we have to decide where to place the diffusion runs between the nets on different rows. Let the set of nets that require diffusion runs be denoted by D. A *diffusion position function* V_P assigns to each member of D a value equal to either left or right. Left (right) means that the diffusion run should be placed to the left (right) of the net. Fig. 13 gives an example. Figs. 13(a), 13(b) and 13(c) show a NOR gate and two gate matrix realizations. We notice that whatever the gate sequence, only series–connected transistors need diffusion runs (if at all). Parallel–connected transistors do not need diffusion runs. But if we can place the gates in such a way that the output gate is on one side of the input gates (not in between), no diffusion run is required even for series–connected transistors. The lesser the number of diffusion runs, the fewer rows we require. We can state the algorithm now.

Algorithm 12: Minimizing both n and p–parts at once.

- Input: The set of nets N and the set of gates G.
- Output: An assignment of the gates, πG and a diffusion position function V_P such that the layout requires minimum number of rows.

  1. Construct the gate graph GG corresponding to the set of gates G and the set of nets N.
  2. Select a vertex $v \in V$ and set the set of vertices V to $V - v$.
  3. From V, select a vertex v' such that the expected number of rows is minimum when v' is placed next to v. Set $V = V - v'$ and $v = v'$.
  4. Repeat step 3 till an assignment of P is obtained (i.e., V becomes empty).
  5. Find a diffusion position function V_P for P.
  6. Apply the left-edge method to obtain the net assignment[15] for P and V_P.
  7. Repeat Steps 2 to 6 for each vertex in GG and output the best layout obtained among them.

When we apply the algorithm on the example of Fig. 12, the layout of Fig. 2(b) is obtained. The algorithm has a complexity $O(|G|^3 \bullet |N|)$ where $|G|$ denotes the number of gates and $|N|$ denotes the number of nets.

This algorithm has used one heuristic for eliminating diffusion runs. However, there are other possible rules, which if followed will eliminate diffusion runs. Our next algorithm incorporates this 'duality' between the series and parallel parts of a logic gate. Also, the above algorithm takes $O(|G|^3)$ time. It is possible to reduce the complexity by using min-net-cut for generating the gate sequence. This brings us to our next algorithm due to Chen and You [20].

7.2 An algorithm that uses several diffusion elimination rules

Other than the duality rule discussed in the above subsection for eliminating diffusion runs, three other such rules can be developed. We give these rules.

- Rule 1: For a set of series-connected and parallel-connected transistors as shown in Fig. 13(a), all the gates should be placed on the same side with respect to the output line to eliminate diffusion runs. This is shown in Fig. 13(c).
- Rule 2: When two sets of series-connected (parallel-connected) transistors are connected in parallel (series) as in Fig. 14(a) (14(b)), the output line should be placed in between these two sets to eliminate diffusion runs. See Figs. 14(c) and 14(d).

[15] Net assignment algorithms are discussed in a later section.

- Rule 3: When a set of series-connected (parallel-connected) transistors is connected in parallel (series) with a single transistor as shown in Fig. 14(e), this single transistor should be placed between the output line and the transistor set. See Fig. 14(f)

- Rule 4: When more than two sets of series-connected (parallel-connected) transistors are connected in parallel (series) as shown in Fig. 14(g), at least one vertical diffusion run will be required.

The min-net-cut algorithm is used to obtain the gate sequence. However, this algorithm is not identical to the earlier min-net-cut algorithm we discussed. Specifically the concept of what nets constitute dynamic nets (D-nets) and fixed nets (F-nets) is determined using the rules (i.e., constraints) stated above. To represent these constraints, a *duality constraint graph* $DCG = (V, E)$ is introduced. This graph is defined as follows:

- A vertex represents a single transistor or a set of transistors which can be permuted.

- There exists an undirected edge between two vertices if and only if there exists a constraint, from the above rules, between them.

Figs. 15(a) and (b) show a circuit and the duality constraint graph of a part of it.

For the earlier min-net-cut algorithm, two transistors of a net are said to be in a D-net if their positions in the final layout are permutable. Here, this definition is extended to take duality constraints into account.

- Two vertices of the duality constraint graph are said to be in a dynamic net (D-net) if there is an edge between them and one of them has more than one transistor in it. Each vertex is called a *component* of this D-net.

- A set of transistors (i.e., gates connected to the transistors) is called a fixed net (F-net) if they are in the same vertex.

- Two vertices, each of them having exactly one transistor, connected by an edge also form a fixed net.

In Fig. 15(b), $N_{f1} = \{f, c\}$ and $N_{f2} = \{e, d\}$ are the fixed nets while $N_{d1} = \{g, N_{f1}\}$ and $N_{d2} = \{g, N_{f2}\}$ are the D-nets. This representation fits the circuit better than the representation adopted in [15]. In [15], this circuit would have been represented as having one D-net $\{f, c, e, d\}$ with the condition that gate g should connect to any gate belonging to this set. Actually, the gate g should connect to one gate belonging to set $\{f, c\}$ and one gate belonging to set $\{e, d\}$.

A min-net-cut algorithm similar to the earlier min-net-cut algorithm is used here. However, the gain function has to be modified to take into account the new representation. Some additional rules have been introduced to calculate the gain function:

- When a gate is an F–net, and no other gates in this F–net are in the current (opposite) block, the move of this gate will increase (decrease) the gain by one.
- When a gate is a component of a D–net, and an F–net which is also a component of the same D–net is in the opposite (current) block (i.e., all the gates of the F–net are in the block), the gain of the gate is increased (decreased) by one.

The procedure is described below:

Algorithm 13: A min–net–cut algorithm using diffusion elimination rules

- Step 1: Construct a duality constraint graph.
- Step 2: Establish an initial partition according to the duality constraint graph constructed in Step 1.
- Step 3: Recursively perform the min-net–cut algorithm, i.e., at each stage, compute the gain functions for all gates and move the gate with the highest gain. After the move of this gate, recalculate the gain function for all gates which have common nets with the moved gate.

The layout of the circuit of Fig. 15(a) is shown in Fig. 15(c). Since this algorithm is also based on the min–net–cut principle, it is of time complexity $O(N \log N)$ where N is the total number of transistors and gate–net contacts.

We now discuss an algorithm due to Huang and Wing [32,33] that searches for a good *net sequence* rather than a good gate sequence. This algorithm is based on an algorithm developed by Asano [34] for one–dimensional arrays. Asano's algorithm is discussed in a later section devoted to one-dimensional arrays. Once we have a net sequence, we can construct a gate sequence corresponding to it.

8 Searching for a good net sequence

At any stage of the net selection process, we can identify the following categories of nets:

- N_L = {the set of I/O nets emerging from the left boundary[16]}.
- N_R = { the set of I/O nets emerging from the right boundary}.
- N_M = { the set of nets connected to gates which have pre–fixed positions}.

[16]This is a practical constraint that earlier algorithms have not dealt with. The positions of some nets have to be restricted to the boundaries to accomodate implementation constraints. Also, it is sometimes necessary to keep certain gates in pre–fixed positions

- m = { the set of nets which have already been selected}
- $t(m)$ = { the set of unselected nets that have some common gate with the set of selected nets}.
- $A(n_i)$ = { the set of nets that have some common gate with net n_i}.
- $A_R(m, n_i)$ = { the set of nets that have some gate in common with n_i, but which belong neither to m nor to $t(m)$}.

Fig. 16 shows the sets m, $t(m)$ and $A_R(m, n_i)$ for some set of selected nets m. Now, to choose a net, we have to evaluate its 'goodness', i.e., we have to obtain an estimate of the number of rows we will require if we choose this net now. From Fig. 16, we know that if we choose n_i, we will not need more than $|t(m)| + |A_R(m, n_i)|$ tracks[17]. So the number of tracks required will be either $|t(m)| + |A_R(m, n_i)$ or number-of-tracks(m) whichever is maximum. Since for all n_i being considered, $|t(m)|$ is common, we only need to evaluate $|A_R(m, n_i)|$. In addition, if there is a tie, the net with most neighbours (larger $|A(n_i)|$ is chosen. So, we can define the evaluation function $f_E(m, n_i)$ as follows:

$$f_E(m, n_i) = \begin{cases} undefined, & \text{if } n_i \text{ belongs to either } m \text{ or } N_R \text{ or } N_M) \\ c \times A_R(m, n_i) - |A(n_i)|, & \text{otherwise,} \end{cases}$$

where c is a sufficiently large constant that restricts the use of $A(n_i)$ to cases which involve ties. Also, a net that belongs to N_M is released from it if all the pre-fixed gates that it is connected to have been assigned positions. We can now state the algorithm.

Algorithm 14: An algorithm for gate assignment by net sequencing

- Input: The sets N_L,N_R and the set of gates G_F, whose column positions C_F are fixed.
- Output: A gate sequence which minimizes the number of tracks and ensures that the constraints placed are obeyed.

1. Set $m = \emptyset$; $t(m) = N_L$; current-col-position = 0.
2. While (current-col-position$< |G| - 2$) repeat steps 3 to 6.
   /* That is, we continue till each gate has been assigned a place. */.
3. For each net n_i that does not belong to $[m \cup N_R \cup N_M]$, calculate $f_E(m, n_i)$. Choose that n_i which has the minimum value of $f_E(m, n_i)$ and update m to include n_i.

[17] In [32], this expression is modified to reflect the role played by *oversized transistors*. Oversized transistors are represented in consecutive rows on the same gate

4. For each gate g_j that is connected to n_i and has not already been assigned a position, do step 5.
5. If current–col–position$\in C_F$, then continue this step; otherwise do step 6.
   Find the gate g_f in G_F that should be assigned to this position. Assign g_f and update N_M. If g_j is different from the gate g_f, assign g_j to the next position and update current–col–position.
6. Since current–col–position is available, place g_j at this position and increase current–col–position by 1.

In Fig. 17, we show examples of net selection and assignment of gates to positions. The time taken by the algorithm depends on the number of nets. We now describe another algorithm that takes into account another practical constraint, the height–width ratio, of gate matrix layouts. A modified version of the net sequencing algorithm is used to get the gate sequence.

In a circuit, it is possible for a large number of gates to have many fewer nets than the lower bound (i.e., the maximum number of nets connected to any gate). In such a case, the methods we have discussed so far will yield a layout that has very low density. Also, the width (i.e., number of gates) to height (required number of tracks) ratio, known as the *aspect ratio* may become unacceptably large. Such a situation often arises because the nets on a gate vary over a wide range. So, it is not necessary to place all gates in a single sequence. We can divide the set of gates into two or more (not necessarily disjoint) parts and place them on 'top' of each other. Huang and Wing introduced gate matrix *partitioning* [35] which does this division in a systematic manner.

9 Gate Matrix Partitioning

The partioning algorithm divides a gate matrix into two blocks at a time, an upper block and a lower block. Gates that have nets in both blocks form the interconnections between the blocks. To avoid the use of any extra routing area, the interconnecting gates are placed in the same column in both blocks. The other gates in a block intersect nets only inside the block, so that the width of each block is smaller. Each partitioned block has width smaller than the width of the original matrix. However, the sum of the heights of all the blocks may be larger than the height of the original matrix (because we stack one block on top of another). The min–cut algorithm is used here to partition the matrix into two blocks. The cutset is made up of the interconnecting gates. The min–cut algorithm can then be called recursively upon the two blocks till the rectangle that circumscribes the entire set of partitioned blocks has the best combination of height and width. Once the partitioning process is finished, the gate matrix layout of each block is done separately. We can now define the problem formally.

9.1 Problem Definition

Consider a gate matrix $M = (G, N, R)$ where G represents the set of gates, N the set of nets and R the set of rows. Let $N(g_i)$ denote the set of nets associated with gate g_i and $G(n_j)$ denote the set of gates associated with net n_j. Let t denote the total number of gate–net contacts and let n be the total number of nets. The symbolic area (i.e., number of grid points) is given by $|G| \bullet |R|$.

Our goal is to partition the net set N into two disjoint subsets, N_1 and N_2, forming two gate matrices, the upper matrix $M_1 = (G_1, N_1, R_1)$ and the lower matrix $M_2 = (G_2, N_2, R_2)$ such that the total area and the aspect ratio of the layout are better than that of the layout M. If we know N_1 and N_2, we can calculate the gate sets G_1 and G_2 as
$G_1 = \{\text{gates intersecting nets in } N_1\} = \cup_{n_j \in N_1} G(n_j)$
$G_2 = \{\text{gates intersecting nets in } N_2\} = \cup_{n_j \in N_2} G(n_j)$

To implement the min–cut algorithm, we had to specify the following:

1. the balance condition, and
2. a method of computing the gain, and
3. a method of updating the gain after an entity is moved from one block (*the source block*) to the other (*the target block*).

9.2 The balance condition

The balance condition employed here ensures that the height (i.e., the number of rows) of one block is roughly the same as that of the other, i.e., $|R_1| \approx |R_2|$. However, when we partition the matrix into two blocks, we do not know the heights of the blocks. The heights of the blocks can only be known after assigning the nets to the rows. This places a severe computational burden at each step of the partioning process. To avoid this burden, we can estimate the height by a lower bound given by the maximum number of nets on a gate in the block. We know that the height can never be lower than the maximum number of nets on a gate. The authors also produce experimental evidence to show that the height increases as the lower bound increases. Hence, the use of the lower bound instead of the actual height is permitted. Just as we defined pseudogates in the original min–net–cut algorithm to maintain the relative positions of the gate sets, we define two virtual nets, n_{top} and n_{bottom} here to maintain the relative vertical order of the blocks. n_{top} intersects all gates extending to the top edge and n_{bottom} intersects all gates expanding to the bottom edge. In the partitioning process, n_{top} must always be in the upper block, and n_{bottom} must be in the lower block.

9.3 Computing the gain function

The gain function for a net has two parts:

1. If by moving the net from one block to another, we can reduce the number of interconnecting gates, we stand to gain by moving the net from one block to another. This is called the *cutgain*. To calculate the cutgain of a net n_i, we go through the following steps:
   - For every gate g_k associated with this net n_i do
     * Define $partialgain(g_k, n_i) = 1$, if n_i is the only net that gate g_k has in the source block (i.e., g_k will be removed from the set of interconnecting gates);
     * else $partialgain(g_k, n_i) = -1$, if gate g_k has no net at all in the target block (i.e., g_k will be introduced into the set of interconnecting gates;
     * else $partialgain(g_k, n_i) = 0$, otherwise.
   - $cutgain(n_i)$ is the sum of the partialgains of n_i over all its associated gates.
2. We would also want to move a net that can reduce the height of the blocks. This gain term is called the *heightgain*. To calculate this, we have to first identify those gates in the block that have maximum number of nets within the block. These nets are *critical nets* in the sense that only nets attached to these gates can reduce the lower bound.

   If a net intersects all critical gates in its source block, then moving it to the other block will cause the lower bound of the source block to be lowered by one. If on the other hand, the net intersects some critical gate in the target block, the lower bound of the target block will be raised by one. Even if the movement of the net does not change the lower bounds of either block, it might reduce the number of critical gates in the source block. This makes it more likely that another net moved from this source block will reduce the lower bound of the block. We want to assign some positive gains for such nets too. We can now state the following five rules.
   - H1: $heightgain(n_j) = 2$, if n_j intersects all critical gates in its block and does not intersect any critical gate in its target block.
   - H2: $heightgain(n_j) = 1$, if n_j intersects some critical gate (but not all) in its block and does not intersect any critical gate in its target block.
   - H3: $heightgain(n_j) = -1$, if n_j intersects some critical gate in its block, and also intersects some critical gate in its target block.
   - H4: $heightgain(n_j) = -2$, if n_j does not intersect any critical gate in its block, but intersects some critical gate in its target block.
   - H5: $heightgain(n_j) = 0$, otherwise.

   Finally, $Gain(n_j) = Cutgain(n_j) + Heightgain(n_j)$.
   Fig. 18 shows the cutgains and heightgains of several nets using these rules.

9.4 Updating the cutgains and the heightgains

Once we have calculated the cutgains and the heightgains after the initial (given) partition, it is not necessary to recalculate this value from scratch after the move of a net from one block to another. The net n_j that has been moved is *locked*. All nets that are not locked are available for movement and are called *free*. The updates of the cutgains and the heightgains of the free nets are done as follows:

- Cutgain update:
  We consider each gate g_i that is associated with net n_j. There are four cases that we have to consider:

  1. If g_i does not have any nets in the target block other than n_j, increase the cutgains of all free nets of g_i (because these nets can now be moved to the target block without increasing the number of interconnecting gates).
  2. If g_i has only one net in the target block other than n_j and this net is free, reduce the cutgain of this net by one (because moving this net is not going to remove g_i from the set of interconnecting nets).
  3. If g_i has no nets left in the source block, reduce the cutgains of all its free nets by one (because these nets cannot be moved without putting g_i into the set of interconnecting nets).
  4. If g_i has only one net left in the source block and this net is free, increase the cutgain of this net by one (because moving this net will remove g_i from the set of interconnecting gates.

  When we go through the four steps above for all gates associated with the moved net n_j, we are done. We now have to calculate the change in height gains of the free nets.

- Heightgain update:
  We perform two steps here. In the first step, we determine the set of free nets D that need updates to their height gains. In the second step, we use the five rules H1 through H5 stated above to calculate the new heightgains of these nets.

  1. Include in D all nets that have some common gate with net n_j. Also, check if the lower bound of the source matrix has changed. If so, make the new set of critical gates. Add to D all the nets that are associated with any of the critical gates. Finally, check if the lower bound of the target matrix has changed. If so, repeat the process that was carried out for the source matrix.
  2. For all nets in D, recalculate the heightgains using the five rules H1 through H5 stated above.

We now summarize the partitioning algorithm:

Algorithm 15: PARTITION

- Input data from the data–list.
- Identify gates which have lengths greater than lower–bound/2 and call them long gates.
- Randomly generate an initial partition (N_1, N_2).
- Repeat PASS until the result (N_1, N_2) is unchanged.

Algorithm 16: PASS

- Initialize the gains of the nets.
- While (a gate to be moved n_c can be found) do
  - Lock n_c and complement its block.
  - Update the gains of the free gates.
  - If (the current partition (N_1, N_2) is better than the recorded one), retain (N_1, N_2).

The algorithm PASS takes $O(t) + O(|N_L|^2)$ time. However, we still have to assign the gates to their respective positions. Also, to ensure that no extra routing area is necessary betweem any two partitioned blocks (i.e., matrices), gates have to be assigned the same (absolute) positions in each matrix. This is acheived by the *simultaneous gate sequencing algorithm* discussed next.

9.5 Simultaneous gate sequencing

After PARTITION, the set of nets has been partitioned into a number of subsets b, i.e. $N = N_1 + N_2, \cdots, N_b$. Each subset N_x comprises the nets of a partitioned matrix $M_x = (N_x, G_x, R_x, C_x)$, in which G_x is the set of gates, R_x the set of rows and C_x the set of columns. The gate set G_x includes the gates intersecting nets in N_x as well as gates that intersect nets in both $N_r (r > x)$ (i.e., in some matrix above M_x) , and $N_s (s < x)$ (in some matrix below M_x) even if these gates do not intersect any net in N_x itself. This is because such gates will have to run through M_x. The matrices to which a gate belongs are called its *home matrices* and this set is denoted as $H(i)$. A column is denoted as c_{xi}, where x is its matrix index and i is the position of the column.

Our problem is to allot each gate in each matrix to a column position such that:

- the total height is minimum, and
- those gates that run in more than one matrix are assigned the same position in all matrices.

An algorithm similar to that of the net sequencing algorithm is used. There are two important changes.

- Associated with each matrix, we have a gate buffer. This buffer holds gates that are to be placed in the matrix. When every buffer has atleast one gate, a set of gates that matches the set of columns $c_{1k}, c_{2k}, \ldots, c_{bk}$, where k is the current column index, is moved from the buffers to the columns [18]. If a gate has multiple home matrices, it must be included in the buffers of all the home matrices.

- The evaluation function $f_E(m, n_j)$ has an objective function similar to that of the net sequencing algorithm. However, the total height of all the matrices put together is evaluated and not the height of a single matrix. The evaluation function is now defined as

$$f_E(m, n_j) = c \sum_{x=1}^{b} h_x(m, n_j) - |A(n_j)|,$$

  where $h_r(m, n_j)$ denotes the height of matrix r if net n_j is selected. As before, a large value for the constant c is selected. This enables us to break ties.

We can now put the algorithm together.

Algorithm 17: SIMULTANEOUS–GATE–SEQUENCING

- Input: The set of partitioned matrices $M_1, M_2, \ldots, M_b$
- Output: A gate sequence that minimizes the total height while allocating the same absolute positions to gates that run in more than one matrix.

1. In each matrix, initialize the set of selected nets m_x to the empty set $\emptyset$ and the height of the matrix $h_x(m)$ to 0. Set the column index p to 0.
2. Define the set of nets to be considered for placement as the set of all nets of all the empty buffers. /* We allocate gates to columns only when all the buffers are non-empty.*/ From this set, choose that net n_c for which $f_E(m, n_c)$ is minimal. Update the buffers of all the matrices to include the gates connected to net n_c. Also, update the evaluations of all nets which are neighbours of n_c.
3. While (there is not an empty buffer) do
   For $(x = 1, 2, \ldots, b)$ do

[18] This can create conflicts of the form shown in Figure 18(a). This is resolved using *doglegs* which increase the total height by one [33]. Only those doglegs which do not go away by exchanging columns are retained. Many doglegs can be discarded by exchanging two columns. However, we shall not discuss this in detail.

Choose a gate g_i from buffer(x).
Remove g_i from buffer(x)
Assign g_i on column p of matrix x.
endfor
$p = p + 1$.
endwhile

The total time complexity of algorithm SIMULTANEOUS–GATE–SEQUENCING has been shown to be $O(t \bullet n)$ where t is the number of gate–net intersections and n is the number of nets.

With this algorithm, we have discussed all algorithms that yield a gate sequence. After a gate sequence is obtained, we need to assign the nets to a minimal number of rows. This is the *net assignment* problem. Before net assignment is done, we can perform an operation known as *net merging* which can reduce the number of tracks we require.

10 Net merging

In the formulation we have used so far, we have assumed that if two nets have any gate in common, then they cannot be placed on the same track. Consider the example of Fig. 19(a). Since nets n_2 and n_3 share only the terminal gate g_3, they can be merged into one net. This is shown in Fig. 19(b). In [6] and [21], net merging is performed after we obtain the gate sequence. In the min–net–cut algorithm [16] however, nets are merged during the base–gate selection process. This gives a correct estimate of the cutset. The rule for merging a dynamic net with a serial net is slightly different, though. When the dynamic net has all its unbound gates on one side and all its bound nets on the other, the D–net can be merged with the fixed net. An example of this is shown in Fig. 19(c) which is the layout for the example of Fig. 10(a).

It is possible to merge two nets even if they do not have a transistor in common. In [36], an algorithm that merges nets in a more comprehensive fashion is presented. In our representation so far, we have assumed that each row–column intersection is either a transistor or a (metal–poly) contact. Actually, we can classify these contacts into six categories as seen in Fig. 20. The number of *units* (of vertical length) that a contact occupies is also shown. The *density* $D(g_i)$ of a gate g_i is defined as the minimum number of units required to accomodate all the nets associated with the gate g_i. The minimum number of tracks required to place all the nets is equal to the maximum density, $D_{max} = max[D(g_i), D(g_2), \ldots, D(g_m)]$, where m is the total number of gates. The algorithm proceeds in two steps:

1. Given a gate–list, the density of all gates and the maximum density is calculated.
2. The left–edge net assignment algorithm[19] is modified to merge nets using information generated in step 1. while assigning nets to rows.

[19]To be discussed in the section immediately following this one.

To represent all the types of contacts we saw in Fig. 20, we denote a gate–net intersection by the 6–tuple: $\langle g, n, t, c, l, r \rangle$[20], where

- g is the gate number (i.e., g_1 has the gate number 1 etc.);
- n is the net number;
- t is the type of intersection where

$$t = \begin{cases} 0, & \text{for the metal-to-poly contact} \\ 1, & \text{for the horizontal metal line} \\ 2, & \text{for the source (drain) of a transistor whose drain (source) is connected to } V_{DD} \text{ or ground} \\ 3, & \text{for the source (drain) of a transistor whose drain (source) is connected to a net} \end{cases}$$

- c is equal to 0, if $t \leq 0$. When $t = 3$, i.e. when the 6–tuple represents the source (drain) of a transistor whose drain (source) is connected to the net numbered c.
- l (or r) is equal to 1 when the intersection is the leftmost (rightmost) endpoint and 0 otherwise.

Fig. 21 shows the 6–tuples for a variety of connections.

We are now ready to describe the algorithms for steps 1 and 2 above.

Algorithm 18: DENSITY–FUNCTION

- Input: The gate permutation and the gate for which we wish to find the density, g_k.
- Output: The density function $D(g_k)$.

1. Initialize $D(g_k)$ to 0.
2. Identify any two 6–tuples $\langle g_k, n_i, 3, c_i, l_i, r_i \rangle$ and $\langle g_k, n_j, 3, c_j, l_j, r_j \rangle$ with $n_i = c_j$ and $n_j = c_i$ as a *pair of 6–tuples*.
3. For each pair of 6–tuples obtained in step 2, if l of one 6–tuple is 1 and r of the other is 1, then add 1 to $D(g_k)$ and delete the pair of 6–tuples (because they terminate at the same gate, these two nets can be merged into one); otherwise, if l (or r) of one 6–tuple is 1, mark the 6–tuple with lh to indicate left half (resp. rh to indicate right half); delete the other 6–tuple and add 1 to $D(g_k)$. This corresponds to Fig. 20(b).
4. For each 6–tuple with $t = 2$ and l (or r)= 1, if there exists a 6–tuple with $t = 1$, mark the 6–tuple of $t = 2$ with lh (or rh), add 1 to $D(g_k)$ and delete the 6–tuple of $t = 1$; otherwise, add 1 to $D(g_k)$ and delete the 6–tuple of $t = 2$. This corresponds to Fig. 20(c).

[20] A n–tuple is a sequence of n elements; for example, (1,1,1,0,0) is a 5–tuple.

5. For each 6–tuple with $t = 0$ and l (or r)=1, if a 6–tuple marked with rh (or lh) can be found, add 1 to $D(g_k)$ and delete the two 6–tuples (these two 6–tuples can share the same space as in Fig. 20(d).
6. For each 6–tuple marked lh, if a 6–tuple marked rh can be found, add 1 to $D(g_k)$ and delete the two 6–tuples (these 6–tuples can share the same space).
7. Finally, add the number of remaining 6–tuples to $D(g_k)$.

The maximum of $D(g_k)$ is taken over all gates g_k. This is the maximum density D_{max}. Fig. 22(a)shows the calculation of $D(g_k)$'s and D_{max} for an example layout. We will now describe the algorithm that actually does the merging of the nets. Net assignment is also done simultaneously using the left–edge algorithm which will be described in the next section.

Algorithm 19: NET–ASSIGNMENT

- Input: The gate permutation, along with the gate densities $D(g_i)$'s and the maximum gate denisty D_{max}.
- Output: An assignment of nets to rows including all possible mergings.

1. For each pair of 6–tuples $\langle g_i, i, 3, j, 1, 0\rangle$ and $\langle g_i, j, 3, i, 0, 1\rangle$, merge the corresponing nets i and j.
2. For a gate column g_i, if the number of 6–tuples with $g = g_i$ is larger than D_{max}, mark the gate column with mg to indicate that there are mergings to be made at this column. (Since we know from DENSITY–FUNCTION that the assignment will require atmost D_{max} rows, any gate column that has more than D_{max} 6–tuples must have some mergings).
3. Apply the left–edge algorithm to assign a net[21]. Once a net has been assigned, let the gate positioned at the endpoint of the net be denoted by g_{end}. If g_{end} has not been marked with mg, then assign a net beginning from $g_{end} + k$, choosing k as the smallest number greater than or equal to 1. If g_{end} has been marked with mg and if the corresponding 6–tuples had been merged during the density function calculation, then assign a net beginning from g_{end}. If not, then start from $g_{end} + 1$.

Fig. 22(b) shows the steps used in obtaining the density of gate F. Using the same method on all the gates leads to the 3–track solution shown.

Once we obtain the gate sequence, we have to assign individual nets to the rows to complete the layout. As we will see, this is not a trivial problem because of the constraints imposed by the gate matrix style. In the next section, algorithms for net assignment are given.

[21]The reader is advised to browse through the material on the left–edge algorithm in the next section.

11 Net Assignment algorithms

The first algorithm that was used for the net assignment problem was the left–edge algorithm developed by Hashimoto and Stevens. This algorithm was originally developed for wire routing. We describe the algorithm below.

Algorithm 20: Left–Edge–First

- Input: The gate permutation and the set of nets.
- An assignment of the nets to rows such that the number of rows required is minimum.

1. Consider each gate succesively from left to right in the given order. For each gate do
   - For each unassigned net connected to this gate do
     * Check if this net can be assigned to any of the existing rows. If it can, assign it to that row and mark the net as 'assigned'. If it cannot be assigned to any of the existing rows, assign this net to a new row, and mark it 'assigned'.

Fig. 23(b) shows the step–by–step assignment of nets for the net–set in 23(a) and the gate sequence $[H, A, B, C, F, G, E, D]$. Notice however that there are *conflicts* which prevent us from realizing the diffusion runs between nets 7 and 8, and between nets 3 and 4. We will prove later that assigning the nets to the rows such that there are no such conflicts is a very difficult problem. Now, we will present two heuristic methods [6], [33] that assign nets to rows such that the number of conflicts is minimized.

11.1 Permuting the rows after Algorithm Left–Edge–First has been applied.

As we saw in the example above, the assignment given by the Left–edge–first algorithm may not be implementable. We can immediately think of a straightforward solution to the problem. Suppose a conflict is occuring at column c_i. We could increase the spacing between c_i and its left (or right) neighbour so that we can realize the diffusion run[22]. For example, in Fig. 23(b), if increase the spacing between columns G and E, we could implement the diffusion run between nets 7 and 8. However, it might be possible to eliminate (or atleast, reduce) the number of conflicts by permuting the rows. In Fig. 23(c), we have placed the rows in the order $[5, 4, 2, 1, 3]$. There are no conflicts now. Wing et al. [6] have given a method of constructively placing the rows. We will discuss that algorithm next.

Algorithm 21: Realizable sequence construction

[22]So, what we said in section 1 about the spacing between columns being fixed is not entirely true.

The algorithm constructs a realizable sequence row by row. To select the 'next' row, we need a representation that depicts the entire relationship between the rows. Using this representation, we can choose a row by keeping in mind the global effect of our choice. Such a representation is provided by the *row adjacency graph* $J(V, E)$. Each row is represented as a vertex. There is an edge between two vertices v_i and v_j, if there is a diffusion run between some net n_p in v_i and some net n_q in v_j. The row adjacency graph of the layout of Fig. 24(a) is shown in Fig. 24(b). A *cycle group* is a set of vertices in which every vertex belongs to a cycle[23] or to a set of cycles in which every pair of cycles have atleast two vertices in common. In Fig. 24(b), $\{1,3,4\}, \{1,2,4\}$ and $\{1,2,3,4\}$ are cycle groups. In the algorithm, a partial sequence is constructed for every cycle group and the positions of the other rows are decided later. This is done because we know that the partial sequence for every cycle group has to be realizable for the entire sequence to be realizable. How do we decide which vertex of the partial group to place as the initial vertex? This is done by considering *extensions*. An *extension* is a path which begins at a vertex of a cycle group and terminates on a isolated vertex of a vertex of another cycle group. In Fig. 25, ($\{1,2\}, \{5,8\}$ and $\{5,6,7\}$ are the extensions. The vertex (row) with the longest extension (the *source*) has the 'deepest' connection with rows not in this cycle group. Hence, a partial sequence is started with the vertex having the longest extension and ended with the vertex having the second longest extension (the *sink*).

How do we choose the other vertices of the partial sequence? At each step, we define a set of *acceptable candidates*, K. A vertex is considered to be acceptable if it has a diffusion run with some vertex that has already been selected *and* if it does not cause a conflict between a vertex that has already been selected and a vertex that has not been selected yet. Fig. 24(c) gives some examples of acceptable and unacceptable candidates. We can now define the following steps:

- If there is only one acceptable candidate, choose that row as the next vertex in the partial sequence.
- If there are more than one acceptable candidates, the tie is resolved by applying the following rules in the order shown.
    1. Choose that candidate which has the maximum number of diffusion runs to rows already selected.
    2. For each candidate, define a *risk function* that gives the risk taken if this candidate is not selected, i.e., the possibility of conflicts being introduced later if this candidate is not chosen. Then, select a candidate that has the maximum risk.
    3. Choose that candidate which is nearest to the sink (in terms of shortest path).
    4. For each candidate, assume that it has been selected and form the fresh list of candidates that are now eligible, $K_{grandsons}$. Choose that candidate which has the largest $K_{grandsons}$.

[23] A cycle in a graph is a sequence of vertices $v_1, v_2, \ldots, v_k$, such that each of $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1)$ are edges in the graph. A graph might or might not have cycles.

Fig. 24(c) shows the layout obtained after the application of the above rules to the layout of Fig. 24(a).

- If the set of candidates is empty, choose a candidate y, and for each conflict induced by y, increase the spacing between columns as necessary to implement all the diffusion runs between y and the set of selected vertices. Choose y in the order of priority shown below:
    1. Choose that candidate which has the minimum number of diffusion runs to rows already selected.
    2. Select a candidate that has the maximum risk.
    3. Choose that candidate which is nearest to the sink (in terms of shortest path).
    4. Choose that candidate which has the largest $K_{grandsons}$.

However, the above algorithm has not been very effective in eliminating conflicts. This is because it uses the left–edge algorithm as its basis, and the left–edge algorithm does not take realizability into account when it assigns the nets. We now discuss another algorithm due to Huang and Wing [32,33].

This algorithm uses the concept of *zones*. A zone is a consecutive set of columns with the following properties:

1. Every net whose left end lies on a column in this set overlaps every other net which also has its left end in this set.
2. Each of these nets overlaps any net whose right end lies on a column in this set.
3. If one more column is added to this set, the set will not have properties 1 and 2, i.e., the set is maximal.

In Fig. 26, a layout and its zones are shown.

Net assignment is done zone by zone from left to right. A row is said to be *available* in zone z_i if it is filled only at the columns to the left of z_i. In Fig. 26, rows 3 and 4 are available in zone z_2. Denote the set of rows available in zone z_i as $AR(z_i)$. The set of associated nets of zone z_i is denoted by $N(z_i)$. In Fig. 26 again, $N(z_2) = \{n_4, n_7\}$. If there are more nets in $N(z_i)$ than rows in $AR(z_i)$, new rows are added to the gate matrix. But, the real advantage of the concept of zones is that an associated net can be assigned to any of the available rows. So, we can assign a net to that row which makes it possible to realize the diffusion runs emanating from the net. Also, define $E(z_i)$ as the set of nets that terminate in zone z_i, and $B(z_i)$ as the set of nets that begin in zone z_i.

Before we describe the algorithm, we have to discuss the nature of two graphs known as the *net relation graph*, G_n, and the *row relation graph*, G_r, respectively. The net relation graph $G_n(N, E_n)$ has each net as a vertex. There is an edge between two vertices (nets) if there exists a diffusion run between the two nets. The row relation graph $G_r(R, E_r)$ consists of each row of the matrix as

a vertex. Rows r_i and r_j are connected if r_i should be placed adjacent to r_j. Initially, the edge set E_r is empty. During the course of the algorithm, edges are added to E_r. We can now describe the algorithm.

Algorithm 22: Zone–Net–Assignment

- Input: The gate sequence and the set of nets.
- Output: An assignment of nets to rows with minimal number of increases in gate spacing.
- Identify the zones of the given gate sequence.
- For each zone z_i do
  - Set $AR(z_i) = AR(z_{i-1}) \cup E(z_{i-1})$ and $N(z_i) = B(z_i)$.
  - Call MATCHING with $AR(z_i)$ and $N(z_i)$¿
  - Update the row relation graph G_r to reflect the row adjacency requirements created by MATCHING
  - For every net $n_i \in N_(z_i)$ that could not be assigned a row, assign a new row and add it to the best possible row position using information in G_r.
- endfor

Algorithm 23: MATCHING

- Input: the set of available rows and the set of nets belonging to a zone
- Output: An assignment of nets to the rows.
- Use the CRITERIA listed below to find a pair of the form (row,net).
- Add the net to the row and update the sets $AR(z_i)$ and $N(z_i)$. Also, update G_r.

The CRITERIA are given below in order of importance.

1. If all rows in $AR(z_i)$ are free, match them to nets arbitrarily.
2. If net n_j is adjacent to n_k in the net relation graph G_n, and n_k is assigned to a row adjacent to $r_i \in AR(z_i)$, match n_j to r_i.

a vertex. Rows r_i and r_j are connected if r_i should be placed adjacent to r_j. Initially, the edge set E_r is empty. During the course of the algorithm, edges are added to E_r. We can now describe the algorithm.

Algorithm 22: Zone–Net–Assignment

- Input: The gate sequence and the set of nets.
- Output: An assignment of nets to rows with minimal number of increases in gate spacing.
- Identify the zones of the given gate sequence.
- For each zone z_i do
  - Set $AR(z_i) = AR(z_{i-1}) \cup E(z_{i-1})$ and $N(z_i) = B(z_i)$.
  - Call MATCHING with $AR(z_i)$ and $N(z_i)$¿
  - Update the row relation graph G_r to reflect the row adjacency requirements created by MATCHING
  - For every net $n_i \in N_(z_i)$ that could not be assigned a row, assign a new row and add it to the best possible row position using information in G_r.
- endfor

Algorithm 23: MATCHING

- Input: the set of available rows and the set of nets belonging to a zone
- Output: An assignment of nets to the rows.
- Use the CRITERIA listed below to find a pair of the form (row,net).
- Add the net to the row and update the sets $AR(z_i)$ and $N(z_i)$. Also, update G_r.

The CRITERIA are given below in order of importance.

1. If all rows in $AR(z_i)$ are free, match them to nets arbitrarily.
2. If net n_j is adjacent to n_k in the net relation graph G_n, and n_k is assigned to a row adjacent to $r_i \in AR(z_i)$, match n_j to r_i.

3. Nets in G_n that have a diffusion run between them are called adjacent nets. A *component* of G_n is a subgraph of G_n such that any two nets in the component have a path between them. We wish to place nets belonging to a component to adjacent rows. So, this step looks for a component p_s of the row relation graph G_r that is *complete*, i.e., a set of rows in which none of the assigned nets are adjacent to any of the unassigned nets (which means that any of the nets can still be placed on these rows). Such a complete component should be in the set of available rows $AR(z_i)$. If the number of rows in the component p_s is greater than the number of nets in the component of G_n (this component of nets must be in the set of unassigned nets $N(z_i)$), we also match some free nets along with the nets belonging to the component. If it is not, we match as many nets as possible onto the rows. The matching should maintain the adjacency properties of the rows and the nets.

4. This criterion is similar to Criterion 3. Here, however, we look for a *complete* set of nets H. This set H contains all the nets of a component of G_n. We look for a component p_s of $AR(z_i)$ that is bigger than H[24]. If we cannot find a p_s bigger than H, we add some free rows. The matching should maintain the adjacency properties of the rows and the nets.

5. Match every free net in $N(z_i)$ to a non–free row in $AR(z_i)$. The longest free net is matched first. That row in G_r which has the maximum number of adjacent rows is matched with this net. We do this because we try to keep the least constrained (in terms of adjacency) rows for nets with many diffusion runs. This minimizes the chance of a conflict.

6. Match every free row in $AR(z_i)$ to a non–free net in $N(z_i)$.

7. Match net $n_j \in N(z_i)$ to some row $r_i \in AR(z_i)$ if n_j does not overlap with the nets already assigned to $AR(z_i)$ and if $AR(z_i)$ has some net n_k which is in the same component as n_j.

8. For the remaining rows in $AR(z_i)$, the row with the smallest number of adjacent rows is matched first. If there is a tie, choose that row which belongs to the smallest component. For the remaining nets in $N(z_i)$, the following three sub–criteria are used to determine the net to be matched:
   - Choose that net which has the smallest number of assigned nets in its component.
   - Choose a net belonging to the smallest component.
   - Choose a net which has a minimum number of adjacent nets.

[24] In the earlier criterion, we looked for a set of nets that was larger than the number of rows; here, we are looking for a set of rows that is larger than the set of nets that we have identified.

The algorithm runs in $O(n)$ time where n is the number of nets. It has succeeded in obtaining realizable layouts for a large number of cases that could not be solved using the left–edge–first algorithm.

We know now that the net assignment problem is quite difficult. In fact, it has been proved to be NP–Complete by Rim and Nakajima [37]. In the following subsection, we present the outline of the NP–Completeness proof as well as the algorithm developed in [37].

11.2 Net Assignment as an NP–Complete problem

Each vertical diffusion run on a column induces a partial order on the nets connected to the column, i.e., if a diffusion runs exists between nets $n_{i_1}, n_{i_2}, \ldots, n_{i_k}$ on column l, the other nets connected to column l must lie *outside* $n_{i_1}, n_{i_2}, \ldots, n_{i_k}$ for the diffusion run to be feasible. This is because all these other nets also have contacts on column l and if they lie in between, a conflict is bound to arise. Fig. 27(a) gives an example. So for each column, we can define a set of *constraints* that must be satisfied for all the diffusion runs to be realizable. Each constraint is of the form $(n_{i_1}, n_{i_2}, \ldots, n_{i_k})n_{i_r}$ which means that n_{i_r} (the *outsider*) has to be placed outside $n_{i_1}, n_{i_2}, \ldots, n_{i_k}$ (the *insiders*). Thus, the *Outsideness problem* described next can be converted to the net assignment problem in polynomial time[25]. Hence, the net assignment problem is considered to be NP–Complete.

The Outsideness problem is:
Given a set A and a collection of constraints R, find an ordering of the elements of A such that all the constraints in R are satisfied. To prove this problem NP–Complete, the authors have demonstrated that a known NP–Complete problem called as the NOT–ALL–EQUAL–3SAT problem [41] can be transformed in polynomial time to the Outsideness problem. The details of the proof are not given here. We will next describe the heuristic algorithm that has been developed by the authors based on the proof.

Given the netlist and the set of diffusion runs, we can create the sets A and R, where A is the set of nets and R is a set of constraints that reflects the vertical diffusion runs. In Figure 27(b), we show how each vertical diffusion run can be expressed as a set of constraints. Based on the sets A and R, we can construct a graph $G(V, E)$ where V is equal to the set of nets N and there is an edge (n_i, n_j) in G if n_i and n_j are an insider and the outsider respectively, of a constraint[26]. Our aim is to create an ordering of the vertices of G. Hence, as we progress through the algorithm, we will assign a direction to each edge. The directed edge (v_i, v_j) tells us that the vertex v_i precedes v_j in the ordering. The edge is *outgoing* from v_i and *incoming* into v_j.

The algorithm first calculates the maximum density D_{max}. This gives us the minimum number

[25] There are some details that have to be looked into before we can make this conversion. The full proof is given in [37].

[26] In the earlier algorithm, there was an edge in the graph if the two nets had a diffusion run in common. Here, it is the other way round.

of rows in which all the nets can be assigned. It then proceeds to find *supervertices* that are collections of nets (vertices) that can be placed together on a single row. The algorithm then allocates supervertices to the D_{max} rows, proceeding row by row. We give a broad outline of the steps below:

1. For each row, a 'seed' supervertex is found.
2. Then, all those supervertices that can be placed to the left are *merged* with the 'seed' supervertex. If there is more than one such supervertex, the supervertices are chosen in an order that minimizes the enlargements.
3. Similarly, all supervertices that can be placed on the right are also merged with the 'seed' supervertex. As before, the order of merging aims to keep the number of enlargements minimal.
4. The three steps above are repeated D_{max} times.

The two operations orientation and merging are discussed below:

- Orientation: When we assign a direction to a edge as discussed above, other edges are affected as well. Consider Fig. 27(c) which shows how the direction of (v_2, v_3) is automatically determined once we orient (v_1, v_2). This is called the *domino effect*. In general, the domino effect may create a directed cycle, or may cause a constraint violation (Fig. 27(d)). In such cases, we have to permit enlargements.

- Merging: Merging takes two supervertices and creates one new supervertex that contains all the nets contained in the other two. If there exists a directed path[27] between any two nets v_i and v_j in the two supervertices, we have to destroy the path by removing some edge (i.e., by removing some constraint). That means, we have to permit some enlargements. The merging operation chooses to delete that edge which minimizes the number of enlargements.

Of all net assignment algorithms developed so far, this one has given the best results. Net assignment is a problem that has not received as much attention as the Gate Matrix problem.

Our next section is devoted to algorithms that solve the gate assignment problem of *one-dimensional arrays*. This problem, as we will see is closely related to the gate matrix layout optimization problem. However, the practical constraints that we take into account while solving the gate matrix problem, are not relevant in the one-dimensional array problem.

[27] If there exists a directed path, it means that an order has been specified between v_i and v_j. But, merging puts them in the same position. This violates the ordering imposed earlier and leads to a directed cycle.

12 One–Dimensional Arrays

The one-dimensional logic gate array was introduced by Weinberger [1]. A complex logic function can be realized with gates interconnected as a one–dimensional array. In the layout, in-gate wiring is done along the vertical columns and connections between gates are made horizontlly. Input/Output lines are also drawn as vertical columns. At the intersection of a vertical column and a horizontal wire, there is either a terminal of a transistor or a straight connector. The width of each column(gate) and the separation distance between two neighboring columns are predetermined depending on physical constraints. The layout schematic of a NAND gate is shown in Fig. 28(a). In Fig. 28(b), a layout of a one–dimensional array with six gates is shown. Additionally, we require two *distinguished gates* g_{left} and g_{right} for input/output purposes. In Fig. 28(c), another layout of the same array is obtained by permuting the gates. We see that the number of rows is lesser in the second case. In general, we want to place the gates so as to minimize the number of rows(also called *tracks*). The one–dimensional array approach is suitable for MOS and I^2L technologies.

In Section 3, we have already seen one algorithm, the minimal augmentation algorithm, that solves the one–dimensional array problem [3]. We will outline four other algorithms that have been developed.

1. Asano's net assignment algorithm 335]
2. The two–terminal net heuristic developed by Fujii et al. [38]
3. The hierarchical contraction heuristic of Yamada et al. [39]
4. The double macrovertex heuristic given by Hong et al. [40]

Of these, a modified version of Asano's algorithm was used in Algorithm Net-Sequencing. We will outline the algorithms but will not go into the details.

12.1 Asano's algorithm

Asano developed a *branch–and–bound* algorithm that obtains optimal solutions to the one–dimensional array problem. The branch–and–bound technique first finds one solution using the depth–first–search method. Then every incomplete path in the search graph is pursued either till it finds a solution of lesser cost or till it exceeds the cost of the best solution so far. An evaluation function is used to determine whether the current node in the search graph can possibly lead to a better solution than the best one so far. The evaluation function used in this algorithm is similar to the function $f_E(m, n_i)$ used in algorithm net–sequencing.

This algorithm takes $O(m^2 * v)$ time, where m is the number of nets and v is the number of nodes generated during the search. v may vary from $O(m)$ to $O(m!)$. Hence, this algorithm is not feasible even for moderate values of m. Asano has suggested an approximate version of the algorithm that

works in a greedy manner. At every stage of the search tree, the approximate algorithm chooses the seemingly best path. Once a solution is obtained, the algorithm stops (there is no backtrack).

12.2 The two–terminal net heuristic algorithm

Fujii et al. developed a heuristic algorithm that first decomposes a problem consisting of multi–terminal nets (nets with more than two gates) to a problem consisting only of two–terminal nets. That is, each multiterminal net with q contacts is broken down into $q-1$ consecutive two–terminal nets. To do this, an arbitrary gate sequence is generated. The multiterminal net is now decomposed into a sequence of two–terminal nets which appear consecutively. This reduced problem, consisting only of two–terminal nets is known as Problem R. A solution for Problem R known as Algorithm R is based on a bidirectional search method. Two partial assignments of vertices are constructed independently: one is started from the left boundary , while the other is started from the right boundary. The search is conducted on a *weighted graph* $G = (V, E)$ which is created using the input to Algorithm R. A complete one–dimensional assignment is obtained when the search from the left boundary visits the same vertex as the search from the right boundary. Algorithm R is repeated for a number of different starting sequences. The best solution obtained is retained as the final gate sequence. It is claimed that such an approach also minimizes the total wire length. Algorithm R runs in $O(|V| \bullet |E| \bullet \log |E|)$.

12.3 The hierarchical contraction heuristic algorithm

There are two stages in the hierarchical contraction algorithm. In the first, gates are *contracted*[28] a prefixed number of times. The 'hierarchy' defines that at each level, nets with minimum number of gates be contracted. Contraction is done using the following two policies:

- Policy 1: The original problem is divided into $i_m + 1$ levels by iterating contraction of nets in increasing order of the number of terminals, where i_m is a control parameter to be selected by the designer, and is an integer between 1 and $p-1$, where p is the total number of terminals on nets. This ensures that atleast one net will be contracted at each level.

- Policy 2: The contraction of nets connecting to the left boundary gate or the right boundary gate is forbidden.

Fig. 30(a) shows a layout, whose contracted layout with $i_m = 2$ is shown in Fig. 30(b). Once we have extracted the subset of gates at each level, we have to arrange them in a row to get a gate sequence. This is done in accordance with the following policy:

- Policy 3: The gates at each level are selected one at a time, and arranged in order such that the total wire length and the number of tracks are locally minimized.

[28] Fig. 29(a) shows a small layout. Fig. 29(b) shows the layout after the contraction of net n_1.

The algorithm runs in $O(i_m \bullet n \bullet p)$ time where n is the total number of gates. It is claimed that this algorithm performs much better than the algorithm due to Fujii et al.

12.4 The double macrovertex heuristic algorithm

Our last algorithm [40], like the algorithm due to Fujii et al., also proceeds to builds a gate sequence from both the left and the right directions. However, this algorithm starts with a weighted graph $G(V, E)$, where V is the set of gates (columns). The edge set E of the graph is generated as follows:

- Construct a *complete graph*[29] for each net. The union of all the complete graphs thus generated is the final weighted graph. The edge set E is the union of the edge sets of all the complete graphs.

Fig. 31(a) shows the weighted graph of the netlist in Fig. 31(b). Since this weighted graph does not depend on any starting sequence etc., it is unique for the given netlist. So, unlike Fujii's algorithm, this algorithm does not have to be repeated for different starting sequences. The subset of vertices already chosen to be placed from the left boundary is called the *left macrovertex* and that from the right boundary is called the *right macrovertex*.

At each step, the direction of ordering (left or right) is first determined using the *minimum cut* as a criterion. In Fig. 30(a), the cut of the left macrovertex L_S is 9 and that of the right macrovertex R_S is 6. So, at this step, a gate is chosen to be placed next to the right macrovertex.

To choose the gate, two criteria are used.

1. The maximum connection criterion: We consider all the gates which are connected to the macrovertex chosen for expansion. That gate which has the edge of maximum weight will be chosen. In Fig. 31(a), c_5 has an edge of weight 3 which is the maximum among the weights. Hence, c_5 is chosen.
2. The Overriding property: In Li's algorithm [7] , we discussed the concept of dominant gates. It was stated that for local optimization, a gate (column) should be placed next to a gate that dominates it. This property is called the *overriding property* here because we choose a gate that either dominates or is dominated by the macrovertex, if such a gate exists. Hence, we are overriding the choice made by the first criterion. In Fig. 31(a), c_5 dominates and hence replaces R_S in Fig. 31(c). The method of obtaining the graph of Fig. 31(c) is explained in the next paragraph.

Once we have chosen the gate, we *merge* it with the macrovertex. For all those gates that were connected to the chosen gate but not the macrovertex, edges are introduced with weights as before. For gates that were connected to both the chosen gate and the macrovertex, a single edge with weight equal to *newweight* is created, where *newweight* is

[29] A complete graph is one in which every vertex is connected to every other vertex by an edge.

- the maximum of the two weights if the gate was chosen because of the overriding property
- the sum of the two weights if the gate was chosen by the maximum connection criterion.

Fig. 31(c) shows the graph after c_5 has been merged with R_S. The above steps are repeated till there is only one gate left (other than the distinguished ones). The algorithm runs in $O(|G|^2)$ time where $|G|$ is the number of gates. However, the actual execution time depends on the number of overrides. In [40], a *simulated annealing* approach to the one-dimensional array problem has also been detailed. It is claimed that the proposed heuristic algorithm gives results of nearly the same quality as the simulated annealing algorithm using much less computation time.

13 Conclusions

In this section, we will discuss the performance of the various examples on some common, benchmark problems. There are some issues that still need to be resolved before the Gate Matrix style becomes as popular as other design styles. It is necessary to try out the Gate Matrix style on large problems that are solvable using standard place-and-route techniques. Future algorithms should provide bounds on the extra area that the Gate Matrix is likely to use as compared to existing place-and-route techniques.

Other open theoretical problems include the development of (or atleast, a result about the existence of) a relative approximation algorithm and the use of other graph-based techniques.

14 Bibliography

1. A. Weinberger, "Large-scale integration of MOS complex logic: A layout method," *IEEE JSSC*, Dec. 1967.
2. A.D. Lopez and H.F.S. Law, "A Dense Gate Matrix Layout Method for MOS VLSI," *IEEE Trans. Electron Devices*, Aug. 1980.
3. T. Ohtsuki, H. Mori, E.S. Kuh, T. Kashiwabara and T. Fujisawa, " One-Dimensional Logic Gate Assignment and Interval Graphs," *IEEE Trans. Circ. Syst.*, Sept. 1979.
4. O. Wing, "Automated Gate Matrix Layout," *Proc. ISCAS 1982.*
5. O. Wing, "Interval-Graph-Based Circuit Layout," *Proc. ICCAD 1983.*
6. O. Wing, S. Huang and R. Wang, "Gate matrix Layout," *IEEE Trans. CAD*, Vol. CAD-4, No. 3, 1985.
7. J.T. Li, "Algorithms for Gate Matrix Layout," *Proc. ISCAS 1983.*
8. D.M. Xu, Y. K. Chen, E. S. Kuh and Z. J. Li, "A new algorithm for Gate Matrix Layout," *Proc. ISCAS 1987.*

9. G.K. Liu and M.L. Liu, "Gate Matrix Layout Algorithms with controllable upper bound," *Proc. ISCAS 1987.*

10. T. Ohtsuki, H. Mori, T. Kashiwabara and T. Fujisawa, "On minimal augmentation of a graph to obtain an interval graph," *Proc. ISCAS 1979.*

11. T. Kashiwabara and T. Fujisawa. "An NP-Complete Problem on Interval Graphs," *Proc. ISCAS 1979.*

12. D.J. Rose, R.E. Tarjan and G.S. Lucker, "Algorithmic aspects of vertex elimination in graphs," *SIAM Jl. Comp.*, No. 5, 1976.

13. N. Deo, M. S. Krishnamoorthy and M. A. Langston, "Exact and Approximate Solutions for the Gate Matrix Layout Problem," *IEEE Trans. CAD*, Jan. 1987.

14. R. Bellman, *Principles of Dynamic Programming*, Princeton University Press, Princeton, NJ, 1962.

15. D. K. Hwang, W. K. Fuchs, S. M. Kang, "An Efficient Approach to Gate Matrix Layout," *Proc. ICCAD 1986.*

16. D. K. Hwang, W. K. Fuchs, S. M. Kang, "An Efficient Approach to Gate Matrix Layout," *IEEE Trans. CAD*, Sept. 1987.

17. B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Syst. Tech. Jl.*, Feb 1970.

18. C. M. Fidducia and R. M. Matheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proc. 19th DAC*, 1982.

19. S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, 1983.

20. C. Y. R. Chen and C. Y. Hou, "A new algorithm for CMOS Gate matrix Layout," *Proc. ICCAD 1988.*

21. H. W. Leong, "A New Algorithm for Gate Matrix Layout," *Proc. ICCAD 1986.*

22. D. F. Wong, H. W. Leong and C. L. Liu, Chapter 7, *Simulated Annealing for VLSI Design*, Kluwer Academic Press 1988.

23. J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

24. D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Publishing Company, Reading, MA, 1988.

25. K. Shahookar and P. Mazumder, "GASP - A Genetic Algorithm for Standard Cell Placement," to be published in *Proc. European Design Automation Conference 1990.*

26. H. M. Chan and P. Mazumder," A Genetic Algorithm for Macrocell Placement," submitted to *IEEE Trans. CAD.*

27. K. Shahookar and P. Mazumder, " An Application of the Genetic Algorithm to Standard Cell Placement with Meta–Genetic Parameter Optimization," to be published in *IEEE Trans. CAD*, May 1990.

28. J. P. Cohoon and W. D. Paris, "Genetic Placement," *Proc. ICCAD*, 1986.

29. J. P. Cohoon, S. U. Hegde and W. D. Paris, "A Distributed Genetic Algorithm for Floorplan Design," *Proc. ICCAD 1988.*

30. Anil S. Chakravarthy and P. Mazumder, "Gate Matrix Layout using the Genetic Algorithm," in preparation.

31. K. Nakatani, T. Fujii, T. Kikuno and N. Yoshida, "A Heuristic Algorithm for Gate Matrix Layout," *Proc. ICCAD 1986.*

32. S. Huang and O. Wing, "Improved Gate matrix layout", *Proc. ICCAD 1986.*

33. S. Huang and O. Wing, "Improved Gate matrix layout", *IEEE Trans. CAD*, Aug. 1989.

34. T. Asano, "An Optimum Gate Placement Algorithm for MOS One–dimensional Arrays", *Jl. Dig. Syst.* Vol. 6, No. 1, 1982.

35. S. Huang and O. Wing "Gate Matrix Partitioning," *IEEE Trans. CAD*, July 1989.

36. Shu, Wu and Kang. "Improved net merging method," *IEEE Trans. CAD*, Sept. 1988..

37. C. S. Rim and K. Nakajima, "Net Assignment in Gate Matrix Layout," *Proc. ISCAS 1988.*

38. T. Fujii, H. Horikawa, T. Kikuno and N. Yoshida, "A Heuristic Algorithm for Gate Assignment in One–Dimensional Array Approach," *IEEE Trans. CAD*, March 1987.

39. S. Yamada, H. Okude and T. Kasai,"A Hierarchical Algorithm for One–dimensional Gate Assignment Based on Contraction of Nets," *IEEE Trans. CAD*, June 1989.

40. Y-S. Hong, K-H. Park and M. Kim,"A Heuristic Algorithm for Ordering the Columns in One–Dimensional Logic Arrays," *IEEE Trans. CAD*, May 1989.

41. M. S. Garey and D. S. Johnson, *Computers and Intractability: A guide to the theory of NP–Completeness*, W. H. Freeman, San Francisco, 1979.

27. K. Shahookar and P. Mazumder, " An Application of the Genetic Algorithm to Standard Cell Placement with Meta-Genetic Parameter Optimization," to be published in *IEEE Trans. CAD*, May 1990.

28. J. P. Cohoon and W. D. Paris, "Genetic Placement," *Proc. ICCAD*, 1986.

29. J. P. Cohoon, S. U. Hegde and W. D. Paris, "A Distributed Genetic Algorithm for Floorplan Design," *Proc. ICCAD 1988.*

30. Anil S. Chakravarthy and P. Mazumder, "Gate Matrix Layout using the Genetic Algorithm," in preparation.

31. K. Nakatani, T. Fujii, T. Kikuno and N. Yoshida, "A Heuristic Algorithm for Gate Matrix Layout," *Proc. ICCAD 1986.*

32. S. Huang and O. Wing, "Improved Gate matrix layout", *Proc. ICCAD 1986.*

33. S. Huang and O. Wing, "Improved Gate matrix layout", *IEEE Trans. CAD*, Aug. 1989.

34. T. Asano, "An Optimum Gate Placement Algorithm for MOS One-dimensional Arrays", *Jl. Dig. Syst.* Vol. 6, No. 1, 1982.

35. S. Huang and O. Wing "Gate Matrix Partitioning," *IEEE Trans. CAD*, July 1989.

36. Shu, Wu and Kang. "Improved net merging method," *IEEE Trans. CAD*, Sept. 1988..

37. C. S. Rim and K. Nakajima, "Net Assignment in Gate Matrix Layout," *Proc. ISCAS 1988.*

38. T. Fujii, H. Horikawa, T. Kikuno and N. Yoshida, "A Heuristic Algorithm for Gate Assignment in One-Dimensional Array Approach," *IEEE Trans. CAD*, March 1987.

39. S. Yamada, H. Okude and T. Kasai,"A Hierarchical Algorithm for One-dimensional Gate Assignment Based on Contraction of Nets," *IEEE Trans. CAD*, June 1989.

40. Y-S. Hong, K-H. Park and M. Kim,"A Heuristic Algorithm for Ordering the Columns in One-Dimensional Logic Arrays," *IEEE Trans. CAD*, May 1989.

41. M. S. Garey and D. S. Johnson, *Computers and Intractability: A guide to the theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

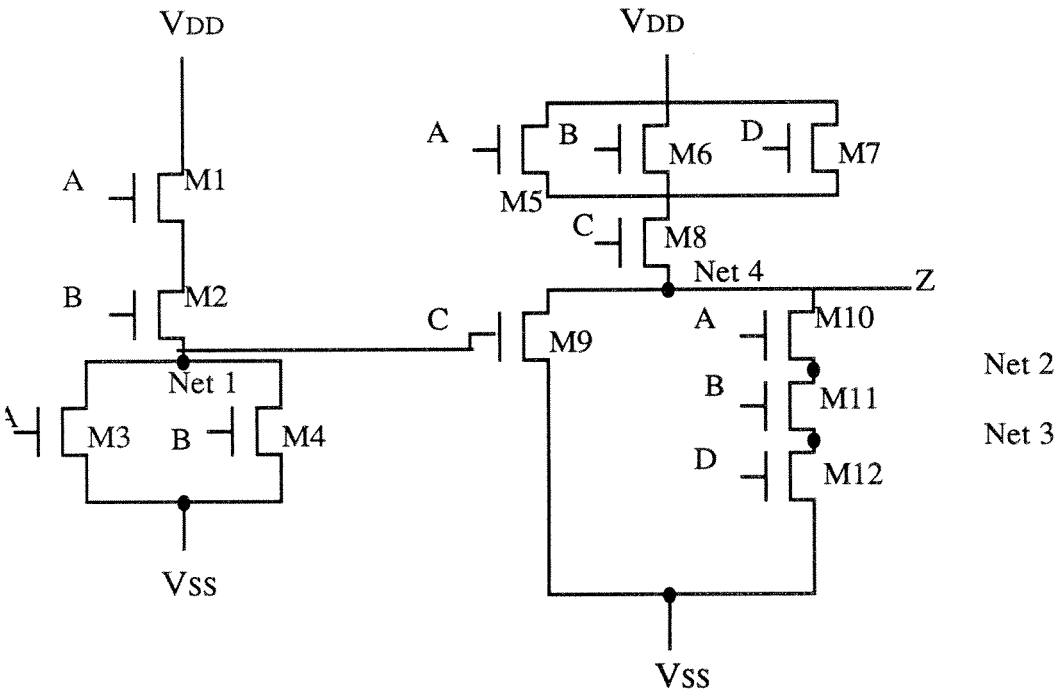


Figure 1(a): A CMOS Circuit.

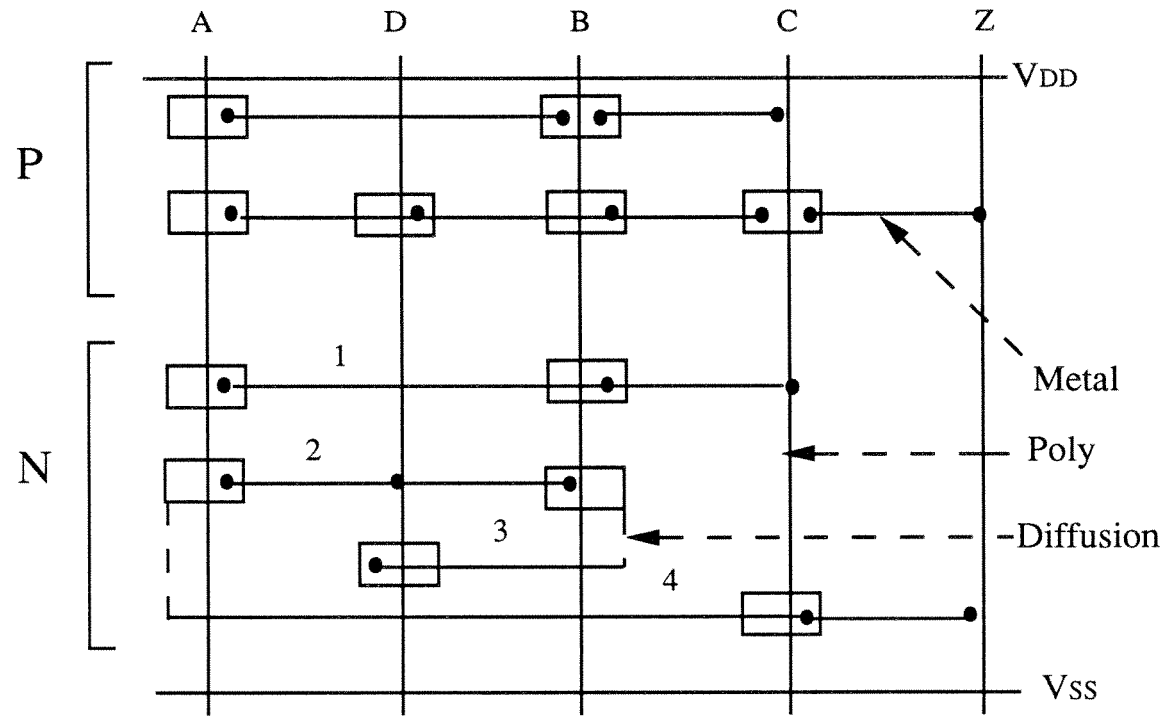


Figure 1(b): Layout of the above circuit.

n1 = { g2, g4, g3 }; n2 = { g1, g3, g4, g5}; n3 = { g2, g4, g8 }; n4 = { g5, g7 }; n5 = { g5, g8, g6 };

Figure 2: A sample netlist.

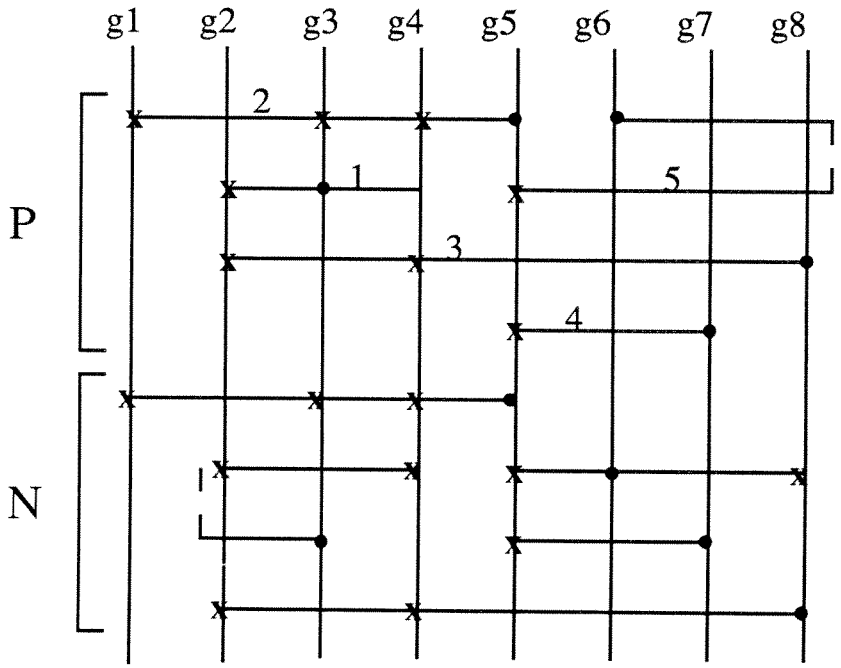


Figure 2(a): Layout of the above netlist.

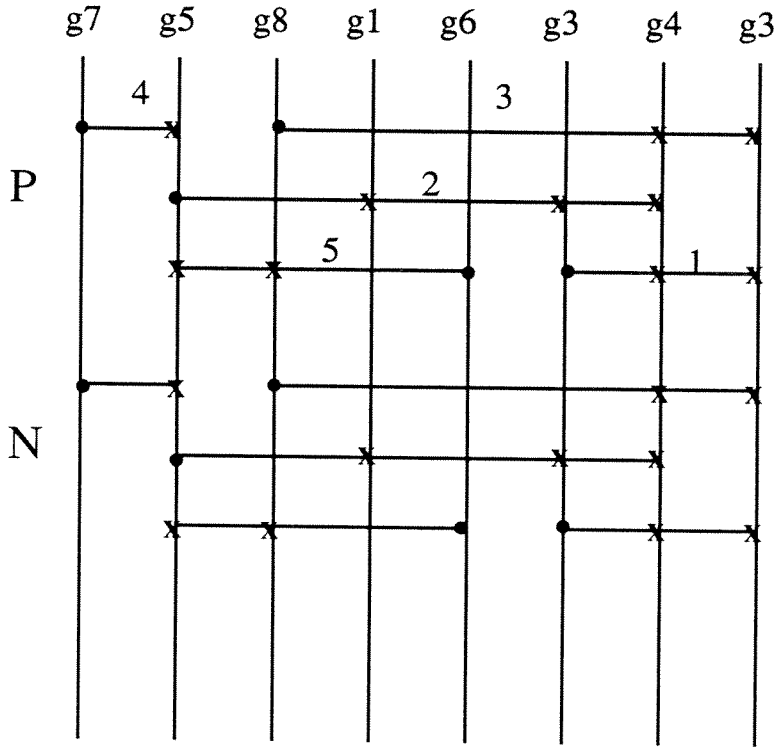


Figure 2(b): Layout of the same netlist requiring fewer rows.

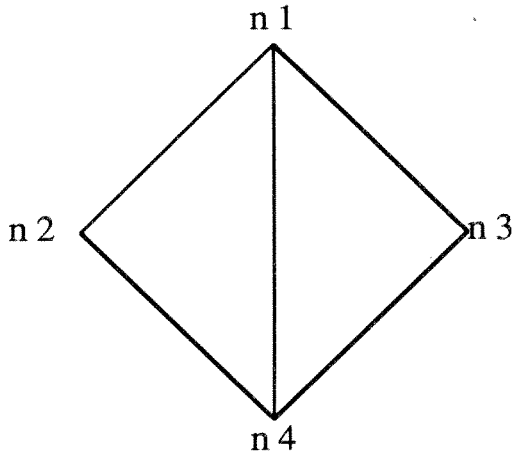


Figure 3(a): An Interval Graph

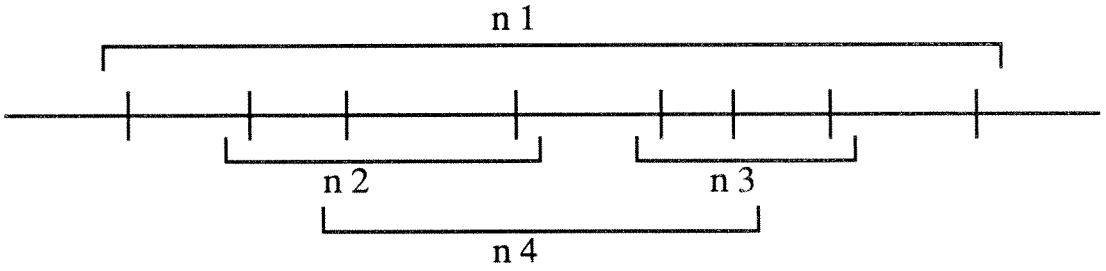


Figure 3(b): Its Realization.

n 1 = { g 1, g 2}
n 2 = { g 2, g 4}
n 3 = { g 3, g 1}
n 4 = { g 3, g 4}

Figure 3(c)

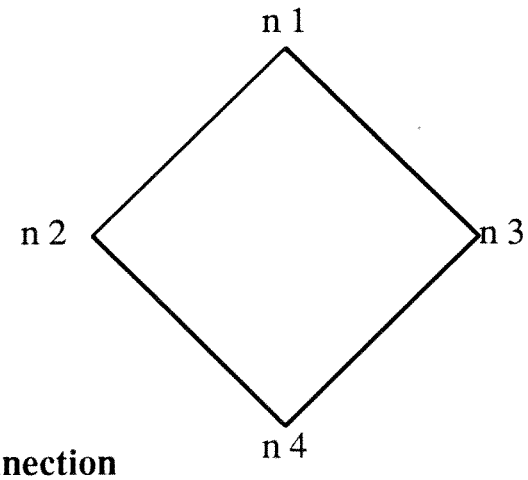


Figure 3(d): Connection Graph of netlist given in Figure 3(c).

Gates	N(g)
g 1	1, 4
g 2	3, 4
g 3	1, 2, 3
g 4	2, 5
g 5	1, 9
g 6	9
g 7	6, 8, 9
g 8	5, 7, 8
g 9	6, 7

TABLE I

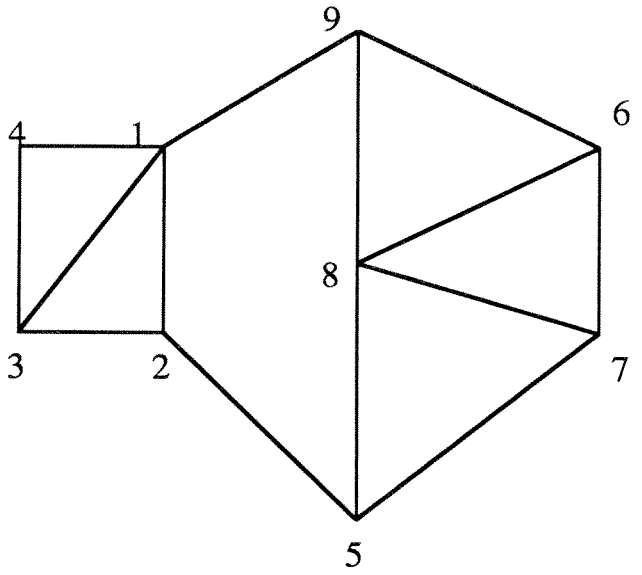


Figure 4(a)

	c1	c2	c3	c4	c5	c6	c7
1	1	1	1	0	0	0	0
2	0	1	0	1	0	0	0
3	1	1	0	0	0	0	0
4	1	0	0	0	0	0	0
5	0	0	0	1	0	0	1
6	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1
8	0	0	0	0	1	1	1
9	0	0	1	0	1	0	0

Figure 4(b)

	c1	c2	c3	c4	c7	c6	c5
1	1	1	1	0	0	0	0
2	0	1	x	1	0	0	0
3	1	1	0	0	0	0	0
4	1	0	0	0	0	0	0
5	0	0	0	1	1	0	0
6	0	0	0	0	0	1	1
7	0	0	0	0	1	1	0
8	0	0	0	0	1	1	1
9	0	0	1	x	x	x	1

Figure 4(c)

x denotes a fill-in.

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

= A

Figure 5(a)

	1	2	3	4	5	6	7	8	9	10	11
2	1	0	0	0	0	0	0	1	0	0	0
4	0	0	0	0	0	0	0	0	0	1	1
5	0	0	0	0	0	0	0	0	1	1	0
7	0	0	0	0	0	0	0	0	1	1	0
8	0	0	0	0	0	0	0	1	1	0	0
9	1	1	0	0	1	0	0	0	0	0	0
10	1	1	0	0	0	0	0	1	0	0	0
11	0	1	1	0	0	0	0	0	0	0	0
12	0	1	0	1	0	0	0	0	0	0	0
15	0	0	0	0	1	1	0	0	0	0	0
16	0	0	0	0	1	0	1	0	0	0	0

= A'

Figure 5(b)

	1	2	3	4	5	6	7	8	9	10	11
2	1	x	x	x	x	x	x	1	0	0	0
4	0	0	0	0	0	0	0	0	1	1	
5	0	0	0	0	0	0	0	1	1	0	
7	0	0	0	0	0	0	0	1	1	0	
8	0	0	0	0	0	0	0	1	1	0	0
9	1	1	x	x	1	0	0	0	0	0	
10	1	1	x	x	x	x	x	1	0	0	0
11	0	1	1	0	0	0	0	0	0	0	
12	0	1	x	1	0	0	0	0	0	0	
15	0	0	0	0	1	1	0	0	0	0	
16	0	0	0	0	1	x	1	0	0	0	

= A'ub

Figure 5(c)

	4	3	2	6	5	7	1	8	9	10	11
2	0	0	0	0	0	0	1	1	0	0	0
8	0	0	0	0	0	0	0	1	1	0	0
5	0	0	0	0	0	0	0	0	1	1	0
4	0	0	0	0	0	0	0	0	0	1	1
7	0	0	0	0	0	0	0	0	1	1	0
9	0	0	1	x	1	x	1	0	0	0	
11	0	1	1	0	0	0	0	0	0	0	
10	0	0	1	x	x	x	1	1	0	0	0
12	1	x	1	0	0	0	0	0	0	0	
15	0	0	0	1	1	0	0	0	0	0	
16	0	0	0	0	1	1	0	0	0	0	

= B

Figure 5(d)

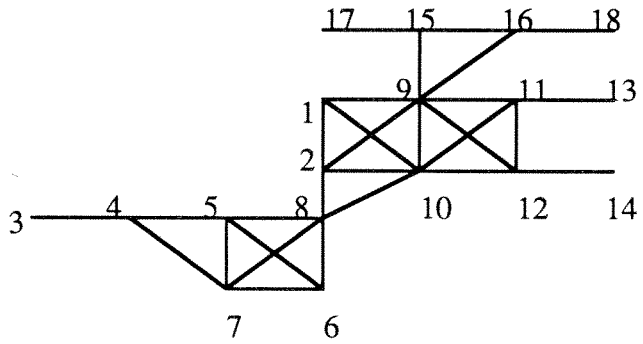


Figure 5(e)

Gates	N(g)	
g 1	1, 2, 11	
g 2	1, 2, 3, 11	D
g 3	3, 4	D
g 4	5, 6, 7, 10	D
g 5	4, 5, 11	D
g 6	3, 7, 10	D
g 7	5, 7, 8, 9	D
g 8	6, 10	
g 9	7, 8	
g 10	8, 9	

Table II

	g2	g3	g4	g5	g6	g7
1	1	0	0	0	0	0
2	1	0	0	0	0	0
3	1	1	0	0	1	0
4	0	1	0	1	0	0
5	0	0	1	1	0	1
6	0	0	1	0	0	0
7	0	0	1	0	1	1
8	0	0	0	0	0	1
9	0	0	0	0	0	1
10	0	0	1	0	1	0
11	1	0	0	1	0	0

= A

Figure 6(a)

	g2	g3	g5	g6	g4	g7
1	1	0	0	0	0	0
2	1	0	0	0	0	0
3	1	1	x	1	0	0
4	0	1	1	0	0	0
5	0	0	1	x	1	1
6	0	0	0	0	1	0
7	0	0	0	1	1	1
8	0	0	0	0	0	1
9	0	0	0	0	0	1
10	0	0	1	1	0	0
11	1	x	1	0	0	0

g3 is selected.

Figure 6(b)

	x	f	s	y	z
g3	2	1	3	0	0
g4	4	2	6	1	1
g5	3	1	4	0	1
g6	3	1	4	0	0
g7	4	2	6	2	2

Table III

Li's Algorithm.

Gates	N(g)
1	5
2	6
3	2, 6
4	4
5	1, 6, 3
6	2, 3
7	5
8	1, 4, 5

Table IV

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\
\left[\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0
\end{array}\right]
\end{array}
= A
$$

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} 1 & 7 & 8 & 4 & 5 & 6 & 3 & 2 \end{array} \\
\left[\begin{array}{cccccccc}
0 & 0 & 1 & x & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & x & 1 & 1
\end{array}\right]
\end{array}
= B
$$

net	1	2	3	4	5	6
dv(i)	4	2	3	2	2	3
din(i)	10	6	9	6	6	9

column	1	2	3	4	5	6	7	8
dc(i)	2	3	5	2	10	5	2	8
dcin(i)	6	9	15	6	28	15	6	22

Figure 7: Xu et al. 's algorithm.

Gates	N(g)
1	5
2	6
3	2, 6
4	4
5	1, 6, 3
6	2, 3
7	5
8	1, 4, 5

Table V

$$
\begin{array}{c}
\begin{array}{cc} & 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10 \end{array} \\
\left[\begin{array}{cc}
1 & 0\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0 \\
2 & 1\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0 \\
3 & 1\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1 \\
4 & 1\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 1 \\
5 & 1\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 0 \\
6 & 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0 \\
7 & 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1
\end{array}\right] = A
\end{array}
$$

LB = 4; Set e = 2, QBD = 1.

$$
\begin{array}{c}
\begin{array}{c} 1\ 2\ 3\ 4\ 5\ 7\ 8\ 10 \end{array} \\
\left[\begin{array}{c}
0\ 1\ 0\ 0\ 0\ 1\ 1\ 0 \\
1\ 0\ 0\ 1\ 0\ 1\ 0\ 0 \\
1\ 1\ 0\ 0\ 0\ 0\ 0\ 1 \\
1\ 0\ 1\ 0\ 0\ 0\ 1\ 1 \\
1\ 0\ 1\ 0\ 1\ 0\ 0\ 0 \\
0\ 0\ 0\ 1\ 1\ 0\ 0\ 0 \\
0\ 0\ 1\ 0\ 1\ 0\ 1\ 1
\end{array}\right] = A^{1}
\end{array}
$$

$$
\begin{array}{c}
\begin{array}{c} 4\ 5\ 1\ 3\ 7\ 8\ 2\ 10 \end{array} \\
\left[\begin{array}{c}
0\ 0\ 0\ 0\ 1\ 1\ 1\ 0 \\
1\ x\ 1\ x\ 0\ 0\ 0\ 0 \\
0\ 0\ 1\ x\ x\ x\ 1\ 1 \\
0\ 0\ 1\ 1\ 1\ 1\ x\ 1 \\
0\ 1\ 1\ 1\ 0\ 0\ 0\ 0 \\
1\ 1\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 1\ x\ 1\ 1\ 1\ x\ 1
\end{array}\right] = B^{1}
\end{array}
$$

Figure 8: Liu's algorithm.

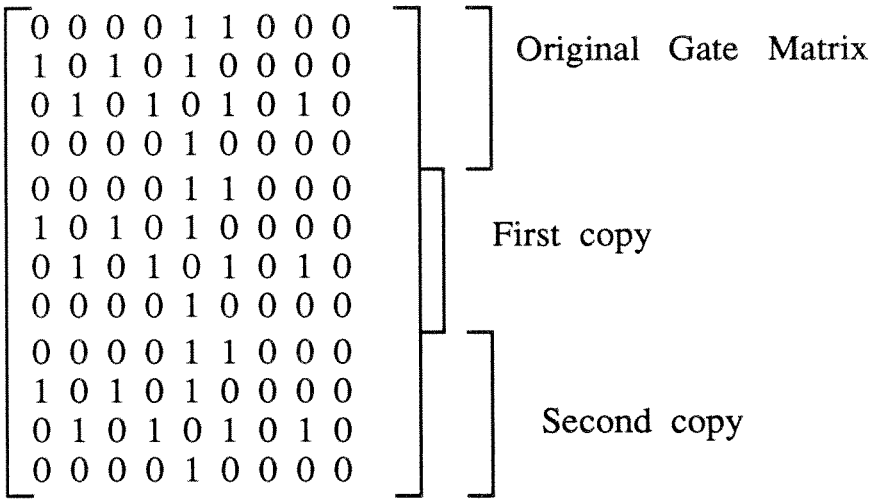


Figure 9(a)

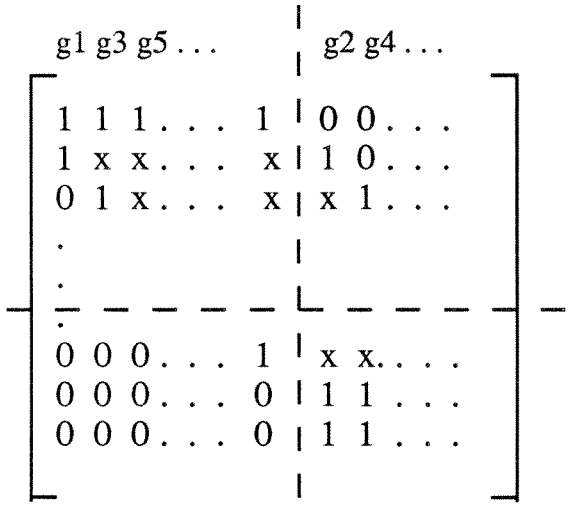


Figure 9(b)

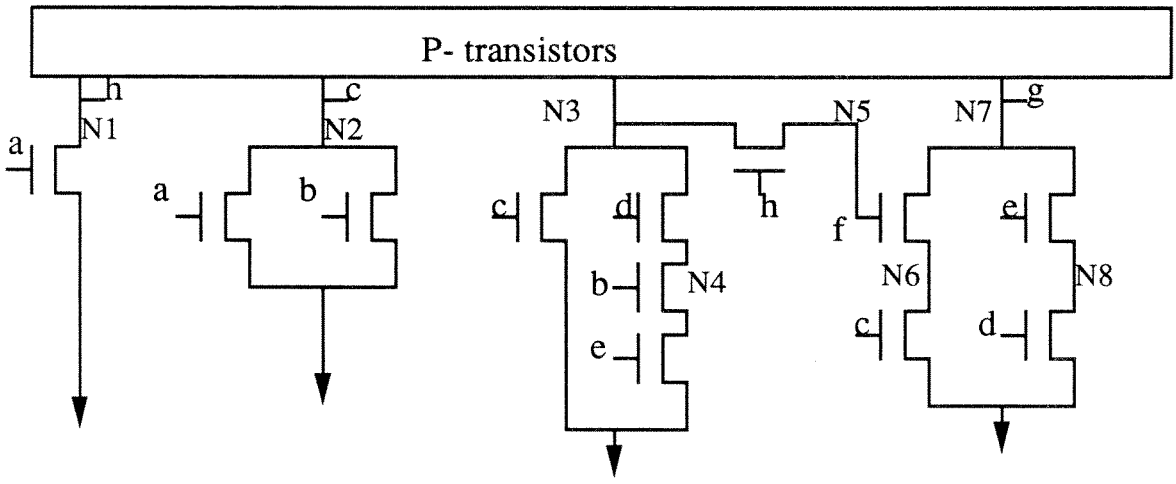
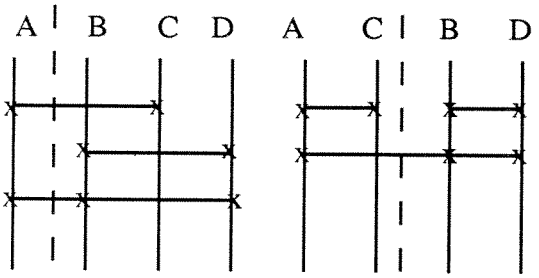
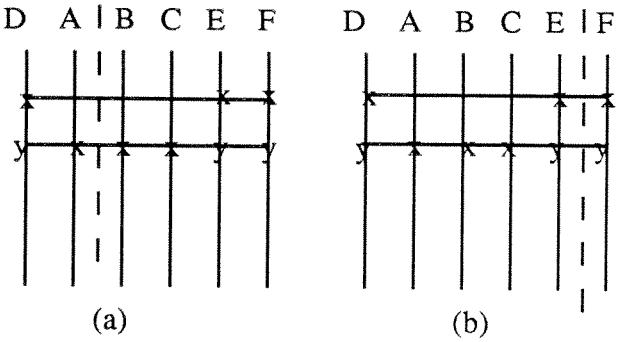


Figure 10(a)



G(A) = 2, G(B) = -1, G(C) = 1, G(D) = -1.
If we move C, cutset becomes 1.

Figure 10(b)



(a) G(A) = +1 by rule 3.
(b) G(A) = G(B) = G(C) = -1 by rule 4.

x denotes a contact; y denotes a series transistor.

Net N1 = [{ g2, g4 }, g3]
N2 = [{ g1, g3, g4 }, g5]
N3 = [{ g2, g4 }, g8]
N4 = [{ g5, g8 }, g6]

g3, g5, g6 and g8 are the output gates.

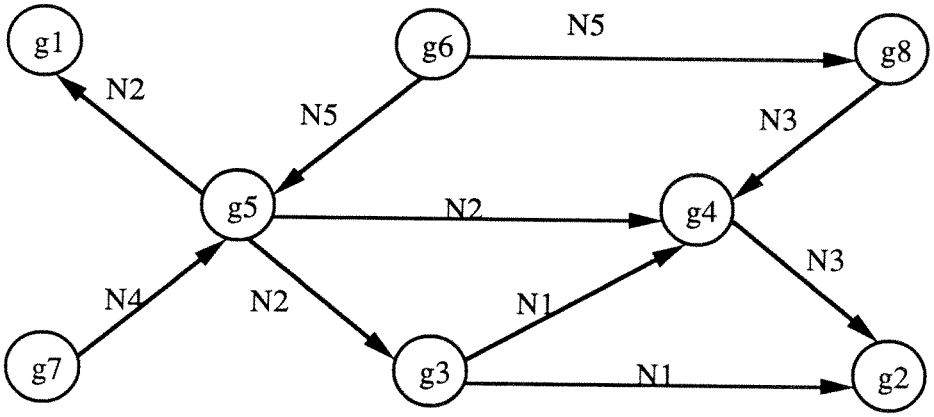


Figure 11

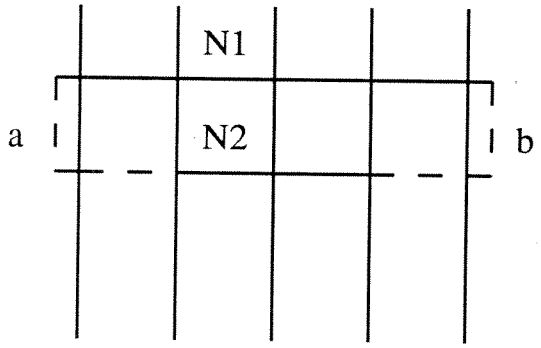


Diagram for the Assignment function

Vp(N1) = Left is given by a.

Vp(N1) = Right is given by b.

Figure 12

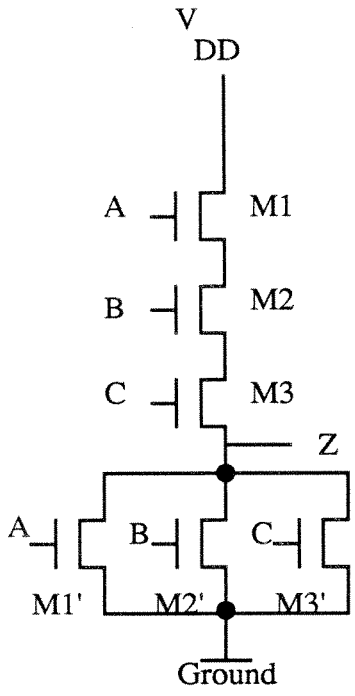


Figure 12(a)

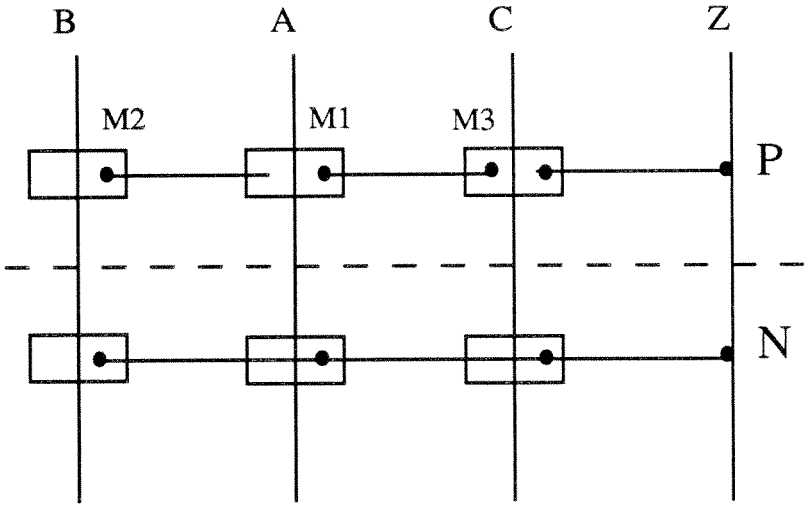


Figure 12(b)

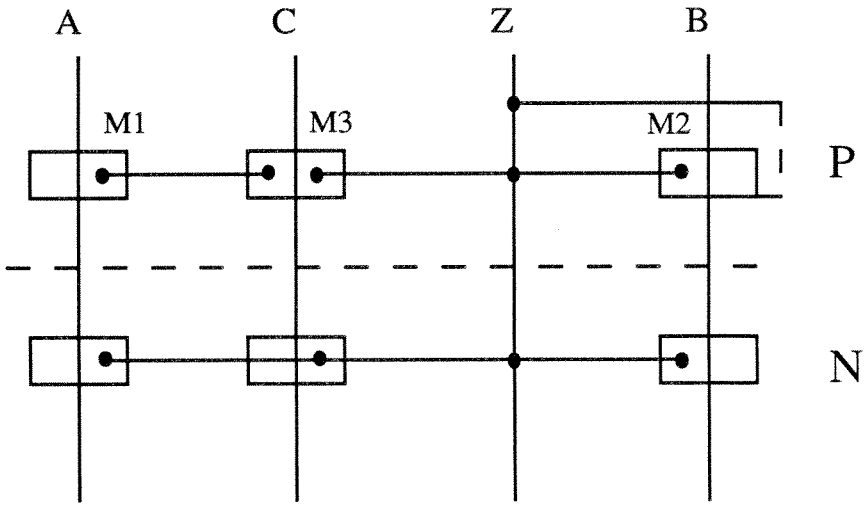


Figure 12 (c)

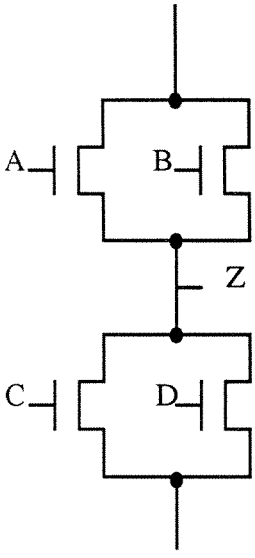


Figure 13(a)

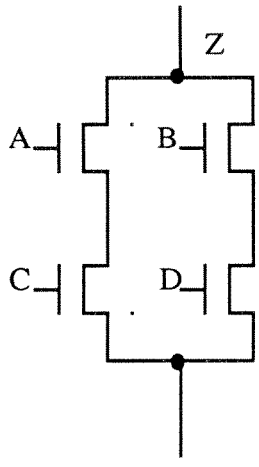


Figure 13(b)

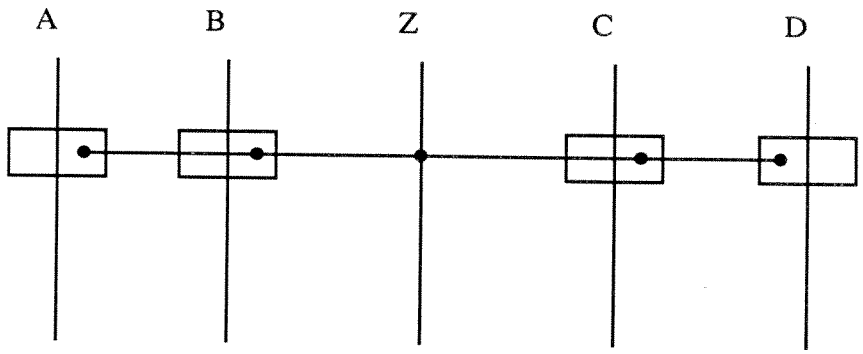


Figure 13(c)

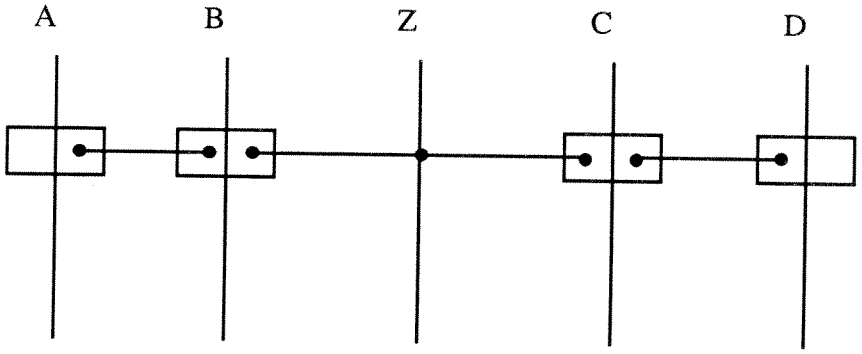


Figure 13(d)

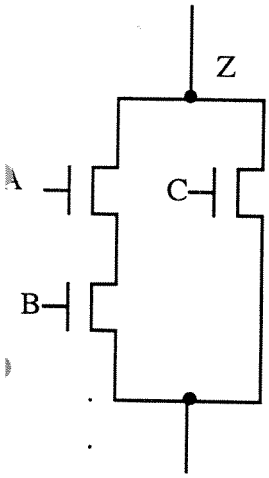


Figure 14(a)

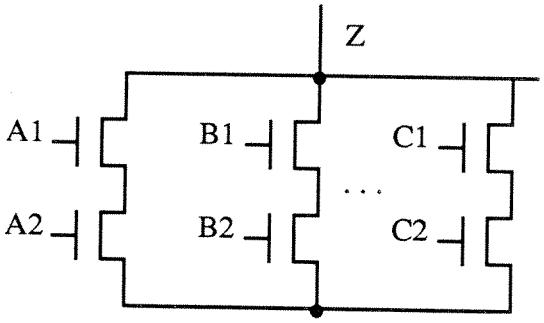


Figure 14(c)

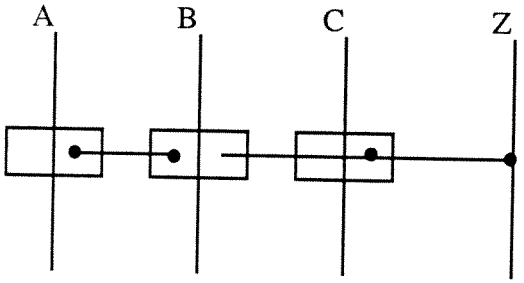


Figure 14(b)

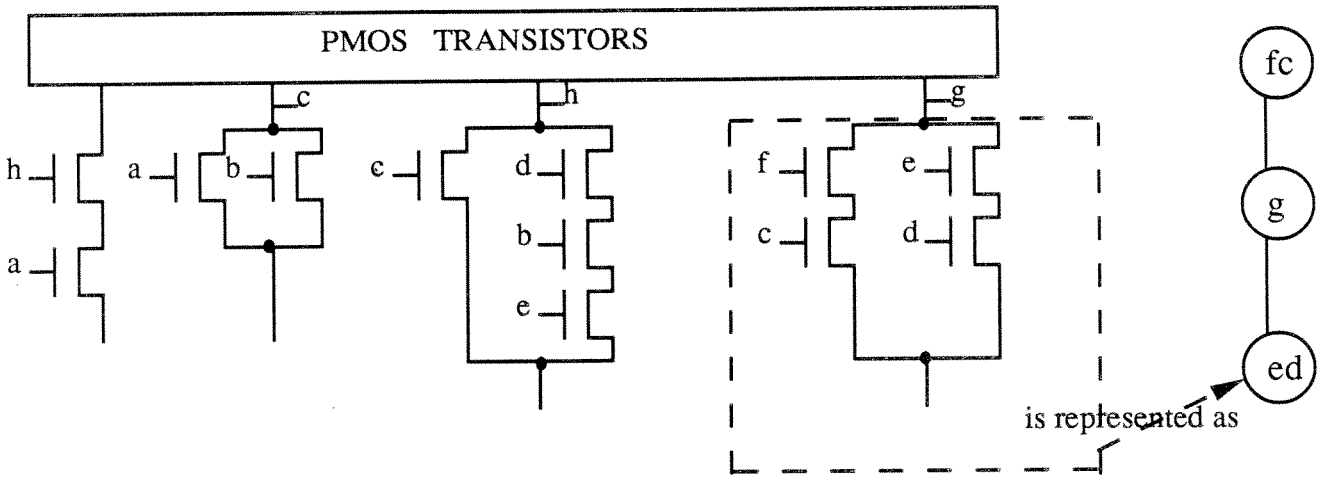


Figure 15(a)

Figure 15(b)

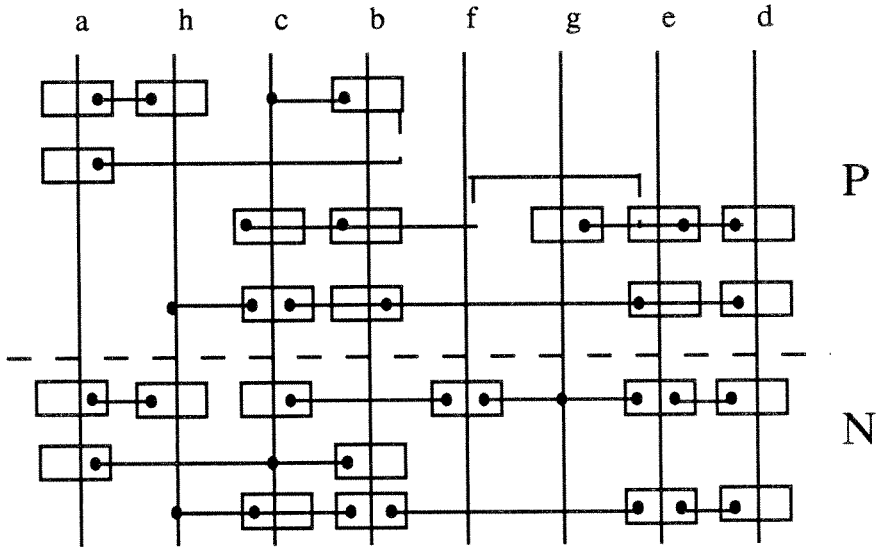


Figure 15(c)

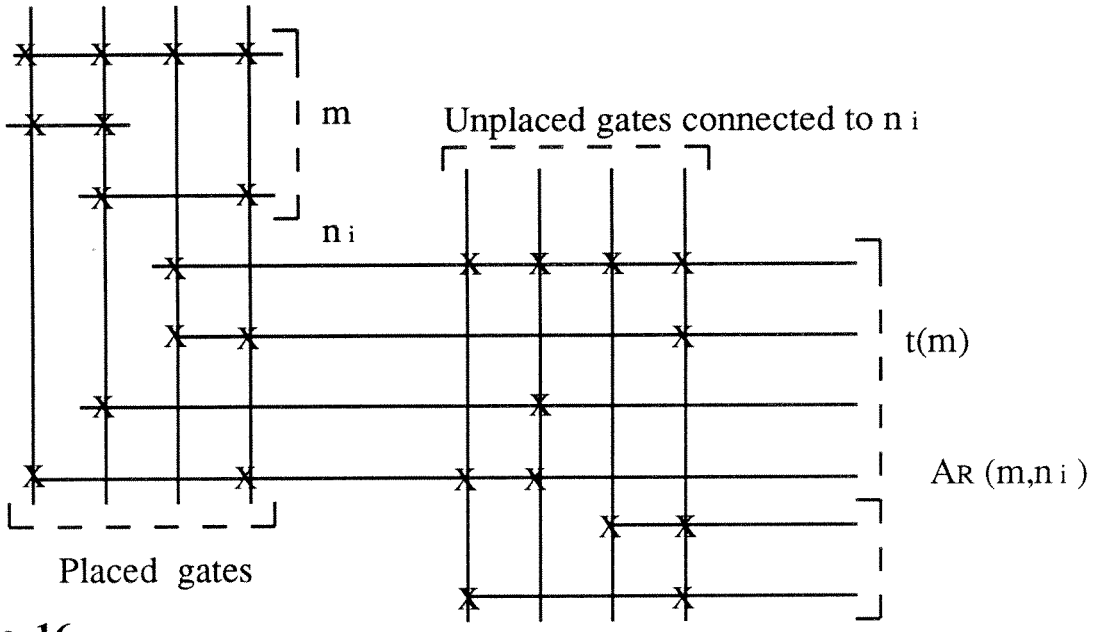


Figure 16

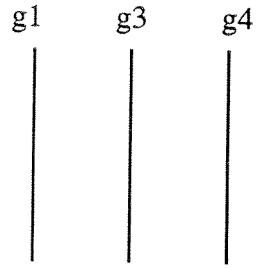
Let the netlist be n1 = { g1, g3, g4}; n2 = { g2, g3}; n3 = { g3, g4 }; n4 = { g2, g5 }

Assume that n1 has been assigned. So, m = { n1 }; t (m) = { n2, n3 }

We have $f_E(m, n_i) = A_R(m, n_i)$.

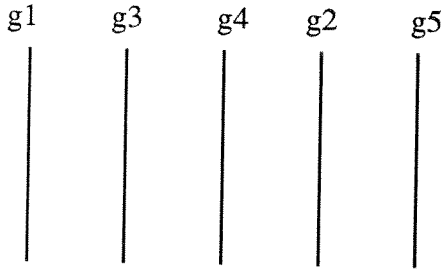
$A_R(m, n2) = 1$; { i.e. n4 }; $A_R(m, n3) = 0$; $A_R(m, n4) = 0$;

So, choose n3. No gates are added. We have the following sequence.



m = { n1, n3 }

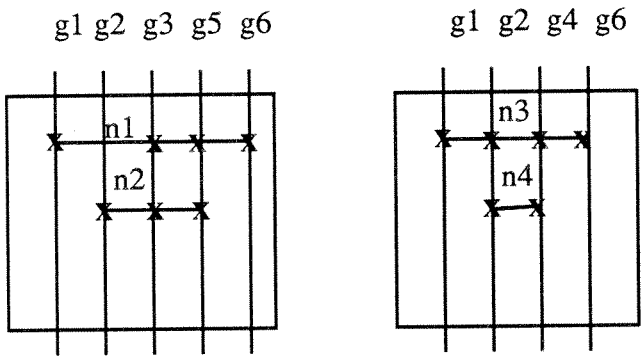
Since no gates are added, AR (m, n2) and AR (m, n4) remain unchanged. Choose n4.



Now, all gates have been assigned.

Figure 17.

et the netlist be n1 = { g1, g3, g5, g6 }; n2 = { g2, g3, g5 }; n3 = { g1, g4, g6};
n4 = { g2, g4 };
et the two blocks be { n1, n2 } and { n3, n4 } as shown in the figure below.



Cutgain (n1) = partialgain (g1, n1) + partialgain (g3, n1) + partialgain (g5, n1)
+ partialgain (g5, n1) = 1 + -1 + -1 + 1 = 0.
Cutgain (n4) = partialgain (g2, n4) + partialgain (g4, n4) = 1 + -1 = 0.

Heightgain (n1) = 2 because critical gates in block 1 are g3 and g5 and in block 2 is g4.
Heightgain (n4) = 2.
Heightgain (n2) = 2.

Figure 18.

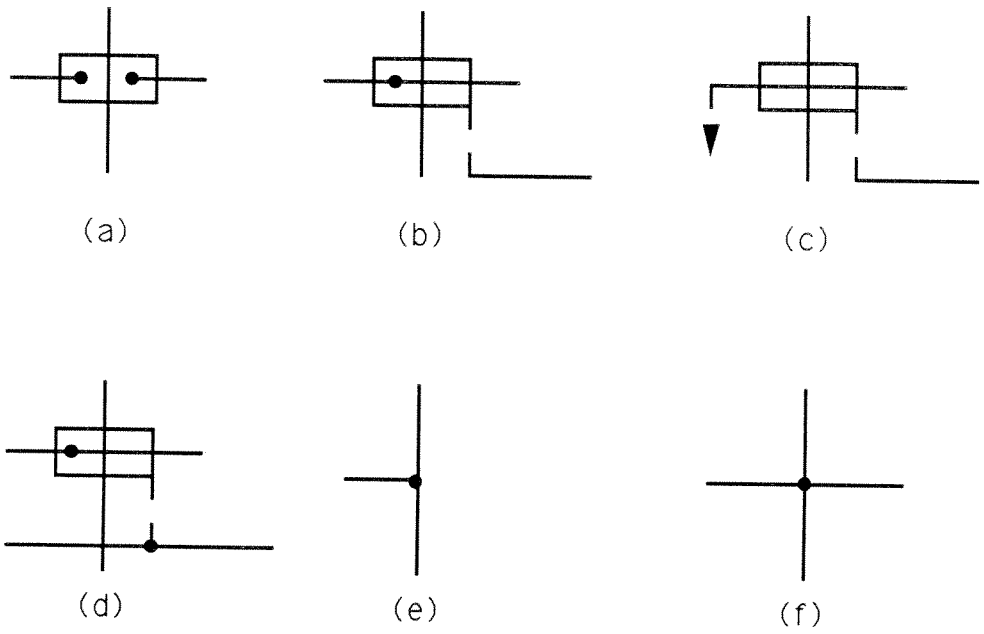
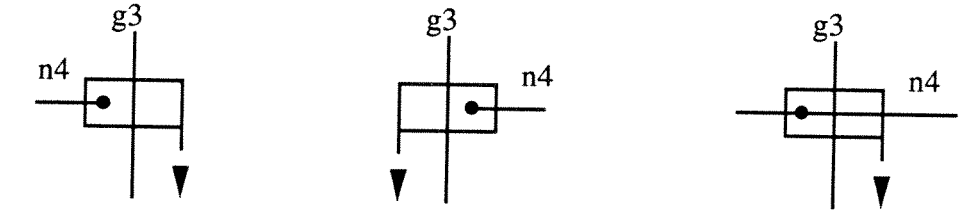
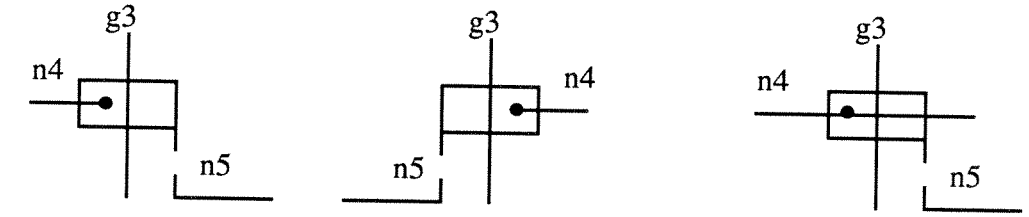


Figure 20.



< g3, n4, 2, 0, 0, 1>
< g3, n4, 2, 0, 1, 0 >
< g3, n4, 2, 0, 0, 0>



< g3, n4, 3, n5, 0 ,1>
< g3, n5, 3, n4, 1, 0>
< g3, n4, 3, n5, 1, 0>
< g3, n5, 3, n4, 0, 1>
< g3, n4, 3, n5, 0, 0>
< g3, n5, 3, n4, 0, 0>

Figure 21

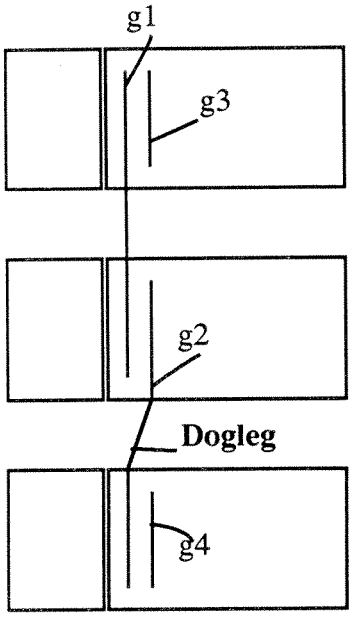


Figure 18(a)

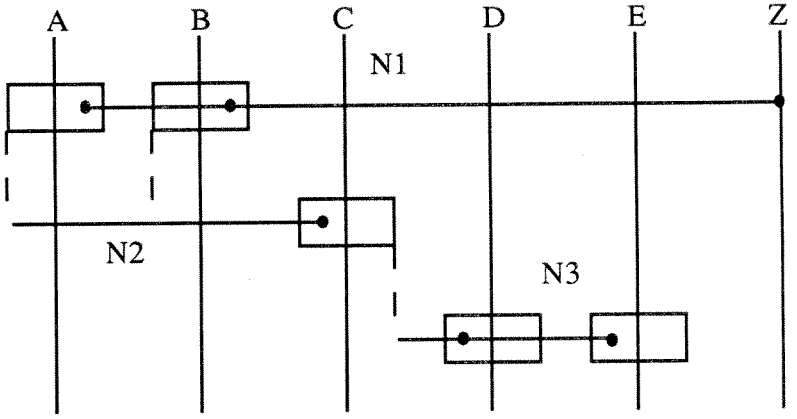


Figure 19(a)

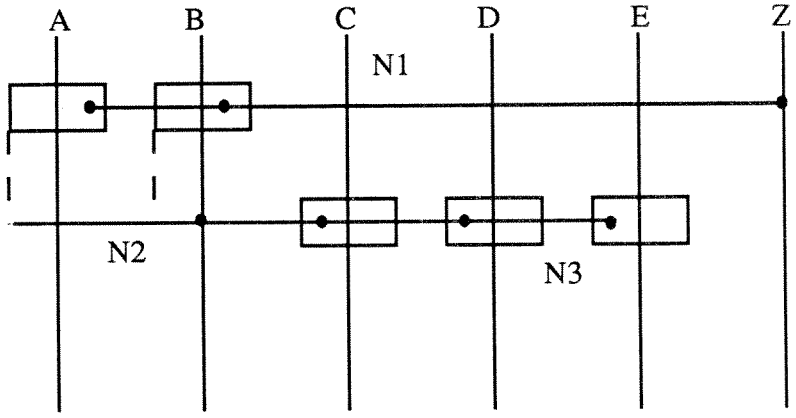


Figure 19(b)

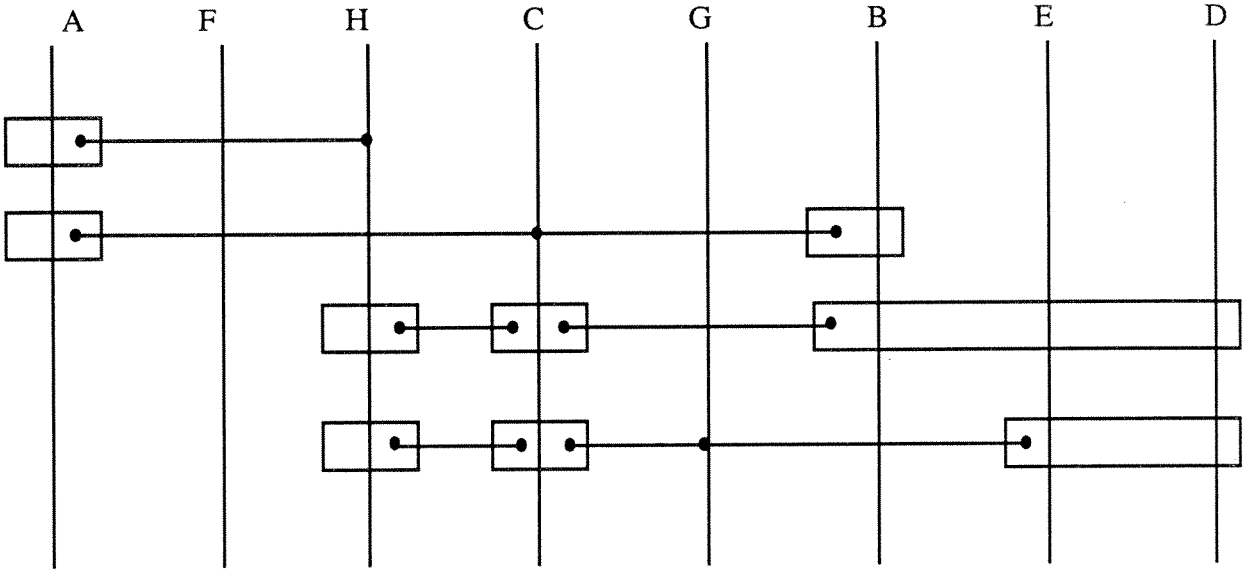


Figure 19(c)

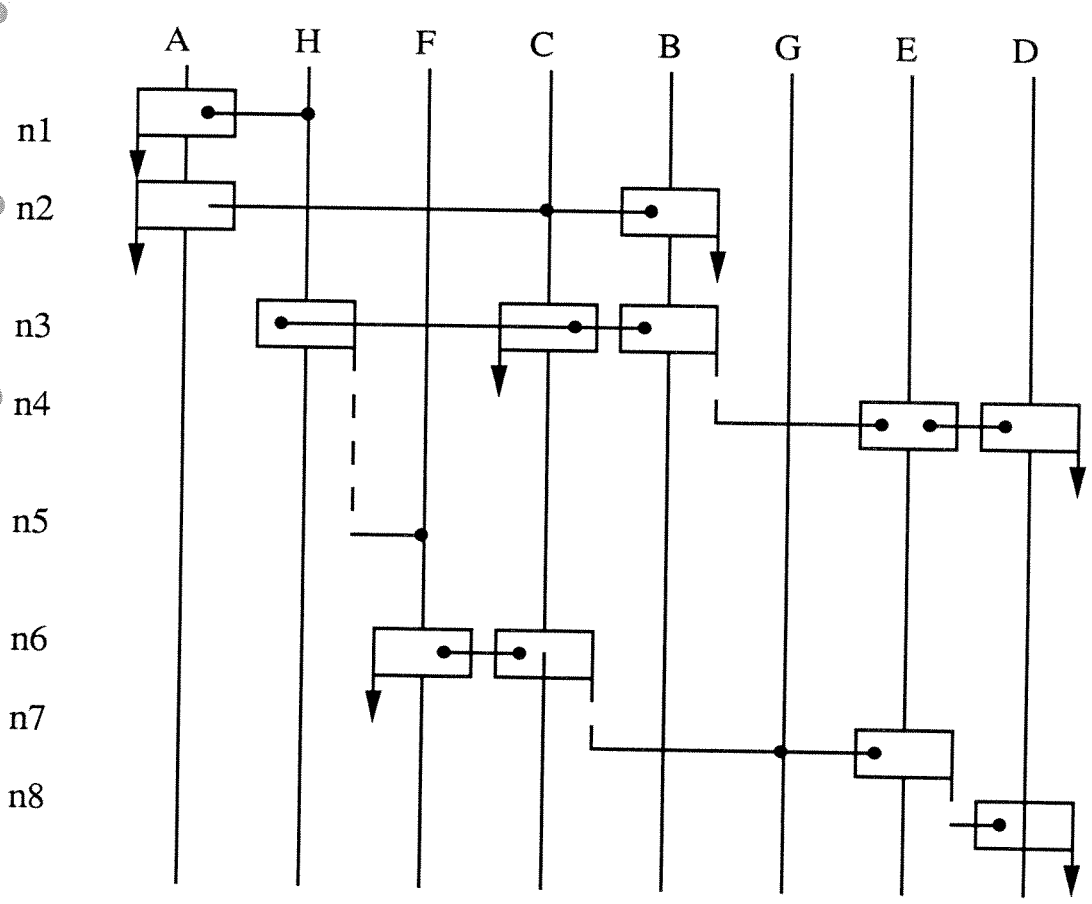


Figure 22(a)

To calculate D(F), the density at gate F.
F has 4 6-tuples <3,3,1,0,0,0>, <3,3,1,0,0,0>, <3,5,0,0,0,1> and <3,6,2,0,1,0>

Step 1: Only one 6-tuple with c = 3; Hence, step 1 is skipped.

Step 2: No pair; Skipped.

Step 3: For <3,6,2,0,1,0>, we have <3,3,1,0,0,0>. Mark <3,6,2,0,1,0> with lh ; delete <3,3,1,0,0,0>; Add 1 to D(C); Hence, D(C) = 1;

Step 4: For <3,5,0,0,0,1>, we have <3,6,2,0,1,0> marked with lh.; delete both; Set D(C) = 2;

Step 5: No 6-tuple marked with lh or rh left in contention.

Step 6: Add 1 to D(C) because <3,2,1,0,0,0> because <3,2,1,0,0,0> is left; D(C) = 3.

After calculating D(A), D(B), . . ., D(H) similarly, we find that $D_{max} = 3$.

Figure 22(b): Continued on the next page.

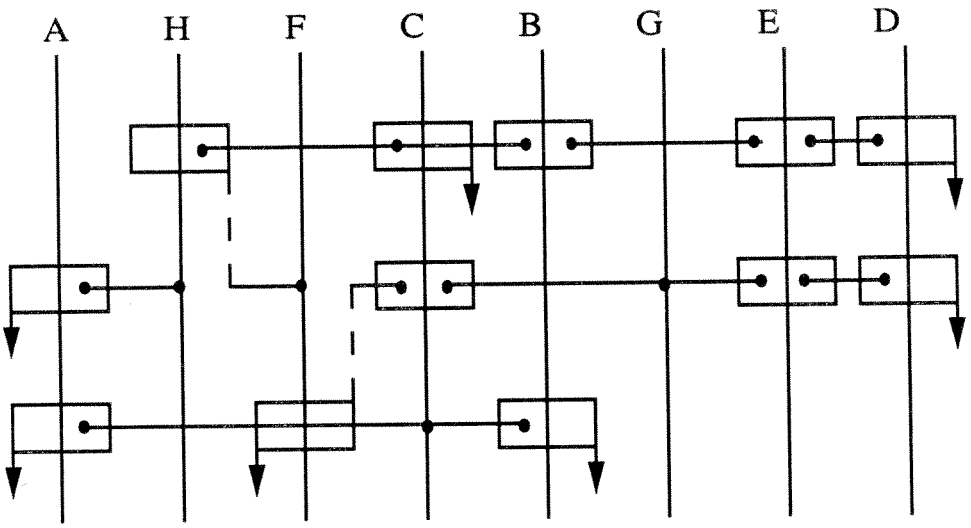
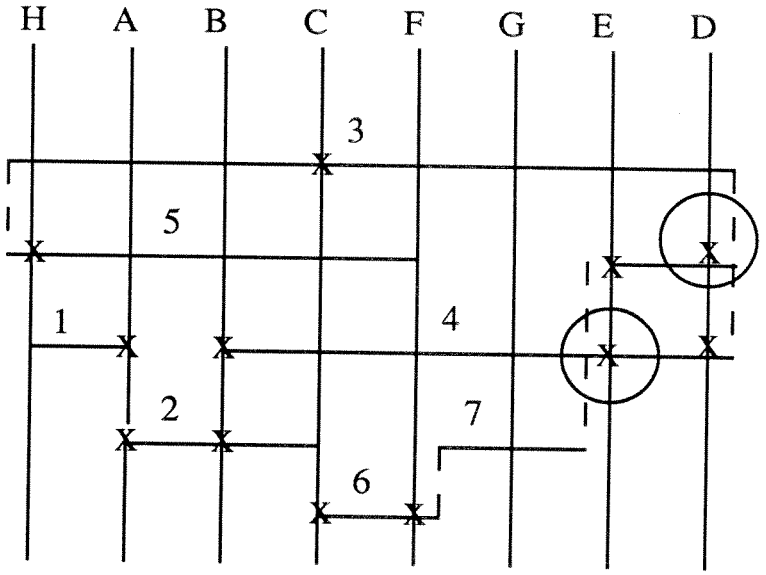


Figure 22(b)

Nets	Gates
1	A, H
2	A, B, C
3	C, D, H
4	B, D, E
5	F, H
6	C, F
7	E, F, G
8	E, D

Figure 23(a).



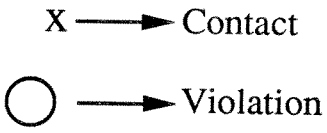


Figure 23(b).

Step 1: Assign nets 1, 3 and 5 to different rows because they are connected to the leftmost gate H.

Step 2: None of the first three rows are empty at gate A. So, put net 2 on row 4.

Step 3: Row 3 is empty at gate B. Assign net 4 to row 3.
and so on.

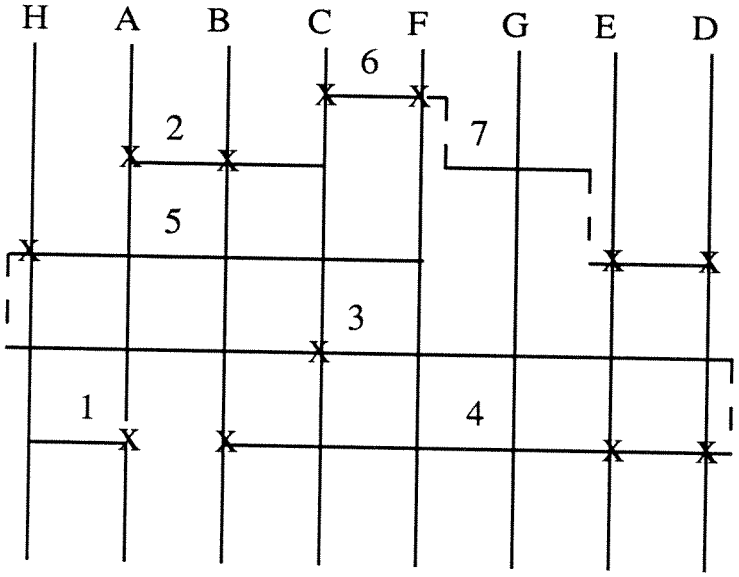


Figure 23(c).

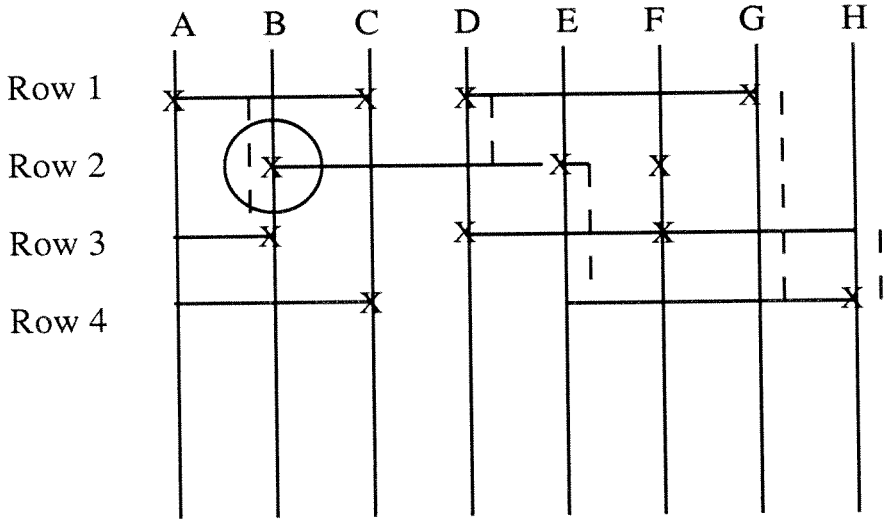


Figure 24(a)

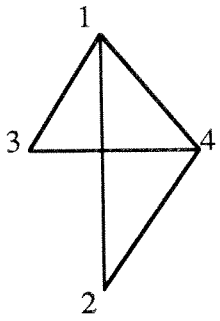


Figure 24(b)

Iteration 1: Only one cycle group here. No extensions. Choose row 1 arbitraily.

Iteration 2: Set of acceptable candidates = { 4}. So, choose row 4.

Iteration 3: Acceptable candidates = {}. Row 3 has minimum number of diffusion runs to rows 1 and 4. So, choose row 3. Enlarge spacing between gates D and E.

Iteration 4: Acceptable candidate set = {2}, because of the enlargement. Choose row 2. This is shown in figure 24(f).

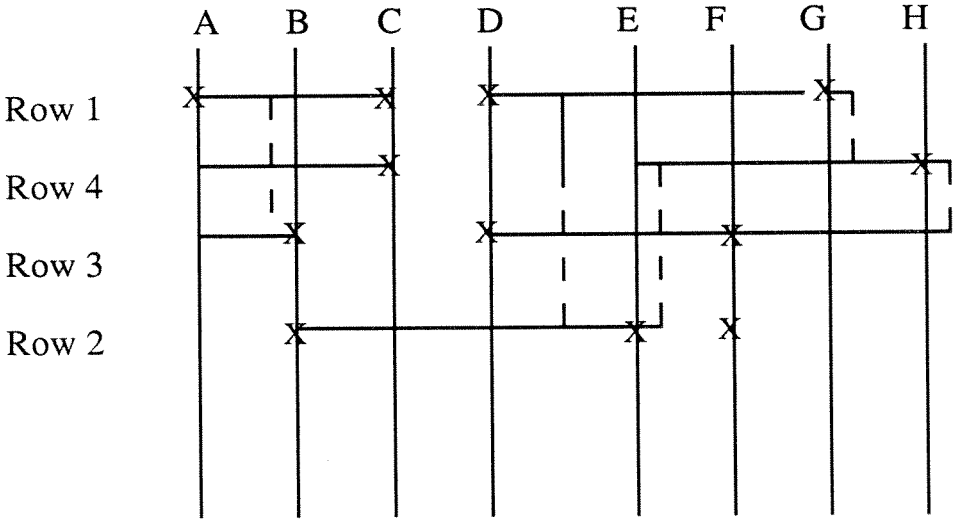


Figure 24(c)

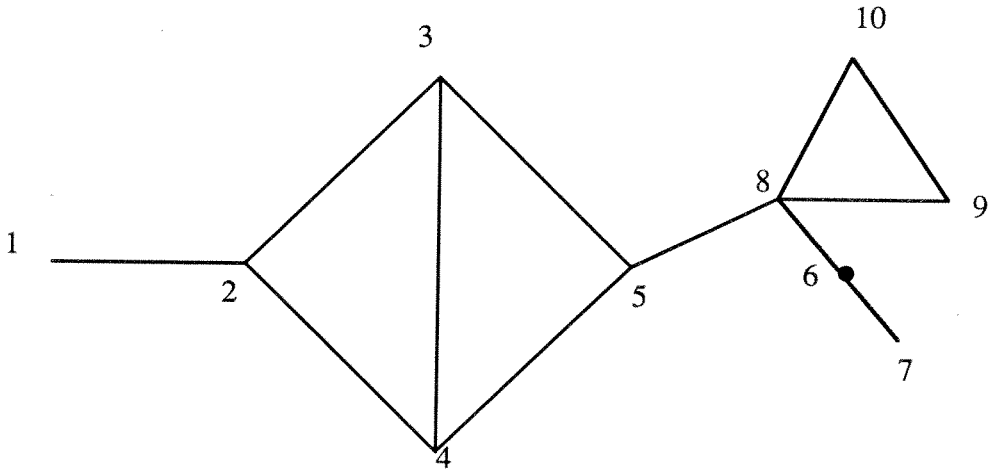


Figure 25.

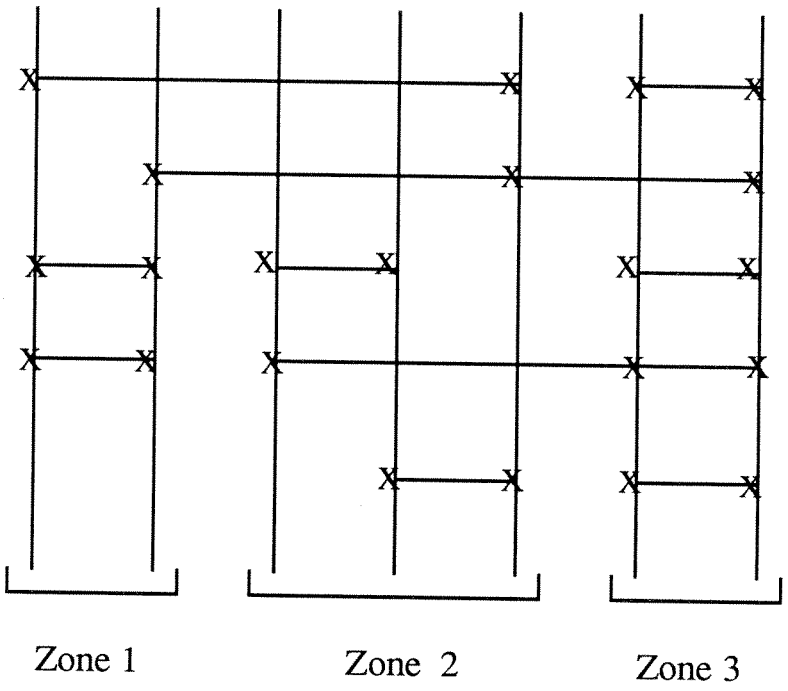
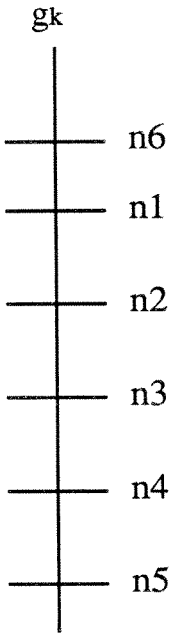


Figure 26.

Suppose we have a diffusion run involving nets n1, n2, n3 and n4 at gate gk. Let the other nets connected to gate gk be n5 and n6. Two orderings of the nets on gate gk are shown. One is realizable, while the other has a conflict.

Realizable.



Unrealizable. Conflict at n6.

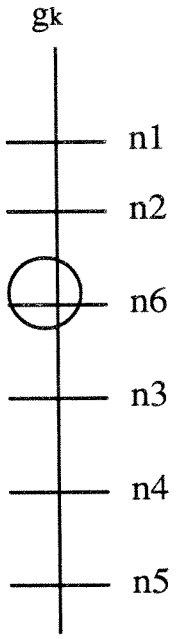
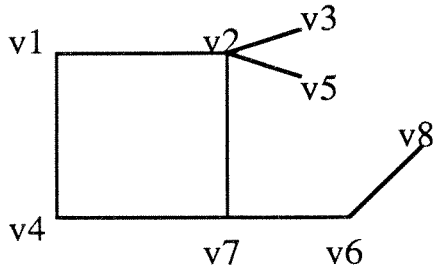


Figure 27(a)

So, for the gate gk, we have two constraints, r1 = (n1 n2 n3 n4) n5 and r2 = (n1 n2 n3 n4) n6. That is of course true only if [n1 n2 n3 n4] is the only diffusion run on gate gk. If, for example, [n3 n4 n5] is another diffusion run, we have to add the constraints r3 = (n3 n4 n5) n1; r4 = (n3 n4 n5) n2 and r5 = (n3 n4 n5) n6.

Figure 27(b)

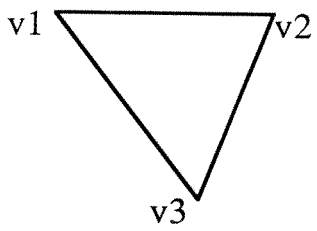
Suppose we have the constraint set R = { (n1 n3) n2, (n2 n4) n7, (n2 n4) n1, (n7 n5) n2, (n7 n8) n6}. The graph G =(V,E) is given by



Suppose (v1 v2) has been directed as v2 ⟶ v1. Now, (v2 v3) has to be oriented as v2 ⟶ v3. Otherwise, we have a conflict because of constraint 1.

Figure 27(c)

Suppose we have the constraint set R = { (n1 n2) n3, (n2 n3) n1, (n1 n3) n2 }.
Hence, we have the graph G =(V,E) below.

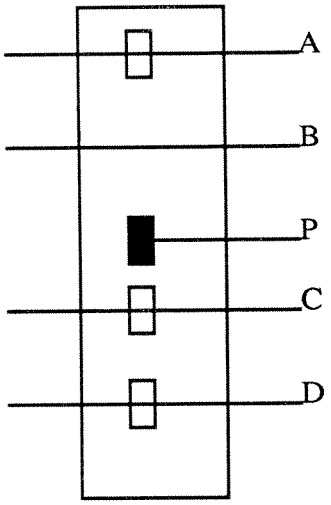


Constraint 1: Orient (v3 v1) as v3 → v1. So, we have to orient (v3 v2) as v3 →v2.

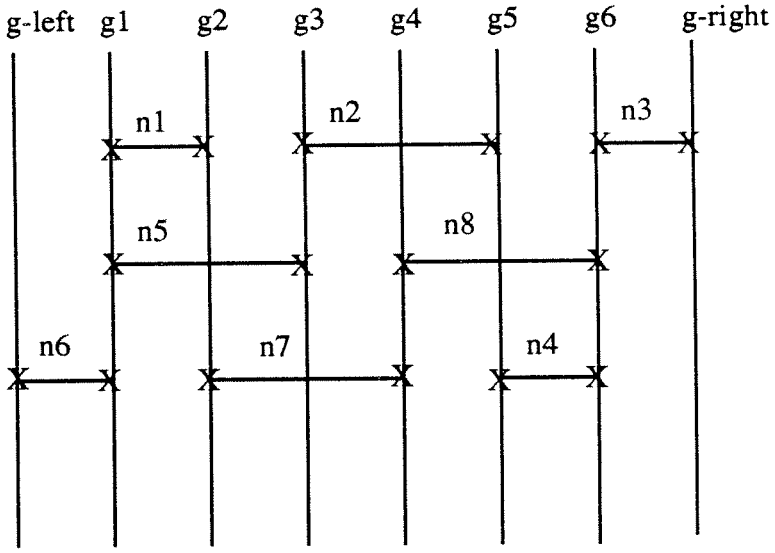
Constraint 2: (v3 v1) is already v3 →v1. So, we orient (v1 v2) as v2 → v1.

Constraint 3: To satisfy this constraint, we need either v2 →v1 and v2 →v3 or v1 → v2 and v3 → v2. But, either of these will violate the orientation induced by the earlier constraints.

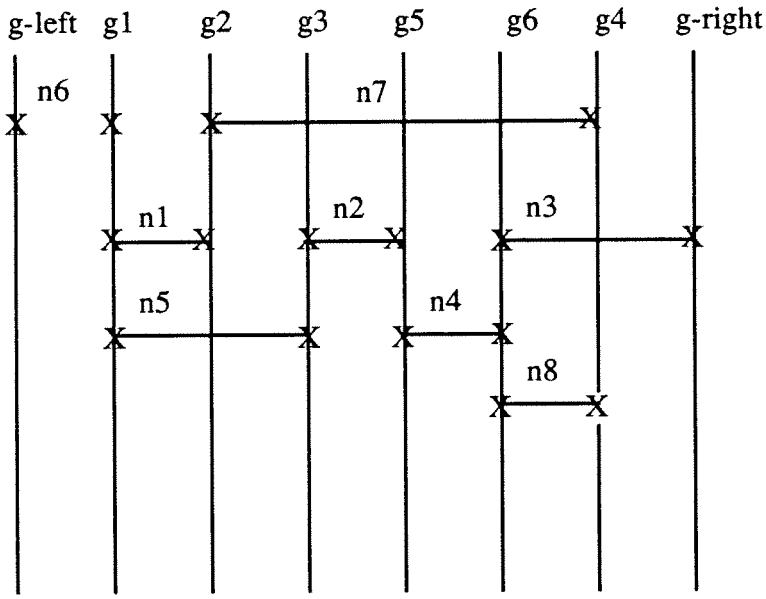
Figure 27(d)



28(a)



28(c)



28(b)

Figure 28.

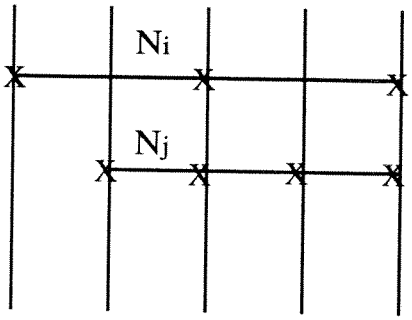


Figure 29(a)

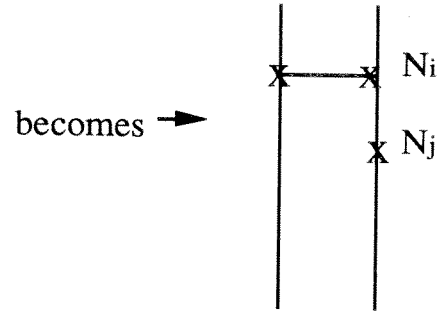


Figure 29(b)

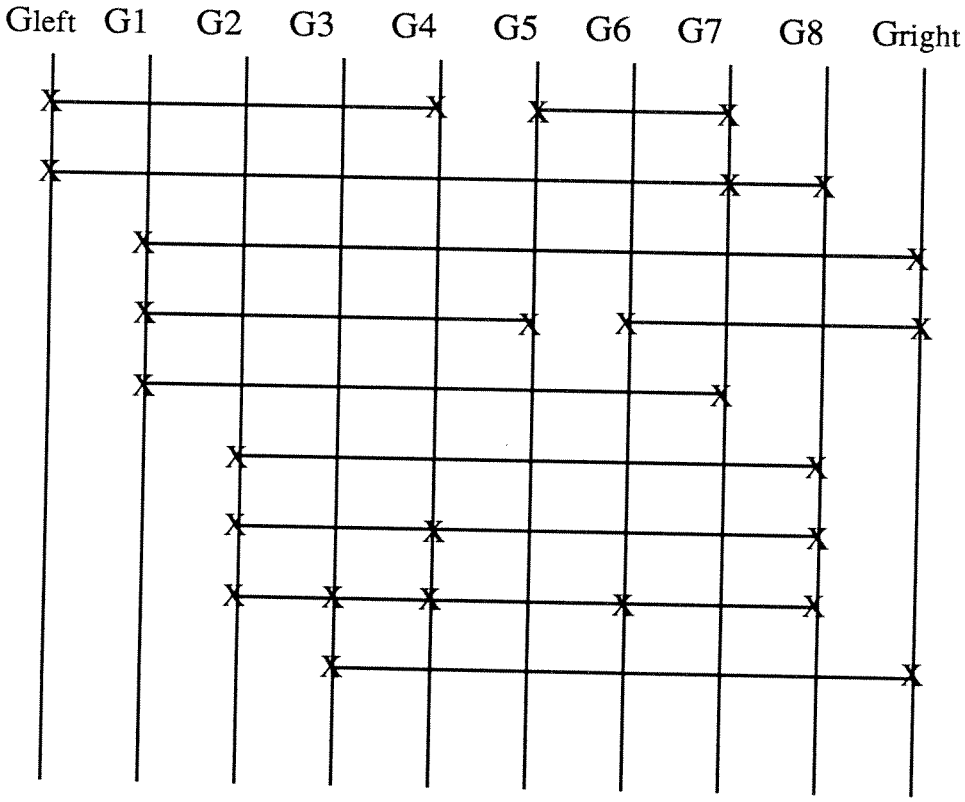


Figure 30(a)

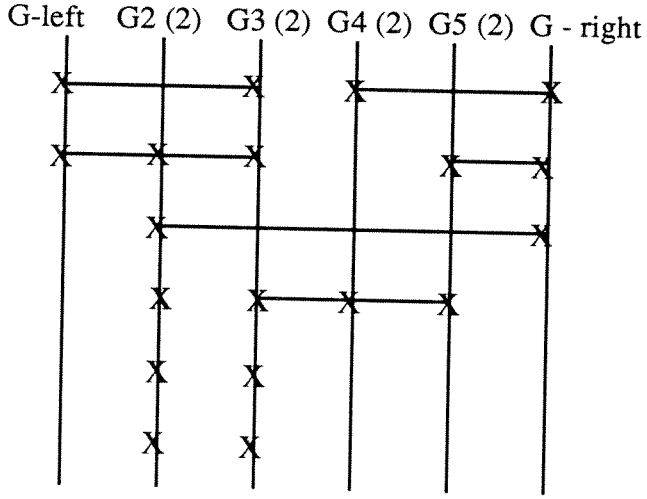


Figure 30(b)

G2 (2) means that 2 levels of contraction have been made to produce gate G2 and so on.

Let n1 = { cL, c1, c2 }; n2 = { c3, c5, cR }; n3 = { cL, c1, c3, c5 }; n4 = { cL, c4, c6 }
n 5 = { cL, c2, c6} ; n6 = { c3, c5, c7, cR }; n7 = { c4, c6, c7}; n8 = { c5, cR }

Figure 31(b).

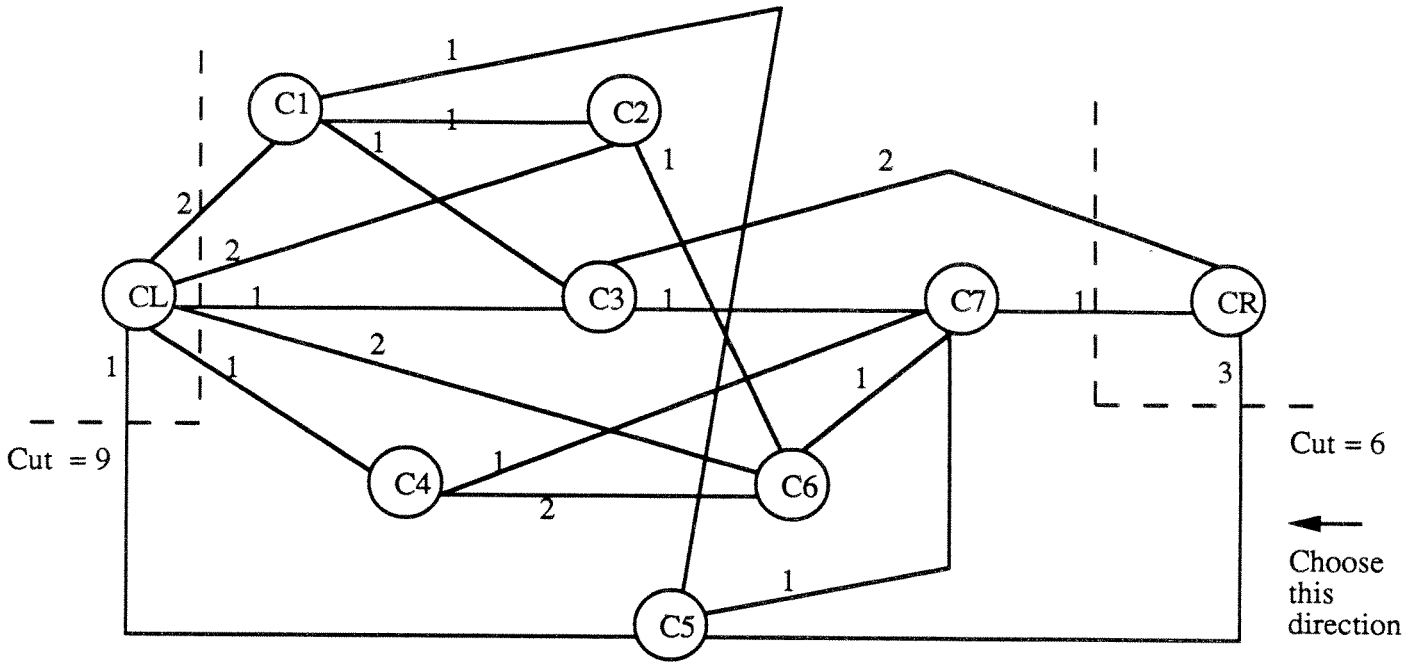


Figure 31(a).

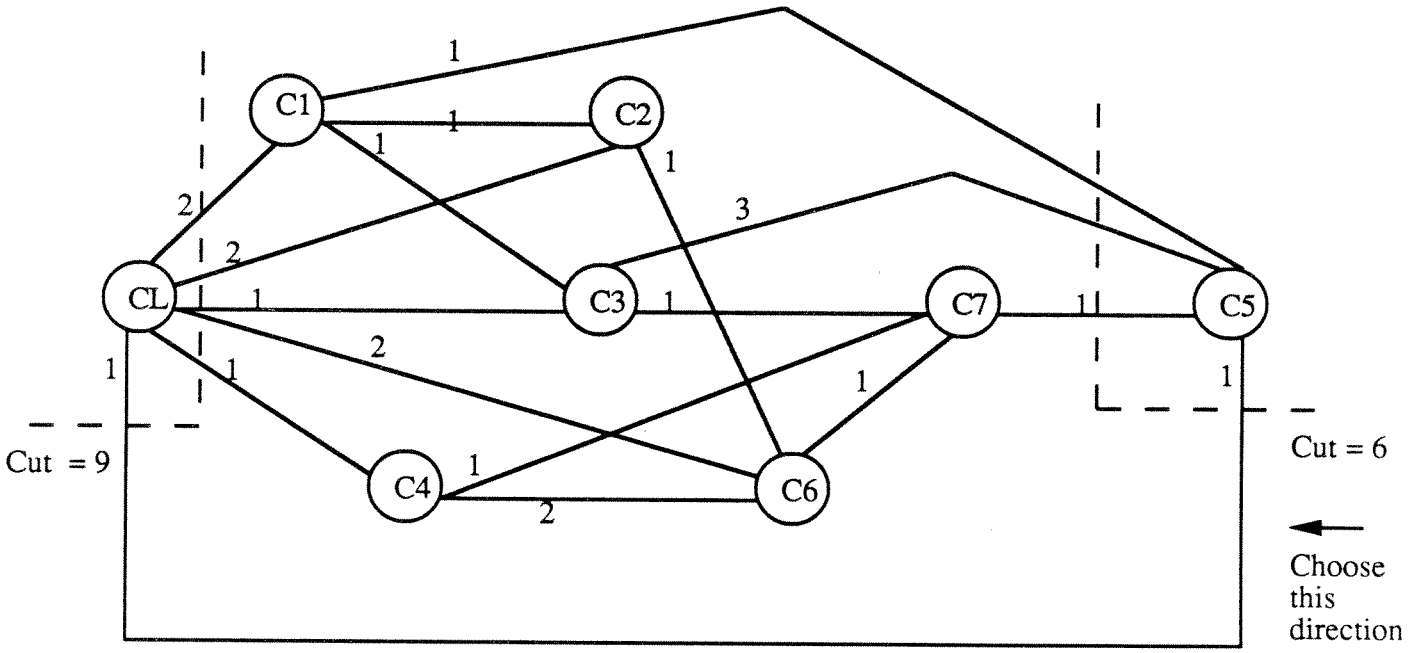


Figure 31(c).

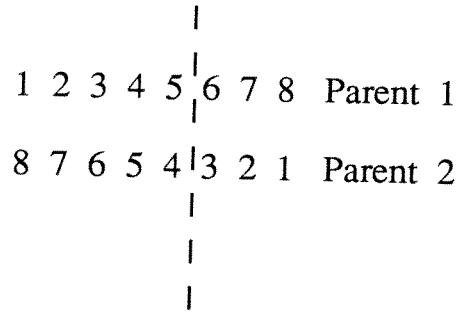
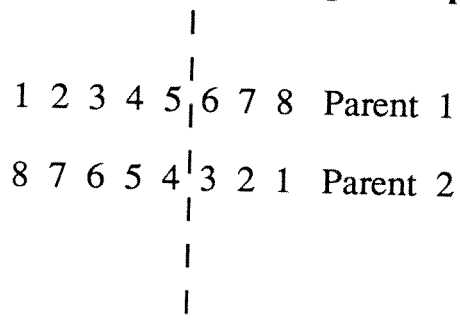


Figure 32(a): Simple Crossover.

1 2 3 4 5 3 2 1 : Illegal Offspring
8 7 6 5 4 6 7 8 : Illegal Offspring

Figure 32(b): Illegal offspring .



1 2 3 4 5 8 7 6 Offspring 1

8 7 6 5 4 1 2 3 Offspring 2

Figure 32(c): Order Crossover. Substrings occur in the same order as in parent.