A Detailed Router Based on Incremental Routing Modifications: Mighty

HYUNCHUL SHIN MEMBER, IEEE, AND ALBERTO SANGIOVANNI-VINCENTELLI, FELLOW, IEEE

Abstract—**For the macrocell design style and for routing problems in which the routing regions are irregular, two-dimensional routers are often necessary. In this paper, a new routing technique that can be applied for general two-layer detailed routing problems, including switchboxes, channels, and partially routed areas, is presented. The routing regions that can be handled are very general: the boundaries can be described by any rectilinear edges, the pins can be on or inside the boundaries of the region, and the obstructions can be of any shape and size.**

The technique is based on an algorithm that routes the nets in the routing region incrementally and intelligently, and allows modifications and rip-up of nets when an existing shortest path is "far" from optimal or when no path exists. The modification steps (also called weak modification) relocate some segments of nets already routed to find a shorter path or to make room for a blocked net. The rip-up and reroute steps (called strong modifiction) remove segments of nets already routed to make room for a blocked connection; these steps are invoked only if weak modification fails. The algorithm has been rigorously proven to complete in finite time and its complexity has been analyzed.

The algorithm has been implemented in the "C" programming language. Many test cases have been run, and on all the examples known in the literature the router has performed as well as or better than existing algorithms. In particular, Burstein's difficult switchbox example has been routed using one less column than the original data. In addition, the router has routed difficult channels such as Deutsch's in density and has performed better than or as well as YACR-II on all the channels available to us.

I. Introduction

CHANNEL ROUTERS have been most successful in performing the detailed routing task for a variety of design styles. However, for the general macrocell or building-block design style, the routing regions cannot be defined so that a channel router can route all of them unless some restrictive assumptions are made on the shape of the cells and on the placement technique used [1]. In our approach to macrocell placement and routing, we have to place and route cells whose bounding boxes are not rectangles and we do not wish to enforce slicing structures to capitalize on the power of simulated annealing algorithms. Hence, we are confronted with the problem of routing regions with irregular boundaries and nonconvex shapes which are delineated by polygons. This approach would not have been feasible if a powerful two-layer two-dimensional router were not available. This paper describes an algorithm and its implementation that allows a very efficient routing of any two-dimensional region with pins on or inside the boundaries of the region. Because of the interest of designers in modifying incrementally their design and in performing part of the routing manually, we have developed the router to take care of partially routed regions.

Several two-layer switchbox routers have been developed in the recent past [2]–[6], but the quality of the results or the running time needed by the routers has not been fully satisfactory. The most successful switchbox router, Weaver [6], is based on knowledge-based expert systems (KBES's) and is certainly very complicated, including many experts and several hundred rules to guide the routing. For this reason, running time is very long and the time needed to develop the program was also very long. In all cases, the routers tried to predict where congested regions were and which nets were difficult to connect and gave priorities based on this estimate. Of course, the estimate is only approximate and problems may develop anyway and force the router to use additional rows and/or columns. In addition, some of the interconnections may be unnecessarily far from being "optimal" with respect to their length and the number of vias.

Only a few rerouting algorithms have been reported, mostly for the routing of printed wiring boards. In 1974, Rubin [7] suggested an iterative process where all failed connections are routed allowing crossings with a substantial scoring penalty and then the connections with the greatest number of crossings are ripped up. Later, this approach was extended using penalty functions by Linsker [8]. In 1977, Agrawal and Breuer [9] suggested a backtracking algorithm which can possibly enumerate all the alternate connections for each net. However, complete backtracking to find an optimal routing is so complicated that it can be used only for small problems. In 1982, Dees and Karger [10] reviewed rip-up and rerouting techniques and suggested two types of modifications—rip-up and shove-aside. Shirakawa *et al.* [11] reported a rerouting scheme for single-layer printed wiring board with two-pin nets only. Recently, Suzuki *et al.* [12] reported that the Lee algorithm [13] is most useful and powerful when applied to interactive rip-up and reroute. However, no published results are available, to the best of our knowledge, on automatic general routing modification and rerouting for two-layer detailed routing with polynomial

Manuscript received December 14, 1986; revised June 1, 1987. This research was supported in part by DARPA, Hughes Semiconductor Division, AMI, Intel, and the MICRO Project of the State of California.

The authors are with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

IEEE Log Number 8716055.

worst-case complexity. Most of the previous rerouting approaches have one or more of the following weaknesses.

1) Rip-up and rerouting are the only modification routine, while relocating part of existing connections can frequently solve the problem [7], [8], [11], [12].
2) Modification is tried only after a completely blocked net appears. But there may be many "bad" quality connections at this time [10], [12].
3) Modification is interactive, and human intervention is necessary [12].
4) Rip-up techniques can be potentially uncontrollable when considering computational complexity [9], [10].
5) The focus is on the routing of two-pin nets on a single layer of printed wiring board [9]-[11].

We took a different approach. Our router is based on an incremental routing technique that favors nets with several pins widely scattered in the routing region. The novel part of the algorithm is the modification of connections already made for the nets in the routing region to allow blocked or "bad" quality connections to find better solutions. These modifications are of two kinds: a weak modification step pushes aside existing connections without removing them to find a shorter path or to make room for a blocked connection, and a strong modification step, invoked when weak modification fails, removes blocking connections. Thus, we have concentrated our attention on developing a router which is an "expert" in modifying and rerouting a partial solution if problems occur. Note that, in our opinion, the most noticeable difference between a human expert and existing routing tools is the ability of rerouting or modifying the existing connections.

The router described in this paper implements a general incremental routing modification strategy for rectilinear polygonal routing regions. This router has obtained excellent results on a number of test cases, outperforming not only existing two-dimensional routers but even channel routers when applied to "standard" channels. Hence, Mighty can be considered a general-purpose two-layer router, being able to route any irregular regions with floating as well as fixed pins on the boundaries of the region. Mighty works on symbolic grids and is an integral part of MOSAICO [14], a general macrocell floor planning, placement, and routing system where it is used to route L-shaped and staircase-shaped channels.

The paper is organized as follows. The basic definitions and problem formulation are described in Section II. Section III outlines the main ideas of our approach and the description of algorithms. The computational complexity of the algorithm is analyzed in Section IV. Section V contains variations of the basic algorithm used for channel routing and special cases. In Section VI, routing results of switchbox and channel examples are evaluated. In Section VII, the sensitivity of the algorithm to parameters is discussed. Finally, conclusions are given in Section VIII.

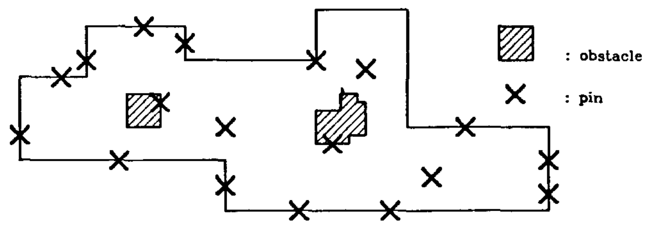


Fig. 1. Routing area with obstacles.

II. Definitions and Formulation of the Problem

The terms used throughout this paper are defined as follows.

channel or **routing area:** A rectilinear polygonal region between circuit blocks that can be used for interconnections (for example, see Fig. 1).

pin: A terminal of a cell facing the routing area, which is to be connected to a set of other terminals. All pins are on grid points because the router works on symbolic grids.

net: A set of pins to be connected and their associated connections.

component: A set of pins and wire segments of a net which have been interconnected. Each unconnected pin is a trivial component.

horizontal (vertical) segment: A piece of wire running horizontally (vertically) on a layer described by two end points. Nets are connected by wire segments.

track or **row:** The symbolic routing area in the horizontal direction.

column: The symbolic routing area in the vertical direction.

contact or **via:** An interconnection of two segments on two different layers, placed at the row and column location the segments have in common.

path: A set of vias and segments implementing the interconnection between two components of a net.

The routing area is a rectilinear polygon with two layers of interconnection as shown in Fig. 1. Pins may be inside or on the boundaries, and some of them may be floating on the boundaries. There may be obstacles of any shape and size in the routing region.

The problem we have to solve is: Connect all the components of each net in the routing region such that each available grid point on each layer is used by at most one net.

The primary objective is to complete the connections using minimum routing area. The secondary objective is to minimize the number of vias and the wire length of each net. Other factors, such as preference of one layer to the other or consideration of coupling with neighboring nets, can be included by appropriately setting cost parameters.

III. The Algorithm

The main algorithm is described in Section III-A. The detailed algorithms are described in the following sections. In the description of the algorithms, all the parameters begin with a capital letter.

A. The Basic Algorithm

The overall flow of the main algorithm is presented in this subsection.

The basic algorithm consists of four main parts: a path finder that searches for a shortest (minimum cost) path among components, a path conformer that implements a particular path proposed by the path finder, a weak modifier that relocates existing interconnections to find a shorter path when the existing one is "far" from optimal, and a strong modifier that removes some connections from the routing region to allow the completion of a blocked path.

As a preprocessing step, the algorithm extends all the pins on the boundaries of the region inside by one unit to avoid possible connections along the boundaries of the routing region. Then the path-finding phase begins. The nets are processed in the order they are entered. From each component (pin) of a net, we start a maze router search to find the minimum-cost path that interconnects any two components of the net. The cost used to direct the search is based on the model in which a layer is mainly used for horizontal interconnections and the other for vertical interconnections. For example, extension of a path in the vertical direction on the horizontal layer, while possible, is penalized. Changing a layer is also penalized to minimize the number of vias. As soon as a path connecting two components of the net is found the search is stopped. The path connecting the two components is recorded on an ordered list organized in increasing cost. Note that the path is not finalized yet.

When all the nets have been processed by the path finder, the path conformer takes over. The paths on the ordered list are popped and examined. If the path is feasible, i.e., if no other interconnection has occupied the same locations as the present path, the path is implemented. Otherwise, the path finder is invoked again and a feasible minimum-cost path connecting any two unconnected components of the net is sought. Note that when this path is looked for, the interconnections laid out already are taken into consideration. Because of the paths that have been implemented before, the new path of the net under examination may be far from the optimal solution. One of the main ideas of our algorithm is to make sure that poor intermediate solutions are not accepted. Hence, if the new path is found to be unacceptable, the modification phase is entered.

It is important to choose a good criterion to classify paths as acceptable or unacceptable. The criterion we selected is based on the number of "bends" that the path has to include. In particular, we use the following definitions.

floating segment: A segment which is not connected to a pin or a pseudopin (see Fig. 2).

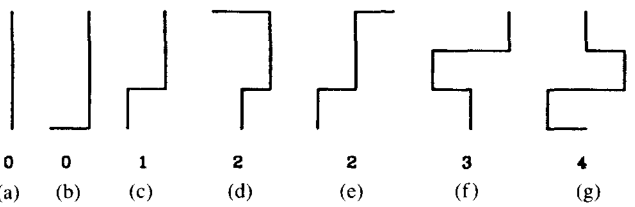


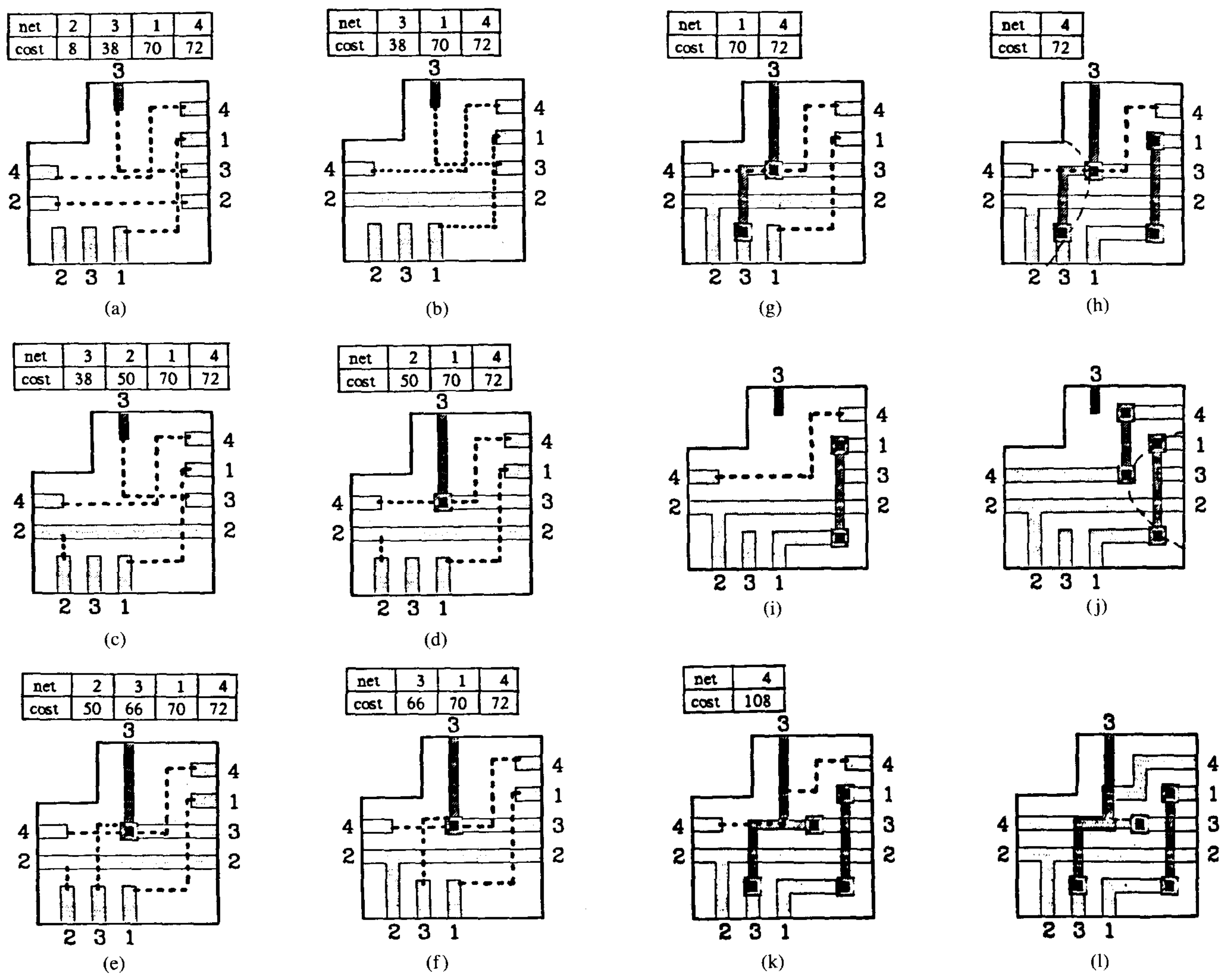
Fig. 2. The number of floating segments of several cases. With $n = 3$, (a) to (e) are good paths, while (f) to (g) are bad paths.

good path: A path which has less than n floating segments. ($n = 3$ to 5 seems to be a good range to use.

bad path: A path which is not a good path.

If a path is found and it is a good path, then it is scheduled again. Otherwise, the modification phase is entered. The weak modifier is first called to push other nets around to make a good feasible connection possible for the net under consideration. If no solution is found, then the strong modification phase is entered. Each of these implies a modification of the existing interconnections—some interconnections have to be either pushed away or removed. In both phases, a variety of alternative "good" interconnections for the net under consideration are examined, the cost of each solution is computed considering the cost of removing existing nets, and the minimum-cost solution is selected. If such solution is above a preset limit, then the router ends its search with a failure. The process is iterated until either a solution is found or a failure is reported.

To show the schedule and routing order, a simple example is shown in Fig. 3. The cost of a via is assumed to be 30, while the cost of a unit length of wire is 2 (if in preferred direction) or 50 (if in nonpreferred direction). Since the cost of a via is large, straight connections are made first and paths with two vias are implemented last. In Fig. 3(a), the minimum-cost path of each net is found as shown by the dashed lines, and all the four nets are scheduled in increasing order of their cost. Note that nets 2 and 3 have three components and that shortest paths connecting any two of the three components are found and scheduled. In (b), net 2 is popped and its path is implemented. In (c), net 2 has two components, so a shortest path which interconnects the bottom pin with the partially routed net is found and net 2 is rescheduled. In (d), a path of net 3 has been implemented. In (e), net 3 is rescheduled with a shortest path connecting its two components. In (f), net 2 is completely routed and three nets are in the queue. In (g), net 3 is completely routed and two nets are in the queue. In (h), net 1 is completely routed. However, the path of net 4 is completely blocked in the figure. Simple rip-up and reroute do not help in this case, as shown in Fig. 3(i) and (j). In (i), the blocking connection (net 3) has been removed, and net 4 is routed as shown in (j). Now net 3 is blocked and the pin on the right-side edge cannot be connected to other pins of net 3. Since Mighty uses weak modifications, the via of net 3 is moved by one



(a) net 2 3 1 4 / cost 8 38 70 72
(b) net 3 1 4 / cost 38 70 72
(g) net 1 4 / cost 70 72
(h) net 4 / cost 72
(c) net 3 2 1 4 / cost 38 50 70 72
(d) net 2 1 4 / cost 50 70 72
(i)
(j)
(e) net 2 3 1 4 / cost 50 66 70 72
(f) net 3 1 4 / cost 66 70 72
(k) net 4 / cost 108
(l)

Fig. 3. An example to show the routing order and the schedule, with shortest paths and their costs.

grid space to the right and a path for net 4 is found, as shown in (k). In (l), the routing is completed and the queue is empty.

Basic Algorithm: The precise algorithm is as follows.

```
mighty()
  /* Mark obstacles and pre-routed nets */
  pre_processing;
  /* Initially, schedule all nets */
  for( $i = 1$; $i \leq$ num_nets; $i$++ )
  {
    if( net $i$ has more than one component )
    {
      find a shortest path, path[$i$], connecting
        any two components of net $i$;
      if( path[$i$] != $\phi$ )
      {
        Schedule net $i$ with path[$i$] in increasing
          order of the path length;
      }
      else
      /* There exists a net which can not be connected
          without changing the routing area */
      {
        report failure;
        exit;
      }
    }
  }

  /* Main routing loop */
  while( schedule is not empty )
  {
    $i$ = the first net in the schedule;
    if( path[$i$] is feasible )
    {
      /* implement path[$i$] */
      conform_path( $i$ )
    }
    if( net $i$ has more than one component )
    {
      find a shortest path path[$i$] connecting
```

```
            any two components of net $i$;
        if( path[$i$] = $\phi$ or (path[$i$] is a bad path) )
            path[$i$] = weak_modification( $i$, path[$i$] );
        if( path[$i$] != $\phi$ )
        {
            Schedule net $i$ with path[$i$] in increasing
                order of the path length;
        }
        else
        {
            strong_modification( $i$ );
        }
    }
}
/* All nets have been connected. Vias and wire-length
    can be reduced further, and metal maximization can
    be done if necessary */
post_processing();
```

Now all the important functions used in the above algorithm are described.

B. Finding a Shortest Path

The routine that finds a minimum-cost path is a modification of Lee's algorithm [13]. The major extension is that our algorithm can find a minumum-cost path connecting any two of the components of each net, i.e., the minimum-cost path which can reduce the number of components by one. Hence, it is not necessary to specify two points (or two components) before finding a path, but the path finder makes use of all the existing pins and connections.

Since the search for the minimum-cost path is started from every component of the net, component number as well as the cost has to be propagated. In the data structure, every grid point p of the routing area can contain two integers, p.net and p.component. If p is a point of a conformed path, p.net contains the owner net. Otherwise, p.net is used to store the negative value of the cost during find_path. All the connected pins and wires of a net belong to the same component.

The algorithm consists of two parts, schedule and search. At the beginning, the grid points belonging to a pin or implemented connections of the net being processed are scheduled with cost value 1. Then the search starts from the first grid point p_f in the schedule, in which the cost value of the point is marked and all available neighbor grid points p_n are scheduled with cost = cost of p_f + *distance*(p_f, p_n), where *distance* is the incremental cost depending on the direction, the layer, and change of layers. The frontiers of search grid points are expanded like a wave front from all the components of the net. It should be pointed out that the path found by this approach is frequently shorter than a minimum path between two specified points, since the search is triggered from all the conformed wire segments as well as pins of the net. When cost = c, the algorithm marks all the free grid points reachable at cost c from any component of a net i. We set all the cost parameters as even numbers, so that we can get an integer even after dividing the cost by 2. The desired result comes from the following theorem.

Theorem 1: If there exists a path of length less than Max_cost, then the find_path algorithm finds a minimum-cost path connecting two components of net i.

Proof: When cost is equal to c, the algorithm has marked all free grid points reachable from each component of net i with cost $< c$ and part of grid points reachable with cost = c. If two different components (wave fronts) meet at a grid point, then a path is found and the cost of the path connecting the two components is calculated. Let *touchcost* be the half of the cost of the path found. If a path is found at *touchcost* = c for the first time, then there does not exist a path with cost $\leq 2(c - 1)$. (Otherwise, we had to find the path at *touchcost* = $c - 1$.) Since all the cost parameters are even numbers, the minimum possible cost is $2c$, and the path found is an optimal one. If there does not exist a path of length less than Max_cost, then the algorithm cannot find a path until *cost* is increased to Max_cost/2 and the algorithm returns nil (ϕ).

Algorithm Find_Path: This algorithm finds a minimum-cost path connecting any two components of a net. In the algorithm, the *searchq* contains all the scheduled grid points in increasing order of cost, with their cost values and component numbers.

```
find_path( $i$ )
    /* Initialization and schedule */
    cost = 1;
    schedule all the pins and existing connections of net $i$

    /* Search for a path.
     Max_cost is a user defined parameter */
    while( cost < Max_cost/2 )
    {
        if( searchq has not a point $p$ of cost )
        {
            cost = cost + 1;
    continue;
        }
        else
        {
            /* get the first point $p$ from searchq */
            pop $p$ with component $comp$ from the searchq;
            if( point $p$ is searched already with less cost
                by the same component )
    continue;
            }
            else if( point $p$ is searched already
                by a different component )
            /* A minimum-cost path has been found */
            {
                find the path, path[$i$], by backtracing;
                return( path[$i$] );
            }
            else
        {
            /* Mark the point $p$ as searched */
            if( $p$.net = $\phi$ )
                $p$.net = $-$ cost;
            $p$.component = comp;
            /* schedule its neighbors */
            for(each neighboring point $s$ of point $p$ )
            {
                if( $s$.net = $\phi$ )
                {
                    /* schedule $s$ */
                    add $s$ in searchq with comp
                        and cost + distance($p$,$s$ );
                }
                /* if $s$ is searched by another component,
                  then a path is found */
                else if( $s$.component $\neq$ $p$.component
                    and $s$.component $\neq$ $\phi$ )
                {
                    /* A path is found which may or may not
                      be a shortest path */
                    touchcost = (cost of $p$ + distance($p$,$s$ ) +
                        cost of $s$ )/2;
                    /* schedule $s$ */
                    add $s$ in searchq with comp
                        and touchcost;
                }
            }
        }
    }
}
/* Path does not exist */
return( $\phi$ );
```

C. *Conformation of a Path*

In the basic algorithm, when net i is processed according to the schedule, if all the points of path[i] are available (they have not been assigned to another net), the routine *conform_path* is called. Conform_path(i) implements the path of net i by assigning the path[i] to net i, and by merging the two components just connected into a big component.

D. *Weak Modification*

As discussed above, during weak modifications, part of the existing paths can be relocated while maintaining all the existing connections. After a minimum-cost path is found, the path is evaluated as either a good path or a bad path according to the number of floating segments (see Fig. 2). If the path is good, it is scheduled and processed. Otherwise, weak modifications are attempted to find a better solution.

We use three types of weak modifications (*unit_push*, *jump_push*, and *point_push*), as shown in Fig. 4. Each of these modifications is in four directions (down, up, left, and right). For the sake of simplicity, only downward modifications are shown in the figure.

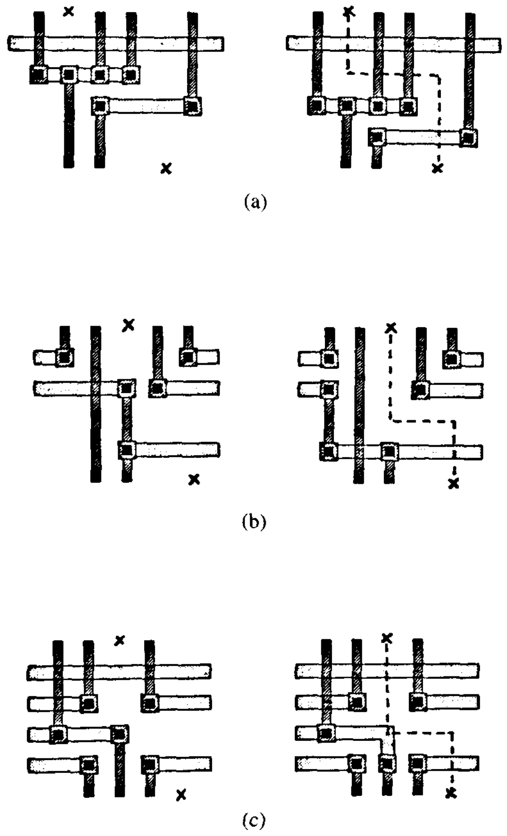


Fig. 4. Examples of weak modifications: (a) *unitpush* (down), (b) *jumppush* (down), (c) *pointpush* (down).

In the algorithm, described below, *weak_modification* calls *modify*, and *modify* calls *tryunitdown*, etc. The routine *tryunitdown* finds a blockage, and if the blockage is modifiable, calls *unitdown*, which actually tries to push the blockage a unit grid space downwards and returns YES if successful.

The routines *jumppush* and *pointpush* are implemented in a similar way, and their algorithms are not included. *Jumppush* differs from *unitpush* in that it can move segments of other nets. *Pointpush* is similar to *unitpush*, but is more powerful in that it can change the layer of a segment at the blocked point if it is necessary. *Unitpush* and *pointpush* are recursive so that they can push a stack of nets (for example, two nets are modified in Fig. 4(a)).

One of the difficult issues in rerouting or routing modification is how to prevent oscillations. To prevent oscillation in weak modifications(push/jump up and down or left and right can be repeated forever), we used history. We ordered the 12 types of weak modifications as shown in *modify*. If one type of modification did not solve the problem, then we make sure that the same modification is not called again unless one of the related nets is rerouted by a strong modification or one more connection is made. The element on the ith row and the jth column of the history matrix contains the type of weak modification to be tried if net i calls weak modification to push net j. When one more connection is made in *conform_path* for net i or when net i is affected during *strong_modification*, all the elements on the ith row or ith column are reset so that all types of weak modification can be tried later.

```
weak_modification( i, path[i] )
   /* i is a net, path[i] is the shortest path of net i */
   /* Save current paths */
   for( j = 1; j ≤ num_nets; j++ )
      savedpath[j] = path[j];
   /* find_cost returns the sum of costs of path[1..num
      _nets] */
   totalcost = find_cost();
   while( (type = modify( i )) ≠ φ )
   {
      path[i] = find_path( i );
      if( path[i] != φ )
      {
         newcost = find_cost();
         if( newcost < totalcost )
         {
            /* better solution is found */
            totalcost = newcost;
            for( j = 1; j ≤ num_nets; j++ )
               savedpath[j] = path[j];
         }
         /* if path[i] is good then break,
           otherwise try more modifications. */
         if( path[i] is a good path )
   break;
      }
   }
   /* Choose the best path found */
   for( j = 1; j ≤ num_nets; j++ )
      path[j] = savedpath[j];
   /* reschedule affected nets */
   for( each modified net k )
      reschedule( k );
   return( path[i] );

modify( i )
   find the two closest blocked pins p1 and p2 of net i;
   if( tryunitdown( p1, p2, i ) )
      return( Unitdown = 1 );
   if( tryunitup( p1, p2, i ) )
      return( Unitup = 2 );
   if( tryunitleft( p1, p2, i )
      return( Unitleft = 3 );
   if( tryunitright( p1, p2, i ) )
      return( Unitright = 4 );
   if( tryjumpdown( p1, p2, i ) )
      return( Jumpdown = 5 );
   if( tryjumpup( p1, p2, i ) )
      return( Jumpup = 6 );
   if( tryjumpleft( p1, p2, i ) )
      return( Jumpleft = 7 );
   if( tryjumpright( p1, p2, i ) )
      return( Jumpright = 8 );
   if( trypointdown( p1, p2, i ) )
      return( Pointdown = 9 );
   if( trypointup( p1, p2, i ) )
      return( Pointup = 10 );
   if( trypointleft( p1, p2, i ) )
      return( Pointleft = 11 );
   if( trypointright( p1, p2, i ) )
      return( Pointright = 12 );
   return( φ );

   tryunitdown(p1, p2, i )
   /* p1 and p2 are blocked pins of net i.
     If possible, move a blocking segment one grid space
     in the downward direction and update history */
   /* find a blockage */
   scan the vertical layer of the routing region downward
      from p1 until a blockage b is found;
   /* Let the location of the blockage be (x, y) */
   if( b is a modifiable net AND history[i, b] ≤ Unitdown)
   {
      if( unitdown( i, b, x, y ) )
         /* The downward push was successful */
         history[i, b] = Unitdown;
      else
         /* Same type of push can not be repeated */
         history[i, b] = Unitdown + 1;
   }
   repeat the above for pin p2;

unitdown( i, b, x, y )
   /* i is the blocked net,
     b is the blocking net,
     (x, y) is the grid point on which the blockage is found*/
   find the interval( from x1 to x2) of net b to be pushed;
   if( there is a pin of net b on the interval )
      return( NO );
   find the blockages that prevent the move of the segments
      of net b on the interval by one grid space downward;
   if( there are no blockages )
   {
      move the segments of net b on the
         interval by one grid space downward;
      return( YES );
   }
   else if( there is a single modifiable net n in the new
      blockage )
   {
      /* push the new blockage first */
      if( unitdown( b, n, x, y+1 ) )
      return( YES );
   }
   else
      return( NO );
```

E. Strong Modification

When weak modification fails to find a path, then strong modification is called. If strong modification fails to find a path within the maximum cost limit, the router reports failure and exits. Since the report includes the locations of failure, more space can be added at the right position to complete the routing.

During a strong modification, we remove part of existing connections so that the blocked net can be connected. First a minimum-cost rip-up path is found. Then all the connections of the nets in the path are removed, and the blocked pins are connected. All the nets disconnected during the rip-up process are rescheduled.

The rip-up cost consists of the number of nets affected, the difficulty values of the nets, the length and the number of vias in the path. The difficulty values are zero at the beginning. If connections of a net are removed during strong modification, the difficulty value of the net is increased by Delta, which is a user-defined parameter. This makes the same net unlikely to be removed repeatedly, prevents oscillations, and makes it possible to try a new path when strong modification is called several times to connect the same pair of blocked pins.

To find an optimal rip-up path between the pins of a blocked net, we evaluate the rip-up cost of all the straight, L-shaped, U-shaped, and Z-shaped paths, and choose the minimum-cost one.

```
strong_modification( i )
  /* find a pair of blocked pins of net i */
  find two closest pins p1 and p2 of net i
     which are not in the same component;
  find a minimum-cost rip-up path connecting p1 and p2;
  /* Limit_cost is a user defined parameter */
  if( rip-up cost ≥ Limit_cost )
  {
     /* rip-up cost is too large */
     report_failure();
     exit();
  }
  else
  {
     /* remove all the connections in rip-up-path.
       reschedule all the affected nets. */
     for( each net k in the rip-up path )
     {
        difficulty[k] = difficulty[k] + Delta;
        remove all the connections of net k;
        /* reschedule with zero cost and nil path */
        reschedule( k );
        clear_history( k );
     }
     while( net i has more than one component )
     {
        find a shortest path[i] connecting any two
           components of net i;
        if( path[i] != φ )
           conform_path( i );
        else
           strong_modification( i );
     }
     /* reset the ith row and ith column of history
       which is used in weak modification */
     clear_history( i );
  }
```

IV. Complexity of the Algorithm

In this section, we prove that the algorithm given in Section III terminates and that its run time is bounded by a polynomial in the size of the input. Let the number of pins be p, and let the number of nets be k. Let L be the complexity of finding a minimum-cost path within the routing area. If there are m rows and n columns, then clearly $L = O(mn)$. Furthermore, without loss of generality, we assume that $m \leq n$ so that $O(m + n) = O(n)$.

To find the computing complexity when weak modifications are used, a "history" of the weak modifications is used. History is a k by k matrix. The element on the ith row and jth column of the matrix has a number which represents the type of weak modification to be tried if net i calls weak modification to relocate net j. The 12 modifications are tried in the order listed in the algorithm in the previous section. If the current weak modification fails to modify, then the next in the order list is tried.

Keeping the information of strong modification is a little simpler. Each net has a difficulty value, which is increased if the net is ripped up during a strong modification. Since this difficulty value monotonically increases whenever a strong modification is called, the algorithm must terminate after a finite number of operations, since Limit_cost is finite.

Parts (a) and (b) of Theorem 2 explore cases of successful routing, and part (c) holds whether the routing is successful or not.

First we prove four intermediate results, and then state the theorem.

Lemma 1: If strong modification is not called, $(p - k)$ connections complete the routing.

Proof: There are p unconnected pins initially; hence, the total number of connected components is p. After completion of routing, there will be k connected components. Since each connection reduces the number of connected components by one, $(p - k)$ connections complete the routing.

Lemma 2: If no modification is used, each of the connections requires at most kL operations.

Proof: All the nets which have more than one connected component are in the queue. Among them, those which are scheduled after the last successful connection have feasible paths because all conformed paths have been considered when finding the paths, and others may not be feasible. Since there are k nets, all the scheduled paths are feasible at most after $k - 1$ failures. Hence, the saved path of the first net in the queue is also feasible and one connection is possible at most after k find paths.

Lemma 3: If strong modification is not used, the maximum number of weak modifications is bounded by $O(kpn)$.

Proof: First we show that at least one element of the history matrix is increased within $2n$ modifications. For any of the weak modifications (unitpush, jumppush, and pointpush) in the directions of left or right, on each row of the routing area, the same type of modification in the same direction can be successful at most $(n - 1)$ times. Since we start the modification from the closest two pins of different subnets, $2(n - 1)$ modifications are possible in the worst case. Similarly, for the up or down modifications, $2(m - 1)$ modifications are possible.

It is clear that the maximum sum of all the element of the history matrix is $(T + 1)\,k(k - 1)$, where T is the number of weak modifications to be tried and is 12 in the current implementation of Mighty. Note that the diagonal elements of history are not used at all, because a net cannot block itself. Each element of the matrix grows from 1 to $(T + 1)$. Since clear-history can be called at most $(p - k)$ times by Lemma 1 and each time $2(k - 1)$ elements are reset to 1, the maximum number of weak modifications is given by $2n(2(k - 1)\;T(p - k) + k(k - 1)\;T)$. Since $k < p$, the number of weak modifications is $O(kpn)$.

Lemma 4: The number of operations necessary for one weak modification is bounded by $O(kL)$.

Proof: In the worst case, all k nets can be pushed during one weak modification. If a moved net has more than one connected component, we need to do find-path and reschedule the net.

Theorem 2:

(a) If the algorithm completes the routing without calling any modification routines, the computing complexity is bounded by $O(kpL)$.

(b) If the routing is completed by using weak modification but no strong modification, the worst-case complexity of the algorithm is $O(k^2pnL)$.

(c) If strong modification is used, then the algorithm terminates after at most $O(k^3pnL)$ operations.

Proof:

(a) The proof follows from Lemmas 1 and 2. Since we need to make $(p - k)$ connections and each connection takes at most kL operations, $(p - k)kL$ operations are enough to complete the routing.

(b) From Lemma 3, the maximum number of weak modifications is $O(kpn)$. From Lemma 4, one modification takes $O(kL)$. Hence the total number of operations required in weak modification in the worst case is $O(k^2pnL)$. To complete the routing, the complexity is $O(k^2pnL) + O(kpL) = O(k^2pnL)$.

(c) Let the limit of rip-up cost be C, and let the incremental rip-up cost be D. Then the maximum number of calls of the strong modification is $k\lceil C/D \rceil$. Since C and D are constants, it follows from (a) and (b) that the number of operations either to complete the routing or to report failure is $O(k^3pnL)$.

The bounds given above are worst-case ones. Empirical results show that the computing time is roughly $O(mn)$ to $O((mn)^{1.5})$.

V. Variations to the Basic Algorithm

Several other features are added to the routing algorithm based on the incremental routing modifications described in Section III. The most important one is the addition of pseudopins to give global information to the maze routing phase in *find_path*. Other features are equivalent pins, floating pins, prerouted nets, critical nets, and handling of pins on either layer.

A. Pseudopins

When we applied the algorithm to long channels of irregular shape, we found that the algorithm may spend large amounts of time and find solutions that are not satisfactory. For this reason, we developed an additional technique that has made the application of the algorithm to large routing regions with irregular shape as well as to long channels with many nets successful. The idea is to guide the selection of the paths by placing "pseudopins" inside the routing region for a number of critical nets. This scheme adds some global information about the routing region to the maze routing phase.

For long channels, such as Deutsch's difficult example, we place pseudopins at the column of maximum density. For wildly irregular channels, pseudopins are added across the columns where the variation in width occurs. To guide the search even further, pseudopins can be added at regular intervals away from the column of maximum density or where the irregularity takes place. The placement of the pseudopins is accomplished by examining the vertical constraint graph (see [15] or [16] for the definitions) of the channel. Pseudopins are not real pins in that the net being routed is not forced to go through them, but if it does not go through them the path is penalized.

Note that the selection of where to insert pseudopins as well as their placement is totally automatic in Mighty. No user intervention is required.

To describe the algorithm used in pseudopin generation, we need the following definitions.

pseudopin: A pin temporarily added to give some global information to the path finder and to help modification routines.

compensated lateral distance The distance in the horizontal direction in which pin density is considered. For example, the conpensated distance between columns $c1$ and $c2$ is given by $|c1 - c2| + a(b + 2d)$, where a is a constant, b is the number of columns having one pin, and d is the number of columns having no pin, between $c1$ and $c2$. This reflects the freedom of routing due to pin density.

vertical constraint: A constraint that exists when two nets have a pin in the same column. The net connected to the top pin must have its horizontal segment above that of the net connected to the bottom pin on the column.

horizontal constraint: A constraint that exists when two nets cross the same column. If pseudopins have to be added to guide the routing of nets that have a horizontal constraint, pseudopins should be added on different rows even though these pins are added on different columns.

Violating vertical or horizontal constraints does not imply that complete routing cannot be achieved. However, satisfying these constraints makes the later routing easier.

As shown below, pseudopins are generated using a linear assignment algorithm so that the number of VCV (vertical constraint violations) or HCV (horizontal constraint violations) and the wire length are minimized. Setting the cost parameters in linear assignment is very important for satisfactory results. The algorithm we use is described in detail in Section V-A-2. Since the path finder does not use any information about vertical or horizontal constraints, we can give some hints to the path finder by generating pseudopins such that each vertical or horizontal constraint violation is penalized. Using the topmost and bottommost tracks is also penalized. To find the level_from_top and level_from_bottom, which are the highest and lowest rows that a net can be assigned without VCV [16], we need to break all the cycles, if any, in the vertical constraint graph. We remove cycles by removing edges which are created by a column farthest from the column on which pseudopins are being generated. In other words, we generate the pseudopins such that vertical constraints are satisfied near the column and let the main detailed router (path finder, path conformer, weak and strong modifiers) choose the exact path of connections to avoid blockages where vertical constraint violations are observed.

1) Pseudopin Generation: A linear assignment algorithm generates a matching of crossing nets and available tracks while minimizing the total cost of the matching. (Linear assignment has independently been used for global routing by Marek-Sadowska and Lauther [17], [18].)

Let $N = \{n_1, n_2, \cdots, n_p\}$ be the set of crossing nets, and let $T = \{t_1, t_2, \cdots, t_q\}$ be the set of available tracks on the column. Note that q should be greater than or equal to p to assign pseudopins for all the crossing nets. A bipartite graph $B(X, Y, U)$ is constructed by associating nodes in X to the nets in N, nodes in Y to the tracks in T and adding an arc $(n_r, t_c) \in U$ with cost c_{rc} for each $1 \leq r \leq p$ and $1 \leq c \leq q$ (c_{rc} being the cost of assigning a pseudopin of net n_r on track t_c). Then a source node s and a sink node t are added. Finally, to make sure that each net is assigned to only one track, arcs (s, n_r) and (t_c, t) are added with capacity 1, for all $1 \leq r \leq p$ and $1 \leq c \leq q$. Now we can obtain an assignment by solving a maximum flow and minimum-cost problem on the graph.

The following algorithm shows how pseudopins are generated on a selected column.

```
pseudopin_generation()
  select a column;
  find available tracks;
  find the nets crossing the column;
  construct vertical constraint graph on proper interval;
  while (there is a cycle)
    break the edges which are caused by constraints on
      farthest columns;
  fill cost matrix;
  solve linear assignment problem;
  add pseudopins;
```

2) Setting Parameters in the Cost Matrix: In the algorithm, each row r of the matrix corresponds to a crossing net $net(r)$ and each column c corresponds to an available track $tr(c)$. Each element $cost[r, c]$ of the cost matrix is obtained by adding the costs from vertical constraints, horizontal constraints, and other pin locations of the net(r).

```
fill_cost_matrix()
  /* VCV_cost: parameter penalizing vertical constraint violation.
     HCV_cost: parameter penalizing horizontal constraint violation.
     Hslope (Vslope): parameter penalizing distance to
        other pins on horizontal (vertical) preferred
        layer. */
  /* Generate pseudopins on a column col of the channel */
  for( each row r )
  {
    /* penalize vertical constraint violation */
    for( each column c )
      if( assigning net(r) on track tr(c) cause VCV )
        cost[r, c] = cost[r, c] + VCV_cost;
    /* penalize horizontal constraint violation */
    for( each column c )
      /* HCV may occur due to other nets or obstacles */
      if( assigning r to c causes HCV )
        cost[r, c] = cost[r, c] + HCV_cost;
    /* look around other pins */
    for( each pin p of net(r) )
    {
      /* by using large value of Hslope, all the pseudo
         pins of a net can be generated on a track. */
      /* Let (x, y) be the coordinates of the pin p */
      if( p is on the horizontal preferred layer )
        /* α is a constant
           dist returns the compensated distance */
        slope = Hslope/(α + dist(x, col));
      else
        slope = Vslope/(α + dist(x, col));
      for( each column c )
        cost[r, c] = cost[r, c] + slope * |tr(c) − y|;
    }
  }
```

Several other features have been added to make the router practical.

B. Preprocessing and Postprocessing

Preprocessing sets up the data structure and marks obstacles in the routing region so that *find_path* can avoid them when finding a shortest path.

Postprocessing is a "cleanup and beautification" step. During postprocessing, all the nets are rerouted from the net with longest net length to the one with shortest net length. Each net is completely removed and routed again with different cost parameters. Since all other nets are connected, we can penalize less routing in the vertical (horizontal) direction on the horizontal (vertical) layer. Metal maximization can be achieved by assigning less cost to one of the two layers. (For example, when the two interconnection layers are poly and metal.) Also, the total wire length and the number of vias can be traded off by setting the cost parameters accordingly.

```
preprocessing()
    move pins on the boundary inside the channel by one
        grid space;
    if( channel is long OR irregular)
        generate pseudopins;
    if( there exist floating pins )
        assign positions for them;
    /* Mark obstacles as blocked */
    if( there exist blocked area )
        mark the blocked grids in the routing area;
    if( there exist equivalent pins )
        merge the pins to form a component;

postprocessing()
    /* via and wire-length minimization */
    change cost parameters in find__path;
    sort the nets by total wire length in decreasing order;
    for( each net i in the order given above )
    {
        remove all segments of net i
        while( net i has more than one component )
        {
            path[i] = find__path( i );
            conform__path( i );
        }
    }
```

C. Equivalent Pins

Some pins are electrically equivalent since they are interconnected outside the routing region already. The algorithm deals with these cases very naturally, since these pins may be given the same "component name" when the maze router is invoked and the interconnection will reflect the additional optimization that results from taking advantage of this feature.

D. Floating Pins

Floating pins at the boundaries of routing regions are often present. The algorithm decides the positions of floating pins during preprocessing, using the same procedures as in the generation of pseudopins. As shown in the preprocessing algorithm, this is done after all the pseudopins are generated.

E. Obstacles and Prerouted Nets

If there are obstacles on any of the two layers, then we mark this area and the path finder is prevented from using it.

If a net is completely prerouted then we can mark the interconnection as an obstacle. If only part of a net is prerouted, then we can start routing from the existing status. For example, if we attach temporary pins on all the grid points used by the prerouted net, the prerouting is used as an initial condition and the path finder will optimize other necessary connections.

If Mighty is used for automatic cell generation or routing over the cells, we may find an obstacle covering a large portion of one layer. In this case, if there are several pins of different nets on the other layer at the same location as the obstacle, modifications of existing connections are limited. To overcome this problem, a routine is activated if strong modification fails because of an obstacle. This routine tries to guide the blocked pins to the region where both of the layers are available for routing.

F. Critical Nets

Some nets may be electrically critical. These nets can be routed first using minimum number of vias and wire length. If necessary, we can use a different set of costs. For example, we can route power nets on metal layer simply by assigning a very high cost to the polysilicon layer. We can force the modifiers not to touch these important nets by assigning high cost to the modification of their interconnections.

G. Routing on the Boundaries

Another important factor that decides the routing area of real examples is the ability to handle pins on an arbitrary layer. Most conventional routers [1], [16] assume that pins on vertical boundary segments are on one layer and those on horizontal boundary segments are on the other layer. However, real examples frequently do not satisfy this assumption, and conventional routers need extra tracks and columns to change layer and make a larger number of vias than necessary.

VI. Experimental Results

All the results described in this section have been obtained by using the same set of cost values, on a DEC VAX 11/785 running 4.3BSD UNIX.

In the tables, "Mighty" is the name of our router. Table I shows a comparison with other well-known routers on Burstein's difficult switchbox. Mighty uses one less column than any other router. Mighty takes 2.7 seconds to complete the connections in this example, and postprocessing takes 1.3 seconds. Note that WEAVER took 1390.0 seconds to route the same example on a VAX 11/780. WEAVER is written in OPS5 and C. Its run time is long in part because of the use of OPS5. A speedup can certainly be obtained by rewriting the router entirely in C. However, in this case, some of the most interesting features of expert systems would be lost. We believe that the complexity of the underlying approach is also responsible for its long run time. Table II shows the results of other switchbox examples.

The detailed router described in this paper is quite symmetric in the vertical or the horizontal direction, as opposed to other algorithms reported in the literature that are sensitive to this factor [2]. We tried to route some of the switchboxes after rotating the channel by 90° and in all cases obtained basically the same results. As an example, two routers are compared on Burstein's switchbox example in Table III.

TABLE I
ROUTING OF BURSTEIN'S DIFFICULT SWITCHBOX

Router Name	#rows	#columns	#vias	total length
Hamachi	15	23	67	564
Luk	16	23	58	577
Mod Detour	15	23	63	567
M-Sadowska	15	23	58	560
WEAVER	15	23	41	531
Mighty	15	22	39	541

TABLE II
ROUTING OF SWITCHBOX EXAMPLES

Example Name	Router Name	#rows	#cols	#vias	total length
simple	Lee	7	7	11	60
	WEAVER	7	7	4	60
	Mighty	7	7	5	60
term inten	Luk	16	23	68	632
	WEAVER	16	23	49	615
	Mighty	16	23	50	629
dense	Luk	18	16	36	527
	Mighty	18	16	32	530
mod dense	WEAVER	17	16	29	510
	Mighty	17	16	29	510

TABLE III
DEPENDENCE ON THE ORIENTATION OF THE ROUTING AREA

scan direction	#tracks used Luk	Mighty
orientation1	23 x 18	22 x 15
orientation2	23 x 20	22 x 15
orientation3	23 x 16	22 x 15
orientation4	24 x 16	22 x 15

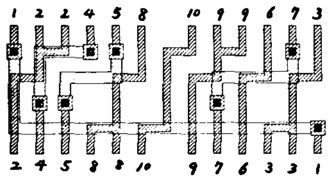


Fig. 5. Channel example routed by Mighty.

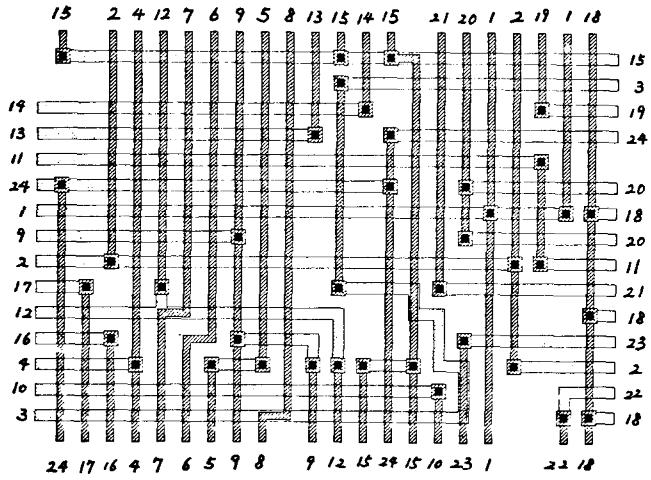


Fig. 6. Burstein's difficult switchbox routed by Mighty.

As pointed out above, our router has been developed to deal with switchboxes and highly irregularly shaped channels as well as rectangular channels. Table IV shows the comparisons on Deutsch's difficult channel example [19]. Table V shows the results when we applied Mighty to a number of difficult channels compared with the results of YACR [16] and Chameleon [20]. In all the examples tried, our router shows equal or better results, as we expect since Mighty has more degrees of freedom. However, note that in the past the use of switchbox routers in channels has not given results as compact as the ones that can be obtained with channel routers. Hence, the above results show how powerful the techniques used in Mighty really are.

Three examples, corresponding to cases when the modifiers were not used, when weak modification was used, and when strong modification was employed, are shown in Figs. 5 through 7. Fig. 5 is the result of our router when applied to the difficult channel presented in [21]. Note that we have not used an empty column in the middle of the channel. Mighty could complete the routing without using any modifications by using four tracks. To route this example with one more empty column, Burstein's router used five tracks and YACR used six tracks. Fig. 6 shows the result of Burstein's switchbox. To complete the routing, two weak modifications and no strong modifications were used. Fig. 7 shows the results of Deutsch's channel example routed by Mighty. During the routing of Deutsch's channel, 27 weak modifications and 16 strong modifications were used. The total CPU time was 176 seconds, including 46 seconds of postprocessing time.

The router has also been used to generate CMOS cells, interfaced with TOPOGEN, which is a synthesis tool that takes a logic description at the gate level and converts it into a symbolic layout of a static CMOS circuit on virtual grids [22]. TOPOGEN generates the sequence of p and n channel transistors on two rows, Mighty generates all the necessary interconnections between the transistors to obtain a desired functional circuit block. Fig. 8 shows a sample result routed by Mighty during the generation of a 16-input CMOS gate. YACR used four more tracks to route the same example.

VII. SENSITIVITY OF THE ALGORITHM

The described algorithm has many parameters. For all the examples described in this paper, the same parameters are used. Since Mighty performed as well as or better than existing algorithms on both channel and switchbox examples, the algorithm seems to be robust with respect to parameters.

The cost parameters in finding a shortest path are the most important ones. Hence, the sensitivity of the algorithm to cost parameters in finding a shortest path is examined. We fixed the cost per wire length in the preferred direction to 2 and the cost of a via to 30; we varied the cost per nonpreferred direction W_{np} from 20 to 70. When applied to Burstein's switch-box example, Mighty gives the same results for (W_{np} = 40, 50, 60 (see Fig. 9). When applied to Deutsch's channel, Mighty completes the routing with 19 tracks for W_{np} = 40, 50, 60, 70; the total wire length is minimum at W_{np} = 50.

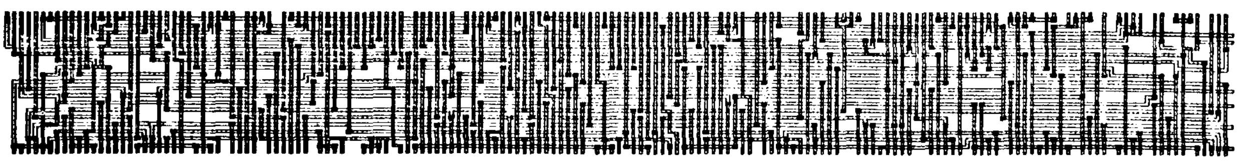


Fig. 7. Deutsch's difficult channel routed by Mighty.

TABLE IV
ROUTING OF DEUTSCH'S DIFFICULT CHANNEL

Router Name	#rows	#vias	total length
Yoshimura,Kuh	20	308	5075
Hamachi	20	412	5302
Burstein	19	354	5023
YACR	19	287	5020
Mighty	19	301	4812

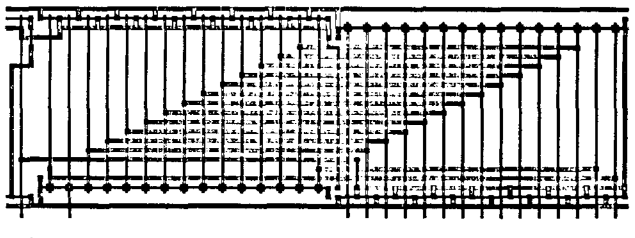


Fig. 8. CMOS gate (NAND (NOR $X2\ X3 \cdots X16$) (NAND $X1\ X2 \cdots X15$)) routed by Mighty.

TABLE V
ROUTING OF CHANNEL EXAMPLES

	channel			#tracks used		
Example Name	#nets	#cols	density	YACR	Chameleon	Mighty
3a	30	45	15	15	15	15
3b	47	62	17	18	18	17
3c	54	103	18	19	19	18
chris1	158	432	49	50	49	49
cycle.t	65	134	16	19	17	17
ex1	235	417	16	17	16	15
ex2	282	421	15	16	16	15
ex3	291	421	11	12	12	11
ex4	270	421	19	22	22	20
r1	77	139	20	22	22	22
r2	77	117	20	21	20	20
r3	78	123	16	18	18	17
r4	74	150	15	17	17	17

Example	Wnp	30	40	50	60	70	80
Burstein switch-box	via length	fail	39 541	39 541	39 541	fail	fail
Deutsch channel	via length	fail	300 4825	301 4812	299 4830	314 4920	fail

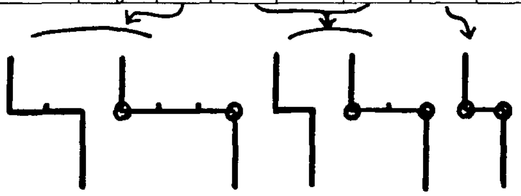


Fig. 9. Sensitivity of the algorithm to cost parameters.

The basic ideas we followed to select the parameters are also shown in Fig. 9. If changing layers by introducing two vias is not wanted, even if the routing is in nonpreferred direction, we may use small values of W_{np} (for example, $W_{np} = 30$). On the contrary, if we like to use one layer for horizontal segments only and the other layer for vertical segments only, then we may use large values of W_{np} (for example, $W_{np} = 70$). Too much restriction on layer usage increases the number of vias unnecessarily, and too much flexibility can create blockages. Hence, we believe that $W_{np} = 40$ to 60 is a good range to use.

Max_cost in *find_path* and Limit_cost in *strong_modification* are not critical as long as they are large enough. Max_cost should be large enough so that all the available paths can be found. Large values of Limit_cost allow the algorithm to search more rip-up paths; however, CPU time increases, as shown in Section IV. Currently, Max_cost = Limit_cost = 500 is used, and further increase of these values could not reduce the routing area for the examples tried. Note that if the routing area is large, pseudopins are added and the length of the shortest paths connecting components of nets can be bounded.

VIII. CONCLUSIONS

We have presented a new general two-dimensional router, Mighty. For this router, rather than trying to avoid blocking the paths of unrouted nets, we developed a new approach in which the minimum-cost path is found and connected incrementally, and the existing connections are then modified or rerouted whenever the incremental interconnections are not satisfactory. The described router is very flexible. For example, we can route on a preferred layer as much as possible or trade off the number of vias and the wire length. The mechanism used to guide this routing is parameter setting.

We are now extending this approach to multilayer routing for macrocell and printed circuit board routing.

The program implementing the algorithms described in this paper consists of about 11 000 lines of C. This router has been developed as a part of an integrated automatic synthesis and layout system for VLSI design [14].

REFERENCES

[1] C. Hsu, "A new two-dimensional routing algorithm," in *Proc. 19th Design Automat. Conf.*, June 1982, pp. 46-50.

[2] W. Luk, "A greedy switch-box router," *Integration*, vol. 3, pp. 129-149, 1985.

[3] Y. Hsieh and C. Chang, "A modified detour router," in *Proc. Int. Conf. CAD*, Nov. 1985, pp. 301-303.

[4] G. Hamachi and J. Ousterhout, "A switch-box router with obstacle avoidance," in *Proc. 21st Design Automat. Conf.*, June 1984, pp. 173-179.

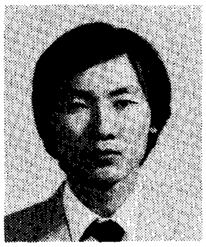
[5] M. Marek-Sadowska, "Two-dimensional router for double layer layout," in *Proc. 22nd Design Automat. Conf.*, June 1985, pp. 117-123.

[6] R. Joobbani, "WEAVER: A knowledge-based routing expert," Carnegie-Mellon Univ., Res. Rep. CMUCAD-85-56, June 1985.

[7] F. Rubin, "An iterative technique for printed wire routing," in *Proc. 11th Design Automat. Workshop*, 1974, pp. 308-313.

[8] R. Linsker, "An iterative-improvement penalty-function-driven wire routing system," *IBM J. Res. Develop.* vol. 28, no. 5, pp. 613-624, Sept. 1984.
[9] P. Agrawal and M. A. Breuer, "Some theoretical aspects of algorithmic routing," in *Proc. 14th Design Automat. Conf.*, 1977.
[10] W. A. Dees and P. G. Karger, "Automated rip-up and reroute techniques," in *Proc. 19th Design Automat. Conf.*, 1982, pp. 432-439.
[11] I. Shirakawa and S. Futagami, "A re-routing scheme for single-layer printed wiring boards," *IEEE Trans. Computer-Aided Design*, vol. CAD-2, no. 4, pp. 267-271, Oct. 1983.
[12] K. Suzuki, Y. Matsunaga, M. Tachibana, and T. Ohtsuki, "A hardware maze router with application to interative rip-up and reroute support," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, no. 4, pp. 466-476, 1986.
[13] C. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electron. Computer.*, vol. EC-10, pp. 346-365, Sept. 1961.
[14] J. Burns, A. Casotto, M. Igusa, F. Marron, F. Romeo, A. Sangiovanni-Vincentelli, C. Sechen, H. Shin, G. Shrinath, and H. Yaghutiel, "Mosaico: An integrated macro-cell layout system," to be published.
[15] T. Yoshimura and E. Kuh, "Efficient algorithms for channel routing," *IEEE Trans. Computer-Aided Design*, vol. CAD-1, no. 1, pp. 25-35, 1982.
[16] J. Reed, A. Sangiovanni-Vincentelli, and M. Santomauro, "A new symbolic channel router: YACR2," *IEEE Trans. Computer-Aided Design*, vol. CAD-4, no. 3, pp. 208-219, July 1985.
[17] M. Marek-Sadowska, "Route planner for custom chip design," in *Proc. IEEE Int. Conf. CAD*, Nov. 1986, pp. 246-249.
[18] U. Lauther, private communication, 1986.
[19] D. Deutsch, "A dogleg channel router," in *Proc. 13th Design Automat. Conf.*, June 1976, pp. 425-433.
[20] D. Braun, J. Burns, S. Devadas, H. Ma, K. Mayaram, F. Romeo, and A. Sangiovanni-Vincentelli, "Chameleon: A new multi-layer channel router," in *Proc. 23rd Design Automat. Conf.*, June 1986.
[21] M. Burstein and R. Pelavin, "Hierarchical channel router," in *Proc. 20th Design Automat. Conf.*, June 1983, pp. 591-597.
[22] C. Sequin, "Tools for macro module construction," presented at Int. Workshop Symbolic Layout and Compaction, Nov. 1986.

*



Hyunchul Shin (M'87) was born in Kunsan, Korea. He received the B.S. degree in electronics engineering from Seoul National University and the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology in 1978 and 1980, respectively. Since August 1983, he has been working toward the Ph.D. degree in the Department of Electrical Engineering and Computer Science at the University of California, Berkeley.

From 1980 to 1983, he was a full-time Lecturer and then an Assistant Professor in the Department of Electronics Engineering, Kum-Oh Institute of Technology, Korea. In 1983, he received a Fulbright scholarship. His current research interests are in the physical design of integrated circuits, automatic cell generation, and algorithms for computer-aided design.

*

Alberto Sangiovanni-Vincentelli (M'74-SM'81-F'83) for a photograph and a biography, please see p. 913 of this issue.