AUTOMATIC DESIGN of "MULTI-CHIP MODULE" WIRING

László Hárs

December 29, 1989

Chapter 1. Introduction

In this report we describe a program based on some new concepts which automates the design of wiring in multilayer technology, especially surface mounted devices. The objective is to design a wiring (routing of disjoint paths) using several layers for the wires.

Chips are mounted on the top layer of a ceramic module. Their pins are to be connected to another pins or to the I/O pins of the module. The wires can go only in allowed channels in a certain layer. For technological reasons these channels are only of the direction North-South on some layers and East-West on the others. Wires can change layers only at allowed points.

We formulate the problem in the following way, using the constraints and constants similar to an existing module. It is similar but at some point different from the model used in [4].

We are given a 3-dimensional rectangular grid, and a list of pairs of grid-points — the *nets*. The endpoints of the nets are to be connected by paths, i.e. by a series of joining grid-edges. (The grid-edges, or for short the *edges* connect two grid-points, or for short *points* or *nodes*, if their distance is 1, the grid-displacement.) Any two paths must be *point disjoint*, i.e. they must not have a common point.

The size of the grid is fixed. The x and y coordinates range from 0 to 182, the z coordinate from 2 to 9.

The grid-points with the same z coordinate constitute a *plane*. We refer the direction of the z axis as *UP*, its opposite as *DOWN* direction. Similarly the direction of the x axis is *NORTH*, the direction of the y axis is *EAST*. Their opposites are *SOUTH* and *WEST* respectively.

Some points are of special type (they correspond to power distribution channels or to the module's pin connection). These points lay on plane 9, and are connected permanently to the points with the same x and y coordinates on all other planes. We call these points *fixed*, the others *free*.

If a point has an incident *vertical* edge, i.e. the other endpoint of an incident edge has different z coordinate, we call it *via* point.

The endpoints of the nets to be connected all have even x and y coordinates, and they lay on plane 2 or 9.

1.1. Wiring rules

The paths connecting the endpoints of the given nets cannot be arbitrary. They must obey the following rules:

1. On a plane with even z coordinate paths must go in East-West direction, except at most two non joining edges. More precisely if a subpath of a path contains only edges with all endpoints having the same even z coordinate, then the endpoints of each edge must have equal y coordinates, with the exception of at most two non joining edges. We call therefore the *even* planes *E-W* planes, too.
2. On a plane with odd z coordinate paths must go in North-South direction, except at most two non joining edges. We call therefore the *odd* planes *N-S* planes, too.
3. The endpoints of the nets and those points on other planes which have the same x, y coordinates may have at most *three* incident edges with their other endpoints on the same plane. In short, net-endpoints and the points above and below them may have at most 3 *horizontal* incident edges, their *horizontal degree* is at most 3.
4. Other points with even x and y coordinates may have *horizontal degree at most 1.*
5. Other points may have *horizontal degree 0 or 2.*
6. Only the points with even x and y coordinates may have incident vertical edges (may be vias).
7. The paths must be simple, i.e. only joining edges may have common point, their common endpoint.

If a path connects the endpoints of a net and does not violate the above rules, we call it *wire*. Connecting the endpoints of all the given nets by pairwise point-disjoint wires is the *wiring*.

The rules 1.–6. still allow loops even on a single plane, where a loop can start and end at a net endpoint, so rule 7. is necessary.

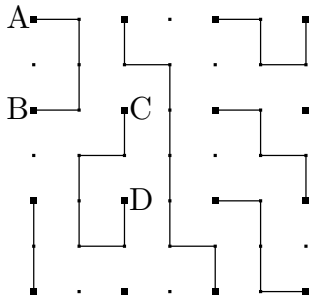

Figure 1

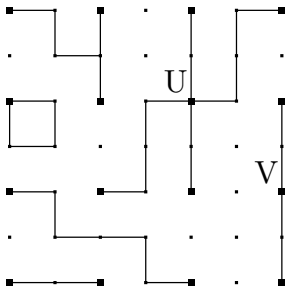

Figure 2

Figure 1 shows a part of some valid wiring on a N-S plane. The larger dots represents the points with even x and y coordinates, the smaller dots stand for the other grid-points on this plane. The wire segments AB and CD are unnecessarily long, but valid ones. Both can be substituted by straight lines of length 2 between the endpoints.

Figure 2 shows some invalid wire segments also on a N-S plane. Here node V is not a net endpoint, therefore its horizontal degree can not be 2 (rule 4.). Even if point U is a net endpoint, its horizontal degree is 4, what is not allowed (rule 3.). The bottom left wire-segment and the one just above it have two joining E-W edges (violating rule 2.). The top left wire segment has a "T junction", a point of horizontal degree 3, and this point can not be a net endpoint, since its y coordinate is odd (in contrary of rule 5.). The remaining wire segment is a loop (violating rule 7.).

1.2. Optimality of the wiring

There are several objective a wiring can aim at.

— The most important of all is minimizing the number of used planes. For technological reasons not all the planes can be saved if they do not have edges with both endpoints on them. What really matters if an even plane and the very next (odd) plane do not have horizontal edges at all, these planes may be excluded from the manufacturing of the corresponding electronic module.
— Less important is minimizing the total length of the wires, or
— Minimizing the number of bends the wires make, i.e. to make them as straight as possible, or
— To minimize the number of via points.

In the present work we concentrate mainly on reducing the number of used planes, and only after having a complete wiring on a small number of planes, we try to modify it improving on other objectives.

Chapter 2. Wiring algorithms

On the multi-chip module the majority of the net endpoints lay in the center region of the planes, where the x and y coordinates range from 40 to 142. (These region corresponds the locations of the chips on the physical modules. The remaining area contains some pins of the module for connection to the outside world, and can also be used for wiring.) We call this center region on the planes the *city*, the outer area the *country*.

In the city there are relatively few free nodes to be used as vias, and many nets are in conflict, i.e. their shortest wires would go in the same *channels* (use the lines $x = 2k + 1$ or $y = 2k + 1$, of odd coordinates). Therefore the most difficult part of the wiring algorithms is the drawing of the city-city connections. We would do it first, separately from the wiring of the remaining nets.

The size of the grid makes it difficult to handle the whole problem together, we have to divide it into smaller sub-wirings. We have chosen to solve the problems on pairs of planes from top-down fashion. The reasons of this choice are

— In a pair of consecutive planes it is possible to connect the net endpoints, if we use vias to change plane, and using the even plane for the E-W, the odd plane for the N-S wire segments.

— If a net endpoint is free, and all the incident nets are already wired, the points below it on all the lower planes become available for vias. As we proceed to lower plane-pairs, we mark these nodes usable before the wiring algorithm starts, what is much simpler than if the algorithm itself determines which points can be vias.

2.1. City-city connections

2.1.1. Allowed wires

A net can be in many different ways wired. Since both the endpoints are now in the city, where there are only few vias available and the wire channels anywhere must be used effectively, we consider only the shortest or nearly shortest wires, with at most one via. But this limits the number of feasible wires very much, the choice is too small to do the whole wiring effectively. Therefore, we allow wires to start and/or end with a wire segment of length 2 to a neighboring via point in the correct direction on the given plane. This consumes a precious via, but does not use wiring channels, since this short connections run on lines of even coordinates.

More precisely, we allow wires only on two consecutive planes, not counting the starting and finishing vertical edges. We call such a wire —connecting points A and E— an *L-wire* if it uses at most 3 via points between these two planes: B, C and D. If B exists, it is connected to A with a wire segment of length 2; if D exists, it is connected to E with a wire segment of length 2.

In one plane there are 4 possibilities for the endings of a wire segment to connect 2 points (if they can be connected at all, and their distance is at least 6): the first 2 edges can be in N-S, E-W or E-W, N-S direction, such as the last 2 edges. To be consistent, in a N-S plane the upper (North) end of wire is always an E-W edge, the lower (South) end is a N-S edge, and in an E-W plane the leftmost (West) edge is always an E-W edge, the rightmost (East) edge is always a N-S edge. With this convention if two wire segments have neighboring endpoints like the ones on a N-S plane in Figure 3, they are not in conflict.

The name L-wire comes from the global shape of the wire, what resembles the letter "L". Figure 4 shows all the edges of an L-wire drawn on one plane. Actually, wire segments AB and CD lay on the N-S (odd) plane, and BC, DE lay on the E-W (even) plane, B, C and D are vias.

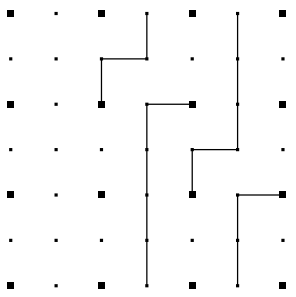

Figure 3

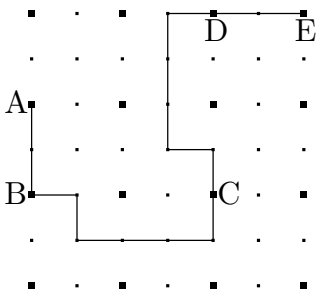

Figure 4

If C exists, according to its location the wire segment AC can use either a N-S or an E-W channel. With appropriate C, this channel can be next to A (if B does not exists) or it can be in distance 3 from A (if B exists). Similarly, there are 4 possibilities for the channel used by segment CD. With these 2 channels fixed, we have to find a via C, to make a connection between them. In general there are 4 possibilities for C, since it has to be in distance 1 from both channels. All together, we have at most 128 possible L-wires to connect the same endpoints. In practice, this number is usually smaller, because some locations for points B, C or D may not be free, or the distance of some of this points are small, so not all of the possibilities exist.

2.1.2. Graph of the L-wires

Two L-wires may not exist on the same pair of planes in the same wiring. (They may both want to use either the same point as a via, or overlapping segments of the same channel). To represent the possible conflicts, we construct a graph whose vertices are the L-wires, and two vetices are connected with an edge if the corresponding L-wires can be drawn in the same wiring.

Fortunately, it is easy to check if two L-wires can coexist. An L-wire will be coded by the coordinates of the points A, B, C, D, E, the coordinates of the first and last points of the used channels. Then conflicts can be detected with a few comparisons.

2.1.3. Weights of the L-wires

What we need for a good wiring is to find a "large" independent set of L-wires. Here the cardinality of this set is not a very good measure of "largeness", since two short wires may be easier to draw than a long one, and the long wire also must be drawn on some planes.

We can assign weights to the L-wires. The natural choice would be the distance of their endpoints. This has the drawback, that a large weighted independent set may contain L-wires which could be substituted by a shorter one, connecting the same points. It can happen because some L-wires are a little longer then others, and are equally good if we consider only the distance of the endpoints. That is the reason why we choose the weight of an L-wire to be ten times the distance of their endpoints, minus the actual length of that L-wire. Now the shorter one has larger weight, and all the wires of a longer net have larger weight than the wires of a shorter net.

2.1.4. Approximation algorithm for the maximum weighted independent set problem

For a "large" set the best would be the maximum weighted independent set, but to find that is hopelessly difficult (in computer time) [1]. Although our graph of wires has special structure, we still do not know polynomial time algorithm for it.

We apply a general purpose approximation algorithm for this problem, which is very fast and gave very good approximate solution in practice. The algorithm is based on the one of M. Carter [2], modified for weighted graphs and improved by two other heuristic steps. There are other good approximate algorithms, e.g. [3], but they are slower, and the quality of the solution does not justify their usage here.

The idea behind our algorithm is simple.

Step 1.

Repeat until the graph becomes independent (has no edges).

Delete the vertex which has the smallest weight free set.

(Here the *free set* of a node means itself and all the vertices which are not connected to it with an edge.)

After finishing this procedure, we have already a large independent set. Then we try to improve it.

Step 2.

Repeat until no more changes

Find the heaviest node not in the independent set, and not connected to any of its nodes. If exists, add to the independent set.

Find the node not in the independent set, but connected to exactly one node of the independent set, and the exchange with it gives the largest increase of the total weight. If this increase is positive, exchange these nodes.

To further improve the quality of the solution, we restart this procedure a few times (10..60). It should be prevented, that a previous solution reappears. One way to do it, to permanently delete a node from the graph of the L-wires, which one was a member of the last found independent set. We compared several versions, and the best was

Step 3.

Delete the node which has the smallest weight free set in the original graph.

2.1.5. Running time

Before starting the algorithm we can calculate for each node its degree, the weight of its free set, and chain them in a linked list. This takes $O(n^2)$ computer time, if the graph has n nodes.

Then in Step 1 we scan the remaining nodes to find the maximum degree (if it is 0, the graph is independent) and the minimum weight free set. This takes $O(n)$ time. Deleting a node from the graph takes $O(1)$ time, but updating the degrees and the free sets' weights requires $O(n)$ time in the beginning, because another scan is necessary on the nodes. Since the independent set can be much smaller then the graph's node-set,

we may have to repeat this procedure $O(n)$ times. These give $O(n^2)$ time complexity on Step 1, too.

In Step 2 one complete scan on the nodes takes $O(nI)$ time, if the cardinality of the independent set is I. It is repeated until no more improvement is found. In practice only 10..20 iterations occur, although theoretically $O(n)$ iterations are possible. Also, $I = o(n)$ in almost all the practical cases, theoretically $I = O(n)$ is possible. These may give an $O(n^3)$ time complexity, which is $O(n)$ in practice.

Deleting a node from the graph in Step 3 takes $O(n)$ time together with the necessary updatings. We restart the algorithm a constant number of times, what gives an $O(n^2)$ algorithm in practice, ($O(n^3)$ is a theoretical upper bound). In practice, the construction and initialization of the graph takes about as much time, as one iteration of the algorithm.

During the initialization process we may set up the adjacency matrix of the graph, to speed up the rest of the algorithm. But this is not much better than calling each time a subroutine to detect conflicts of the L-wires when it is required. The reason of this is simple: the algorithm needs the edges only when a node is deleted or when we scan the nodes for a possible add/exchange in Step 2. It means, during one restart of the algorithm each (directed) edge is asked at most twice. On the other hand, the graph has usually some 16000 nodes, so the adjacency matrix needs about 256 millions of bits, or 32MB. Even when we do not allocate space for the upper half of this matrix (it is symmetrical), this storage requirement approaches the limit of today's commonly used computers. Also packing the bits like this makes the access to the information slower, then not much advantage remains for this version.

2.1.6. VS-Pascal program

The program is written in VS-Pascal and ran under VMS, on an IBM mainframe computer.

The nodes of the graph are stored in the array `Node`. Each node is a record of several fields, as the listing shows below. The found large independent set will be stored in the array `Indpt`, where each element is the corresponding node's index in the array `Node`. `Indpt[0]` is the number of nodes the set has.

```
Const
      MaxX     = 182;  MaxY     = 182;
      MinZ     =   2;  MaxZ     =   9;
      MaxXY    = 183;                      {Max( MaxX, MaxY) + 1     }
      NO       = MaxXY;                    {Marks non-existing points}
      NNO      = 256 * NO;                 {MaxXY<256:2 coords packed}
      MaxNets  = 1000;                     {Maximum of given nets' nb}
      MaxNodes = 16000;                    {Max of L-Wire-graph-nodes}
```

```
Type
      Short    = packed -32768..32767;     {16-bit integer          }
      Coord    = packed 0..MaxXY;          { 8-bit integer now      }
      Coord2   = packed 0..NNO;            {16-bit integer now      }
      NodeType = record net,               {Corresponding net's index}
                        deg,               {Incident edges' number  }
                        deg0,              {        - " -       fixed}
                        next   : Short;    {Next live node's index  }
                        wght,              {Weight of the node      }
                        free,              {Wgtsum of nodes indp of v}
                        free0  : Integer;  {        - " -       fixed}
                 end;
      LType    = record nnet               {Corresponding net's index}
                   case integer of
                   1:( A, B, C, D, E : Coord2);        {A=(xa,ya),...}
                   2:( xa,ya, xb,yb, xc,yc,
                       xd,yd, xe,ye,
                       xv, yl, yu,         {V.channel:(xv,yl)-(xv,yu)}
                       yh, xl, xu: Coord); {H.channel:(xl,yh)-(xu,yh)}
                 end;
      IArray   = Array[0..MaxNets] of Short;
      NodeArray= Array[1..MaxNodes] of NodeType;
Var
      Node     : NodeArray;                {Graphnodes for Cliquefind}
      Nodes    : Integer;                  {Number of graph nodes   }
      First    : Integer;                  {Index of first graph node}
      Indpt    : IArray;                   {Nodes of the Indpt set  }
```

The following procedure performs Step 1 of our algorithm. It is called by another procedure, which performs Step 2 and Step 3.

```
(*********************************************************************)
(*                                                                   *)
(* Procedure BigWtInd                                                *)
(* Checks if the given graph is independent. If yes,returns with the *)
(* better graph. If no,deletes the worst node(lowest estimated wght) *)
(* Repeat this until the graph is independent, or the upper bound    *)
(* of the solution becomes lower than the best known indpt. set.     *)
(* An upper bound on weight of independent sets is always updated.   *)
(*                                                        28.9.1988.*)
(*********************************************************************)

Procedure BigWtInd( var  First,            {First node's index       }
                         Nds,              {Number of nodes          }
                         wopt,             {Weight of best known set }
                         wupb,             {Upperbound on lost sets  }
                         TotWght: Integer; {Total weight of Indpt set}
                    var  Indpt  : IArray); {Best set found, [0]: size }

Var  i, j, k, n, w, minfree, maxfree, mini, maxdeg : integer;

Begin
n := Nds;
repeat
  minfree := MaxInt;  maxfree := 0;  maxdeg := 0;
```

```
  i := First;
  Repeat With Node[i] do Begin              { Find min/maxfree,maxdeg }
    if free < minfree
      then Begin minfree := free; mini := i end;
    maxfree := Max(free,maxfree);
    maxdeg := Max(deg,maxdeg);
    i := next  end;
  Until i = First;

  if maxdeg = 0 then Begin                        {Graph is independent}
    if minfree > wopt then Begin
      wopt := minfree;  j := 0;                       {Better set found}
      i := First;
      Repeat With Node[i] do Begin
        j := j+1;  Indpt[j] := i;
        i := next  end;
      Until i = First;
      Indpt[0] := j;                            {Size of independent set}
    end;
    LEAVE

  end  else                                             {Not independent}
    if maxfree <= wopt then RETURN                        {Can't improve}
    else with Node[mini] do Begin
      n := n - 1;  wupb := Max(wupb,minfree);
      First := next;
      TotWght := TotWght - wght;  w := wght;
      i := First;  EdgeEnd(mini);
      While TRUE do With Node[i] do Begin
        if EdgeTo(i) then deg := deg - 1
                     else free := free - w;
        i := next; if next = mini then  LEAVE;
        next := First                                    {Close the loop}
      end;
    end;
until FALSE;
end;
```

On completion of this procedure, the indecies of the nodes of the best found independent set are stored in the array `Indpt`. The number of its nodes is stored in `Indpt[0]`.

The procedure `EdgeEnd` fixes an endpoint. Function `EdgeTo(i)` is true, if there is an edge from `Node[i]` to the previously fixed endpoint. The `LEAVE` statement causes an immediate termination of the innermost loop, which is a VS-Pascal extension of the standard Pascal language.

The following procedure calls `BigWtInd`, performs Step 2 and Step 3 of our algorithm and restarts the search `maxtry` times.

```
(*********************************************************************)
(*                                                                   *)
(* Procedure MaxWtInd                                                *)
(* Restarts some times the BigWtInd procedure to find better indepen-*)
(* dent set of L-Wires, writes out best known, marks used nets,draws *)
(* L-Wires on the planes.                                            *)
(*                                                        29.9.1988.*)
(*********************************************************************)
```

```
Procedure MaxWtInd( maxtry : Integer);
Var  wmx0, minw, mini, maxw, maxi, xw,
     LL, Last, TotWght,
     i, j, k, w, ii, j1, jj : Integer;
Label  0;

Begin
  wmx0 := 0;
  for k := 1 to maxtry do Begin
    if k > 1 then Begin                          {Remove lightest node}
      minw := Maxint;
      i := First;
      Repeat With Node[i] do Begin                    {Find min weight}
        if free0 < minw
          then Begin minw := free0; mini := i end;
        i := next  end;
      Until i = First;

      With Node[mini] do
        Begin w := wght; free := -1 end;        {-1 marks deleted node}
      TotWght := 0;  Nodes := Nodes - 1;
      for i := Nodes downto 1 do                {Find non deleted node}
        if Node[i].free >= 0 then Begin Last := i; LEAVE end;

      EdgeEnd(mini);
      for i := Last downto 1 do With Node[i] do       {Modify deg/free}
        if free >= 0 then Begin
          TotWght := TotWght + wght;
          if EdgeTo(i)
            then Begin deg0 := deg0 - 1; Edges := Edges - 1 end
            else Begin free0 := free0 - w end;
          deg := deg0; free := free0;
          next := LL; LL := i;                        {Link the nodes}
        end;
      First := LL; Node[Last].next := First;           {Circular list}
    end;

    wmx := 0;  wup := 0;
    BigWtInd( First, Nodes, wmx, wup, TotWght, Indpt);

    Repeat                                  {Add - exchange heuristics }
      maxw := 0; xw := 0;
      for i := 1 to Nodes do Begin
        EdgeEnd(i);  j1 := 0;
        for j := 1 to Indpt[0] do
          if EdgeTo(Indpt[j])
          then if (i = Indpt[j]) or ( j1 > 0)
               then GOTO 0                            {This j is bad}
               else j1 := j;                          {  - " -  good}

        if j1 > 0                               {candidate for exchange}
        then if Node[i].wght - Node[Indpt[j1]].wght > xw
             then Begin xw := Node[i].wght - Node[Indpt[j1]].wght;
               ii := i;  jj := j1  end
             else
        else if Node[i].wght > maxw                 {candidate for add}
             then Begin maxw := Node[i].wght;  maxi := i end;
    0:end; {for i}
```

```
      if maxw > 0 then Begin                                          {ADD}
        Indpt[0] := Indpt[0] + 1;
        Indpt[Indpt[0]] := maxi; wmx := wmx + maxw end
      else if xw > 0
           then Begin Indpt[jj] := ii;  wmx := wmx + xw end;
    Until (maxw = 0) & (xw = 0);                              {No changes}

    if wmx > wmx0 then Begin wmx0 := wmx; WriteMaxInd end;
  end; {for k}

  ReadMaxInd;
  TCorrection(z); TCorrection(z xor 1);
end; {MaxWtInd}
```

Here procedures `WriteMaxInd` and `ReadMaxInd` write to or read from a disk file the found maximum indepedent set respectively.

2.1.7. T-correction

In the first section of this chapter we put down a convention on the starting and ending edges of the wire segments. In most of the cases it works fine, at very rare occasions it may fail.

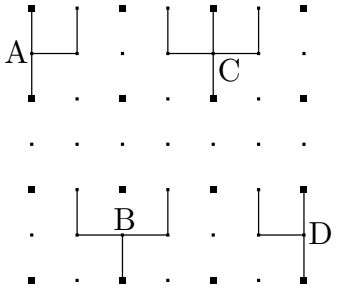

Figure 5

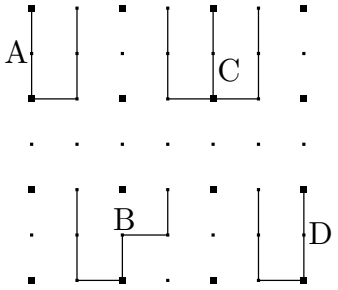

Figure 6

E.g. on a N-S plane if the wires of 2 or 3 nets start at the same point and they all go upward, they make a "T" or a "+" junction. These are shown on Figure 5. At points A, B, C and D only two edges are allowed to meet. Since the wires are coded by their channel endpoints, these L-wires are not found to be conflicting, and can exist in a maximum independent set. Fortunately, it is easy to correct these configurations, as shown on Figure 6. This correction is the task of procedure `TCorrection(z)` on plane `z`. Since we always work on consecutive pairs of even–odd planes, the other one has number `z xor 1`, where `xor` is the bitwise *exclusive or* operator.

Although it may happen that the points needed for the corrections are used by other wires, and so we would have to change those wires too, and so on, in our example wiring problems it was not the case. In the 3 or 4 cases when a correction was necessary, it succeeded without recursion, i.e. the grid-points were free where the corrected wires went.

2.1.8. Experiments

We tried several hundreds times versions of this algorithm. Our experiments show, that Step 1 gives a quite good independent set, Step 2 gains about 5% in the weight of the independent set, and restarting the algorithm according to Step 3 10..20 times finds an independent set of another 5% weight-increasing. Further restarts may give some more improvements (we tried up to 60 repetitions), but these were usually not more than 1% better then the best independent set after the twentieth iteration.

2.2. City-country, country-country connections

There are no nets connecting country-country points, because there are no chips in the country area to be connected. So actually we deal with only city-country connections, but our algorithm would work with country-country nets too. We assume, that some nets are already wired by the earlier stage of the wiring process, and they are now fixed.

The critical part of the algorithm is finding the shortest wires between two points. This will be the topic of the next chapter. Now we assume we have a subroutine which does this job either exactly or approximately. The exact solution (`procedure ShtWire`) is very slow, therefore we imposed restrictions on the wires' shape, rejecting wires which are too complex in some sense. This makes the speed of the algorithm acceptable, but some nets were reported "unwireable" when only complex wires exist for them. The approximate solution (`procedure SWire`) is much faster, but it may generate invalid wires (with loops). It is easy to detect this defect of a wire, but hard to fix it. Therefore we treat the corresponding nets as unwireable (they often are). The result is the same as with `ShtWire`, sometimes we do not find wires where we could. But it is not a serious problem, because this occurs very rarely, and if we do not draw a wire, its place is left free for others.

Because the maximum-independent-set approach was so successful for the city-city connections, the first idea we had to finish the wiring was to generate somehow a large number of possible wires for all the remaining nets, then find a large independent set among them. (Here again two wires are *independent* if they can exist in the same wiring.) But we failed with this. If there is a wire between two nodes which passes through a sparse area, i.e. where relatively few points are used by already drawn wires, there are extremely many possible other wires connecting the same points, and these differ only a little from the first one. We have to choose only some of them to be considered by the maximum independent set procedure. Several variations were tried, like

- Allow wires which are only a little longer then the shortest one. (This meant within 10–20%, no more than 10–12 edges longer, and so on.)

- Allow wires which change direction only a few more times then the straightest one.
- Allow wires which change planes only a few more times then the minimum possible. (This is similar, but different from the previous condition, since wire segments can have 0,...,4 changes of direction on a single plane.)
- Allow wires which differ from each other at least at a given number of positions. (To find the largest set of wires like this is not easy at all. Even to generate the wires in a certain order and reject the last one if there exist already a wire which does not differ from this one very much takes too long computation time.)

In all the cases the number of wires turned out to be very instable, that is changing the parameter to its next possible value at some step caused drastic changes in the number of wires. The result was we could not find a reasonable size set of wires to search for the maximum independent set. It had either too few wires, and the independent set could not be large enough, or it had so many wires that the problem did not fit the computers memory, or the wires did not differ enough from each other and the long computation gave the same result as if we had included only a few wires. These were the reasons why we used different approach at this step.

2.2.1. Multi-stage algorithm

Using the exact shortest wire algorithm it is too slow to connect the endpoints of a net, especially if one endpoint is in a sparse area, the other is in the dense city, and we start at the former one.

Therefore we divide the nets into categories, and treat them separately. The categories distinguish between the nets according the location of their endpoints.

1. Inner–inner nets, which have both of their endpoints in the central region of the city, (in the *centrum*) i.e. their x and y coordinates range from 46 to 136.
2. Inner–outer nets, which have one of their endpoints in the centrum.
3. Outer–outer nets, which have both of their endpoints outside the central part of the city.

The reason to handle these categories separately is the shortage of the free points or channels in the inner city. If a wire can find a way out of the inner city, it is almost sure, that we can continue to a point in the outer region, because naturally the L-wiring uses the border area of the city less heavily.

2.2.1.1. Stage 1. Wire segments out of the inner city

First we consider the inner–outer nets, because the city L-wiring algorithm already wired a lot of inner–inner nets, and to draw more wires of this category would cause that on this plane pair the country would be almost blank, (there were no room to start the inner–outer nets in the inner city), and on later plane-pairs there were too few inner–inner nets compared with the other category nets.

Our algorithm is simply:

Repeat for all unwired category 2 nets

Find the shortest wires from the inner city endpoint to the border of the inner city.

It is a greedy way to start this wires, and the number of found wire segments depend on the order we try to draw them. We then use a simple exchange procedure to improve this solution:

Repeat a given number of times

Delete one randomly chosen just drawn wire segment.

Repeat for the unwired category 2 nets in random order

Find the shortest wires from the inner city endpoint to the border of the inner city.

This later step is quite time consuming and usually gives no significant improvement. We used it only on the last plane pair, where there are less unwired nets, but they must be all wired there.

2.2.1.2. Stage 2. Finishing the wire segments

We must try immediately to complete half-drawn wires from Stage 1, for later some of them could not be continued, when outer–outer nets had consumed outer channels.

The unfinished wire segments end on a *side* of the inner city's square.

Repeat until the wire gets finished

If the line of this side separates the inner city and the other endpoint of the net being wired (i.e. the other endpoint of the net is "visible" from the last point of the drawn wire segment) then we simply use `ShtWire` *to finish the wire.*

Otherwise the wire has to go around the city. We determine which starting direction gives the shorter wire, and use `ShtWire` *to continue the wire to some point on the line of the neighboring side (of the inner city's square) in the chosen direction. From that last point continue the wire to the net's endpoint. (If it is visible, direct connection, else to a point of the line of the neighboring side of the centrum, and so on.)*

In some occasions it happened that the algorithm was unable to finish a wire. In this case we choosed to stop the algorithm and let a human finish the wire by hand (possibly after redrawing some other wires). Strictly speaking this is not an automatic wiring step, but this is not essential. We could just

Delete the unfinished wire segment, and continue.

After this stage of the algorithm terminated, and there were deleted wire segments (and the space they used became free),

Restart while new wires could still be added.

The hand wiring facility mentioned above will be discussed in Chapter 5.

2.2.1.3. Stage 3. Inner–inner net wiring

This stage should come next, because wires may need to go to the outer region. They still leave room for the outer–outer nets' wires, which have a lot of freedom to choose their way, but if the outer–outer nets were wired first, they would block some inner–inner wires from coming out of the city.

In this stage we simply call procedure `ShtWire` for all unwired inner–inner nets. We can hope only a few (from 2 to 5) nets to become wired, but these are precious.

2.2.1.4. Stage 4. Outer–outer net wiring

This stage remains last. Again simply

For all unwired outer–outer nets do
  If the endpoints can "see" each other (assuming that the city is not transparent), call procedure `ShtWire` *to connect them.*
  Else determine in which direction should the wire go around the city.
  Repeat until the wire is complete
    Draw the wire until some point on the line of a visible side of the inner city. (This point should be somewhere on that segment where the round tour crosses the line of the side.)

The wires now contain long straight segments, and so usually the order of the wiring does not influence the number of wires we get. (They use long segments on all available channels, and no two wires could have short segments on the same channels.) Therefore in this stage it does not pay using a last exchange cycle.

2.2.2. Single stage algorithm

Using the approximate shortest wire algorithm `SWire` we can connect the endpoints of a net in reasonable time. This allows us not to distinguish between nets, treat them all at once.

Again we use a greedy algorithm first, but restart several times to get better solution.

Wire-rest

Repeat a given number of times
  For all unwired nets do (in random order)
    Wire the net using `SWire`.
    If loopless wire found, add it to the set of wired nets.
  If the last wire-set is the best found, save it.

Then some exchange steps follow to further improve.

XChange-wires

Repeat a given number of times
If all nets are wired, STOP.
Delete a randomly chosen wire.
For all unwired nets do in random order
Wire the net using `SWire`.
If loopless wire found, add it to the set of wired nets.
If the last wire-set is the best found, save it.

These two algorithms are used differently in the case of the last and previous plane pairs. On the last plane pair only a few unwired nets are left, therefore we can afford and should do a lot of iterations of procedure XChange-wires, because they each cost very little time. But on earlier plane pairs even one iteration can take too long time. There we perform instead a larger number of iterations of algorithm Wire-rest.

Chapter 3. Shortest wire algorithms

As it was mentioned earlier we have two shortest wire algorithms which were successfully used in the wiring algorithm. We tried many versions of them, but here only the most efficient ones are discussed.

The main difficulty with finding the shortest wires is that it is not a shortest path problem in the grid graph. On a plane any point (not on the border) is connected to the four neighbor points by an edge. But not all of them can always be used, depending on the previous edge of the wire. E.g. in a N-S plane if the wire arrived from West to a point of odd x and y coordinate, it must not use the edge to East, for the wire would have two consecutive E-W edge, what is forbidden. On the other hand, if the wire arrived from South, it is OK to turn to East here. If we use a Dijkstra–type shortest path algorithm, we have to store at each node a distance value, which is updated many times until it gets its final value, the real distance from the fixed starting point. Now it is not that simple. With the previous example if the wire —which gave the smallest distance so far from the starting point— came from South, this value should not be used when calculating the distance at the point to the East of it. But we have to remember the distance which was found when the wire arrived there from West.

Actually we have to keep three copies of the nodes, that is to store three distance values for the original points: which were got when the wire came from N-S, E-W or U-D direction. These nodes have different neighbors. And it is still not sufficient because the wires are not allowed to make loops. If we just follow a Dijkstra type procedure, to choose the point which is connected to the nodes with already determined distance by an edge which gives the smallest distance value to this node, we could easily get a loop: $(0,0,3) \to (0,1,3) \to (0,2,3) \to (0,2,2) \to (1,2,2) \to (1,3,2) \to (0,3,2) \to (0,2,2) \to (0,2,3) \to (0,3,3) \to (0,4,3)$. Here at each node where edges join the wire

is locally valid, but we got a loop $(0,2,3) \to (0,2,2) \to (1,2,2) \to (1,3,2) \to (0,3,2) \to (0,2,2) \to (0,2,3)$. We can not cut off this loop from the wire, because then the node $(0,2,3)$ would have horizontal degree 2, what is forbidden.

The points $(0,0,3)$ and $(0,4,3)$ may not be connectable by wires (e.g when all the nearby channels are occupied except those used by the above cyclic wire). If this is the case, no minimality criterion could exclude the looping wires.

These show that finding the shortest wire between two points even on consecutive even-odd plane pairs is different from finding the shortest path.

3.1. Exact shortest wire algorithm

To be able to exclude looping wires we use backtracking. A wire is built up from the starting point until the target is reached. In a dead end path we step back and try other directions:

For all possible starting plane/direction pairs do

GO: If target is hit, SAVE the wire.
If this node is in current wire or fixed, try OTHER edge.
If the wire reaches the limit of badness, try OTHER edge.
If the wire is longer than the shortest one found earlier, try OTHER edge.
Store the distance from the starting point to this one with the dirrection the wire came from.
Take an incident edge and GO one step further.

SAVE: Store the nodes and the length of the wire.
Try OTHER edge.

OTHER: If we stepped back to the starting point, begin processing the next plane/direction pair.
Take the next, in this wire unused incident edge.
If it exists, GO on with this edge, else step BACK.

BACK: Remove last edge of the wire.
Continue with an OTHER edge.

In this algorithm it is not specified what should be the target, in which order the incident edges are taken at a point, what is the cost of an edge. These can vary according to the application.

3.1.1. Target

- If the task is to find a way out of the inner city, the target is any point on the sides of the inner city.
- If the procedure is used to go around the city, the target is any point on the continuation of a visible side of the city, in the desired direction.
- If the task is to connect two points, the target is naturally the other point, where the wire should end.

3.1.2. Order of the edges

The algorithm works with any order of the edges, but with different running time. The best was to chose the edge which makes the largest reduction in the distance between the wire end and the target. With this choice the wire goes in the wrong direction only when it is necessary. Of course one can easily construct an example where this order performs poorly, but most of the time this is the best we can do.

3.1.3. Edge costs

With giving the edges costs we can put preferences to some wires. The via holes are precious in the city, therefore we can give the edges higher costs if they change planes there. Also, if the wire changes direction, we can add extra cost to the last edge. We prefer wires going farther from the dense center of the city, so we can give larger costs to the edges which are closer to the center.

3.1.4. Starting point

In the case of connecting two points the algorithm may have different running time if we exchange the starting and finishing points. Roughly speaking, if the wire starts in a large blank area, and has to go through a little “door” to get in a densely wired area, where the endpoint lays, the algorithm will spend a lot of time to find that little door, calculating unnecessarily the distances of almost all nodes in the blank area.

But if the wire starts inside the dense area, there are only a few nodes are reachable, it will soon gets out of the bush, and can head to the endpoint. It will find the shortest segment immediately, and needs only a little time to confirm it, since it does not continue a wire if it gets longer then the shortest one already found.

Therefore we always choose for the starting point the one, which is closer to the center of the grid, since it is more likely in a denser area.

3.1.5. Exclusion of too complex wires

At each step of growing the wires we can assign to the edges a "badness" value. These sum up as the wire expand and if it reaches a given limit, we consider the wire to be bad, and try another edge.

This can reduce the running time of the algorithm, reducing vastly the number of wires to be considered. Usually there are much more wires which are "complex" than "simple". But sometimes the algorithm will not find any wire when only complex wires exist, or the found one is not the shortest (in the case the shortest wire was too bad and rejected). But still the found wire is valid.

3.1.6. VS-Pascal program

```
Type
      PontType = record x, y, z : Coord end;
Const
      MaxWireSize = 5000;
Var
      Pont : Array[1..MaxWireSize] of PontType;
      Next : Array[0..MaxX,0..MaxY,2..3] of Byte;  {Bit i set if wire-}
                                                   {edge goes in dir i}

(*********************************************************************)
(*                                                                   *)
(*  Procedure ShtWire                                                *)
(*                                                                   *)
(*  Finds the shortest wire starting at point A until HIT, on planes *)
(*  2, 3; allowing at most BadLim badness                            *)
(*                                                        10.10.1988.*)
(*********************************************************************)

Procedure ShtWire(xa, ya, za,                    {Starting point      }
                  BadLim : Integer;              {Max badness         }
              var Size,                          {Wire's node number  }
                  WireLen: Integer);             {Shortest wirelength }
Const inf  = 65535;
      used = 255;
      N = 1;  E = 2;  S = 3;  W = 4;  U = 5;  D = 6;       {Directions}
Type  HalfWord = packed 0..inf;
Var   i, j,
      x, y, z,                                   {Coordinates         }
      bad,                                       {Total badnes        }
      dir,                                       {Wire came from      }
      edge,                                      {Last used edge, 1..4}
      Quad,                                      {Target's direction  }
      StTop,                                     {Stack-top index     }
      Length : Integer;                          {Wire length to here }
      Dist : Array[0..MaxX,0..MaxY,2..3,1..6] of HalfWord;
```

```
Label Go, Save, Other, Back;
Begin
  WireLen := MaxInt;
                                                        {Initialization}
  for x := 0 to MaxX do for y := 0 to MaxY do for z := 2 to 3 do
    if Free(x,y,z) & (Next[x,y,z] = 0)
    then for i := 1 to 6 do  Dist[x,y,z,i] := 65535
    else for i := 1 to 6 do  Dist[x,y,z,i] := 0;  {Mark Used/fix nodes}

  for i := 1 to 8 do Begin
    case i of
     1: Begin  za := 2; dir := N  end;
     2: Begin  za := 2; dir := E  end;
     3: Begin  za := 2; dir := S  end;
     4: Begin  za := 2; dir := W  end;
     5: Begin  za := 3; dir := N  end;
     6: Begin  za := 3; dir := E  end;
     7: Begin  za := 3; dir := S  end;
     8: Begin  za := 3; dir := W  end;
    end; {case i}
    x := xa;  y := ya;  z := za;
    Quad := SetQuad(xa,ya,za);
    StTop := 0;  bad := 0;
    Length := 0;
    Dist[x,y,z,dir] := Length + 1;                          {for start}

  Go:                                                  {Go to a given dir}
    if Dist[x,y,z,dir] <= Length then GOTO Other;
    Dist[x,y,z,dir] := Length;
    Length := Length + EdgeLen( x,y,z, dir);
    case dir of
      E : Begin x := x - 1; if x < 0 then GOTO Other end;
      W : Begin x := x + 1; if x > MaxX then GOTO Other end;
      N : Begin y := y - 1; if y < 0 then GOTO Other end;
      S : Begin y := y + 1; if y > MaxY then GOTO Other end;
      U : z := z - 1;
      D : z := z + 1;
    end;

    bad := bad + Badness( x,y,z, dir);

    if Hit(x,y,z) then GOTO Save;                              {Found}
    if bad > BadLim then GOTO Other;                       {Too complex}
    if Next[x,y,z] = used then GOTO Other;               {Already on wire}
    if Dist[x,y,z,1] = 0 then GOTO Other;                {Not free to use}

    edge := 1;                                          {first edge chosen}
    Push( StTop, x,y,z, Quad, bad, dir, edge, Length);
    Next[x,y,z] := used;
    Quad := SetQuad(x,y,z);
    dir := Step(Quad,x&1,y&1,z&1,dir,edge);
    if Length < WireLen then GOTO Go else GOTO Other;

  Save:
    WireLen := Length;
    With Pont[1] do Begin x := xa; y := ya; z := za end;
    Size := StTop + 2;
    Pont[Size].x := x; Pont[Size].y := y; Pont[Size].z := z;
    for j := StTop+1 downto 2 do
      With Pont[j] do Pop( StTop, x,y,z, Quad, bad, dir, edge, Length);
    StTop := Size - 2;                                    {Restore stack}
    GOTO Other;
```

```
  Other:                                              {Go to another dirtn}
    if StTop = 0 then CONTINUE;
    Top( StTop, x,y,z, Quad, bad, dir, edge, Length);       {Last node}
    edge := edge + 1;                                            {Next}
    dir := Step(Quad,x&1,y&1,z&1,dir,edge);
    if dir = 0
      then GOTO Back                                {No more incident edge}
      else GOTO Go;
    end;
  Back:                                                    {Go a step back}
    Next[x,y,z] := 0;                                 {Mark node as unused}
    StTop := StTop - 1;                                   {Remove stacktop}
    GOTO Other;
  end; {for i}
end;
```

In this procedure some other subroutines are called.

- The Boolean function `Free` tells if the given node is free or not.
- `Push, Pop, Top`. The first parameter of these procedures is the stackpointer, which is incremented, decremented, left unaltered respectively. `Push` stores the parameters, `Pop, Top` loads the parameters from the stack.
- The integer function `Step` is actually a large constant array. For each quadrant (in that the target lays) and point, according to the direction the wire came from it contains the directions the wire can continue going. These directions are in the correct order and can be accesed setting the last parameter to 1, 2, ... After the last usable direction it returns 0, indicating the end of the possibilities. For the directions only the parities of the coordinates are necessary. This is why we have in the program `x&1,y&1,z&1`. The `&` operator is the bitwise logical *and*.

The following 4 subroutines are to be changed according to the application of the `ShtWire` procedure.

- The Boolean function `Hit` returns a TRUE value iff the wire reached the target (e.g. if x = xe, y = ye, the target point).
- The integer function `EdgeLen` gives the length of the edge starting at the given point and going to the given direction. In the simplest case it is always 1.
- The integer function `Badness` gives the badness of the edge given with the starting node and the direction. If it is always 0, no badness is calculated, and we have the normal shortest wire procedure.
- The integer function `SetQuad` determines the direction of the target (in which plane-quadrant of the given point it lays). This information is used to determine the order of the edges incident to the given point.

3.2. Approximate shortest wire procedure

The procedure essentially a modified version of Dijkstra's shortest path algorithm, see [5]. Suppose, the edges have nonnegative length, and these length are all less then a given constant k. Then the algorithm works as follows:

Let the starting point get distance 0, and form alone List(0).
Let the current list be List(0).
Repeat with next list, until all lists are empty
Repeat until the current list gets empty
Remove last node v from the current list.
If it HITs the target, Find and Save the wire. STOP.
If v is not processed earlier,
For all its neighbors do
If neighbor is free and the edge to it is allowed, add neighbor to the List(current distance + edge length).
If all lists are empty and target is not HIT, report graph is disconnected.

Dijkstra's algorithm require at each step to attach to the determined-distance node set a new node, if it would get the smallest determined-distance. In our algorithm the first nonempty list contains these nodes.

3.2.1. Running time

We assumed that the edges are shorter then a constant length k. Therefore finding the next nonempty list takes at most k comparisons, because we have processed all the nodes with smaller distance than the current distance, and some nodes with the current distance, and non of their neighbors can be in distance larger than the current distance $+ k$. Only these nodes get in the lists.

At each node removed from a list the work to be done is proportional to the number of its neighbors, if it is processed here. Otherwise only a constant number of steps are done here. A node can get into a list only when one of its neighbors is processed, therefore at most as many times as the number of the neighbors.

All together the number of steps the algorithm performs until it determines the distance of the target is $O(E)$, proportional to the number of edges. To find the wire of this distance the same amount of time is enough:

Starting at the target until the starting point is reached do
Check all neighbors of the current node until one's distance + the edge length to the current node is equal to the current node's distance from the starting point
Let this neighbor be the current node.

(Another possibility is to save the neighbor the wire came from at each node when it gets its distance. This saves time to construct the path, but uses time and memory to store these neighbors in the main body of the algorithm.)

Therefore our algorithm works in $O(E)$ time. In our case, the degree of every node is at most 6, so the running time is $O(n)$, the best possible.

3.2.2. Memory requirements

We have to store only at most k lists. In an array of length k the index (or address) of the last elements of these lists are stored. $List(i)$ (the list of the distance i) corresponds to the position i mod k in this array. (If k is a power of 2, a bitwise *and* with a mask will do in place of the modulo.)

The lists themselves can be linked lists. In this case, the total number of cells is at most $2E$, because a node can appear in the lists at most d times.

In the program a simpler version is used. We store the nodes in a $k \times 4n$ array. The rows of this array correspond to the distances (mod k), and new nodes are added to the end of the corresponding row. Longer than $4n$ length rows are not necessary if the graph has n nodes, since the maximum degree is 4. (Actually a much smaller number is used, what proved to be enough in practice.)

3.2.3. Edge costs

Again we can influence the found shortest path with manipulating the edge costs. E.g. higher costs in the center, for vias, etc.

Another possibility is to use "future costs". (See [6] and [7]). It will give the same shortest path as if the normal lengths are used for costs, but faster in the average. The future cost means to add to each *directed* edge the increase of the distance from the target, if we go through the edge. (The distances should be measured in units, which allow avoiding negative edge costs. If the distance decreases at some edge, we had to add a negative value to its length. It must not become negative, since with negative edge costs the Dijkstra-type algorithms do not work.)

We applied this future cost edge length function in our program. It gave some improvement in the running time of this part (about 30%), although in other applications it gives usually much more speed up. In our program, in most of the cases the nets are unwireable, and to confirm this all the nodes in a connected component must be processed. To calculate the future costs takes a bit more time than the normal costs, and so no speed up is achieved at failures. The successful wire constructions finished really faster.

At the present implementation the maximum edge length was 14 (the minimum is 0). Therefore $k = 16$ is a good choice, and the mod k is replaced by just taking the last 4 bits of the numbers.

3.2.4. Loops, nodes

As we already discussed, the shortest wire problem is not a shortest path problem, because edges depend on from where the wire arrives at a node. One possibility to handle this problem, to replace each node of the grid with some new ones. These are (*node,direction*) pairs, where that direction is considered from where the wire gets to the node. These new nodes have different neighbors.

This can ensure local validness of the wires, but now the shortest path algorithms can produce looping wires. A path can come back to a different copy of the same node (from different dirrection) and can continue its way on an edge, which was not allowed at the first copy of the node. It shows, that we cannot simply cut the loop off the wire.

Because a loop increases the length of a wire, it usually occurs, when no loopless wire exists. The exceptions are the possible shortest loops in a plane: one edge in both directions or a square of sidelength 1. The first can easily be excluded, if we distinguish the copies of a node with direction South and direction North (or East/West), but the square remains possible. It does not add too much to the length of a wire, so the algorithm could produce this instead of a bit longer valid wire.

To prevent this short loops get in the wires, we chose to have even more copies of the grid nodes. The nodes of our actually used graph are (*node,tail*) pairs, where the "node" stand for a grid node, the "tail" means the 3 last edges of a valid wire. Now, with correct definition of the edges we can exclude the square loops, too. There are many possible tails, but at a given node not all are valid. Also, some behave similarly, we need not differentiate between them. It turned out, that considering at most 12 tails is enough. Defining the new edges is a tedious job. We set up some tables (constant arrays). It is sufficient to take the coordinates of the nodes modulo 2. One table shows at a given (*node mod 2,tail-code*) pair what directions are valid to go on, another table gives there the new tail-codes. Further tables are needed to tell from which (*node mod 2,tail-code*) pair could the wire get here (for finding the wire at the end).

3.2.5. VS-Pascal program

```
Type  NdType = record x,y,z,t : Coord end;
Function Nd(x0,y0,z0,t0 : Coord): NdType;              {Type conversion}
Begin Nd.x := x0; Nd.y := y0; Nd.z := z0; Nd.t := t0 end;
Function Hit( v: NdType): Boolean;                     {Target reached?}
Function Len( p, q: NdType): Short;                        {Edge length}
Function Dist( v: NdType): Short;                      {Distance from A}
Procedure SetDist( v: NdType; dist: Short);               {Set distance}
Function FirstNbor( v: NdType): NdType;            {First adjacent node}
Function NextNbor: NdType;                          {Next adjacent node}
Function NoMoreNbor: Boolean;                      {All nbors are used?}
Procedure PathFind( v: NdType; Var loop: Boolean);{Find/Save wirenodes}
```

```
(*********************************************************************)
(*                                                                   *)
(*  Procedure SWire                                                  *)
(*                                                                   *)
(* Finds the shortest wire between points A and E with starting tail *)
(* TA. Uses modified Dijkstra, (for small costs) with future-cost.   *)
(* Nodes have 12 copies corresponding to the endings of wires there. *)
(*                                                    18.01.1989.*)
(*********************************************************************)
Procedure SWire( xa,ya,za, ta,                    {Starting point, tail}
                 xe,ye,ze : Coord;                {Target point        }
             Var Size,                            {Wire's nodes' number}
                 WireLen,                         {Wire's length       }
                 OK : Integer);                   {0:loop,1:OK,2:discon}
Const
      MaxT = 11;                                  {Wire tails:  0..MaxT}
      MaxL = 4;                                   {Bits of Max Len,1..8}
      Mask = 15;                                  {2^MaxL - 1          }
Var
      i,l,m,t : Integer;
      MinD, MD: Short;
      x, y, z : Coord;
      v, u    : NdType;
      loop    : Boolean;
      List : Array[0..Mask,1..50*MaxXY] of NdType;
      Last : Array[0..Mask] of Short;
Begin
  for x := 0 to MaxX do  for y := 0 to MaxY do  for z := 2 to 3 do
  for t := 0 to MaxT do  SetDist( Nd(x,y,z,t), inf);
  for i := 0 to Mask do Last[i] := 0;                {Initializations}
  MinD := 0; MD := 0;                              {Min unfinished dist}
  v := Nd(xa,ya,za,ta); SetDist(v,0);                 {Start at point A}
  List[0,1] := v; Last[0] := 1;                          {A into list0}
 Repeat
  While Last[MD] > 0 do Begin                 {More points in MinD dist}
    v := List[MD,Last[MD]];
    if Hit(v) then Begin                                      {Found!}
      PathFind(v,loop);
      if loop then OK := 0
      else Begin WireLen := MinD; OK := 1 end;
      RETURN end;
    Last[MD] := Last[MD] - 1;                           {No HIT: Delete}
    if Dist(v) = MinD then Begin                          {Process now?}
      u := FirstNbor(v);
      Repeat                                          {Nbors into lists}
        l := Len(v,u); m := MinD + l;
        If (l < Mask) & (Dist(u) > m) then Begin      {l=Mask:forbidden}
          SetDist(u, m);
          i := m & Mask;
          Last[i] := Last[i] + 1;
          List[i,Last[i]] := u;
        end;
        u := NextNbor;
      Until NoMoreNbor;
    end; {if}
  end; {While}
```

```
  m := MinD + Mask;                                          {Dists' limit }
  Repeat                                                     {Next distance}
    MinD := MinD + 1;  MD := MinD & Mask;
    If MinD > m then Begin OK := 2; RETURN end;               {Disconnected}
  Until Last[MD] > 0;
 Until FALSE;
end;
```

Chapter 4. Secondary optimization

As it was mentioned earlier, after the wiring is done on the smallest possible number of planes, we can improve on it taking other objectives into considerations, too.

4.1. Z-correction

The L-wires, according to our convention, can make at one end an unnecessary turn ("Z" ending: e.g $W \to E, N \to S, W \to E$ instead of $W \to E, W \to E, N \to S$). A simple cycle on each plane can find and straighten them out (if the point is not used by another net, where the corrected wires would go).

4.2. V-correction

If a via is under a different net's endpoint, (has the same x and y coordinates) it generates a warning in the manufacturers program, because they may want later to use that vertical channel for some test or extension purposes. Therefore we have to reduce the number of vias of this type.

Again a simple cycle searches for this vias. If one found, we remove the whole wire it belongs to. Then, giving a very large costs to the undesirable vias, we redraw the wire. We use `SWire` for this. If it fails finding a correct wire, we can not do better than redraw the original wire which was removed.

If bad vias remain, the hand drawing facility is called, with the cursor positioned at one of them. A human may redraw some nearby wires to make room for eliminating the undesirable via.

Chapter 5. Hand wiring facility

With the earlier versions of the program it sometimes happened, that some wires remained unfinished, or on the last plane pairs some wires were still reported unwireable. To handle this situations the hand wiring facility was developed. Even when perfect wiring was found, sometimes it was desirable to make minor changes, or just to look at the solution. That is why the hand drawing facility was kept in the last version of the program, too.

It is an interactive subroutine package written in VS-Pascal and run under VMS operating system on an IBM mainframe computer. Although graphic terminals are available, they are accessible only through GDDM, a huge graphics programming package. With this, the conversation became so slow, it was practically useless.

That is why we chose to use normal alphanumeric terminals in character mode. To represent nodes and wires, the semigraphic character set is enough. The program draws a piece of a plane on the screen. (A window is opened to the grid.) On the last line some information are written (which part is shown, which plane, what net is to be wired). When the picture is ready, a command can be entered. The following commands are interpreted:

- Show help screen. The available commands and their syntax is written on the screen. Pressing the Enter key the previous grid part comes back to the screen.
- Screen moovement. We can move the window to any direction to any position by specifying the new top or right edge's coordinates or the relative displacement by the window moves. The default displacement is the screen size.
- Change plane. Show the same part of the given plane. The default (when no plane number is given) is the pair of the currently shown plane.

- Show net. On the last line of the window the endpoints of a net are written. We can change this net to the next one, previous one, the next undrawn one or the previous undrawn one, using the order the nets were given.
- Move cursor. The cursor can be positioned to any absolute position on the plane or can be moved by relative displacement (when + or - sign are written before the numbers.)
- Draw line. A straight wire segment is drawn to the given direction (parallel to the x or y axis) to an absolute position or of the given length. The wire stops if it hits an already drawn other one on the same channel.
- Delete line. Deletes a straight wire segment in the given direction to an absolute position or to a given length. If the wire segment is shorter, de deletion stops at the last point.
- Delete wire segmet. Deletes the whole wire segment in the current plane starting at the cursor position.
- Delete wire. Deletes the whole wire from the current pair of planes, starting at the cursor position.
- Clear planes. Removes all the wires from all the planes. Before doing it asks if it was really intended. If the answer is not "y" or "Y", the planes are not cleared.
- Print planes on laser printer. The given or current planes are plotted on a QMS laser printer, together with the wires. The printing starts immediately, without terminating the program. This is very handy when the wiring is too complex to take in from the screen.
- Write data to disk. The partial results of the wiring can be written to the disk. With it the program can continue working from this point at any latter time.
- Read data from disk. Read back partial results of the wiring from the disk. Then the program can continue its run at the point where the data were saved.
- Show/print/clear previous messages. The program generates messages on the quality or stage of the partial wiring it works on, or if it is unable to finish a wire, cannot correct a "T" connection, etc. These messages are printed on the screen and also are stored in a disk file. On request these can be printed out, shown again on the screen or cleared from the disk.
- Construct L-wires. This is actually the first step of the algorithm. The codes of the generated L-wires are stored on disk, too.
- Build graph of L-wires. The degrees and free set sizes are calculated and stored on disk. (The adjacency matrix is too large to set up and store.)
- Find maximum weight independent set. The corresponding algorithm is called, the result stored on disk.
- Transform L-wires to paths. Decode L-wire information for enable the program to remove, redraw L-wires like the wires generated by procedures `SWire` or `ShtWire`.
- Wire remaining nets. The *Wire-rest* algorithm of section 2.2.2. is run to try to wire the undrawn nets.
- Wire-exchange. The *XChange-wires* algorithm of section 2.2.2. is run to try to wire more undrawn nets.
- Change/restore free node information. Marks a net endpoint free on the current

plane pair, if all incident nets are wired on higher planes. (These nodes are to be used only if absolutely necessary.)

- Do Z-correction. Straighten the endings of the wires on the current plane if the nodes are free for the correction.
- Do V-correction. Modify the wiring to avoid using the nodes as vias, which are not free on the topmost plane.
- Quit. Stop the program.

After a command completes, the screen is redrawn, and other commands can be entered. Some parts of the program generate messages. These are written on the screen, after the grid is cleared from it, and also are stored in a disk file. This messages are like the size of the last found independent set, the number of nodes the graph under construction has, etc.

References

[1] M. Garey and D. Jonson "Computers and intractability: A guide to the theory of NP-completeness," *W.H. Freeman & Co., San Francisco,* **1979**.

[2] M. Carter "A practical algorithm for finding the largest clique in a graph," *Working paper, Dept. Industrial Engineering University of Toronto* **1984**(#84-08).

[3] R.E. Tarjan and A.E. Trojanowski "Finding a maximum independent set," *SIAM J. on computing,* **6 No 3**(1977), pp. 537–546.

[4] H. Mueller "Automatic multilayer routing for surface mounted technology," *ISCAS'88,* **IEEE**(1988), pp. 2189–2192.

[5] A.V. Aho, J.E. Hopcroft and J.D. Ullman "The design and analysis of computer algorithms," *Addison-Wesley, Reading, MA* (1974).

[6] P.E. Hart, N.J. Nilsson and B. Raphael "A formal basis for the heuristic determination of minimum test paths," *IEEE Trans. Systems Science and Cybernetics,* **SSC-4/2**(1968), pp. 100–107.

[7] A. Martelli "On the complexity of admissible search heuristics," *Artificial Inteligence* **8**(1977), pp. 1–13.