YACR2: YET ANOTHER CHANNEL ROUTER

by

James B. Reed

Memorandum No. UCB/ERL M85/16

28 February 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

James Reed

YACR2: Yet Another Channel Router

RESEARCH PROJECT

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, in partial satisfaction fo the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee: [illegible], Research Advisor

2/ 85 Date

[illegible]

2/12/85 Date

YACR2: YET ANOTHER CHANNEL ROUTER

James B. Reed

Electronics Research Laboratory

Electrical Engineering and Computer Science Department

University of California

Berkeley, California 94720

ABSTRACT

YACR2 is a channel router that minimizes the number of through vias in adition to the area used to complete the routing in a two-layer channel. This report explains the algorithms used and their implementation in C.

ACKNOWLEDGEMENTS

First I must mention that YACR was the idea of Prof. Alberto Sangiovanni-Vincentelli, my research advisor, and Prof. Mauro Santomauro. I thank them for giving me the opportunity to develop it into a usable program.

Indeed, enough cannot be said about Alberto. There have been times when I thought I would never see the end of his suggestions, requests, and even occasional demands. Trying to keep up with Alberto is both a challenge and a pleasure; his energy is difficult to match.

I thank Rick Rudell and Deirdre Ryan for working with me to interface YACR2 to the HAWK graphics editor and SQUID data base system. I would also like to thank Chris Marino for being the first to use this interface to design a chip. There are several features of YACR2 that would not have been added if I hadn't been able to see plots of YACR2's routing.

I must thank Aart de Geus for getting me interested in CAD. More importantly, Aart was the one who talked me into staying at Berkeley to finish my degree when it seemed easier to leave that to get accustomed to living here. Life in Berkeley is quite different from life in Arkansas.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. History

Channel routing has been used extensively in the layout of integrated circuits (ICs) and printed circuit boards in the past ten years [1-6]. It is flexible enough to allow its use in various design styles such as gate-arrays, standard cells and macro-cells (building blocks).

A channel router is designed to route nets that interconnect terminals on two opposite sides of a rectangular region called the **channel**. Most of the channel routers proposed assume that only two layers are available for interconnection and that components of a routed nets are horizontal and vertical **segments**. It is also assumed that segments in one direction (say horizontal) are run on one layer and segments in the other direction are run on the other layer. Terminals are placed at regular interval, and identify the **columns** of the channel. Horizontal segments are placed so that design rules are not violated. This constraint identifies the **rows** of the channel. When a net is broken into two or more horizontal segments occupying different rows, **doglegging** is used. Doglegging is effective to reduce the number of rows of the channel, but it requires additional vias.

The goal of a channel router is to complete the interconnections in minimum area. Number of through vias and length of nets are also important to evaluate the quality of the routing. These criteria are in general considered secondary objectives, but we believe that they should be given more emphasis because of their effects on the reliability and speed of the circuits laid out.

The first algorithm for channel routing, the **Left-Edge Algorithm**, (LEA) was proposed by Hashimoto and Stevens [1]. This method uses a **row-by-row** approach, assigning nets to each rows and trying to minimize the number of rows needed, i.e.the width of the channel. Various modifications of this basic approach have been proposed [2,3]. Yoshimura and Kuh [4] proposed a channel router based on graph theory concepts. Fiduccia and Rivest [5] proposed a **column-by-column** greedy algorithm; the router scans the channel from left to right and completes the routing for one column before proceeding to the next one. All these methods try to minimize the area of the channel by minimizing the number of rows. Even though they have similar performances, Yoshimura and Kuh and Rivest and Fiduccia report the best results in terms of speed and number of rows used, almost always equal to the theoretical minimum, the density of the channel.

Kawamoto and Kajitani [6] proposed a column-by-column router that guarantees routing with upper bound on the number of rows equal to the density plus one, but additional column are needed to complete the routing. Thus, the goal of this router is to minimize the number of columns of the channel.

This paper describes **YACR2**, (Yet Another Channel Router) that can route channels with cyclic constraints and uses a virtual grid.

YACR2 basically uses one layer for vertical interconnections and the other layer for horizontal interconnections as other channel routers, but allows some exceptions on horizontal interconnections, this feature being the key to its performance. YACR2 uses at most as many tracks as the best published results on a number of test cases. In particular, it routed the famous Deutsch difficult example using 19 tracks as Burnstein's hierarchical router[6], but using less through vias, much less computer time and with no need of interactive parameter adjustments. In a number of industrial examples, YACR2 performed substantially better than existing channel routers.

YACR can route channels with so called cyclic constraints with no modification as Rivest and Fiduccia (the channel router of Yoshimura and Kuh needs a non trivial modification [7]).

1.2. Formulation of the Problem

The formulation of the channel routing problem has been given in several papers. We recall here some basic definitions and give a more general formulation. We assume here that the channel is gridded and that the boundaries of the channel are parallel lines.

Channel area: the space between two parallel rows of cells that have to be interconnected. Since the interconnecting paths lay on an orthogonal grid, a measure of this area is given by $m \times n$, where $m\,(n)$ is the number of rows (columns) of the grid. Two different layers (metal and poly) are available for the interconnections.

Horizontal (Vertical) segment: a segment of interconnecting path that lays on a row (column) of the grid. A horizontal (vertical) segment is specified by the row (column) which it lays on and by the column (row) ends.

Top (Bottom) terminal list [5]. $T = (t_1, \ldots, t_m)$ $(B = (b_1, \ldots, b_m))$. $t_i\,(b_i)$ is a positive integer identifying the net that enters in the channel at the top (bottom) of the i^{th} column. If $t_i\,(b_i) = 0$, no net has to be connected to the i^{th} terminal (null terminal).

Left (Right) connection set R(L): the set of nets that enter in the channel from the left (right) end of the channel. It can be an ordered set if the segments have to be laid out in a specific way.

Local density and density [4]. The local density of a column j, d_j, is the number of nets crossing that column. The density k of a given channel is defined by $k = \max_{1 \leqslant j \leqslant m} d_j$. In figure 2.1, $d = (3, 4, 4, 4, 4, 3, 4, 3, 4, 4, 3)$ and $k = 4$.

Unless otherwise specified, horizontal segments lay on one layer and vertical segments on the other one. Contacts located at their intersections (vias) are used to electrically connect the two layers.

Two horizontal segments on the same layer, which belong to two different nets and have in common at least a column, cannot overlap and must be assigned to different rows. We call this type of constraint **horizontal constraint.**

Similarly two vertical segments on the same layer, which belong to two different nets and lay in the same column, cannot overlap. The lower endpoint of the upper segment **must** be placed in a row **above** the upper endpoint of the lower vertical segment. We call this type of constraint **vertical constraint.**

Horizontal segments of a net can lay on the same row and, for each top or bottom terminal, vertical segments can lay on the same column. The splitting of horizontal (vertical) segments of a net in more then one row and/or column is called horizontal (vertical) **doglegging.** Doglegging can be allowed only at terminal position (restricted doglegging) [4] or in any position of the wire (unrestricted doglegging) [2,3] . Doglegging is used to reduce the channel area.

The **vertical constraint graph** $G(V, E)$ is a directed graph, where each node represents a horizontal segment (a net if horizontal doglegging is not allowed). A directed edge from node i to node j means that horizontal segment i must be placed above horizontal segment j because of a vertical constraint. If there is a loop in G, we have a **Cyclic Vertical Constraint** and the routing requirement cannot be satisfied without doglegging.

The channel routing problem can be stated as follows:

PROBLEM P1

Find an assignment of horizontal and vertical segments to rows and columns such that:

i) the connection requirements specified by T , B , L , R are satisfied

ii) no violation of horizontal or vertical constraints occurs.

In this general formulation, the problem always has a solution, because constraints can be removed by adding rows and/or columns or allowing horizontal and/or vertical segments on the two layers. Often additional constraints are added and several criteria are used to evaluate the quality of the solution. Typical additional constraints are: (i) fixed breadth of the channel (i.e. no columns added to the channel) [4]; (ii) fixed width of the channel (i.e. no more than one row over density) [6]; (iii) no doglegging (or only restricted doglegging).

Typical factors that can be used to evaluated the performances of a router are: (i) the area of the channel; (ii) the number of vias; (iii) the total length of the segments; (iv) the length of the longest net.

These criteria are often competing. For example doglegging may reduce the area of the channel but increases the number of vias.

YACR2 routes acyclic channels and most cyclic channels in fixed breadth and the remaining cyclic channels with the addition of few columns at the end of the channel. The area of the channel is the primary goal, but the number of vias and the net length are also effectively minimized.

1.3. Organization of Report

This report is organized as follows. Chapter 2 outlines the main ideas of our approach to channel routing and describes the algorithm in detail. In chapter 3 the details of implementation are given. Chapter 4 describes the implementation of the algorithm in a routing system. Chapter 5 gives a detailed comparison with existing channel routers and statistics on the running time, number of vias, and net length. Concluding remarks are presented in chapter 6.

CHAPTER 2

THE ALGORITHM

2.1. Basic Ideas

Since various versions of the channel routing problem has been shown to belong to the class of NP-complete problems [8], heuristic algorithms have been proposed for its solution.

Our method is based on the following two considerations:

i) The Left-Edge Algorithm (LEA) assigns horizontal segments to the rows of the channel without overlap in density, i.e. using exactly k rows [1]. Since horizontal constraints are satisfied while vertical constraints are disregarded, **vertical constraint violations** may be introduced by LEA.

ii) After routing a channel, there are usually several horizontal and vertical segments that are not used for routing any of the nets. This space is available "for free" and may be used to remove vertical constraint violations.

With these considerations in mind, the basic idea of YACR2 is to start with a number of rows equal to k and then:

1. apply a modified LEA that preserves LEA's characteristics of routing in density but exploits all degrees of freedom to reduce the number of vertical constraint violations;
2. attempt to remove vertical constraint violations by exploiting the "free" space with simple maze routing techniques;
3. if all violations are not removed, increment the number of rows and return to step 1.

Modifications to this basic strategy allowed us to route cyclic channels with the possible additions of external columns to the channel. In addition, we were able to devise a strategy that did not always discard the previous routing if all the vertical constraint violations were not removed. This strategy produced in some cases better area utilization and faster running time in the most complex channels.

2.2. The Net Assignment Algorithm

The overall algorithm has four phases: the first three assign nets to tracks trying to minimize vertical constraint violations and using as many tracks as given; the fourth (described in section 3.4) attempts to remove vertical constraint violations.

In this section, we describe the first three phases for channels with no cyclic constraints. In Section 4 we will describe the modifications to the basic algorithm for channels with cyclic constraints.

The first three phases are similar. In the first one the nets crossing a column of maximum density are assigned to tracks, in the second the nets to the right of this column are assigned and finally the nets to the left of this column are taken care of in the third phase. The third phase is identical to the second one provided that the words left and right are changed throughout. For this reason, the details for this phase will be omitted. Before giving the algorithm used to assign nets we need the following definition:

Definition 1. $S(x) = \{s_1, s_2, \ldots, s_p\}$ is the set of p rows to which net x may be assigned without a horizontal constraint violation.

Assignment Algorithm

Data: a channel and the set of nets to be routed, the number of rows to be used, the density of the channel, k.

```
Phase I (Maxcol nets assignment)

        ( Initialization)
        Choose a column, maxcol, s.t. $d$ ( maxcol ) = $k$ ;
        let N = $\{n_1, n_2, \ldots, n_k\}$ be the nets crossing maxcol;
```

```
while ( N $\neq \emptyset$) {

        ( Choose an $n_s \in$ N to be assigned)
        $n_s$ = select(N);

        ( assign $n_s$ to a row in $S(n_s)$)
        assign ($n_s$, $S(n_s)$);

        ( update)
        Remove $n_s$ from N;

}
```

Phase II (Modified Left Edge Algorithm)

```
( Initialization)
set $j$ = maxcol + 1;
set N = $\emptyset$;

while ( $j \leqslant$ numcols ) {

        ( Collection)
        if (there exists one (or two) net(s) with left endpoint on $j$ ) {
                N = N $\bigcup$ the net(s) with left endpoint on $j$ ;
        }

        ( Assignment)
        if (there exists a net $n_l$ with right endpoint on $j$ ) {
                while ( N $\neq \emptyset$) {
                        $n_s$ = select(N);
                        assign ($n_s$, $S(n_s)$);
                        Remove $n_s$ from N;
                }
        }

        ( Move to the right)
        $j = j + 1$;
}

( Assign nets exiting right edge)
while ( N $\neq \emptyset$) {
        $n_s$ = select(N);
        assign($n_s$,
        Remove $n_s$ from N;
}
```

Phase III (Modified Right Edge Algorithm)

Omitted.

The algorithm described above is not fully specified since the two procedures ***select*** and ***assign*** have not been described. However, we can still prove some properties of the algorithm by looking at its structure. In particular, we can prove that the Assignment Algorithm lays all the nets in k rows, where k is the density of the channel, as long as the selection of the nets to be placed, and the rows in which the nets are placed, are made as specified in the algorithm, i.e. the selection from the set N and the assignment using rows in $S(n_s)$.

Lemma 1. If the Assignment Algorithm has processed column j and $n_s \in$N, $S(n_s)$ is the set of rows that do not contain a net in column j.

Proof: The proof is obvious for phase I. For the sake of simplicity, we will consider only the columns scanned by phase II. If n_s is added to N at column j, i.e., it has its leftmost point on j, then the lemma follows from the fact that no net has been assigned which has its leftmost point to the right of j. Assume now that n_s has been added to N in column $\hat{j}$. Then, the lemma holds for $\hat{j}$. Since between $\hat{j}$ and j no nets in N nor any nets previously placed terminate, the set of rows which do not contain a net in $\hat{j}$ is identical to the set of rows which do not contain a net in j.

Theorem 1 Let k be the density of the channel. The Assignment Algorithm lays all nets in k rows.

Proof: For the sake of contradiction, assume that the algorithm was not able to lay net n_s in k rows of the channel, i.e. $S(n_s) = \emptyset$. By Lemma 1, for $S(n_s)$ to be empty, there must be k nets crossing column j. Therefore, the density of column j must be at least $k + 1$.

2.3. Select and Assign

The procedures ***select*** and ***assign*** have the same basic goals: place nets so that (1) the vertical constraint violations are minimized; and (2) it is easy for the simplified maze routers to complete the routing. Unfortunately, it is impossible to determine all the vertical constraint violations caused by the placement of a certain net. In fact, some of the vertical constraint violations may occur between the net under consideration and nets that have yet to be placed. Since the nets are placed with no dogleg, we can use the vertical constraint graph, G, to estimate how likely is that an assignment cause violations, and how difficult it is to remove a violation if it occurs.

Select and ***assign*** are fairly complicated procedures. Many heuristic ideas have been combined to produce an assignment which has given excellent practical results. Note that the heuristics described in this section are the results of many experiments carried out on examples from industrial chips, university designs and artificially generated test cases. The introduction of the heuristics is done to favor understanding of the concepts. The actual implementation is reported in chapter 3.

G, the vertical constraint graph, is used to determine rows in which a net, n_s, cannot be placed without causing at least one vertical constraint violation. If $p(n_s)$ is the longest path passing through the vertex of G corresponding to n_s, then the nets that come before n_s in $p(n_s)$ must be placed above n_s (each in their own row) and the nets that follow n_s in $p(n_s)$ must be placed below n_s. Using $p(n_s)$ we define two sets: $\overline{P}_t(n_s)$ contains the top $t(n_s) - 1$ rows, where $t(n_s)$ is the number of nets that preceed n_s in $p(n_s)$; likewise, $\overline{P}_b(n_s)$ contains the bottom $b(n_s) - 1$ rows, where $b(n_s)$ is the number of nets that follow n_s in $p(n_s)$. We call the union of $\overline{P}_t(n_s)$ and $\overline{P}_b(n_s)$, $\overline{P}(n_s)$. If n_s is assigned to any row in $\overline{P}(n_s)$, there will definitely be a vertical constraint violation.

We define the set $P(n_s)$ to be the rows not in $\overline{P}(n_s)$. The set

$$S_1(n_s) = S(n_s) \bigcap P(n_s)$$

is the set of rows in which we would most like to assign n_s. We prefer to assign n_s to a row in $P(n_s)$ because such an assignment will not necessarily cause a vertical constraint violation. Note that we cannot guarantee that if n_s is placed in $S_1(n_s)$ no violations will occur. In fact, violations may occur if some of the nets above n_s were (or will be) placed too low, or some of the nets below n_s were (or will be) placed too high. However, the information carried by $S_1(n_s)$ is strongly related to the structure of the channel and we have experimentally observed that it tracks well with the number of vertical constraint violations after the assignment algorithm is completed and with the ease of removing vertical constraint violations.

If $S_1(n_s)$ is not empty, then we have to decide which row in it will be chosen. To do this, we compute the number of vertical constraint violations that placing n_s in each row of $S_1(n_s)$ will cause with respect to nets already placed. Of course, the best position will be the one which will cause the minimum number of vertical constraint violations with respect to nets already placed.

At a first glance, it may seem that choosing a row that has the minimum number of vertical constraint violations with respect to the nets already placed, should be the primary criterion. However, when we tried this selection strategy, we obtained worse results. A possible explanation for this fact is the following: If we were forced at a certain step to place a net in the "wrong" position, it would not be wise to force other nets to follow this "mistake" but it would be better to place the nets by looking at the structure of the channel which is best summarized in $S_1(n_s)$.

After this criterion has been applied, we may still have a tie among some rows. This tie is broken by choosing the row that is closest to the "ideal" row in which to place n_s.

Placing n_s in row $t(n_s)$ forces all the nets preceding n_s in $p(n_s)$ to go in one and only one position in the channel to avoid vertical constraint violations with n_s. This limits the degree of freedom of subsequent assignments and is likely to produce vertical constraint violations. Similar considerations apply for $b(n_s)$. Thus, it is convenient to place n_s as far away as possible from the two bounds, i.e., in the center of the desired range.

However, this will give nets preceding and following n_s in the longest path the same degrees of freedom. It is better to give more degrees of freedom to the nets preceding n_s if they are more numerous than the ones following n_s. With these things in mind we define an "ideal" row in $P(n_s)$ by the following relation:

$$\frac{ideal(n_s) - t(n_s) - 1}{k - b(n_s) - ideal(n_s)} = \frac{t(n_s)}{b(n_s)}.$$

where k is the number of rows in the channel. Note that $t(n_s)$ and $b(n_s)$ give the bounds of $P(n_s)$ and $idedl(n_s)$ will always be between $t(n_s)$ and $b(n_s)$. If $t(n_s)$ is larger than $b(n_s)$, $ideal(n_s)$ will be closer to $b(n_s)$; and if $b(n_s)$ is larger than $t(n_s)$, $ideal(n_s)$ will be closer to $t(n_s)$. Especially, if $t(n_s)$ is zero, $ideal(n_s)$ is the top row; and if $b(n_s)$ is zero, $ideal(n_s)$ is the lowest row.

Note that $S_1(n_s)$ may be an empty set for either of two reasons: $S(n_s)$ and $P(n_s)$ are disjoint; or $P(n_s)$ is empty. Let us consider the latter case first. If $P(n_s)$ is empty, $\overline{P}_t(n_s)$ and $\overline{P}_b(n_s)$ overlap covering the entire channel. Any assignment of n_s will lead to at least one vertical constraint violation, so we should choose the assignment so that the violation(s) will be relatively easy to remove. Since it is impossible to predict where the violations will be, the only thing we can do to make them easy to remove is to assign n_s to a row where the violations will be "short", i.e. vertical segments connecting the top and the bottom pin on that column overlap for as few rows as possible. The best way to keep the violations short is to assign n_s to a row in the region where $\overline{P}_t(n_s)$ and $\overline{P}_b(n_s)$ overlap. We define

$$S_2(n_s) = S(n_s) \bigcap \overline{P}_t(n_s) \bigcap \overline{P}_b(n_s)$$

to be the set of "second choice" rows in which to place n_s. To choose among the nets in this set, we use the same criteria as before. First the row(s) which cause the minimum number of vertical constraint violations with respect to nets already placed are selected. To break further ties, we also define $ideal(n_s)$ to be the "best" row for n_s by a relation similar to that given above except t and b are exchanged throughout.

In cases where both $S_1(n_s)$ and $S_2(n_s)$ are empty, we must assign n_s to a row in $\bar{P}(n_s)$. We assign n_s to a row causing as few violations as possible with respect to nets already placed and as close to *ideal* (n_s) as possible, but we avoid the top and bottom rows unless they are the only choice. The reasons for excluding the top and bottom rows is that it is difficult for the maze routers to go around the vertical constraint violations if they appear in these extreme rows. The following two sets are taken into consideration in the assignment algorithm in sequence:

$$S_3(n_s) = S(n_s) \bigcap (\bar{P}(n_s) - \{s_t, s_b\})$$

$$S_4(n_s) = S(n_s) \bigcap \{s_t, s_b\}$$

where s_t and s_b are the top and bottom rows.

We can now give the algorithm used to determine the row, $\hat{r}$, to which n_s is assigned.

```
$R(n_s) = S_1(n_s)$;
if ($R(n_s) == \emptyset$) {
        $R(n_s) = S_2(n_s)$;
        if ($R(n_s) == \emptyset$) {
                $R(n_s) = S_3(n_s)$;
                if ($R(n_s) == \emptyset$) {
                        $R(n_s) = S_4(n_s)$;
                }
        }
}
( now we have $R = \{s_{r_1}, \ldots, s_{r_l}\}$ )
$\hat{r} = s_{r_1}$;
for ($i = 2; i \leqslant l; i++$) {
        if ($VCV(n_s, s_{r_i}) < VCV(n_s, \hat{r})$) {
                $\hat{r} = s_{r_i}$;
        }
        else if ($VCV(n_s, s_{r_i}) == VCV(n_s, \hat{r})$) {
                if ($abs(s_{r_i} - ideal(n_s)) < abs(\hat{r} - ideal(n_s))$) {
                        $\hat{r} = s_{r_i}$;
                }
        }
}
```

$VCV(n_s, r)$ is a function which returns the number of vertical constraint violations that will occur between net n_s and nets already assigned, if net n_s is assigned to row r.

The assignment algorithm finds the "best" position among the ones which are available for a given net n_s. Note that the sequence in which the nets are placed, affects future placements since the row used by net n_s cannot be used for the nets in N which have not been placed yet. We choose the "most difficult" nets first. Thus we consider first the nets for which $R(n)=S_4(n)$, then the ones for which $R(n)=S_3(n)$ and so on until all the nets have been considered. Among the nets in each of the sets we have identified, we choose the net such that the set $R(n)$ has minimum cardinality. If there is a tie, then the net for which the rows in $R(n)$ are the most distant from its ideal position is selected first.

2.4. Removing Vertical Constraint Violations

After the assignment has been decided, the horizontal segments for the nets are placed in the channel and the appropriate vertical segments are placed in all columns that do not have vertical constraint violations. Next, we enter Phase IV where the columns with vertical constraint violations are examined one at a time to search for a legal connection between the nets and their pins. We could use a general purpose maze routing algorithm to find a connection (if one exists) between the nets and their pins exploiting the "free" space of the channel. However, the cost of this step would be quite susbtantial and a large number of vias may be introduced. Instead, we observed that a simple search very often resolves all the vertical constraint violation introduced by the assignment phases. Maze1, maze2 and maze3 described below are the three strategies we follow. Note that maze1 does not introduce any additional vias, maze2 introduces at most two additional vias per violation, and maze3 introduces at most four additional vias per violation. If the maze routing algorithms are unsuccessful, the channel is made larger by adding a row.

2.4.1. Maze Routing

To explain the operations of these algorithms, we assume that column j has a vertical constraint violation. Let the net that connects to the top edge of the channel be n_t, and the net that connects to the bottom edge be n_b. Net n_t has been assigned to row s; net n_b has been assigned to row p. Row s is below row p. All other columns with unresolved vertical constraint violations are "off-limits" during the maze routing.

Maze1

The first attempt at routing a column with a vertical constraint violation uses one or both adjacent columns, $j + 1$ and $j - 1$, and short *horizontal* segments on the layer used predominately for vertical segments. Any capacitive coupling caused by the overlap of two layers will be small since the overlapping length is the distance between adjacent columns. Note that this is the only time that YACR2 deviates from the common assumption that all the horizontal segments are on one layer and all the vertical segments are on another layer.

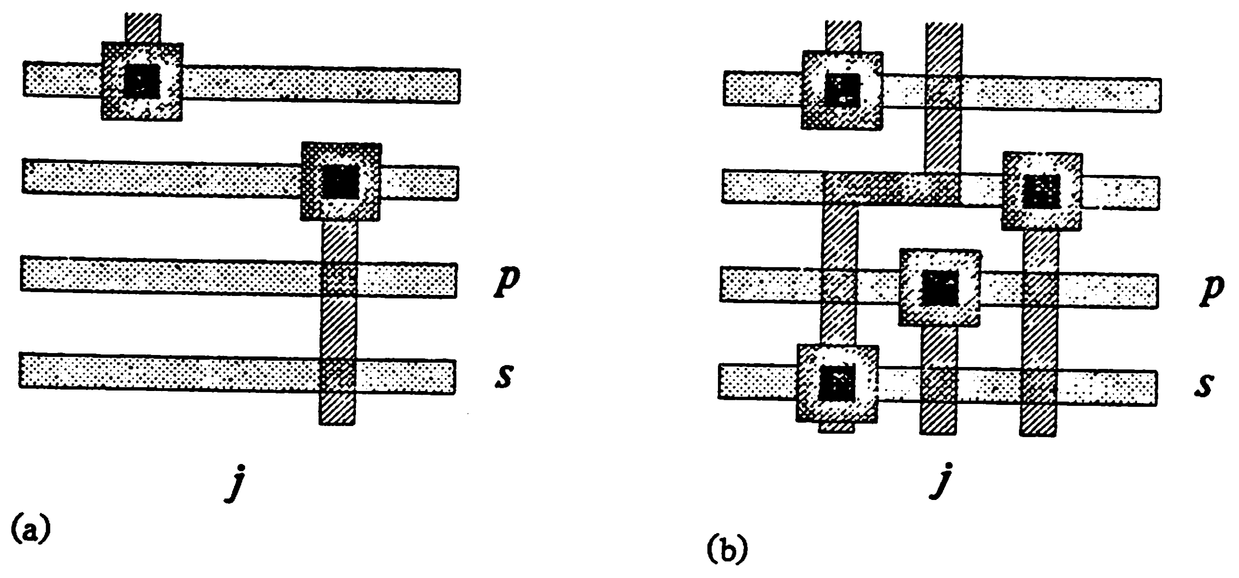
Maze1 checks for one of the following conditions:

(i) no vertical segments exist between $p - 1$ and s on column $j - 1$ or $j + 1$;

(ii) no vertical segments exist between p and $s + 1$ on column $j - 1$ or $j + 1$ (this is a reflection of case i);

(iii) no vertical segments exist between $p - 1$ and some h, between p and s, on column $j - 1$ and between $h - 1$ and $s + 1$ on column $j + 1$, or vice versa.

Figure maze1.a shows an example where case (i) holds true; no vertical segmenst exist between $p - 1$ and s on column $j - 1$. The completed routing is shown in figure maze1.b. Net n_b is routed with a vertical segment in column j connecting the pin at the bottom of the column to row p. Net n_t is routed with the following: a vertical segment in column j from the top of the column to row $p - 1$; a horizontal segment in row $p - 1$ from column $j - 1$ to column j; and a vertical segment in column $j - 1$ from



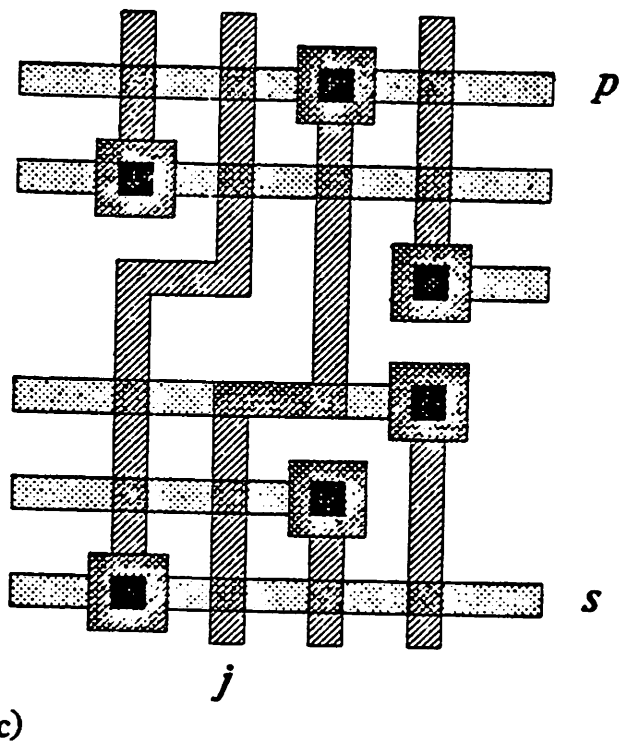


Fig. 1. Examples of maze1 routing. (a) Before maze1 routing. (b) After maze1 routing. (c) More complex example of maze1 routing.

row $p - 1$ to row s. Figure maze1.c shows a routed example where case (iii) was true.

Maze2

If a vertical dogleg of only a single column could not be used to route the pins in column j, maze2 searches for a dogleg that spans more than a single column.

Maze2 checks for one of the following:

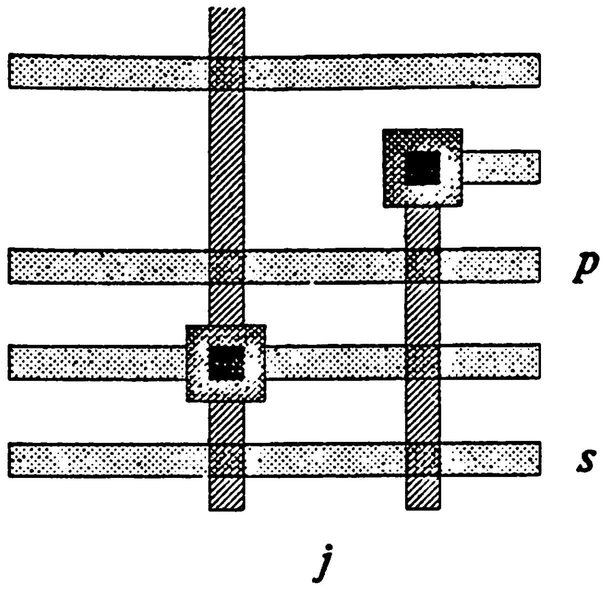
(i) a row, column pair (r, c) such that: (a) there are no horizontal segments in row r between columns j and c; (b) there are no vertical segments in c between r and s; (c) the horizontal segment of net n_t in row s either crosses column c or can be extended to c without causing a horizontal constraint violation; and (d) r is above p.

(ii) a row, column pair (r', c') such that: (a) there are no horizontal segments in row r' between columns j and c'; (b) there are no vertical segments in c' between r' and p; (c) the horizontal segment of net n_b in row p crosses c' or can be extended to c' without causing a horizontal constraint violation; and (d) r' is above s.

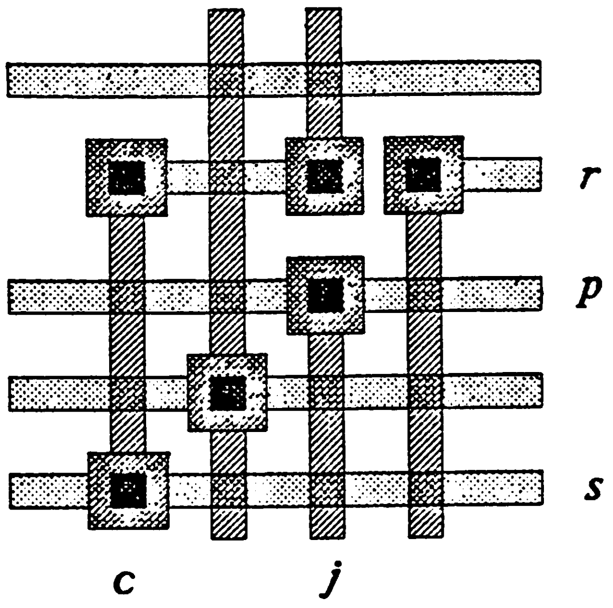
Figure maze2a shows an example where case (i) holds true; row r is the row above p and column c is the second column to the left of j. The completed routing is shown in figure maze2b. net n_b is routed with a vertical segment in column j connecting the pin at the bottom of the column to row p. Net n_t is routed with the following: a vertical segment in column j from the top of the channel to row r; a horizontal segment in row r from column j to column c; and a vertical segment in column c from row r to row s.

Maze3

If a vertical dogleg of only one net cannot be used to route the pins in column j, maze3 searches for a way to dogleg both nets.



(a)



(b)

Fig. 2. Example of maze2 routing. (a) Before maze2 routing. (b) After maze2 routing.

Maze3 checks for **both** of the following:

(i) a row, column pair (r, c) such that: (a) there are no horizontal segments in row r between columns j and c; (b) there are no vertical segments in c between r and s; (c) the horizontal segment of net n_t in row s either crosses column c or can be extended to c without causinga horizntal constraint violation; and (d) r is between p and s.

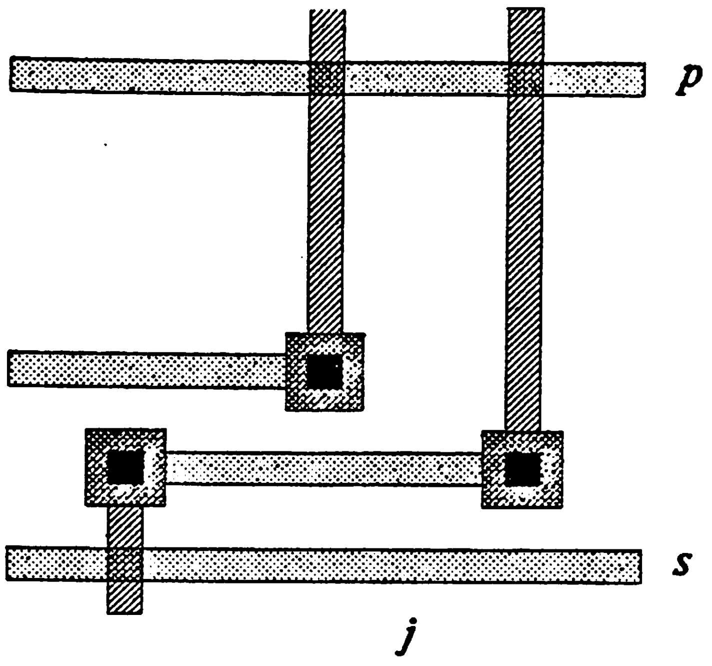
(ii) a row, column pair (r', c') such that: (a) there are no horizontal segments in row r' between columns j and c'; (b) there are no vertical segments in c' between r' and p; (c) the horizontal segment of net n_b in row p crosses c' or can be extended to c' without causing a horizontal constraint violation; (d) r' is between r and s; and (e) $r \neq r'$ and $c \neq c'$.

Figures maze3a and maze3b show an example before and after maze3 routing.

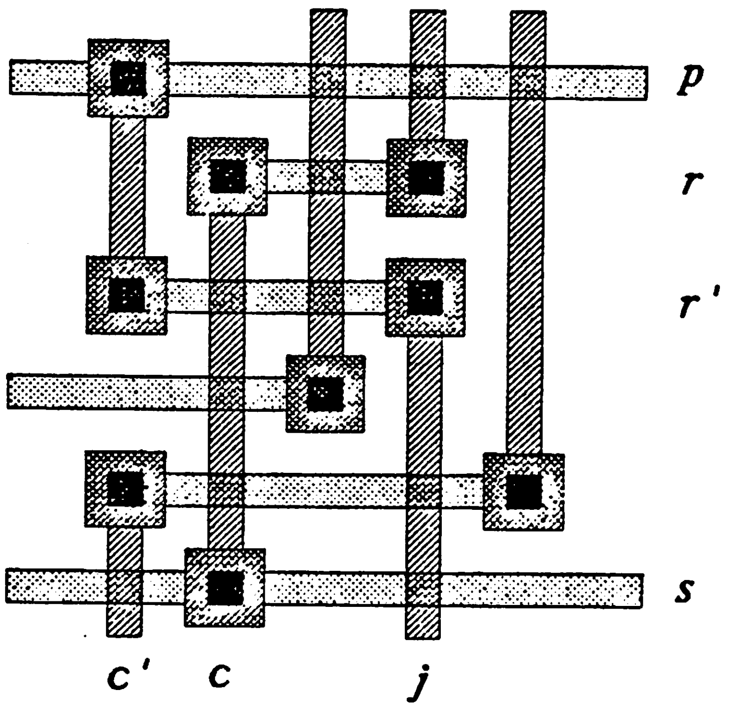
2.4.2. Adding a Row

If the above maze routing algorithms are not sufficient to complete the route, the channel must be made larger. Our basic strategy is to return to the beginning of the algorithm using a channel with one additional row, discarding all of the work so far accomplished. The extra row creates more "white space" that can be used for maze routing to remove vertical constraint violations. More important is that the heuristics will assign nets to rows differently when the number of rows is increased; there will hopefully be fewer vertical constraint violations to remove.

However, it would be nice if we could save some, or all, of the previous work. Thus, before discarding the partial routing completed after the application of maze1, maze2, and maze3, we attempt to add a row in such a way that the unresolved vertical constraint violations may be removed with the help of maze2. Note that since we want to add only one row, maze2 is the only candidate; because maze1 does not use horizontal segments on the horizontal layer, and maze3 requires two rows and we are only considering



(a)



(b)

Fig. 3. Example of maze3 routing. (a) Before maze3 routing. (b) After maze3 routing.

adding a single row.

If there are n rows in the channel, there are $n + 1$ locations in which a row may be added. Each of these locations is checked to see if it could serve as row q for maze2 routing of all the columns that have unresolved vertical constraint violations.

If we assumed that either the nets connecting to the top or the nets connecting to the bottom edge are doglegged, then the only locations we would need to check for new rows would be above the top row and below the bottom row. However, there may be cases where a new row inside the channel can be used for doglegging the net connecting to the top edge in some columns and the net connecting to the bottom edge in the remaining columns. It is therefore necessary to check all possible locations for an additional row.

If it is not possible to complete the route with maze2 and an additional row, we are forced to discard what we have done and begin again with a channel that has one more row than previously.

2.5. Termination of the Algorithm

As we show in chapter 5, the algorithms described in this section have been very effective on a number of test cases. However, to have confidence on the preformance of the algorithms, it is important to give a bound on the number of rows used.

It is quite difficult to obtain a sharp bound because of the maze routing step. This step depends on too many problem related parameters to be able to characterize it completely. Thus, a bound can be determined only by considering the assignment algorithm. In particular, we should show that $\tilde{r}$ rows are needed by the assignment algorithm to assign nets to tracks without any vertical constraint violations.

We have found this bound to be larger than the number of nets in the channel. This is due to the way in which we compute the ideal position of a net. Clearly this bound is too large to be accepted as the worst case behavior of the algorithm. For this reason, we decided to implement the following "escape" mechanism in the very unlikely case the algorithms use many rows. We employ a fast alternative assignment algorithm that will always place the nets without vertical constraint violations in a number of rows less than or equal to the number of nets.

Alternative Assignment Algorithm

Data: the set of nets to be routed, a channel with its number of rows equal to the number of nets, and the vertical constraint graph.

```
$p$ = length of longest path in $G$;
for ($i$ = 1; $i \leqslant p$; $i$++) {
        N = $\{n_j : t(n_j) == i\}$;
        while (N $\neq \emptyset$) {
                $n_s$ = select2 ($N$);
                assign2 ($n_s$);
                remove $n_s$ from N;
        }
}
delete any rows from the bottom of the channel that contain no horizontal segments;
```

select2 chooses any net in N. *assign2* (n_s) assigns n_s to the highest row in which it will have no horizontal or vertical constraint violations with respect to nets previously assigned.

Note again that this algorithm is extremely fast and crude. It is used only to provide robustness to the basic algorithm from an abstract point of view. In fact, it has never been actually used by YACR2 in hundreds of channels we tested, where YACR2 has used at most two rows above density.

2.6. Channels with Cyclic Vertical Constraints

When a channel has a cyclic Vertical Constraint Graph then vertical constraint violations become unavoidable for channel routers which do not use doglegs. Even for the ones which use doglegs, it may be impossible to route a channel without vertical constraint violations. Indeed, the problem may not be solved unless additional columns are added to the channel. An example of this situation is the channel described by $T = (1, 2)$, $B = (2, 1)$.

In this section, we present an algorithm that can always complete the routing of a cyclic channel possibly adding a few columns at the edges of the channel. The basic algorithm is essentially the same as in the case of acyclic G.

The only step of the assignment algorithm that explicitly depends on the vertical constraint graph being acyclic is the computation of $p(n_s)$. The longest path through a node is computed by leveling G; if G has cycles, it cannot be leveled. Because the assignment based on leveling G is very effective, we remove edges from G until G becomes acyclic.

2.7. Breaking Cycles

The leveling algorithm stops when it cannot assign a level to a set of nodes of G. At this point, the cycle detection algorithm of [10] is used to determine which edges of G form cycles. Then, an edge of each cycle is removed to break all the cycles.

The choice of the edges to remove is critical for the quality of the final routing. Since the edge is removed from G, the corresponding vertical constraint will be ignored by the assignment algorithm. The maze routers are the only tools which can remove the vertical constraint violation due to this vertical constraint. Therefore, we try to eliminate the edges whose corresponding vertical constraints are related to columns of the channel with more white space around them. Let E_c be the set of edges forming a cycle in G. Let Y be the set of columns of the channel where the vertical constraints corresponding to E_c are generated, then we rank the edges according to how easy is to route the corresponding vertical constraint. Ease of routing is evaluated by looking at the columns adjacent to the columns in Y. In particular, the following criteria are used where the easiest to route come first.

both adjacent columns to $y \in Y$ are empty (contain no pins);

one is empty, the other is half-empty (contains one pin);

one is empty, the other is full (contains two pins);

both are half-empty;

one is half-empty, one is full;

both are full.

The graph, G, is made acyclic with the following algorithm:

```
$l = \emptyset$;
while ($G$ has a cycle) {
        $E_c$ = the set of edges that form a cycle in $G$;
        $e$ = edge in $E_c$ corresponding to easiest to route constraint;
        add $e$ to $l$;
        delete $e$ from $G$;
}

while ($l \neq \emptyset$) {
        $e_l$ = edge in $l$ corresponding to the most difficult to route constraint;
        if ($e_l$ does not close a cycle in $G$) {
                reinsert $e_l$ in $G$;
        }
        remove $e_l$ from $l$;
}
```

Note that no attempt is made to find the minimum set of edges that removes all the cycles of G. This problem, also called the minimum feedback edge set, belongs to the class of NP-complete problems and hence very difficult to solve. In addition, we are most interested in constraints which are easy to route. However, the second part of the algorithm has been inserted to make sure that the set of edges removed from G is a *minimal* set, i.e., the removal of no proper subset of it can make G acyclic. The redundant edges are added back to G, giving priority to the most difficult to route.

2.8. Adding Columns

If the YACR2 algorithms presented in Section 2.4, have failed to remove all the vertical constraint violations, and the only ones left to route are a subset of those constraints corresponding to the edges removed to break the cycles, there is little hope that repeating the algorithm with one more row will route the channel. Thus, we propose an algorithm which introduces a number of columns at the sides of the channel to remove the vertical constraint violations corresponding to the removed edges.

Each vertical constraint violation is removed with routing similar to maze2. Consider j, n_b, p, n_t and s as in Section 2.4.1. We first identify the net such that one endpoint is the closest to an edge of the channel. Let n_t be such a net and let the right edge be the closest edge of the channel. Then, we add a column, k, to the right of the right edge of the channel, and possibly two rows r_1 above p and r_2 which is adjacent, either above or below, row s.

The following segments are used to eliminate the vertical constraint violation in column j.

(i) (n_b), a vertical segment in column j from the bottom of the channel to row p;

(ii) (n_t), a vertical segment in column j from the top of the channel to a row r_1;

(iii) (n_t), a horizontal segment in row r_1 from column j to the added column, k;

(iv) (n_t), a vertical segment in column k from row r_1 to a row r_2;

(v) (n_t), a horizontal segment in row r_2 from column k to column l, l contains the endpoint of n_t closest to k;

(vi) (n_t), a vertical segment on the horizontal layer in column l from row r_2 to row s.

Note that if rows r_1 and r_2 are new rows that are added just for this routing, the only empty space needed is the vertical layer in column j. The availability of this column is guaranteed by the fact that no other maze routing is allowed to use vertical segments in columns with unrouted violations. If there are rows that have no segments between column j and the edge of the channel, they may be used to save the addition of r_1 and/or r_2.

We can easily compute an upper bound on the number of additional columns and rows to remove i vertical constraint violations: i columns and $2i$ rows. Note, however, that this is a pessimistic bound and that we never needed so many rows and columns to complete the routing.

The construction described above with the remarks of Section 2.5, guarantee that the algorithms in YACR2 can route any channel, with or without cycles.

CHAPTER 3

IMPLEMENTATION OF YACR2

3.1. Data Structures

The two primary data structures used throughout YACR are: the vertical constraint graph, containing all the information about the nets being routed; and the channel, containing all the information about how the nets have been routed. Two other structures that need to be described are used to temporarily store possible routes made up of rectangles and paths found by maze2 and maze3. Other structures include a variety of stacks and linked lists which are very straightforward and will not be described in any detail.

3.1.1. Vertical Constraint Graph

Each node in the graph represents a net with the following structure:

```
typedef struct a_net
{
   int name;
   int level_from_top, level_from_bottom;
   int top_preferred, bottom_preferred;
   int pin_placement;
   int row;
   PIN_LIST leftmost, rightmost;
   NET_LIST_M parents, children;
   int available;
} NET, *NETPTR;
```

name

All of the net structures that will be used for a channel are allocated at the same time as an array. The *name* field is just the index for the structure in the array of nets.

level_from_top and level_from_bottom

level_from_top and *level_from_bottom* are the number of nets in the longest path from the current net to the top and bottom, respectively, of the vertical constraint graph. A net is at the top if its *parents* list is empty; it is at the bottom if its *children* list is empty. Nets at the top of the graph have *level_from_top* of one; nets at the bottom, have *level_from_bottom* of one.

top_preferred and bottom_preferred

top_preferred and *bottom_preferred* fields are numbers of the top and bottom rows in the preferred range of a net. The preferred range is described in the section on algorithm.

pin_placement

The field *pin_placement* is used to signify if the majority of the pins for a net are on the top edge of the channel or the bottom (or neither). This is set in routine store_pin. *pin_placement* is used only for nets which have *level_from_top* and *level_from_bottom* both equal to one, then the net is assigned to a row as close as possible to the edge of the channel with the most of its pins.

row

The number of the row that the net has been assigned to is stored in *row*. If the net has not yet been assigned to a row, *row* is zero. If the algorithm has to start over, *row* is reset to zero for all the nets.

leftmost and rightmost

The fields *leftmost* and *rightmost* are pointers to the ends of a doubly-linked list of pins that a net is to connect. Each element in the list contains the column number of the pin, the edge on which the pin appears, a pointer to the next pin on the left, and a pointer to the next pin on the right. The following structure is used for each element in the list:

```
typedef struct doubly_linked_list_of_columns
{
   int col;
   int edge;
   struct doubly_linked_list_of_columns
         *right_list, *left_list;
} PIN_LIST_ELEMENT, *PIN_LIST;
```

parents and children

The *parents* and *children* fields contain the edges that tie the nets together to form the vertical constraint graph. An edge is represented by two elements, one in the *parents* list of one net, and the other in the *children* list of the second net. Both pieces of the same edge share the same physical *col_list*; there is only one list of columns that is pointed to by each element of the edge.

The *parents* field is the head of a linked list of pointers to nets that must be placed above the given net to avoid vertical constraint violations with the given net. The *children* field is a linked list of pointers to nets that must be placed below the given net. The order of the elements in the lists is not important. Both lists are built by routine build_graph. Each element in the lists is an element of the following structure:

```
typedef struct markable_linked_list_of_nets
{
   struct markable_linked_list_of_nets        *next;
   struct a_net *net;
   int marked;
   INT_LIST col_list;
   int break_value;
} NET_LIST_ELEMENT_M, *NET_LIST_M;
```

If, for example, the element is in the *parents* list of net 12, the field *net* points to some net that must be above net 12. If, on the other hand, the element is in the *children* list, then the net pointed to by *net* must be below the first net.

marked is a flag used while leveling the graph to signify that a net has been visited by a certain parent or child.

col_list points to the head of a list of integers representing all the columns that require the edge to be in the graph. The list of all columns is kept so that when edges are removed to break cycles in the graph, we will be able to estimate the ease of removing the resulting vertical constraint violations. The measure of ease of routing is stored in the field *break_value.*

available

The *available* field is a flag used by routine find_cycle to help detect cycles in the vertical constraint graph.

3.1.2. The Channel

The channel is represented by the following structure:

```
typedef struct a_channel
{
   int **vert_layer, **horiz_layer;
   int left_column;
} CHANNEL, *CHANNELPTR;
```

The fields *vert_layer* and *horiz_layer* are two dimensional arrays representing the poly and metal layers of routing in the channel. Each row in the arrays is a track in the channel, with an extra row on top and bottom. Each column in the arrays is a column in the channel, with an extra column on the left and right edges. The upper-leftmost element in the channel has index [1] [1]. The number stored in each element of the arrays is the index (in *net_array*) of the net that is routed using that part of the channel; a zero indicates that no net currently uses that part of the channel. All the elements in the added top and bottom rows are "-1". This forms a border that cannot be used for routing, but serves to signify the edge of the channel to the maze routing routines.

The field *left_column* is used to indicate how many columns have been added to the left edge of the channel. Initially it is zero. This information is used to make sure the columns on the output are numbered so that column "1" is the leftmost column specified in the input.

3.1.3. Rectangles and Paths

During maze routing, it is often desirable or necessary to store several potential maze routes. One segment of a route is called a *rectangle.* All the rectangles needed for the maze routing of one net in a column are linked together in a list called a *path.* All the paths that could be used for maze routing one net in a column are linked together in a list called a *path_list.* A path list is ordered so that the paths with shortest total rectangle length are at the begining of the list.

The following structure is used to represent a rectangle:

```
typedef struct linked_list_of_rectangles
{
   struct linked_list_of_rectangles *next;
   int layer;
   int orientation;
   int num1, num2;
   int num3;
} PATH, *PATHPTR;
```

The field *layer* indicates whether the rectangle is on the *horiz_layer* or *vert_layer* of the channel.

If *orientation* is set to HORIZ, then *num3* is the row number of the rectangle, *num1* is the number of the column containing the left end of the rectangle, and *num2* is the column with the right end of the rectangle. If *orientation* is set to VERT, then *num3* is the number of the column containing the rectangle, *num1* is the number of the row at the top end of the rectangle, and *num2* is the row at the bottom end of the rectangle.

The field *next* is used to link several rectangles together to form a path. The routine insert_path is used to allocate a new rectangle and link it into an existing path. insert_path also make sure that *num1* is less than *num2.*

The following structure is used to save a number of paths for routing the same net in the same column:

```
typedef struct linked_list_of_paths
{
   struct linked_list_of_paths *next;
   PATHPTR path;
   int cost;
} PATH_LIST_ELEMENT, *PATH_LIST;
```

The field *path* is a pointer to a single path. *cost* is the sum of *num2* - *num1* for all the rectangles in the path. *next* is a pointer to the next element in the linked list. The function *merge_path_lists* calculates the *cost* of a path and inserts into the proper location in a given path list.

3.2. Code

Chapter 2 describes the important steps in the algorithm of YACR covering the actual implementation as little as possible. This section describes the details of implementation including more detail on algorithms which are not necessary to understand the overall operation of YACR2, but should be understood by anyone doing further work on the program.

Outline of Routine main( )

```
set global variables;
process command line;
read input file; /* allocates net data structures */
build and level vertical constraint graph; /* breaks cycles */
find density of each column;
choose starting column; /* if not given */
done = NO;
num_rows = maximum density - 1;
while (done == NO)
{
   num_rows++;
   Vcv_cols = NULL;
   allocate channel data structure;
   allocate and initialize cost matrix;
   Phase I;
   Phase II;
   Phase III;
   remove adjacent parallel vertical lines;
   Phase IV; /* sets done if routing is completed */
}
clean channel;
```

```
maximize metal; /* if requested */
print output;
verify output;
```

Many of the above "statements" are described in more detail below.

Set Global Variables

Two lists of integers, *Vcv_cols* and *Cycle_cols*, are set to NULL. *Vcv_cols* is a list of columns that contain vertical constraint violations. Columns are added to this list in routine place_one_net and are removed when they are routed by the maze routines. *Cycle_cols* is a list of columns for which there is no edge in the vertical constraint graph. Columns are added to this list in routine break_cycle.

Process Command Line

Global flags are set to either their default value or the value specified on the command line. If the starting column was specified, it is stored so that a starting column will not be calculated. The input and output files are opened and pointed to by global variables.

Read Input File

Aside from the obvious reading of the input file, this routine calls allocate_nets to allocate the array of net structures and then sets all the fields that relate to the pins for each net.

store_pin is called for each terminal in the channel. A net's *pin_placement* is incremented for each terminal on the top edge of the channel and decremented for each terminal on the bottom edge. An element for each terminal is inserted into the list of pins for its net. The doubly linked list is ordered from leftmost pin to rightmost pin, with *leftmost* pointing to the leftmost pin in the list, and *rightmost* pointing to the rightmost pin in the list.

The arrays *top, bottom, left,* and *right* are constructed by the input routine as the four edge lists are read from the input file. Each element of these arrays is a pointer to the structure for the net that is to connect to the specific column, e.g. *top[i]* is a pointer to the net that connects to the top edge of the channel in column *i*. There is a net structure with *name* = 0 that is used to represents locations where no net is to be connected.

This routine is also responsible for setting the global variable *place_relative* to indicate if the order of nets on either the left or right edge has been specified.

Build and Level Vertical Constraint Graph

For each column in the channel, make_graph calls insert_edge to modify the vertical constraint graph. If there are not two different nets in the column, insert_edge does nothing. If there already exists an edge in the proper direction between the two nets, then the *col_list* field of the edge is updated. If the desired edge does not exist, it is created by allocating the two elements and inserting them at the beginning of the *parents* and *children* lists.

The algorithm for calculating *level_from_top* for all nets is as follows:

```
set level_from_top to 1 for all nets;
push all nets with no parents;
while (not all nets leveled)
{
    if (stack is empty)
    {
        break all cycles in the graph;
        push all unleveled nets with no unmarked parents;
    }
    while (stack is not empty)
    {
        pop top net off stack and call it working_net;
        foreach child of working net
        {
            if (child's level_from_top <= working_net's level_from_top)
            {
                child's level_from_top = working_net's level_from_top + 1;
            }
            in child's parent list, mark the edge to working_net;
            if (all of child's parents have been marked)
```

```
                {
                    push child on stack;
                }
            }
        }
    }
```

Calculating *level_from_bottom* is slightly simpler. The *level_from_top* of all nets is calculated before the *level_from_bottom*, so the graph is acyclic when the *level_from_bottom* is calculated. There is no need to check for cycles in the graph.

Find Density of Each Column

Allocates an integer array, *density* with one element for each column in the channel. The array values are initially set to zero. The density is calculated by looping through all the nets, incrementing the value in *density* for each column that the net crosses.

Choose Starting Column

If the pins on either the left or right edge have a specified order, the starting column will be either column 1 or the rightmost column. If the left and right pins have no specified order, the starting column specified on the command line is used. If the starting column is not specified by either method, then it is chosen to be the column with maximum density whose nets cross the most other columns of maximum density.

It is sometimes the case that several adjacent columns will be of maximum density and contain the same nets. These columns are considered equivalent and treated as a single column.

Allocate Channel Data Structure

The routine that allocates the channel structure, allocate_channel, also initializes all the elements in rows 1 through *num_rows* to be zero. As stated before, the rows above the top and below the bottom of the channel are dummy rows that are filled with -1's.

Phases I, II, and III

Phases I, II, and III are described in chapter 2. ***select*** and ***assign*** are described conceptually there, the details of efficient implementation are given here.

select and ***assign*** are implemented using a cost matrix M which has as many rows as the channel and one column for each net. An entry $M[r, n_s]$ represents the cost of assigning net n_s to row r. The larger the value, the less desirable the assignment. Note that the most restricted net chosen by ***select*** according to the heuristics described in Section 3.3, is the one with the greatest column sum in the cost matrix. We use four levels of cost when determining an entry in the matrix: LOW (1), MEDIUM (100), HIGH (10,000), and INFINITY (1,000,000).

$M[r, n_s]$ is set to INFINITY to indicate that assigning n_s to row r will cause a horizontal constraint violation, i.e. r is not in $S(n_s)$.

Rows in $P(n_s)$ are assigned a cost of MEDIUM $\times \mid \bar{P}(n_s) \mid$, where $\mid \; \mid$ denotes the cardinality of a set.

If $P(n_s)$ is empty, then the rows in $\bar{P}_t(n_s) \bigcap \bar{P}_b(n_s)$ are assigned a cost of MEDIUM $\times (\mid \bar{P}(n_s) \mid - \mid \bar{P}_t(n_s) \bigcap \bar{P}_b(n_s) \mid$.

Rows outside the preferred region are assigned a cost of HIGH.

Finally, each row r with M [r, n_s] < INFINITY is incremented by LOW times the distance from *ideal* (n_s) to r.

Once the cost matrix is constructed, ***select*** (N) simply returns the $n_s \in$ N which has the greatest column sum in M. The algorithm for ***assign*** becomes:

```
$R(n_s) = \{s_i : M[s_i, n_s] <$ HIGH$\}$;
if ($R(n_s) == \emptyset$) {
        $R(n_s) = \{s_i : s_i \neq s_t$ and $s_i \neq s_b$ and $M[s_i, n_s] <$ INFINITY$\}$;
        if ($R(n_s) == \emptyset$) {
                $R(n_s) = \{s_i : M[s_i, n_s] <$ INFINITY$\}$;
        }
}
```

```
( now we have $R = \{s_{r_1}, \ldots, s_{r_l}\}$ )
$\hat{r} = s_{r_1}$;
for ($i = 2; i \leqslant l; i$++) {
        if ($VCV(n_s, s_{r_i}) < VCV(n_s, \hat{r})$) {
                $\hat{r} = s_{r_i}$;
        }
        else if ($VCV(n_s, s_{r_i}) == VCV(n_s, \hat{r})$) {
                if ($M[s_{r_i}, n_s] < M[\hat{r}, n_s]$) {
                        $\hat{r} = s_{r_i}$;
                }
        }
}
```

Note that the sets $S_1(n_s)$ and $S_2(n_s)$ are combined because one of the two is always empty.

Allocate and Initialize Cost Matrix

The cost matrix has a column for every net, and a row for each row in the channel (plus a row 0 that is not used).

The only reason the cost of assigning a net to a given row may change as other nets are assigned, is that a row may become obstructed so that its cost goes to INFINITY. Therefore, the basic cost of assigning each net to each row is calculated only once. Then, each time a net is assigned to a row, the cost for assigning any net that would overlap it in that row is set to INFINITY.

Remove Adjacent Parallel Vertical Lines

Soon after YACR was interfaced to HAWK and SQUID, it became apparent that adjacent parallel lines of the same net on the same layer were both aesthetically unpleasing and wasteful. Because the translator from YACR output to SQUID had to be able to make rectangles oriented both horizontally and vertically on each layer, every pair of adjacent parallel lines looked like a ladder. It was obvious that much of the material making up the ladder could be removed while still maintaining proper connectivity. This removal is done before any of the maze routing so that there will be more room for maze routing.

The procedure remove_parallel searches for a pair of adjacent columns that have the same net connecting to the same edge. If both have been connected without vertical constraint violations, then all but the edge-most piece of poly for one is removed.

Clean Channel

There are several ways in which the routing algorithms can generate segments of metal that are useless. For example, any piece of metal exactly two grid units long can be removed, because either it connects two pieces of poly (which would be connected without the metal) or it doesn't connect two pieces of poly (and hence serves no purpose). Another case occurs when maze2 routes the leftmost pin of a net with a dogleg to the right, then the primary horizontal segment of the net extends farther to the left than is necessary and can be shortened. (Maze2 does not shorten the segment because it may be needed to remove another vertical constraint violation for the same net.)

Maximize Metal

The metal maximization routine will potentially change a lot of poly to metal, but may also leave some that could be changed but is not. At each pin on the top and bottom edge, the algorithm searches toward the opposite edge untill it reaches the row where the poly connects to the net's horizontal segment (if it indeed does). The it steps backward one grid unit at a time, changing the poly at a grid point to metal if the vertical segment on the metal layer can be extended without overlapping the metal from another net. If the same net connects to both the top and bottom edges in a column, the piece of poly connecting to the row of metal can only be removed if the metal can be extended towards both pins. This method does not add any contacts and it does not attempt to maximize the poly used to dogleg in maze2 or maze3.

Print Output

Originally a separate fprintf statement was used for each number in each layer of the channel. Profiling showed that, on some examples, printing the output required about 50% of the entire execution time. Now, YACR does its own translation from integer to character representation, and uses putc to output the numbers one character at a time. This more complex method of output requires "only" about 25% of the total execution time.

Verify Output The verification routine was originally used as a debugging tool, but has been left in to ensure that if there are any more bugs, they will be detected and the user will be warned. There are three reasons for verifying the output after printing it instead of before: first, detecting an error is not the same as knowing how to correct the error; second, if correct output cannot be obtained, partially correct output is better than nothing; and third, the algorithm used to verify the route also destroys it.

The following recursive procedure is used to verify a single net.

```
verify (row, col, layer)
{
    layer[row] [col] = 0;
    if (there is a pin at this location)
    {
        delete this pin from the net's list of pins;
    }
    if (current net exists at layer[row - 1] [col])
    {
        verify (row - 1, col, layer);
    }
    if (current net exists at layer[row + 1] [col])
    {
        verify (row + 1, col, layer);
    }
    if (current net exists at layer[row] [col - 1])
    {
        verify (row, col - 1, layer);
    }
    if (current net exists at layer[row] [col + 1])
    {
        verify (row, col + 1, layer);
    }
    if (current net exists at other_layer[row] [col])
    {
        verify (row, col, other_layer);
    }
    return;
}
```

To verify that all the pins for a given net are connected, the above routine is called with the location of the leftmost terminal of the net. After the routine returns, all of the net connected to the leftmost terminal is erased from the channel. More importantly, the only terminals in the net's list of terminals (pointed to by *leftmost* and *rightmost* fields of the net's structure) are terminals NOT connected to the leftmost terminal of the net. Therefore, if the net's *leftmost* and *rightmost* are not NULL, then the net is not properly routed, and the terminals in the list are reported as being "not properly routed."

CHAPTER 4

INTERFACE TO HAWK

The channel routing algorithm presented has been interfaced with the HAWK graphics editor and SQUID data base system [11]. This work was done with Deirdre Ryan and Richard Rudell.

The router is called as a command from HAWK with the following actions taking place: the user is requested to type the cell name (output file) for the routing geometries; the user points to opposite corners of the rectangular channel to be routed; pin locations are extracted from the SQUID database; columns are defined in the channel; the input file for the symbolic router (described in chapters 2 and 3) is created; the symbolic router is run as a separate process; the output from the symbolic router is read and translated to geometries in the SQUID database; then the routed channel is displayed for the user.

This interface is divided into three portions; the symbolic router already decribed; a preprocessor that generates input for the symbolic router from the database; and a postprocessor that translates the symbolic routing to actual geometries.

4.1. Preprocessor

The preprocessor extracts the pin locations from SQUID and generates a symbolic grid (the lists T, B, L, and R) for the symbolic router. It makes sure that the columns are as wide as the contact-to-contact spacing for the particular technology used. No pin may straddle the dividing line between columns. If this would happen, one column is made wider than the minimum spacing. An important feature of the preprocessor is that it defines columns without pins whenever possible. This generates white space that may be used for maze routing.

Even though the channels have a specific minimum width, some of the pins may be closer together than this minimum. The postprocessor will place small jogs at the channel edge to center the vertical segments in their respective columns. However, if the pins are too densely packed, not all of them can be routed.

4.2. Postprocessor

There are two phases in the postprocessor, symbolic and physical. The symbolic phase performs metal maximization. The physical phase translates the symbolic routing into physical geometries that are design rule correct and ensures proper connections to the pins along the edges of the channel. The symbolic postprocessing is done by the same program that does the routing, the physical phase is done by a separate program.

Metal maximization is a one-for-one exchange of pieces of poly (vertical layer) for pieces of metal (horizontal layer). Only pieces of poly that are adjacent to metal are eligible to be changed, as this keeps the number of vias from increasing.

Before any actual geometries are generated, the spacing between rows must be defined. When the column spacing was defined there was no way of knowing the actual space needed between columns, as the routing had not yet been done. We wait to define row spacing until the routing is completed, then we place adjacent rows as close together as possible. If two adjacent rows both have a contact in the same column, the rows must have contact-to-contact spacing, otherwise metal-to-contact spacing is used.

The horizontal and vertical segment geometries are centered in their respective rows and columns. Along the top and bottom edges the vertical segments often need small jogs to connect to the pins.

CHAPTER 5

EXPERIMENTAL RESULTS

The algorithms have been coded in C and have been run on a DEC VAX 11/780 under both VMS and UNIX operating systems.

Table 1 gives the results for a number of channels. The CPU times are from runs on a VAX 11/780 under UNIX 4.2bsd. The channels labeled as YK.3a, YK.3b, and YK.3c are from [4]. DEUTSCH is the famous Deutsch's difficult example[2]. R1 through R5 are random channels generated by Rivest's program [9], the channels have been generated so that they are of uniform density with many short nets making them difficult to route. CYCLE is a modified version of DEUTSCH that contains seven cycles (with length ranging from four to eleven nets) in the vertical constraint graph. SOAR is the largest channel of the CMOS implementation of SOAR designed at Berkeley by Chris Marino. The Hughes channels have been provided by Dr. C.P. Hsu of Hughes Aircraft Company, Newport Beach. The AMI channels are the channels of an entire standard cell chip.

Table 2 shows the relative amounts of time YACR2 spends in various parts of the code. The profile information is cumulative over all the examples listed in table 1. Note that the maze routing which gives the algorithm its robustness only requires about 1% of the total run time.

Table 3a is a comparison between YACR2 and several other routers for the Deutsch difficult example. Notice that YACR2 used fewer vias and had smaller total net-length than any of the other routers. YACR2 required no multiple runs or setting of parameters for this result.

Table 3b is a comparison between YACR2 and the channel router implemented in the BBL system based on the algorithm of Yoshimura and Kuh. We have chosen this router for extended comparison because it was easily available to us, for this we thank the group of Prof. Kuh for the collaboration in carrying out the experiments. The runs of the BBL router on all the examples but the ones from Hughes have been performed by Dr. J.T. Li. The ones for the Hughes example were peformed by Dr. Hsu.

The initial column is indeed an important parameter for the performance of channel routers. As described in Section 3.2, we start from a column of maximum density. However there may be many columns with maximum density. We choose among all the columns of maximum density the one which shares more nets with other columns of maximum density. By doing so, we place the most critical nets first. Even though the choice of the initial column is important, we noted that YACR2 is very robust with respect to this choice. In fact, in all the examples we run, the difference between the number of tracks needed starting with the "optimal" column and any other (even one with non-maximum density) is at most two. The column tried even include the first and the last, generally the poorest choices. We believe that this characteristic of the router is due to the maze routers that can eliminate vertical constraint violations quite effectively.

Example	Columns	Nets	Density	Rows	Vias	Metal Length	Poly Length	CPU Time (sec.)	Notes
YK.3a	45	30	15	15	65	591	437	0.6	
YK.3b	62	47	17	17	101	911	738	1.1	B
YK.3c	103	54	18	18	152	1401	1047	1.5	B
DEUTSCH	169	72	19	19	287	2689	2331	2.8	
R1	139	77	20	21	201	2389	1754	3.2	B
R2	117	77	20	20	180	1947	1413	2.0	B
R3	123	78	16	17	199	1620	1478	1.9	B
R4	150	74	15	17	245	2085	1531	3.0	
R5	150	112	18	18	224	2017	1632	2.6	B
CYCLE	134	65	16	18	233	1888	1738	3.0	A,B
SOAR	433	158	50	50	402	13787	7394	14.4	A
HUGHES1	417	221	16	17	358	3827	3627	7.1	
HUGHES2	421	252	15	16	380	3670	4078	7.6	
HUGHES3	421	234	11	12	335	3199	2924	6.2	
HUGHES4	421	230	19	21	351	3740	4405	11.0	B
AMI5	616	122	18	18	436	6358	2056	8.5	A
AMI6	644	123	15	15	444	6027	2137	7.7	
AMI7	676	144	18	18	481	7899	2616	9.7	
AMI8	676	166	20	20	499	9104	3457	10.8	A
AMI9	676	184	25	25	498	11684	4106	12.6	A
AMI10	660	173	21	21	499	9541	3990	10.9	
AMI11	654	153	22	22	481	9190	3537	10.7	A
AMI12	662	157	22	22	491	8143	3621	11.3	A
AMI13	662	159	19	19	507	8328	3614	10.1	
AMI14	656	166	23	23	484	9186	3600	11.4	A
AMI15	636	175	21	21	471	8401	2991	10.9	A
AMI16	640	192	22	22	523	8494	3685	11.1	
AMI17	640	203	18	18	579	7968	3364	9.8	
AMI18	648	208	19	19	596	6795	3132	10.9	A
AMI19	654	211	20	20	608	7296	3327	11.1	A
AMI20	654	199	20	20	580	6997	3292	10.5	A
AMI21	640	176	16	16	534	6348	2779	8.9	
AMI22	640	156	18	18	476	7194	2668	9.3	A
AMI23	600	125	17	17	428	6412	2272	8.0	
AMI24	586	108	19	19	390	7613	2162	8.1	A

Table 1.

YACR2 routing of several channels.

Explanation of notes:

A - channel has cyclic constraints.

B - starting column was specified for these results, in each case, one more row was required for automatic selection of starting column.

Portion of Code	Per Cent of Run Time
Read Input and Build Net Data Structure	22
Allocate and Initialize Channel Structure	4
Build and Level Vertical Constraint Graph	5.5
Choose Starting Column	2.5
Select and Assign Nets to Rows	20
Maze Routing and Adding Extra Row	1
Cleanup and Metal Maximization	8
Verification of Completed Route	14
Output Routed Channel	23
Total	100

Table 2.

Profile of YACR2 Code.

Router	Tracks	Vias	Net-length
YACR	19	287	5020
Hamachi	20	412	5302
Burstein	19	354	5023
Yoshimura,Kuh	20	308	5075

Table 3a.

Comparison of various routers for the Deutsch difficult example.

Example	Columns	Nets	Density	YACR	BBL
YK.3a	45	30	15	15	15
YK.3b	62	47	17	17	17
YK.3c	103	54	18	18	18
R1	139	77	20	21	21
R2	117	77	20	20	20
R3	123	28	16	17	17
R4	150	74	15	17	20
Hughes1	417	221	16	17	24
Hughes2	421	252	15	16	20
Hughes3	421	234	11	12	15
Hughes4	421	230	19	21	22

Table 3b.

Comparison of YACR2 and the BBL channel router.

APPENDIX A

YACR User's Guide

The following pages give a complete description of the input/output formats and command line arguments of YACR2. Also included are suggestions for obtaining best results from YACR2.

YACR User's Guide

James Reed

1. Problem Formulation

The channels that YACR can route have the following characteristics:

i) a strictly rectangular region with no obstructions;

ii) two layers to be used for routing, referred to in this guide as "metal" and "poly" but the actual material is unimportant;

iii) fixed terminals on two opposite edges of the rectangular region, called the "top" and "bottom" edges;

iv) (optional) terminals on the other two edges, "left" and "right", of the rectangular region, exact position of these terminals is not specified, but order may be specified for one edge.

All of the routing is done within the boundaries of the rectangular region, but the exact boundaries cannot be specified by the user. The distance between the top and bottom edges is determined by YACR, but will be minimized. The length of the top and bottom edges are given by the user, but in rare cases may be entended by YACR to complete the route. The top and bottom edges will be extended only by adding empty columns to their left or right ends; the spacing between terminals on the top and bottom edges will never be altered by YACR.

YACR is guaranteed to make all of the connections specified in its input. The connections to the terminals on the top and bottom edges will be made with poly. The connections to the terminals on the left and right edges will be made with metal.

YACR routes on a symbolic manhattan grid. The metal layer is used for almost all routing segments that run parallel to the top and bottom edges; poly is used for almost all segments parallel to the left and right edges. This choice was made because the length of segments running from top to bottom is usually less than the length of segments running from left to right and metal has lower resistance than poly.

YACR is a *symbolic* channel router; it does not specify routing geometries in terms of actual dimensions or location on a chip, but shows their relative positions in the channel. The output is comparable to a stick diagram of a circuit layout.

An interface to the HAWK graphics editor and SQUID data base that translates the symbolic output of YACR into physical geometries has been developed. For information on how this interface is used see *User's Guide to Routing in HAWK.*

2. Yacr Input and Output

2.1. Yacr Input File Formats

Yacr can use either of two formats for input. The information that is supplied by both formats consists of: the number of nets to be routed; the number of columns in the channel; a list of pins that connect to the top edge of the channel; a list of pins that connect to the bottom edge of the channel; the number of pins that connect to the left edge of the channel; the list of pins on the left edge; the number of pins that connect to the right edge of the channel; and the list of pins on the right edge. First is a description of the default format, then the format called for by the "-H" command line argument to yacr.

The number of nets in the channel is specified with the following line:

```
nnet= #
```

The number must be separated from the equals sign by at least one white space character, a space, a tab, or a newline. No other spaces are allowed.

The number of columns in the channel is specified with the following line:

```
ncol= #
```

As with the number of nets, the only space allowed is between the equals sign and the number. The space is also required.

The list of pins on the top edge of the channel is begun with the keyword "top_list". The keyword is followed by a white space separated list of numbers, representing nets. The list must contain "ncol" integers. The numbers need not be consecutive. "0" is a special number that means "this location has no net connected to it."

The list of pins on the bottom edge of the channel is begun with the keyword "bottom_list". The restriction for the top list also apply to the bottom list.

The list of nets connecting to the left edge of the channel is begun with the keyword "left_list". The keyword is followed by a number telling how many nets there are in the left list. This is followed by the list of nets. If there are no nets connecting to the left edge, this category may be omitted. By default, the order in which the nets are listed is not meaningful. If a specific order is desired, "left_list" should be preceeded by the keyword "relative". The order in which the nets are listed is then the order (from top to bottom) in which they will appear in the final route. There is no way at this time to specify the exact placement of nets on the left edge of the channel.

The list of nets connecting to the right edge of the channel is begun with the keyword "right_list". This category is exactly like the left list category. An important restriction to note is that relative order can be specified for only the left list <u>or</u> the right list. If it is specified for both, it will be ignored for the second list.

The "-H" format is identical to the default format except that all keyword are omitted (except "relative") and all categories must be included in the following order:

number of nets

number of columns

top list

bottom list

left list

right list

The left list and the right list must be included, if there are no nets connecting to the edge of the channel, the list will be a single "0". The file will contain nothing but integers separated by white space, unless the keyword relative is included for the left or right list.

2.2. YACR Output File Formats

Just as there are two input formats, there are two output formats. One corresponds to the default input format and the other is used with the "-H" flag. The default format is described first, then the differences between it and the -H format follows.

The file begins with the following information: input file; number of nets; number of columns; number of nets in the left list; number of nets in the right list; a list of edges removed from the vertical constraint graph to make it acyclic (if any); a list of columns with maximum density; a list of columns that appear to be the best choices for starting column; and the actual starting column.

The next section of the output contains a block for each "major attempt" YACR2 makes at routing the channel. Each block has the number of rows used for the attempt, followed by a list of vertical constraint violations (VCVs) that occurred, followed by a list of how each VCV column was routed (if it was). If a row was added to complete the route without starting over, that is included at the end of the last block.

The next and most important section gives the routing of the channel. The horizontal (metal) layer is given first, followed by the vertical (poly) layer. Each layer is described by a matrix of integers. Each row in the matrix represents a column in the channel (this keeps the lines shorter so a printout is easier to read). There are rows to represent the left and right edges of the channel; the extra rows are used to indicate nets that connect to the edges. Each integer in the matrices represents the net that occupies a given space in the grid. "0" means the space is empty. Vias are not explicitly given, but implied at each location that the metal layer has the same net as the poly layer.

Finally, there is a summary that tells the number of vias used in the route, the total net length of all the nets, the longest net, and how much metal and poly were used to route the longest net.

The "-H" output format contains a minimum amount of information. It contains no keywords, just integers. The first two lines have the number of rows and columns, respectively, in the channel. The next row is the top row of the metal layer of the channel, listed from left to right. The entire metal layer is given followed by the poly layer. There are no extra columns representing the edges of the channel. The last line of the file has three integers, the first is the number of the net with the longest route, the amount of metal used to route it, and the amount of poly used to route it.

2.3. Examples

2.3.1. Example 1, Input (Default Format)

```
nnet= 2 ncol= 3
top_list
 2 0 1
bottom_list
 1 0 2
```

2.3.2. Example 1, Output (Default Format)

```
Input file: ex/2

num_nets= 2, num_cols= 3, num_left_nets= 0, num_right_nets= 0
Not closing cycle with edge from 1 to 2
Columns with maximum density: 1
Best choice(s) for starting column is (are): 1
initial_column = 1
num_rows = 2
VCV: nets  1  2, column   3
column 3 was not routed
Row 1 is being used to complete the route.
The final result is:

The horizontal layer (turned sideways):
 0  0  0  col =  0
 0  2  2  col =  1
 0  2  0  col =  2
 0  2  0  col =  3
 0  0  0  col =  4

The vertical layer (turned sideways):
 0  0  0   col =  0
 1  0  2   col =  1
 1  1  1   col =  2
 2  2  1   col =  3
 0  0  0   col =  4

There are 2 vias
The total net_length is 12
The longest net is 2, metal length = 4, poly length = 3
```

2.3.3.
Example 2, Input (Default Format)

```
nnet= 72
ncol= 169
top_list
 1  2  4  6  8 10 11 13  3  9 16  5 17 11  5 14 14  7 12 17 19  1 20 21
23 24  0 16 10  3 11 25  0 26 11 26 11  0 27 28 11  3  9 16 30 27  5 31
 1  5  1 20 32 23 24  0  9  1 20 29 23 24  0  3  8 30 38 28 19  6 40 27
35 41 42  6 19 34 43 30  8 31 43 39 46 36 46 47 48 31  0 24 23 45 20  1
51  0 40 39 40 39  0  8 30 50 54  0  0 55 49 19  6  0 47 42 47 42  0 53
58  6 19 49 50 30  8 60 62 59 54 55 54 56 63 55 65  0 66 68 66 68  0 60
68  0 46 44 46 44  0 69  0 55 58 55 58  0 64 71  0 72 63 72 63  0 57 62
54
bottom_list
 0  3  5  7  9  5 12 14 15  7 12 14  7  4 13  8  6 15 18 14  8  6 11 22
21  0 18 16 18 16  0  8  6 26 11  0 24 23 25 20  1 29  0 22  3 22  3  0
 0  9  2  9  2  0 32 23 33 19  6  8 30 27 34 35 36 37 39 31 39 35 38 31
 8 30 37 41 19  6 44 45  0 33 31 33 31  0 27 35 36 48 49 31 39 46 47 50
52 20 53 24  0 47 39  0 24 51 20 52 20 52 23  8 30 50 56  0  0 57 49 19
 6  6 19 49 59  0  0 61 50 30  8 55  0 24 64 20 52  0 67 68 63 55 24 52
20 69 24  0 46 62 63 68  0 24 65 20 52  0 70 60 62 54 63  0 24 71 20 52
67
relative right_list 6
68 55 63 70 67 61
```

3.

Running YACR

3.1.

Description

YACR2 is run with the following command:

```
yacr [options] [file1 [file2] ]
```

File1 contains the input, file2 gets the output. If file2 is omitted, the output goes to standard out. If no files are specified, input is read from standard input, and output goes to standard output.

The command line options described below can be specified in any order, but must come before the input and output file names.

-a Add columns at the ends of the channel if necessary to complete the route. By default columns are not added, but if the channel has cyclic constraints, additional columns may be necessary. If *yacr* cannot complete a route, it will give a message recommending that this flag be used.

-c n Begin routing the channel starting in column "n". If this is not specified, or if "n" is less than one or greater than the number of columns in the channel, *yacr* will chose a starting column. If the left_list or right_list is specified in "relative" order, the "-c" option will be ignored.

-d Sets the debug flag so that the channel will be printed (in the desired format) at the following times: when the current number of rows is found to be insufficient; when the route is completed, before metal maximization and cleanup; after metal maximization and cleanup. Useful only for debugging purposes.

-H Use an alternate input and output format. Also forces input to be read from standard input, and output to be written to standard output.

-m Do not perform metal maximization.

3.2. Diagnostics

The following error messages are all written to stderr.

"net n is not properly routed
bad pins are: col1/edge1 . . . "

The verification routine has discovered a problem with the route. The pin of net "n" in column "col1" on edge "edge1" was not connected to the leftmost pin of net "n". This message will only appear when a bug in the code manifests itself.

"The channel cannot be routed without additional columns,
please use the "-a" command line argument."

The channel has cyclic constraints that cannot be routed in the space provided. This has not happened on any of the hundreds of industrial channels routed by YACR to date.

"top_list already specified, first list used."

The input file tries to specify the top_list portion of the input more than once, all lists after the first are ignored. There are similar messages for the bottom_list, left_list, right_list, nnet, and ncol.

"net n has only 1 pin."

There is only one pin with number "n". Unfortunately YACR usually will not route any nets if this error occurs.

3.3. Suggestions for Best Results

The best suggestion that can be given here is: Let YACR make as many decisions as possible. There are two ways in which the user may make decisions for YACR, either by specifying that the left_list or right_list is "relative", or by specifying the starting column with the "-c" command line argument. One decision that the user should make for YACR is whether or not it should be allowed to add columns to the channel ends to complete the route.

Relative Order

If the left_list or right_list is given a specfic order YACR will almost always require more rows for routing that if the order was not given. This happens for two reasons: first, there are more restrictions that must be met by the final route; and second, in order to guarantee the restrictions are met, YACR must begin routing at either the left or right end of the channel (usually a bad place to start).

Use of -c

Most of the time that YACR is allowed to choose its starting column it will route a channel in density. Occasionally, a small number of extra rows will be required to complete the route. Since the number of rows needed for the route will vary by one or two depending on which column YACR started with, the user can sometimes get YACR to route a channel in density by forcing it to begin in a column it would not normally choose.

Note that if the pins on the left or right edge have a specified order, the user is already forcing YACR to start in the first or last column.

The YACR output file contains information that is useful in picking a starting column that might be better than the column YACR chose. Consider the following portion of a YACR output file (only the routing of the channel is omitted):

```
Input file: ex/3c
num_nets= 54, num_cols= 103, num_left_nets= 0, num_right_nets= 0
Columns with maximum density:  58 69
Best choice(s) for starting column is (are):  58 69
initial_column = 58
num_rows = 18
VCV: nets 38 45, column  71
VCV: nets 21 45, column  63
VCV: nets 21 24, column  75
column 75 was routed by maze1a
column 63 was not routed
column 71 was not routed
Row 1 is being used to complete the route.
```

In this example YACR was allowed to choose a starting column. It chose column 58, and ended up with two vertical constraint violations (VCVs) (in columns 63 and 71) that it could not remove. We could hope for better results by forcing YACR to start in any of columns 69, 63, or 71. We might choose 69 because YACR said it might be a good choice, or we might choose 63 or 71 in hopes that a difficult-to-remove VCV would be avoided. If we choose to force YACR to start routing in column 63 (with the command "yacr -c 63 inputfile outputfile") we get the following results.

```
Input file: ex/3c
num_nets= 54, num_cols= 103, num_left_nets= 0, num_right_nets= 0
Columns with maximum density:  58 59 69 70
initial_column = 63
num_rows = 18
VCV: nets 41 26, column  72
VCV: nets 49 35, column  90
column 72 was routed by maze2
column 90 was routed by maze2
```

With the forced starting column YACR was able to route the channel in density.

There are several interesting things should be pointed out about the two output files. Notice that the first one lists 58 and 69 as the columns with maximum density and the second lists colums 58, 59, 69, and 70. This is not a contradiction, in this example columns 58 and 59 are crossed by the same nets (the same is true of columns 69 and 70). If YACR began routing in either column 58 or 59, it would achieve exactly the same results, so it removes all but one of the redundant columns from consideration. When the user specifies a starting column, YACR does not waste time figuring out if adjacent columns of maximum density are identical.

The other interesting fact is that there were not only a different number of columns with vertical constraint violations, but the columns were totally different when YACR began routing in a different column.

Use of -a

There are some cases in which YACR will think it needs extra columns, but could actually complete the route by adding only rows. Since the times when YACR really needs extra rows are rare (it has not yet happend in a industrial example), you should only use the "-a" command line argument after YACR tells you it is necessary.

3.4. Reed's Rule of Routing

Routers do weird things.

When looking at the output of YACR (or any other router) it is always a good idea to keep in mind this variant of the widely known Murphy's law.

APPENDIX B

YACR man Pages

The following pages are the on-line descriptions of how to run YACR2 and the input and output formats.

NAME

yacr — Yet Another Channel Router

SYNOPSIS

yacr [options] [file1 [file2]]

DESCRIPTION

Yacr is a two-layer symbolic channel router. Its input is a list of pins that are to be connected on each edge of the channel. The output is in the form of two arrays, one for each layer of the route. Each non-zero entry in an array is the name of the net that occupies that space in the routed channel. For information on input and output formats, see YACR(5).

File1 contains the input, file2 gets the output. If file2 is omitted, the output goes to standard out. If no files are specified, input is read from standard input, and output goes to standard output.

The command line options described below can be specified in any order, but must come before the input and output file names.

-a Add columns at the ends of the channel if necessary to complete the route. By default columns are not added, but if the channel has cyclic constraints, additional columns may be necessary. If *yacr* cannot complete a route, it will give a message recommending that this flag be used.

-c n Begin routing the channel starting in column "n". If this is not specified, or if "n" is less than one or greater than the number of columns in the channel, *yacr* will chose a starting column.

-d Sets the debug flag so that the channel will be printed (in the desired format) at the following times: when the current number of rows is found to be insufficient; when the route is completed, before metal maximization and cleanup; after metal maximization and cleanup. Useful only for debugging purposes.

-H Use an alternate input and output format (see YACR(5)). Also forces input to be read from standard input, and output to be written to standard output.

-m Do not perform metal maximization.

SEE ALSO

YACR(5)

AUTHOR

James Reed

DIAGNOSTICS

The following error messages are all written to stderr.

"net n is not properly routed
  bad pins are: col1/edge1 ..."

The verification routine has discovered a problem with the route. The pin of net "n" in column "col1" on edge "edge1" was not connected to the leftmost pin of net "n". This message will only appear when a bug in the code manifests itself.

"The channel cannot be routed without additional columns,
please use the "-a" command line argument."

"top_list already specified, first list used."

The input file tries to specify the top_list portion of the input more than once, all lists after the first are ignored. There are similar messages for the bottom_list, left_list, right_list, nnet, and ncol.

Yacr Input File Formats

Yacr can use either of two formats for input. The information that is supplied by both formats consists of: the number of nets to be routed; the number of columns in the channel; a list of pins that connect to the top edge of the channel; a list of pins that connect to the bottom edge of the channel; the number of pins that connect to the left edge of the channel; the list of pins on the left edge; the number of pins that connect to the right edge of the channel; and the list of pins on the right edge. First is a description of the default format, then the format called for by the "-H" command line argument to yacr.

The number of nets in the channel is specified with the following line:

```
nnet= #
```

The number must be separated from the equals sign by at least one white space character, a space, a tab, or a newline. No other spaces are allowed.

The number of columns in the channel is specified with the following line:

```
ncol= #
```

As with the number of nets, the only space allowed is between the equals sign and the number. The space is also required.

The list of pins on the top edge of the channel is begun with the keyword "top_list". The keyword is followed by a white space separated list of numbers, representing nets. The list must contain "ncol" integers. The numbers need not be consecutive. "0" is a special number that means "this location has no net connected to it."

The list of pins on the bottom edge of the channel is begun with the keyword "bottom_list". The restriction for the top list also apply to the bottom list.

The list of nets connecting to the left edge of the channel is begun with the keyword "left_list". The keyword is followed by a number telling how many nets there are in the left list. This is followed by the list of nets. If there are no nets connecting to the left edge, this category may be omitted. By default, the order in which the nets are listed is not meaningful. If a specific order is desired, "left_list" should be preceeded by the keyword "relative". The order in which the nets are listed is then the order (from top to bottom) in which they will appear in the final route. There is no way to specify the exact placement of nets on the left edge of the channel.

The list of nets connecting to the right edge of the channel is begun with the keyword "right_list". This category is exactly list the left list category. An important restriction to note is that relative order can be specified for only the left list the right list. If it is specified for both, it will only be meaningful for the one that appears second in the input file.

The "-H" format is identical to the default format except that all keyword are omitted (except "relative") and all categories must be included in the following order:

```
number of nets
number of columns
top list
bottom list
left list
right list
```

The left list and the right list must be included, if there are no nets connecting to the edge of the channel, the list will be a single "0". The file will contain nothing but integers separated by white space, unless the keyword relative is included for the left or right list.

Yacr Output File Formats

Just as there are two input formats, there are two output formats. One corresponds to the default input format and the other is used with the "-H" flag. The default format is described first, then the differences between it and the -H format follows.

The file begins with the following information: input file; number of nets; number of columns; number of nets in the left list; number of nets in the right list; a list of edges removed from the vertical constraint graph to make it acyclic (if any); a list of columns with maximum density; a list of columns that appear to be the best choices for starting column; and the actual starting column.

The next section of the output contains a block for each "major attempt" yacr makes at routing the channel. Each block has the number of rows used for the attempt, followed by a list of vertical constraint violations (VCVs) that occurred, followed by a list of how each VCV column was routed (if it was). If a row was added to complete the route without starting over, that is included at the end of the last block.

The next and most important section gives the routing of the channel. The horizontal (metal) layer is given first, followed by the vertical (poly) layer. Each layer is described by a matrix of integers. Each row in the matrix represents a column in the channel (this keeps the lines shorter so a printout is easier to read). There are rows to represent the left and right edges of the channel; the extra rows are used to indicate nets that connect to the edges. Each integer in the matrices represents the net that occupies a given space in the grid. "0" means the space is empty. Vias are not explicitly given, but implied at each location that the metal layer has the same net as the poly layer.

Finally, there is a summary that tells the number of vias used in the route, the total net length of all the nets, the longest net, and how much metal and poly were used to route the longest net.

The "-H" output format contains a minimum amount of information. It contains no keywords, just integers. The first two lines have the number of rows and columns, respectively, in the channel. The next row is the top row of the metal layer of the channel, listed from left to right. The entire metal layer is given followed by the poly layer. There are no extra columns representing the edges of the channel. The last line of the file has three integers, the first is the number of the net with the longest route, the amount of metal used to route it, and the amount of poly used to route it.

Example 1, Input (Default Format)

```
nnet= 2 ncol= 3
top_list
 2 0 1
bottom_list
 1 0 2
```

Example 1, Output (Default Format)

```
Input file: ex/2

num_nets= 2, num_cols= 3, num_left_nets= 0, num_right_nets= 0
Not closing cycle with edge from 1 to 2
Columns with maximum density: 1
Best choice(s) for starting column is (are): 1
initial_column = 1
num_rows = 2
VCV: nets  1  2, column   3
column 3 was not routed
Row 1 is being used to complete the route.
The final result is:

The horizontal layer (turned sideways):
 0  0  0  col =  0
 0 2 2 col =  1
 0 2 0 col =  2
 0 2 0 col =  3
 0 0 0 col =  4

The vertical layer (turned sideways):
 0 0 0  col =  0
 1 0 2  col =  1
 1 1 1  col =  2
 2 2 1  col =  3
 0 0 0  col =  4

There are 2 vias
The total net_length is 12
The longest net is 2, metal length = 4, poly length = 3
```

Example 2, Input (Default Format)

```
nnet= 72
ncol= 169
top_list
  1  2  4  6  8 10 11 13  3  9 16  5 17 11  5 14 14  7 12 17 19  1 20 21
 23 24  0 16 10  3 11 25  0 26 11 26 11  0 27 28 11  3  9 16 30 27  5 31
  1  5  1 20 32 23 24  0  9  1 20 29 23 24  0  3  8 30 38 28 19  6 40 27
 35 41 42  6 19 34 43 30  8 31 43 39 46 36 46 47 48 31  0 24 23 45 20  1
 51  0 40 39 40 39  0  8 30 50 54  0  0 55 49 19  6  0 47 42 47 42  0 53
 58  6 19 49 50 30  8 60 62 59 54 55 54 56 63 55 65  0 66 68 66 68  0 60
 68  0 46 44 46 44  0 69  0 55 58 55 58  0 64 71  0 72 63 72 63  0 57 62
 54
bottom_list
  0  3  5  7  9  5 12 14 15  7 12 14  7  4 13  8  6 15 18 14  8  6 11 22
 21  0 18 16 18 16  0  8  6 26 11  0 24 23 25 20  1 29  0 22  3 22  3  0
  0  9  2  9  2  0 32 23 33 19  6  8 30 27 34 35 36 37 39 31 39 35 38 31
  8 30 37 41 19  6 44 45  0 33 31 33 31  0 27 35 36 48 49 31 39 46 47 50
 52 20 53 24  0 47 39  0 24 51 20 52 20 52 23  8 30 50 56  0  0 57 49 19
  6  6 19 49 59  0  0 61 50 30  8 55  0 24 64 20 52  0 67 68 63 55 24 52
 20 69 24  0 46 62 63 68  0 24 65 20 52  0 70 60 62 54 63  0 24 71 20 52
 67
relative right_list 6
68 55 63 70 67 61
```

SEE ALSO

YACR(1)

AUTHOR

James Reed

REFERENCES

[1] A. Hashimoto and J. Stevens, Wire routing by optimizing channel assignment, *Proc. 8th Design Automation Conf.*, pp. 214-224, 1971.

[2] D. Deutsch, A "dogleg" channel router, *Proc. 13th Design Automation Conf.*, pp. 425-433, 1976.

[3] G. Persky, D. Deutsch, and D.G. Schweikert, LTX - A minicomputer-based system for automatic LSI layout, *J. Des. Automat. Fault-Tolerant Comput.*, Vol. 1, n. 3, pp. 217-256, 1977.

[4] T. Yoshimura and E. S. Kuh, Efficient algorithms for channel routing, *IEEE Trans. CAD of ICs and Systems*, vol. CAD-1, n. 1, pp. 25-35, 1982.

[5] R.L. Rivest and C.M. Fiduccia, A "greedy" channel router, *Proc. 19th Design Automat. Conf.*, pp. 418-424, 1982.

[6] T. Kawamoto and Y. Kajitani, The minimum width routing of a 2-row 2-layer polycell layout, *Proc. 16th Design Automation Conf.*, pp. 290-296, 1979..

[7] M.L. Liu, An algorithm for two-layer channel routing with cyclic constraints, UCB/ERL M81/82, Nov. 1981.

[8] T.G. Szymanski, Dogleg Channel Routing is NP-Complete, *IEEE Trans. CAD of ICs and Systems*, vol. CAD-4, n. 1, pp. 31-41, 1985.

[9] R. Rivest, "Benchmark" channel-routing problems, private communication, 1982.

[10] E.M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 346-353.

[11] K. Keller, An Electronic Circuit CAD Framework, *PhD Thesis*, Department of Electricla Engineering and Computer Sciences, University of California, Berkeley, March 1984.

[12] C. Sechen and A. Sangiovanni-Vincentelli, The TimberWolf Placement and Routing Package, *Proc. 1984 Custom Int. Circuit Conf.*, May 1984.