Fast Printed Circuit Board Routing

Jeremy Dion

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Abstract

This report describes the problem of printed circuit board routing. An overview of circuit board construction is given. The algorithms in a printed circuit board router used for fully automatic routing of high-density circuit boards are described. Running times of a few minutes have resulted from a new data structure for efficient representation of the routing grid, quick searches for optimal solutions, and generalizations of Lee's algorithm for maze routing.

1. Introduction

Even though printed circuit board routing is a venerable problem in computer-aided design, fully automatic routing of densely packed boards remains an elusive goal. In current industry practice, a program is used to make most connections automatically. The remainder is left for manual completion. This procedure is a poor second to fully automatic routing. It leaves the possibility for introducing errors in the routing of the final connections. More seriously, it is an investment in time and effort that makes subsequent logic changes more difficult.

It is always easy to specify a routing problem that is too hard for a program to solve. One need only add wiring to the problem, or remove routing layers. In this sense, designing a completely automatic router is an impossible task. A better program will simply encourage engineers to design harder problems. The only realistic goal for a routing program is to solve practical layout problems well enough that manual intervention is unnecessary.

This report describes the printed circuit board router *grr* (*gr*eedy *r*outer). *Grr* was developed during the construction of the Titan computer [Nielsen 86], a high-performance scientific workstation designed at the Digital Equipment Western Research Laboratory. It was used to route all thirteen boards of the Titan, with run times of 5 to 30 minutes of VAX 11/785 CPU time and no failures.

2. Printed Circuit Boards

A digital logic circuit consists of a collection of interconnected *parts*. Most parts are integrated circuits, but in practical designs there are usually a number of passive components and connectors. Each part has one or more *pins*, which are terminals at which electrical connections are made. Integrated circuits may have hundreds of pins on a single package. The pins are interconnected by *nets*. Each net is a collection of pins that must be electrically interconnected in the final realization.

A circuit board is built as a stack of layer pairs. Each pair starts as an insulating sheet with copper deposited on one or both sides. The copper sides of the sheet are first etched with different wiring patterns. Then the sheets are stacked into a sandwich separated by insulating material. Small holes are drilled into the board. Finally, the holes are plated with metal, so that electrical contact is made with each layer that has copper left at the hole location. This means that a hole can form a conductive path between two or more layers called a *via*.

There are two ways of attaching parts to boards. The current industry standard is to solder the pins into a via pattern in the circuit board[1]. The solder connects the pin to any layers that have metal pads at the via locations. This method is simple and reliable, but does not allow the pins to be packed very closely. Spacings of 100 mils (thousandths of an inch) are common for through-hole pins. The second newer method is surface mounting. In this technique, the pins are flat strips that are soldered to the surface of the board without penetrating it. Surface mounting allows increased pin density, and so larger numbers of pins for a given package size. Surface mounting also leads to a harder routing problem, but not one that is qualitatively different. In the body of this report, only through-hole pins will be considered, and surface

[1]In more modern PCB technologies, the drill holes used for vias can be made very much smaller than those in which pins must be inserted. This can result in substantial increases in via density, but does not substantially affect the problem or algorithms described here.

mounting will be revisited in the Limitations section. For our purposes, the term *via* will mean *either* a hole that connects two or more signal layers, or a hole that connects a pin of a part to one or more signal layers.

In all but the simplest printed circuit boards, the nets that interconnect the pins of chips are divided into two classes. A small number are singled out for special treatment as *power nets*. These supply power and ground to the parts on the board, and nearly every part will be connected to at least two of them. Because a power net carries large currents, one or more layers of the circuit board are devoted entirely to it, and become *power layers*. It is not uncommon to have one power layer for each of the supply voltages, and several for ground. In multi-layer circuit boards, often half of the copper layers are reserved for power and ground.

The etching pattern for power layers is simple. The layer is left as solid copper except at pin and via locations that are not to be connected to the power net. At these locations, a small disk is etched away so that no electrical contact will be made during drilling and plating. Figure 22 on page 25 shows an example of a power layer. The generation of power layer patterns is straightforward once the complete pattern of vias is known. More details are given in the appendix.

The remainder of the nets are *signal nets*, which carry the digital logic values. These connect to far fewer pins than the power nets and have small current flows, but there are thousands of them. Signal nets are routed on the remaining *signal layers* by adding *traces* and vias to the circuit board. A trace is a thin wire lying entirely on one signal layer that is formed by etching away the surrounding metal. Figure 21 on page 24 shows a signal layer.

Figure 1 shows actual board dimensions for an example process. In the figure, traces must be at least 8 mils wide and at least 8 mils apart. Pads for vias must be 60 mils in diameter to allow for error in drilling and plating a 37 mil via.

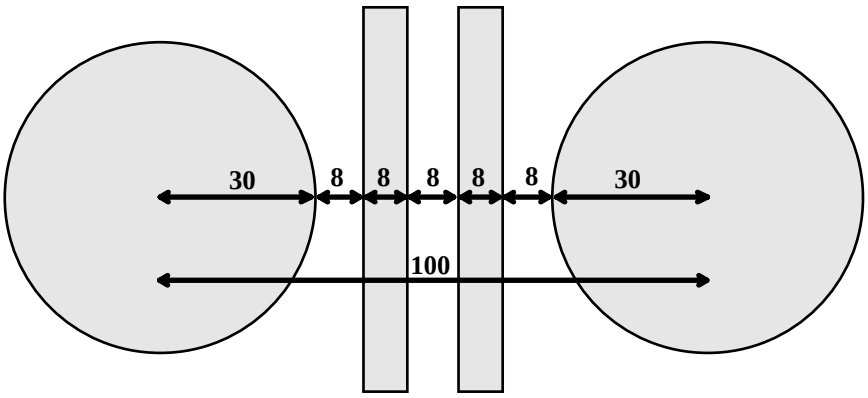

Figure 1: Printed Circuit Board Dimensions

3. Stringing

Any graph of traces and vias may be chosen to connect a net subject to the constraints of technology [Soukup 81]. Some logic families, such as TTL, permit arbitrary interconnection patterns. Others, such as ECL, require nets to be transmission lines, so that the pins of a net must be connected in a chain, with the output at one end and a terminating resistor at the other end.

The router described here was optimized for ECL circuits. (In fact it was designed to replace the inadequate commercial router that had been used in the first attempts to route the Titan boards.) Because of this, a very simple approach to net topology was taken. Nets are connected as chains, and the connection order is chosen by a separate program, the *stringer*, which prepares input for the router.

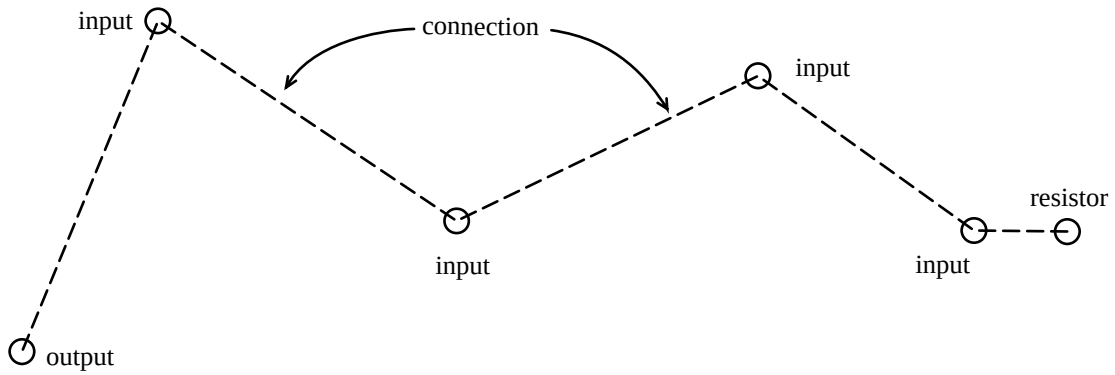

Figure 2: Stringing an ECL Net

Stringing is done as shown in figure 2. Starting at the output pin for the net, the next nearest input pin is repeatedly added to the chain, until the whole net has been connected. Then for ECL nets, the nearest free terminating resistor is added to the end of the net. Sometimes ECL nets have multiple output pins. Any output may start the chain, but all output pins must precede the input pins. In this case, and for TTL nets where pin order is not important, the stringing is repeated for each legal starting pin. The shortest overall path is then chosen.

Stringing prior to routing results in a simplification of problem presented to *grr*. The router input consists of a number of pin-to-pin *connections*, which can be considered independently and in any order. Any realization that makes the required connections will connect the nets correctly. In the remainder of this report, only pin-to-pin connections will be discussed, and their relation to nets will be ignored. Figure 20 on page 23 shows a typical example of router input produced by the stringer. Each line in this figure represents one connection in a signal net.

It is clear that this stringing algorithm is suboptimal. TTL allows nets to be joined by trees, not just chains. Also pin order is decided before any routing congestion information is known. It is clear from experience that net ordering is very important. In one experiment, the router was given two versions of the same routing problem that differed only in the stringing. In one, the stringing was chosen by the method described above. In the other, it was random. The router completed both problems successfully, but there was factor of 25 difference in the run times. The random problem took 50 minutes of CPU time, and the better ordered problem took 2 minutes.

4. Data Representation

The choice of a data representation is fundamental and critical. For printed circuit board routing, the choice lies in the generality of the patterns that must be represented. If arbitrary width wires at arbitrary angles are required, and vias may be at any location on the circuit board, then a very general list structure must be used. This structure will be expensive to search and update. However, if restrictions can be

placed on the location of vias and wires, then more efficient data representations can be used. This trade-off between generality and efficiency is typical of design automation problems.

The major restriction in *grr* is to introduce a routing grid on which all traces must lie. The points of the grid are spaced so that parallel traces on adjacent grid lines are legal under the manufacturing rules. As a further simplification, only rectilinear traces are allowed and diagonals are forbidden[2].

Vias and pins are restricted to lie at regular intervals on the routing grid, on a coarser via grid. This via grid must be fine enough to represent the pin arrangements on all parts that will be used. In Figure 3, an open circle denotes a point on both the routing and via grids. A small filled circle denotes a point on the routing grid only.

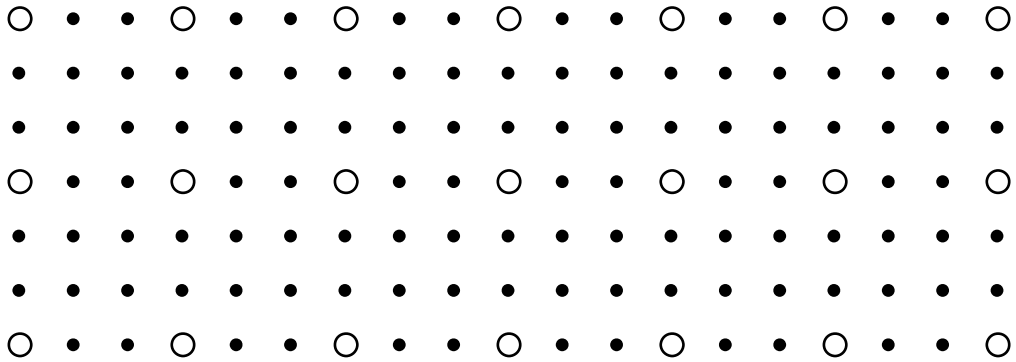

Figure 3: The Routing Grid

The grid of Figure 3 matches the manufacturing process illustrated in Figure 1. The embedding of the via grid in the routing grid was chosen as follows. The minimum pin pitch on the parts to be used was 100 mils. This fixed the via grid at 100 mils between adjacent via points. The fabrication process allows two signal traces between vias at this pitch. So two routing points could be placed between via points. Repeated symmetrically in two dimensions, these constraints produced Figure 3.

Note that the routing grid is irregularly spaced. Between a via point and the nearest routing point there is a space of 42 mils. Between two adjacent routing points there is a space of 16 mils. Because of this distortion the model can't represent the 4 minimum-spaced traces that could be put in place of a single 60 mil via pad. This means that the grid model cannot represent wiring at maximum density.

The second restriction for efficiency is that on any particular layer, traces are presumed to be predominantly horizontal or vertical. In any circuit board, therefore, one or more horizontal and one or more vertical layers are required.

Given these restrictions, an efficient data structure for representing traces and vias can be built. It is *not* a bit map, with one bit for each grid point. Bit maps are large, show poor locality of reference (since adjacent bits in some dimension in the map are far away in memory), and are inefficient to read and write. Instead, each layer is represented as an array of *channels*. For a vertical layer the channels are aligned vertically, so the array runs in the horizontal dimension. For a horizontal layer, the array runs vertically. Each channel is a doubly linked list of *segments*. A segment defines an interval in the channel that is used

[2]In Figure 21 on page 24 there are many diagonal traces. These are caused by postprocessing the rectilinear output of *grr*.

by some trace. It is linked to the next lower and next higher segments in the same channel. Another link through each segment connects the segments of a single trace, so that all space occupied by a trace can be found easily. Free space is not represented explicitly. It is inferred by the absence of a segment.

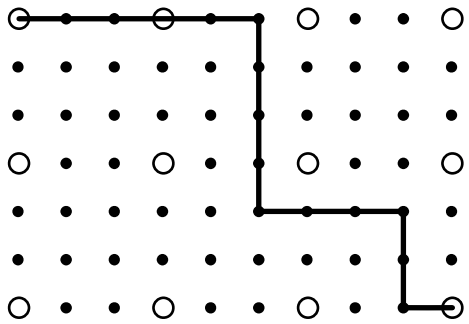

Figure 4: An Example Trace

Figure 4 shows an example trace on the routing grid. It shows a trace running over a via site, but this is avoided where possible in practice. Figure 5 shows the way the trace would be representated on horizontal and vertical layers. Each dark line represents a segment stored in a channel. The arrows in the figure indicate the array of channel list heads. Note that any trace can be represented on both layer types, so any rectilinear wiring pattern can be described on a layer. However, the representation will be more efficient if the trace is put on a layer with the appropriate orientation. This example is artificial in that there is no best orientation.

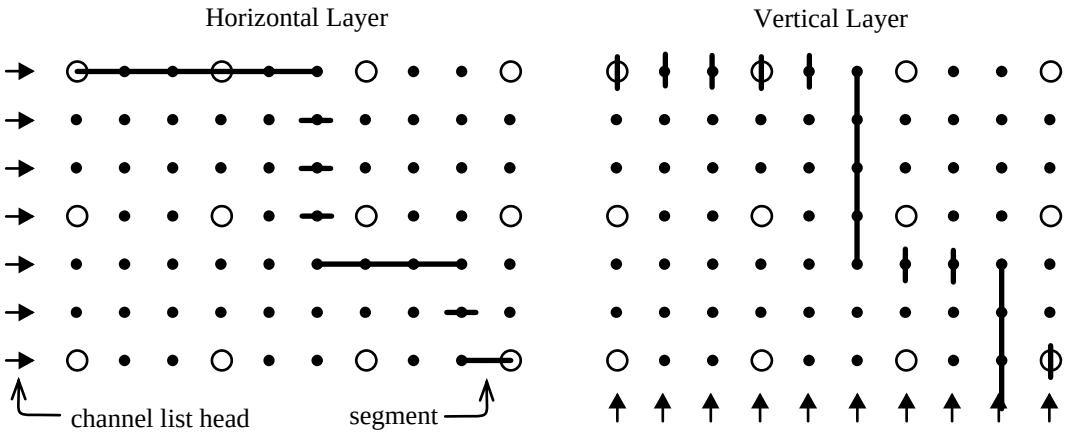

Figure 5: Representing the Same Trace on Different Layers

Vias and pins are represented in the channel array by covering the grid coordinates of the via by a segment of unit length. This is done on all routing layers because a drill hole makes a potential connection to all layers. However, inquiries about the availability of via sites are two to four orders of magnitude more frequent than updates of via site usage. An inquiry requires a probe of the grid location on each layer, because a trace on any layer will prevent a via being drilled there. To make via inquiries efficient, a separate *via map* is maintained, and updated each time segments are added and deleted from a layer. The via map is indexed by (*x*,*y*) in via coordinates (simple integer quotients of the grid coordinates) and holds the number of traces that are using this via location on any layer. This number will be zero if the via location is free. It will be greater than zero if traces on different layers run over the via location. It will be equal to the number of signal layers for a used via. Since updates to the routing layers are much rarer than probes, maintaining the via map results in significant performance improvements.

5. Routing Strategies

The routing algorithms used by *grr* can be grouped into a collection of strategies, which will be presented in the next sections. *Connection sorting* is used prior to routing so that the best connections are attempted first. While routing each connection, the *single-layer* algorithms are used to attempt connections on a single layer. If these fail, the *multiple-layer* algorithms join traces on several layers to complete the connection.

6. Connection Sorting

Connection sorting is the first routing strategy used by *grr*. In any routing problem such as shown in Figure 20, there will be thousands of connections to be made. Attempting the connections in the correct order can make the difference between success and failure.

Grr attempts the easiest connections first, but deciding which are the "easiest" connections may not be immediately obvious. They are not, for instance, simply the shortest connections. If routing is made by rectilinear traces, then there are many Manhattan paths between any two points. Some of these are of minimal length, and in general, there are many of these as well. The easiest connection to route is the one that has the fewest possibilities for a minimal path between its end points.

If the endpoints of a connection are separated by *dx* horizontally and *dy* vertically, then any minimal path will have horizontal length *dx*, vertical length *dy*, and total length *dx + dy*. Thus the total number of minimal paths is the number of ways of choosing *dx* horizontal or *dy* vertical steps from the total path length of *dx + dy* steps.

An approximation to this ordering results from sorting the connections using two sort keys. The keys in decreasing order of importance are *min (dx, dy*) and *max (dx, dy*). So the keys sort by straightness, then by length within straightness. The shortest straight connections will attempted first. The longest diagonal connections will be attempted last.

7. Single-Layer Algorithms

The connections to be routed are treated one by one in the order given in the last section. For each connection, several strategies are used to route the connection. These strategies divide into two parts. The single-layer algorithms are concerned with making connections on one routing layer only, and have no knowledge of multiple routing layers. The multiple-layer algorithms make connections using traces on more than one layer. These algorithms deal with the choice of vias to connect traces on different layers.

All the information that needs to be known about routing on a single layer can be encapsulated in three procedures. These are the single-layer algorithms, and they are simple variations of one underlying method. *Trace* tries to find a path between two given vias. It is used to construct all routes in the final output. *Vias* finds out which free via sites can be reached from a given via. It is used to find the "neighbors" of a via in Lee’s maze routing algorithm. *Obstructions* finds the connections that are near a given via. It is used to select victims to be ripped up when a connection cannot be made.

7.1. Trace

The basic single-layer algorithm is the procedure *Trace*. It answers the question "Is there a trace between *a* and *b* on layer *l* lying entirely within *box*?". If there is, then a linked list of channel segments is returned which leads from *a* to *b*. An example of the problem is shown in Figure 6.

Trace attempts to find a list of free segments, the first touching *a* and the last touching *b*, in which successive segments are in adjacent channels and have a non-empty overlap. The method used is a depth-first recursive search of free space. In Figure 6, the search starts at the segment covering *a* heading for *b*, and the free segments traversed are shown in gray. The action of the recursive procedure is straightforward. Given a free segment, the two channels on either side are searched for further free segments. The one nearest the destination is searched first. In Figure 6, the search at segment *s0* would encounter the used segment *s1* and the free segment *s2* in the search of the channel above. The next recursive call would be with *s2*. The search stops either when some free segment covers *b*, or when all free segments in *box* have been examined. If the target is reached, the resulting segment list is constructed as the recursive calls unwind. As shown in Figure 6, adjacent free segments can have large overlaps. During the unwinding, these overlaps are trimmed back to a single point as the segment list is built. Figure 7 shows the trace that would be created in this example.

Each recursive invocation of the procedure takes a free segment and enumerates its adjacent free segments in best-to-worst order. The cost of the algorithm is thus proportional to the number of segments examined, and not to the distance between the end points. In the absence of obstacles, it is just as fast to make a connection across the board as to the neighboring pin.

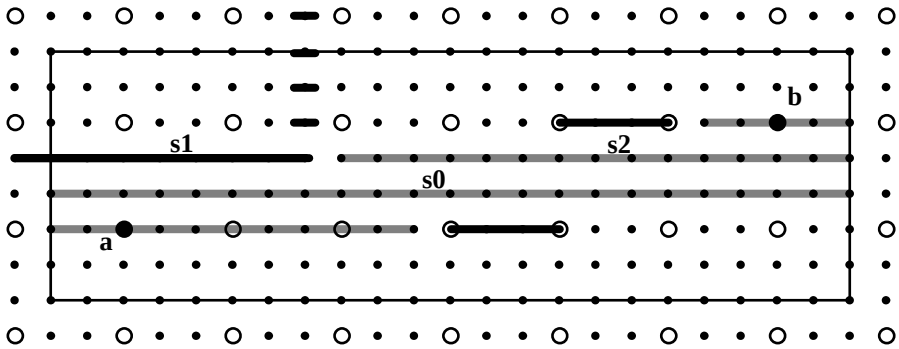

Figure 6: Trace (a, b, l, box)

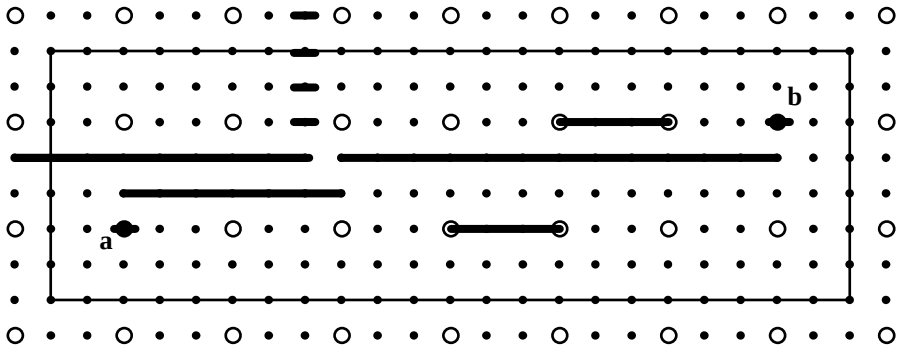

Figure 7: Resulting Trace

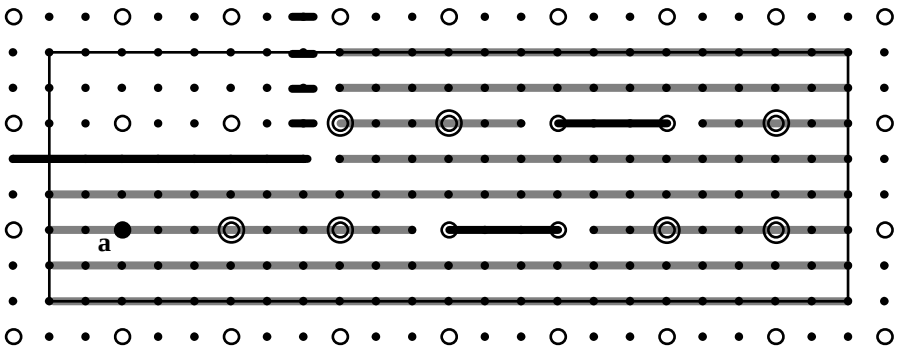

Figure 8: Vias (a, l, box)

7.2. Vias

The second of the three single-layer algorithms, *Vias*, is a minor variation of *Trace*. It answers the question "What via sites are reachable from point *a* on layer *l* by paths lying entirely within *box*?". The reply is given in an array of *(x,y)* pairs.

Figure 8 shows an example of a call to *Vias* and its solution. Again, the algorithm uses recursive examination of free space, but this time, all free segments accessible from the starting point *a* are examined. In Figure 8, the gray areas show the free segments searched. The coordinates of all via sites covered by free segments are added to the output array. These would be the ringed via sites in the figure.

7.3. Obstructions

The third of the single-layer algorithms is *Obstructions*. This procedure answers the question "What connections are near point *a* on layer *l* lying in *box*?". The reply is given as an array of connection identifiers.

As for *Vias*, the method is recursive enumeration of the free space surrounding *a*. The enumeration is exhaustive, and in Figure 8 the same set of free segments would be searched. This time, however, the connection identifier of each used segment and via encountered in the search is added to the array. So a call of *Obstructions* produces the list of immediate obstacles that surround a point on a given layer. It gives the information needed to rip up previously routed connections near some point.

8. Multiple-Layer Algorithms

The single-layer algorithms determine the wiring patterns on individual routing layers. The multiple-layer algorithms manage the choice of vias to join traces on different layers. They use only *Trace*, *Vias* and *Obstructions* to access each individual routing layer.

The multiple-layer algorithms represent a collection of strategies of increasing desperation applied in turn to each connection until a solution is found. Connections that cannot be realized as simple traces on one layer are constructed as a chain of traces joined by vias. As we shall see, these strategies *always* succeed, although in the worst case they may rip up previously routed connections. The strategies in the order they are applied are: try the optimal solution; try Lee’s algorithm; rip up obstructions.

8.1. Optimal Connections

Before describing optimal connections, a control parameter called the *radius* must be introduced. This parameter controls the degree to which orthogonal movement is allowed on a routing layer. Orthogonal movement is horizontal movement on a vertical layer, and vertical movement on a horizontal layer. The parameter *radius* is specified in via grid units, and defines a strip of accessible vias on each layer as shown in figure 9. Suppose a connection from point *a* to point *b* is to be made, and that the points differ by *dx* via units in their X-coordinates and *dy* via units in their Y-coordinates. Then a direct connection from *a* to *b* can be attempted on a horizontal layer only if $dy \leq radius$, and on a vertical layer only if $dx \leq radius$. In figure 9, $radius = 1$, $dy = 1$, and a direct connection on a horizontal layer is permitted. A connection on a vertical layer would not be permitted. Typical values of *radius* are 1 or 2. Increasing *radius* allows more vias to be reached, but increases channel blockage for later connections. Large values of *radius* are counterproductive for this reason.

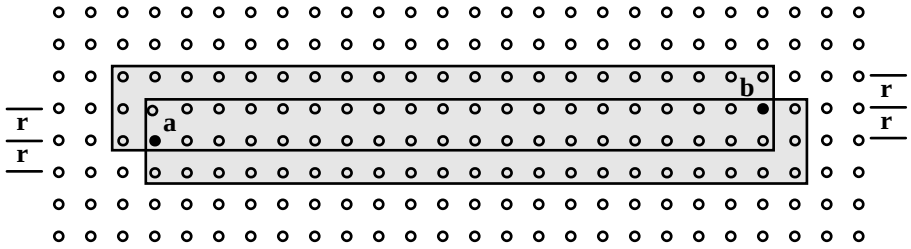

Figure 9: Radius

The first routing strategy applied to each connection is the simplest. Its *dx* and *dy* are tested against *radius* to see if a zero-via solution is acceptable on some layer. If so, then a call of *Trace* is made for each such layer, and we stop after the first successful call.

If there is no such layer, or if the board is sufficiently congested to block the zero-via solution, then the next best choice is a one-via solution. Figure 10 shows this situation.

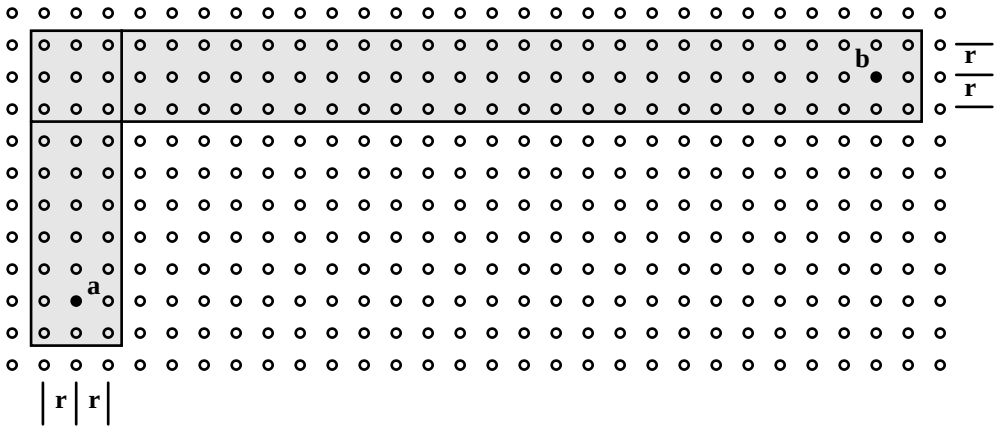

Figure 10: One-Via Solutions

To find a one-via solution connecting *a* to *b*, we must find an intermediate via *v* for which there are two zero-via solutions *a* to *v* and *v* to *b*. The number of candidate via sites for *v* is determined by the *radius* parameter. As Figure 10 shows, there are $(2radius+1)^2$ vias in each of the two squares at diagonally opposite corners of the rectangle bounding *a* and *b*. Figure 10 only shows one of these squares

for clarity. In this example, *radius* = 1, and 9 of the 18 candidates for *v* are shown. Clearly, there are marginally better and worse candidates. The vias at the center of each square, for instance, are the best since connections to them will block the fewest channels. To find the optimal one-via solution, the candidates for *v* are enumerated in best-to-worst order. For each candidate the two zero-via problems are attempted.

As a matter of practical experience with heavily congested circuit boards, it is essential that about 90% of the connections be routed with these optimal strategies. A figure much below this indicates that too few routing layers have been provided and that successful completion of the problem will be impossible. Aramaki [Aramaki 71] also optimizes simple connections before trying more expensive methods.

This is a divide-and-conquer approach to finding one-via solutions. A candidate via is chosen, and two simpler zero-via problems are attempted. It is tempting to consider extending this method to two-via solutions, and in fact this strategy was tried early in the development of *grr*. When a one-via solution can't be found, one might choose an intermediate via and attempt a zero-via connection to one of the pins and a one-via connection to the other. This proposal has the merit of being simple to program, and is similar to pattern-based routers [Asano 82]. Unfortunately there are usually too many possibilities to examine exhaustively. The problem is that the large number of candidate vias is tried in a pre-determined order without concern for local congestion. The approach becomes combinatorially intractable for three-via solutions, and a more effective method must be found.

8.2. Lee's Algorithm

After 90% of the connections are completed with optimal zero- and one-via solutions, hundreds of connections may remain. Finding solutions for these represents well over 90% of CPU time for difficult boards. The method used in *grr* is Lee's algorithm [Moore 59, Lee 61], perhaps the oldest algorithm in design automation.

> *Lee's Algorithm:* Suppose we want to find a connection between vias *a* and *b*. Begin by marking one of the vias, say *a*, and put it on a list of routing points, the *wavefront*. Now repeatedly remove the first point on the wavefront list, and examine all the immediately neighboring points at distance 1 from it. For each neighbor:
>
> - If the neighbor is *b*, then we're done. Retracing the path to the starting via *a* gives a connection from *a* to *b*.
> - If the neighbor is already marked or covered by an obstacle, ignore it.
> - If the neighbor is free and not marked, mark it and put it on the end of the wavefront list.

The algorithm as described above is $O(\mathbf{n}^2)$ in the distance between the vias. Three modifications can be made to make it much faster. The first modification arises from the concept of a "neighbor" of a point. As stated, the "neighbors" of a point are those at distance 1 from it, that is, the adjacent points on the routing grid. This choice leads to very slow searches, since many individual grid points must be scanned to advance a small distance across the board surface. It also leads to paths with a large number of unnecessary bends.

There is, however, a much better definition of "neighbor". All that is required of neighbors is that they should be connectable as the completed route is retraced. This leads to the following modification.

> *Modification 1:* The neighbors of a via are those via sites that can be directly connected to it by a trace lying entirely on one layer.

A "neighbor" of a via is thus a via site reachable in one "hop". This is exactly the purpose of the *Vias* procedure described in the last section. To find the neighbors of a via, *Vias* is called once for each layer, and the result added to an accumulating list. As a result, the neighbors of a via may lie across the board from it. As Figure 11 shows, the potential neighbors lie in a cross shape centered on the via. The vertical strip is due to vias reachable on vertical layers within the *radius* constraint, and the horizontal strip is due to horizontal layers.

This concept of neighbors radiating in lines from a via is a generalization of the line-searching method of Hightower [Hightower 69]. Combinations of the Lee and Hightower algorithms have also been made by Mikami [Mikami 70] and Heyns [Heyns 80].

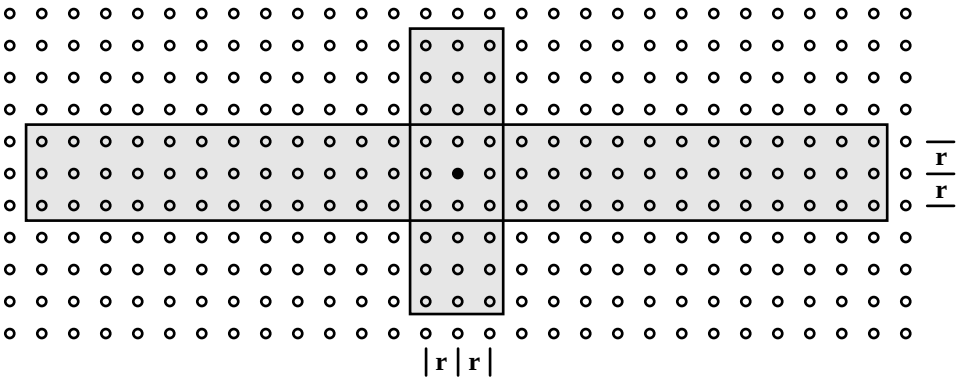

Figure 11: Potential Neighboring Vias of a Point

The second modification to Lee's algorithm results from considering blocked connections. The fact that a connection is impossible is important, because it means that the current connection can only be completed by ripping up others. A blocked connection is detected when the wavefront becomes empty without ever reaching the destination. Unfortunately, the common case is that one of the ends of the connection is heavily congested and can reach only one or two free vias. The other end of the connection is usually uncongested and can reach most other points on the circuit board. If the marking starts from the free end, the blockage will be detected only after marking a very large number of points. This suggests the second modification to Lee's algorithm:

> *Modification 2:* Spread wavefronts from both ends of the connection simultaneously. When the wavefronts touch, retrace from the point of meeting to the two sources. If either one of the wavefronts is exhausted, the connection is blocked.

The third modification to Lee's algorithm is to introduce a cost function [Rubin 74, Korn 82, Clow 84]. Lee's algorithm guarantees that if a connection can be made between two points, then the one found is of minimum distance. But a high price is paid for this guarantee. The algorithm ensures that before any path of length *n* is examined, *all* paths of length *n-1* have been examined. Restated in the light of modification 1, the new guarantee is that *n*-via solutions are attempted only after all *(n-1)*-via solutions have been tried. The cost function is used to direct the attention of the algorithm to places where a solution seems likely, rather than in all directions impartially. The guarantee of minimum number of vias in the solution is traded for a probability, and a much shorter search time.

> *Modification 3:* For each point *p* we define a cost function *cost(p)*. We maintain the wavefront lists for the two ends of the connection in increasing cost order. At each step, only the lowest cost point *p* on a list is examined. For each neighbor *n* of *p*, *cost(n)* is used to determine *n*'s point of insertion in the wavefront list.

The choice of cost function is critical. The original behavior of Lee's algorithm is characterized by

$$cost(n) = cost(p) + 1$$

This cost function minimizes the number of vias in the solution. A better cost function takes into account the position of *n* in relation to the target. If we are connecting point *a* to point *b*, and *n* is being inserted into *a*'s wavefront list, then define

$$cost(n) = distance(n, b)$$

to be the Manhattan distance between n and b. This cost function has the effect of concentrating effort on those points that seem to be leading towards the target. It can also lead to solutions that use many vias to circumvent minor obstacles near the target. The cost function actually used after much experimentation is a compromise between the goals of minimum number of vias and minimum search time. It is

$$cost(n) = distance(n, b) * hops(n, a)$$

where *hops(n, a)* is the number of vias between the neighbor *n* and the source of the wavefront of which it is a part. The effect of this cost function is to magnify the distance remaining to the target by the number of vias in the tentative path from the source. Its intuitive meaning is to insist that each via used in a path must bring progress towards the target.

Figures 12 through 15 show how the implemented algorithm operates. In Figure 12, the neighbors of *a* are found using *Vias* and inserted into *a*'s wavefront list. In Figure 13, the neighbors of *b* are found using *Vias* and inserted into *b*'s wavefront list. In Figure 14, the lowest cost neighbor of *a* is chosen, and its neighbors are found using *Vias*. One of them is a neighbor of *b*, and the search terminates. The completed route found by retracing from the meeting point to *a* and *b* and joining the halves is shown in Figure 15. The links in the retraced path are constructed with *Trace*. They may all be on different layers.

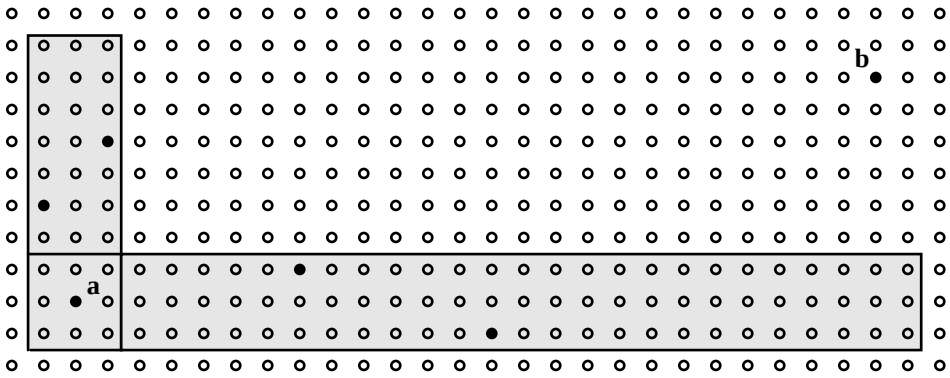

Figure 12: Finding Neighbors of *a*

8.3. Ripping Up

The combination of optimal search and the improved Lee's algorithm is sufficient for complete routing of all but the densest boards. Occasionally, a connection cannot be completed due to local congestion at one or other of the ends. When this happens one or more already-routed connections must be ripped up to make room for the new connection [Dees 82].

Recall that failure of Lee's algorithm is signalled by one of the wavefront lists becoming empty. To select the victims for ripping up, additional information is recorded. For each wavefront list, the identity

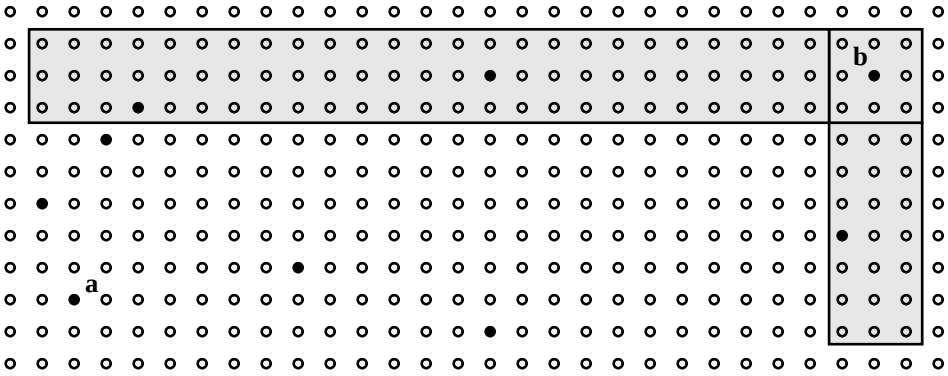

Figure 13: Finding Neighbors of *b*

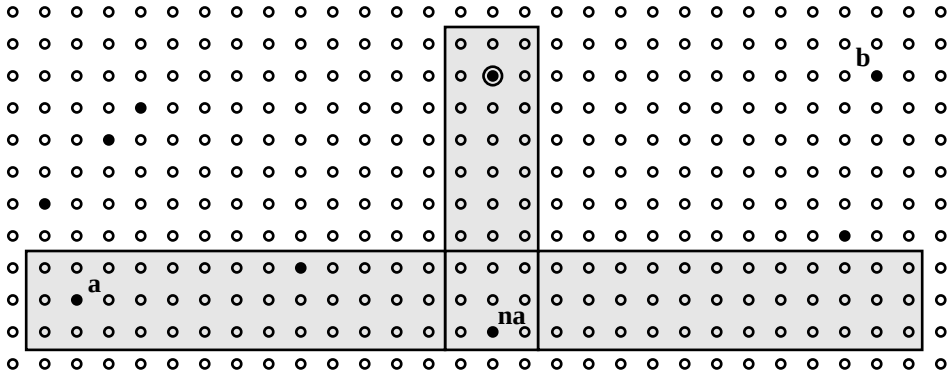

Figure 14: Best Neighbor of *a*

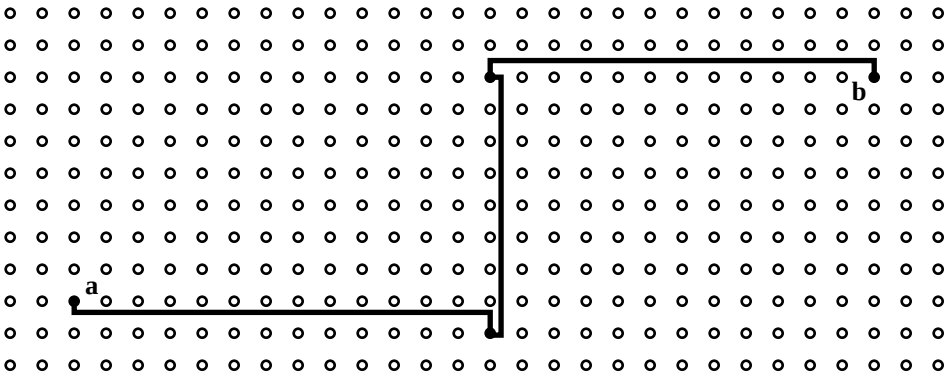

Figure 15: Resulting Connection

of the least cost point ever inserted into it is remembered. Thus when the list is exhausted, the point that made the most progress towards the target is known. This is the point around which obstacles are removed. To do this, the last of the three single-layer procedures, *Obstructions*, is called once for each routing layer. This procedure returns the identities of the connections that are using vias or traces in the immediate neighborhood of this point. These connections are ripped up, but a record of where they were is kept so that they can be re-inserted at very low cost. Now the attempt to route the current connection is

restarted from the beginning. This process of ripping up and restarting continues until enough obstacles have been removed that the route can be completed. At this stage, the new connection has been added, but tens of previous connections have may been ripped up. Now, an attempt is made to put the ripped-up connections back exactly where they were. Most can be re-inserted, since the new trace is unlikely to have blocked very many others. The remaining few must be marked for re-routing in the connection list.

8.4. The Complete Algorithm

The complete routing algorithm can now be presented. Recall that the list of connections in the problem is initially sorted. In the absence of rip-ups, it will take only one pass through the list, routing the connections one by one, to solve the problem. If there have been rip-ups, random connections in the list must be re-done, and so a further pass is made.

```
(* sort the connections using the two sort keys *)
sort (connections)
while progress and (unrouted > 0) do
    (* one pass through the connections *)
    for i := 1 to lastconnection do
        (* route connection[i] from a to b *)
        loop
            if alreadyrouted (a, b)
            or zerovias (a, b)
            or onevia (a, b)
            or lee (a, b) then
                (* the connection is made *)
                exit
            else
                (* rip up connections near selected point *)
                ripup (bestpoint)
            end
        end;
        (* put back as many rip-ups as possible *)
        putback ();
    end;
end;
```

Experience shows that there is a very fine line between ripping up a few connections to deal with local congestion and ripping up connections indefinitely when attempting a problem that is too hard. The symptom of an impossible problem is that on average one wire must be ripped up for each wire routed. In the above code fragment, *progress* is true only while each successive pass through the connection list leaves fewer unrouted connections. This stops infinite looping on impossible problems.

9. Results

Grr has been used to route all the circuit boards of the Titan, several boards from the Firefly computer designed at Digital's Systems Research Center [Thacker 87], and a number of production boards at Digital's Mid-Range Systems Group in Littleton. There are 13 different board types in the Titan. These range from 16 by 22 inch processor boards, to a 15 by 15 inch backplane, to 11 by 16 inch I/O and memory boards. Most are completely filled with SSI and MSI parts. All boards were routed entirely automatically with no manual intervention. Logic revisions were always made by re-routing the entire board, never by manual wiring fixes.

board	layers	conn	pins in^2	% chan	% lee	rip ups	vias	CPU min
kdj11	2	1184	27.5	76.7		>300		
nmc	4	2253	29.9	52.3	14	20	.99	28.5
dpath	6	5533	37.3	46.0	8	1	.65	21.5
coproc	6	5937	36.0	40.5	6	0	.62	11.3
kdj11	4	1184	27.5	38.4	8	0	.70	4.6
icache	6	5795	36.6	36.5	3	0	.41	6.1
nmc	6	2253	29.9	34.9	3	0	.68	2.2
dcache	6	5738	36.4	33.5	2	0	.40	5.2
tna	6	2789	43.4	27.1	3	6	.50	4.8

Table 1: Results

The results for the more difficult boards are shown in decreasing order of difficulty. All boards but *kdj11* and *nmc* are from the WRL's Titan computer. *Kdj11* is a single-board PDP-11, and *nmc* is the VAX 8800 memory controller board. The first line shows the router failing on a problem that is too hard. The program gave up after about 30 minutes having completed about 80% of the connections. The same problem is easily solved by adding two more routing layers, as the fifth line shows. The *layers* column shows the number of routing layers allowed. All the successfully routed boards have four or six layers. Routing boards of even medium density on two routing layers is difficult.

The *conn* column gives the number of connections in the problem. The *pins/in*^{2} column is the average pin density of the board in number of pins per square inch. The *% chan* column is an estimate of the wiring density of the problem. This figure is calculated by dividing the total Manhattan length of all connections to be made by the total available channel space on all layers. This gives the percentage channel demand to channel supply. As a rough estimate, it is clear that completely automatic routing will fail where channel demand is much more than 50% of channel supply.

The *% lee* column shows the percentage of all connections that were routed by Lee's algorithm. In denser boards with lower free space ratios, the percentage is higher, since congestion prevents optimal solutions to later connections.

The *rip ups* column shows the number of connections ripped up during the routing process. This number is very small in proportion to the number of connections, except in the case of the failure. Ripping up is clearly only being used to alleviate local congestion in a few areas, but does enable entirely automatic routing in marginal cases.

The *vias* column shows the number of vias added per connection. This number is below 1 for all examples, which indicates that most connections are routed with zero or one vias. To some extent this is due to the large number of straight terminating resistor connections in these ECL boards (10% to 25% of connections). These connections can be routed simply because the terminating resistors were chosen carefully by the stringer.

The *CPU min* column shows the number of CPU minutes on a VAX 11/785 (about 1.5 MIPS) required by each problem. Most real problems take only a few minutes, and the harder ones take less than half an hour. The Titan can route its own printed circuit boards in less than two minutes each.

10. Extensions

As in many engineering situations, actually building the high-speed circuit boards of the Titan required solving problems that appeared to be peripheral at the outset. These problems took a substantial part of the total effort. They required two extensions to the algorithms presented above. The first extension was the ability to tune specific connections to precise lengths. The second was a method to separate traces carrying ECL signals from those carrying TTL signals.

10.1. Length Tuning

Since a routing program is designed to make connections as short as possible, it is perhaps surprising that making connections longer is also important. In ECL circuits, signal nets are connected as transmission lines for highest speed. This is done so that logic values propagate from the output in a single voltage transition that is absorbed by the resistor at the far end without electrical reflections. Because of this, the distance between the output and an input determines the time it will take the signal to travel between them. In TTL circuits, the voltage change reflects many times, and there is no such simple relationship. The transmission-line nature of ECL circuits means that adjusting the length of connections is a way of controlling the delays in the circuit. This precise control of signal timing is the largest difference between ordinary and high-speed logic design. In common epoxy/glass printed circuit boards, signals propagate at around six inches per nanosecond. So length tuning can be used to adjust propagation delays to accuracies of a few hundred picoseconds.

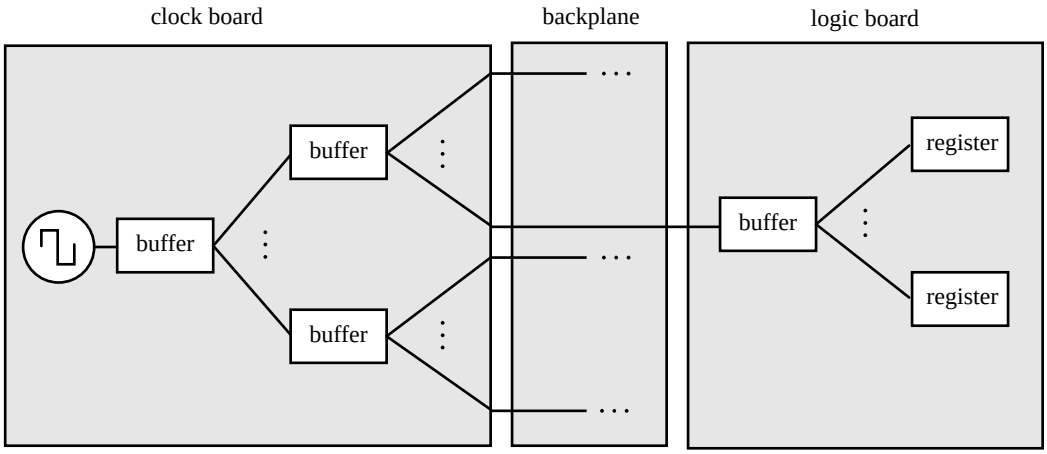

Figure 16: Clock Distribution Tree

The best example of delay tuning is in clock generation[3]. In the Titan, all clock pulses are derived from a single oscillator. The oscillator is at the root of a tree of nets as shown in figure 16. The nets are connected by buffers. These amplify and distribute copies of the clock to the pipeline registers at the leaves of the tree. It is essential that clock pulses reach each register simultaneously, so the delays from the root of the tree to each leaf must be the same. If the buffer delays are matched, the root-to-leaf delays can be matched by making the trace delays equal at each level in the clock tree. For instance, the trace delays of all clock signals on the backplane must be equal.

[3]Although in the Titan, this technique was also used to adjust register set-up and hold times.

Length tuning is complicated by the fact that signal propagation speeds on different layers of a circuit board are not the same. Signals propagate about 10% faster on the two outer layers than on inner layers. Unfortunately, the user is interested in specifying the delays of particular connections, not their lengths. Constructing a tuned connection from traces on different layers is a complicated optimization, especially when there is local routing congestion.

Length tuning is implemented in *grr* by specifying a propagation time for each pin-to-pin connection to be tuned. This time must of course be greater than the propagation time on the minimum-length path on the fastest layer. Two attempts were made at implementation. The first attempt modified the cost function of Lee's algorithm to do length tuning. The cost function was adjusted to select points whose total path delay from the source plus estimated delay to the destination is close to the target delay. Unfortunately, the delay to the destination proved difficult to estimate. The path could be constructed either on fast layers, or slow layers, or a mixture of the two. Also, it might not be close to the Manhattan length. This meant that many candidate solutions for the path were found, which when completed with *Trace* proved to be too fast or too slow. The variation in layer speeds made the cost function inaccurate enough that Lee's algorithm was overwhelmed with false solutions. Length tuning implemented this way turned out to be unacceptably slow.

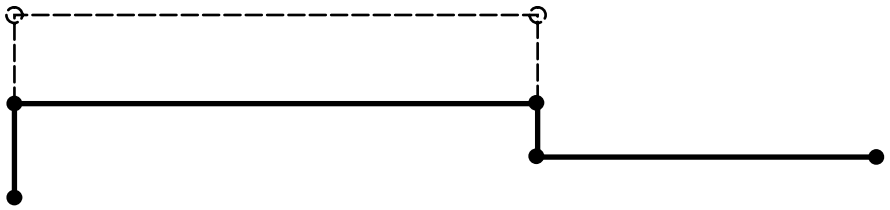

Figure 17: Length Stretching by Detour

The second and current implementation is based on adding detours to shorter paths. This method starts from a path created by the standard method, and then attempts to stretch it by adding a detour. Figure 17 shows the types of detour attempted. Between every pair of pins or vias in the shorter path, the stretching algorithm attempts to add a two-via detour. The search for detours of two vias only is done by restricting a Lee search to add only points one via away from the source to the wavefront. The restriction is needed to make length tuning run in acceptable time by searching only a small class of detours. If a detour is found that lengthens the path, but not enough, the detour process is repeated using the newly added vias in the path. This algorithm leads to acceptable performance if there are a few tens of length-tuned wires on a board. It is slow for hundreds of tuned wires.

It was unsatisfying and surprising that the simple modification to the cost function could not extend Lee's algorithm to do length tuning as well. It is probable that the variation in layer speeds had much to do with this. It probably caused too many plausible but unacceptable solutions to be generated. It is quite possible that by restricting length tuning to inner layers of uniform speed, the Lee approach could work well. This situation is typical of the heuristics employed in design automation. Relatively small algorithmic changes often lead to large changes in performance. Usually the only option is to try it and see.

10.2. Separating ECL and TTL

The second extension to *grr* to construct real circuit boards was to find a way to route boards with mixed ECL and TTL parts. Even though the Titan is an ECL computer, the memory subsystem and the I/O adapters needed chips available only in TTL. The problem here lies in the different signal propagation

characteristics of the two technologies. Because ECL signal swings are less than 1 volt, a nearby TTL signal change of 5 volts can induce enough noise to cause a false ECL logic value. Signals of the same family can be placed close together on the same layer, but ECL and TTL signals must be separated to prevent this problem.

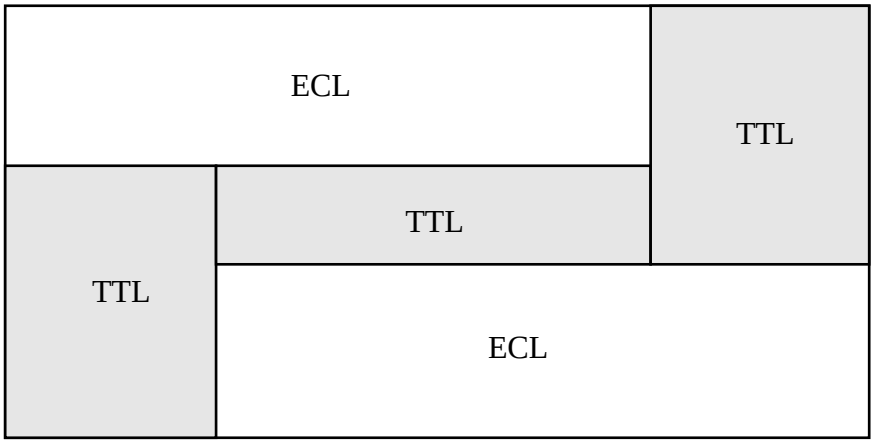

Figure 18: Tesselation for ECL/TTL Separation

Grr uses a method developed by J. Prisner and R. Kao of WRL to solve this problem. This method effectively pushes the problem of ECL/TTL separation back to the designer. Each signal layer can be tesselated into areas reserved exclusively for ECL or TTL wires. Figure 18 shows an example layer. For each TTL (or ECL) connection, *grr* will run traces and allocate vias only in TTL (ECL) tiles of the signal layers. The designer must arrange that under every TTL pin there is at least one signal layer with a TTL tile. And there must be an overlapping sequence of TTL tiles, possibly on different layers, leading to the other pin of the connection. In general, this arrangement is easy to find. Usually the chips of one or other technology can be arranged in a compact area on the board. The signal layers under this area are reserved for that technology. Where there are ECL/TTL level translator chips, some of the signal layers under these chips are marked as ECL and some as TTL[4]. Thus, placement must be done with a tesselation in mind, and the tesselation is defined precisely after the placement is fixed. In practice, this has been an easy task for board designers.

To route boards with mixed ECL and TTL, *grr* treats the board as two separate but superimposed routing problems. The algorithms described in the previous sections are applied in two passes over the board. Before starting the ECL pass, *grr* fills all empty space in TTL tiles, making them unavailable for traces or vias. Routing of ECL signals is then done as described above. After all ECL connections are made, the TTL "filler" is removed. Then to route the TTL connections, the ECL tiles are filled before the routing pass begins. The ECL filler is removed after the end of the pass. In the boards routed to date, this method of separating ECL and TTL has worked well, with little effort required on the part of the board designer or the programmer.

[4]If there is a power plane between them, or they run in orthogonal directions, two adjacent signal layers can carry ECL and TTL signals at the same point on the circuit board.

11. Limitations

The data structure presented in this report imposes four limitations on the circuit boards that can be routed:

- There is no easy way to represent traces of different widths, or non-rectilinear geometries. This makes the routing algorithms presented here inappropriate for analog PCB design. It does not appear to be an important limitation for digital PCB design except in the older design style that mixed signal and power routing on the same layers.
- The grid model does not allow traces to be placed at minimum pitch. Because drill holes are usually larger than trace widths, the routing grid positions occupied by vias cover several legal trace widths, not just one. So where there are many parallel traces but no vias, the traces cannot be spaced at the minimum legal pitch.
- Pins are assumed to be connected to all routing layers. This excludes surface-mount devices, whose pads connect only to the surface routing layer.
- Pins are restricted to lie on the via grid. The via grid must be defined to be at the finest pitch of the pins of any part on the board. In practice, this was 100 mils for the examples in this report.

The limitation on trace density is difficult to quantify. It is true that the imposition of the grid model reduces the maximum trace density on any signal layer, but the availability of via sites at regular intervals increases the connectivity between the signal layers. Thus the grid model trades the ability to route dense bundles of parallel connections for the ability to route irregular crossing connections.

Removing the restriction on surface-mount devices is straightforward. It simply requires that the router not assume that it can start a connection on any layer. Surface mount devices have been used with *grr*, though in a somewhat clumsy way. A hand-designed *dispersion pattern* was generated to connect the pads to a regular array of vias by traces lying only on the top surface. The router was told to consider the vias as the end points of the connections. If the pin pitch of the parts is small compared to the via and trace pitch of the circuit board, using a dispersion pattern is probably the correct practice to follow. Allowing the router to drill vias as needed would result in an irregular via pattern, which would almost certainly create blockages around some of the pads. Where the trace and via pitch of the board are similar to the pad pitch of the surface-mount parts, the fully automatic approach would be better.

Forbidding off-grid pins has proved to be an annoying rather than a serious limitation, and is not fundamental to the data structures or algorithms. In the boards routed to date, parts with off-grid pins were also handled by manually creating a dispersion pattern to nearby vias. The reason for the existence of the via grid is to allow rapid searches for available via sites. This is one of the critical paths of the routing algorithm. However, there is no particular reason to restrict the end points of a connection to lie on the via grid. In the author's opinion, this restriction can (and should) be removed by generalizing *Trace* to connect arbitrary grid points rather than only via points. This would allow off-grid pins, though all vias would still be on the via grid.

12. Conclusions

The router described here uses a collection of heuristic methods to solve the printed circuit board routing problem. These include initial sorting of the connections, attempts to find optimal solutions quickly, and a generalized version of Lee's algorithm. These methods take advantage of a compact representation of the board wiring, and follow a consistent strategy of trying the simplest solution first.

Printed circuit board routing is a global optimization problem in which choices made about individual connections interact. Decisions made early in the solution may turn out to be wrong later. In view of this, a general approach underlies the algorithms presented here. When many interacting decisions are made that may have to be revised, it is much more important to make a quick reasonable choice than an optimal choice. This approach is based on the following assumption. Though the optimal solution to the routing problem may be very difficult to find, there are a large number of acceptable solutions that can be found in reasonable time.

One of the surprises in the development of this set of heuristic algorithms is the number of times that a large increase in speed has been gained by a relatively minor change to the algorithms. In this sense they are unstable. A small change to one of the algorithms can cause unpredictable global effects when repeated in thousands of connections. One example is the representation of the channel data structure. In the current data structure, a channel is a linked list of segments. In earlier versions, each channel was represented as a binary tree of segments, since binary trees have better performance for random probes. In reality, however, the access pattern to a channel is far from random. It is localized to a small part of the channel when routing any given connection. The change from binary tree to doubly linked list with a moving head-of-list pointer halved the running time on most problems. Similar sensitivity was shown to changes in the way that the *Trace* procedure constructs traces between vias. Unfortunately, of the many changes made in the hopes of improving speed or routing density, only a few were beneficial, and these could not be predicted ahead of time. It is likely that improvements remain to be made in the algorithms as they stand. Nearly all heuristic methods seem attractive when proposed; almost none work in practice.

The most effective tools for improving program performance were careful analysis of the router output to find inefficient routing patterns, statistical measures of routing patterns, and profiles of the CPU usage of each procedure in the program. The profiles allowed design effort to be concentrated in that small part of the program where there were large potential performance gains.

Rapid, entirely automatic routing of printed circuit boards is feasible with today’s board technology and algorithms. Manual editing of circuit boards is unnecessary for most digital logic circuits. The development of denser PCB technologies with smaller vias can only improve this situation, and there seems no fundamental reason why the pin densities associated with surface mount devices should pose serious obstacles.

13. Appendix

The four figures in this appendix show a real routing problem and its solution. The problem is the Titan floating point coprocessor, the *coproc* board of table 1. This circuit board has six power planes and six signal layers for a total of 12 manufactured layers.

Figure 19 shows the board placement. Each of the 330 24-pin ECL 100K integrated circuits is flanked by a 12-pin single in-line package containing termination and pull-up resistors. The placement was generated manually using an interactive graphics editor over a period of months. Most of the time was devoted to shortening the critical timing paths found by the timing verifier.

Figure 20 shows the routing problem this placement poses. This is a graphical representation of the stringer output. Each line in the figure is a pin-to-pin connection that must be made in the solution.

Figure 21 shows one of the six routing layers of the solution found by *grr*. This is a photographic positive, so copper is left only where the image is black. The rectilinear *grr* output was postprocessed to generate this photoplot. Local modifications were made to produce the rounded corners and diagonal traces, and also to spread apart long parallel trace runs. These optimizations improve the manufacturing yield and electrical characteristics of the circuit board.

A ground plane of the solution is shown in figure 22. It was automatically generated after routing. It is shown as a photographic negative, so that copper is etched away where the image is black. The small round circles prevent connections to non-power pins and vias. The circles with dashed outlines indicate power pin connections to this layer. Some metal is removed near a power connection to provide thermal resistance. This prevents the heat needed for soldering from passing straight into the copper mass of the power layer. The large black circles in the image prevent the power layer connecting to the mounting screws holding the circuit board to its frame.

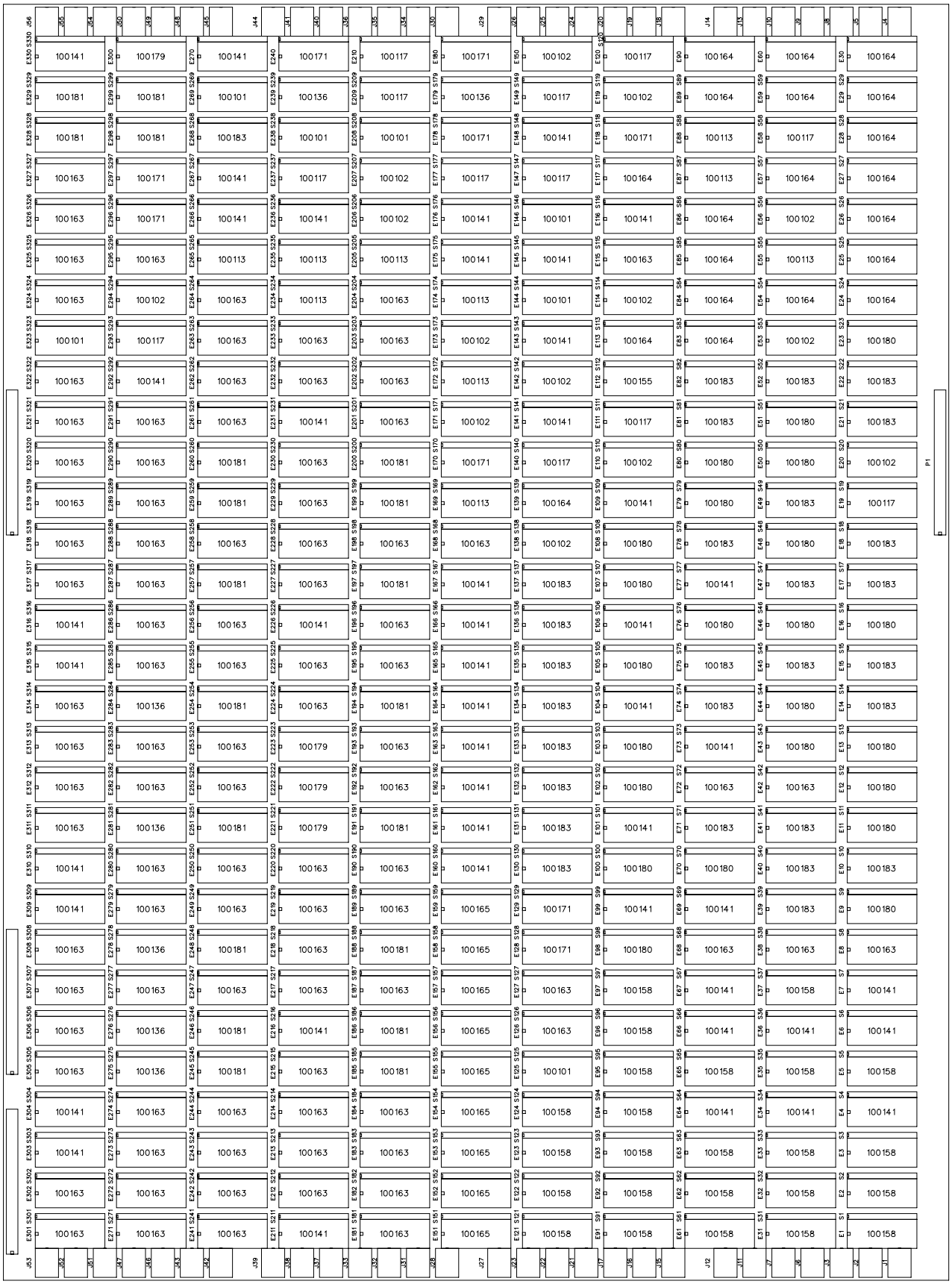

Figure 19: Titan Coprocessor Placement

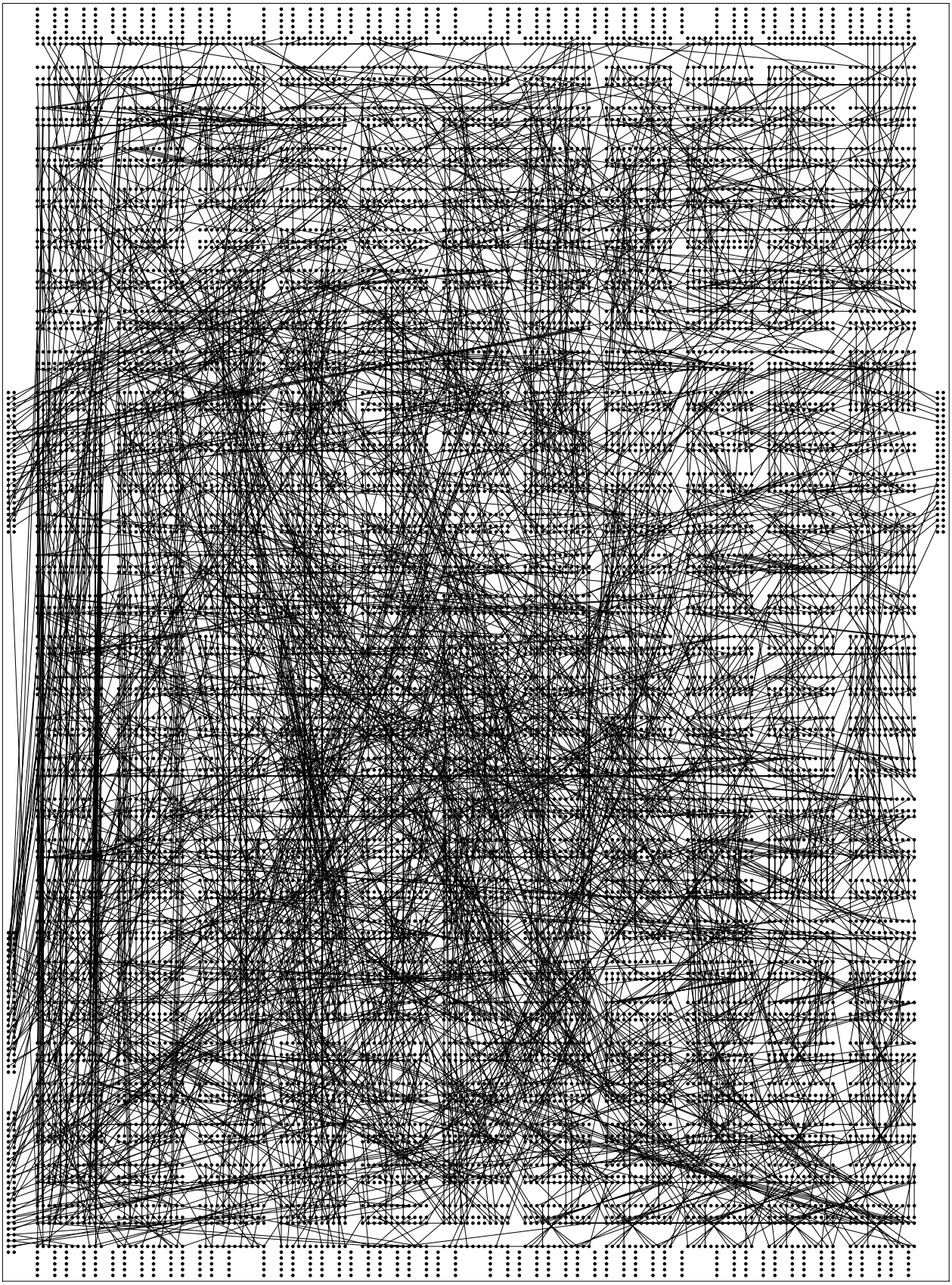

Figure 20: Titan Coprocessor Routing Problem

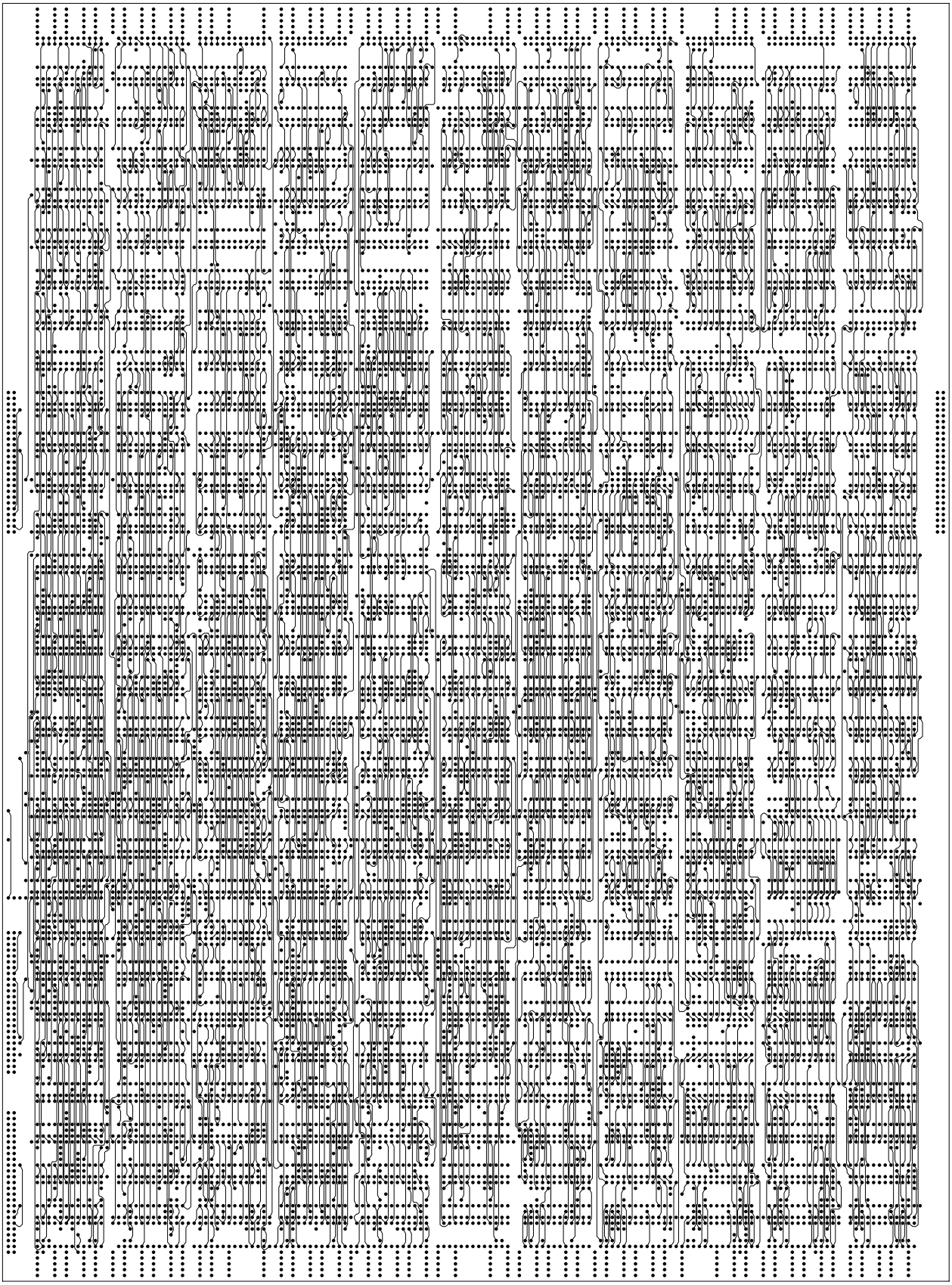

Figure 21: Titan Coprocessor Signal Layer

Figure 22: Titan Coprocessor Ground Plane

References

[Aramaki 71] Aramaki, I. et al.
Automation of Etching-Pattern Layout.
CACM 14(11):720-730, November, 1971.

[Asano 82] Asano T.
Parametric Pattern Router.
Proc. 19th Design Automation Conference :411-417, June, 1982.

[Clow 84] Clow G.W.
A Global Routing Algorithm for General Cells.
Proc 21st Design Automation Conference :45-51, June, 1984.

[Dees 82] Dees W.A. Jr., Karger P.G.
Automated Rip-Up and Reroute Techniques.
Proc. 19th Design Automation Conference :432-439, June, 1982.

[Heyns 80] Heyns W., Sansen W., Beke H.
A Line Expansion Algorithm for the General Routing Problem with a Guaranteed Solution.
Proc. 17th Design Automation Conference :243-249, June, 1980.

[Hightower 69] Hightower D.
A Solution to Line Routing Problems on the Continuous Plane.
Proc. Design Automation Workshop :1-24, 1969.

[Korn 82] Korn R.K.
An Efficient Variable-Cost Maze Router.
Proc. 19th Design Automation Conference :425-431, June, 1982.

[Lee 61] Lee C. Y.
An Algorithm for Path Connection and its Applications.
IRE Transactions on Electronic Computers :346-365, September, 1961.

[Mikami 70] Mikami K., Tabushi K.
A Computer Program for Optimal Routing of Printed Circuit Connectors.
IFIPS :1475-1478, 1970.

[Moore 59] Moore E.F.
The Shortest Path Through a Maze.
Proc. Int. Symp. on Switching Theory :285-292, 1959.
Harvard University Press.

[Nielsen 86] M.J.K. Nielsen.
Titan System Manual.
Digital Equipment Western Research Laboratory Research Report 86/1 , 1986.

[Rubin 74] Rubin F.
The Lee Connection Algorithm.
IEEE Transactions on Computers C-23:907-914, 1974.

[Soukup 81] Soukup J.
Circuit Layout.
Proc. IEEE 69(10):1281-1304, October, 1981.

[Thacker 87] C.P. Thacker, L.C. Stewart, E.H. Satterthwaite.
Firefly: A Multiprocessor Workstation.
Technical Report 23, Digital Equipment Systems Research Center, 1987.

Table of Contents

List of Figures

List of Tables