Optimal Two-Terminal Routing

James P. Cohoon and Dana S. Richards
Department of Computer Science
University of Virginia

Computer Science Report No. TR-85-26
Revised December 1, 1985

OPTIMAL TWO-TERMINAL WIRE ROUTING

J. P. Cohoon and D. S. Richards

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

INTRODUCTION

Wire routing problems have received a great deal of attention over the past two decades. One method of attacking the general routing problem is to decompose the routing of K nets into K problems of routing a single net. Such a decomposition requires an intelligent heuristic to order the nets for routing. The simplest significant single net routing problem is the optimal interconnecting of two terminals with wires on a grid in the presence of obstacles (i.e. modules and previously routed wires). It is this problem that we primarily address here. Generalizations to multiple nets and multiple terminal nets are addressed in a remarks section at the end. We consider two different optimization criteria - minimum number of bends and minimum wire length. The minimum number of bends criterion is considered since an interconnection with a minimum number of bends tends to also have minimum length and is usually easier to compute.

Classic techniques for the two-terminal routing problem involve exploring the underlying grid systematically [HIGH69, LEE61, RUBI74]. There is an enormous literature on improvements to these techniques, so that in practice they yield acceptable algorithms (e.g. [HIGH74, SOUK81]). However, their worst-case performance is still lower-bounded by the number of grid points available for routing. This number is typically quite large and theoretically independent of the (much smaller) number of constraints (i.e. modules and previously routed wires).

Our algorithms have running times that are polynomial in m, where m is a measure of the number of constraints. These algorithms are the first to achieve to this bound. The algorithms exploit new theoretical results that limit the search [COHO84b]. Further, they exploit techniques from the field of computational geometry, which are not applicable to the classical techniques [BENT80, LIPS84]. In fact for the problem of routing with the minimum number of bends our algorithm's time complexity is $O(m \log m)$. Using a result of Lipski [LIPS84] we can conclude that a lower bound on the worst-case time in determining a minimum bend wiring is $\Omega(m \log m)$. Thus our algorithm is optimal with respect to worst-case run-time. (Our measure is with respect to the decision model of computation [HORO78]).

There are two primary approaches to the two-terminal routing problem: maze routers [LEE61, RUBI74] and line search routers [HEYN80, HIGH69]. Maze routers find optimal solutions under a wide variety of optimization criteria. However, they have extremely large memory requirements to represent reasonable-sized circuits and can be slow to find long interconnections. Line search routers do not store the entire circuit medium and thus have more reasonable storage requirements than maze routers. In addition, line search routers have good run-time performance for up to medium size problems and their solutions are typically close to the minimal number of bends. However, line search routers do not guarantee finding a solution and their performance degrades and their storage requirements increase with dense circuits [SOUK81]. The line intersection approach described here is a synthesis of the maze and line search approaches. This synthesis allows our router to possess the advantages of both methods without their disadvantages.

THEORETICAL BACKGROUND AND PROBLEM FORMULATION

A *circuit medium* is one or more rectangular continuous regions (layers). A *via* is a feed-through from one layer of a circuit medium to adjoining layers.

A digital device is often decomposed into a collection of functional units or *modules*. The modules occupy portions of the circuit medium. Individual modules may have

different sizes and shapes. The only restriction for a module is that all its boundary edges must be rectilinear. A similar restriction is applied to the layout of wires. Though they decrease generality, such restrictions allow for the compact representation of modules and wires, and typically reflect the capabilities of automatic fabrication equipment.

Various subsets of the modules may require that information be shared for the operation of the digital device. Each such subset is a *net* and the information is a *signal.* A *terminal* lies on the boundary of a module and is an interconnection point for a net. A signal is shared by interconnecting the terminals that compose a net with wires. Often nets are decomposed into a tuple of terminal pairs. The terminal pairs are then wired one at a time in tuple order.

A *routing region* is that portion of the circuit medium unoccupied by modules.

A *routing segment* implements all or part of an interconnection. It has direction, position, and width on a specified layer of the routing region. The position of a routing segment is denoted by its endpoints. Formally, a *wire* is a series of routing segments that connects two terminals. Two successive routing segments of a wire share either a common endpoint on the same layer or have endpoints that share a via. A *design rule* is a restriction on the way wires are positioned, formed, or interact. Restrictions are imposed to guarantee the presence or absence of some electrical condition.

The routing problem has the following formal specification:

> **Input:** A circuit medium, a set of modules, a routing region, a net composed of two terminals, a set of vias, a set of design rules, and an optimization criterion of either bend minimization or length minimization.
> **Output:** A set of optimal routing segments that implements all nets.

The line intersection approach described here examines a limited subregion, S, of the routing region. S is organized as a collection of possible routing segments and is guaranteed to contain an optimal wiring if such a wiring exists. The guarantee is the result of applying two search limiting theorems [COHO84b]. The routing segments in S are examined in either a *breadth-first* or *shortest-path search* order (depending upon the optimization criterion).

These efficient search techniques enable the router to quickly determine an occurrence of an optimal wiring. The design rule that our router employs concerns the separation of routing segments - all wires are at least one separation unit apart.

We now present the necessary terminology to define the set of segments S.

A horizontal segment $u=((x_u,y_u),(w_u,y_u))$ *covers* horizontal boundary segment $v=((x_v,y_v),(w_v,y_v))$ iff the intersection of the closed intervals $[x_u,w_u]$ and $[x_v,w_v]$ is nonempty. A definition follows analogously for a vertical covering. In Figure 1.a, t and u both cover (horizontally) v, and q and r both cover (vertically) s.

A horizontal segment $u=((x_u,y_u),(w_u,y_u))$ in the routing region is *maximal* iff neither of the points (x_u-1,y_u) and (w_1+1,y_u) are in the routing region. The offset of 1 represents the basic unit of separation. A definition follows analogously for a maximal vertical segment. In Figure 1.a, u and r are maximal and t and q are nonmaximal.

The vertical boundary segment that contains the point (x_u-1,y_u) for maximal segment $u=((x_u,y_u),(w_u,y_u))$ is the *left limiting* boundary segment of u. The vertical boundary segment that contains the point (w_u+1,y_u) is the *right limiting* boundary segment of u. Definitions follow analogously for *lower limiting* and *upper limiting* horizontal boundary segments. In Figure 1.a, p is a left limiting segment for u and z is an upper limiting

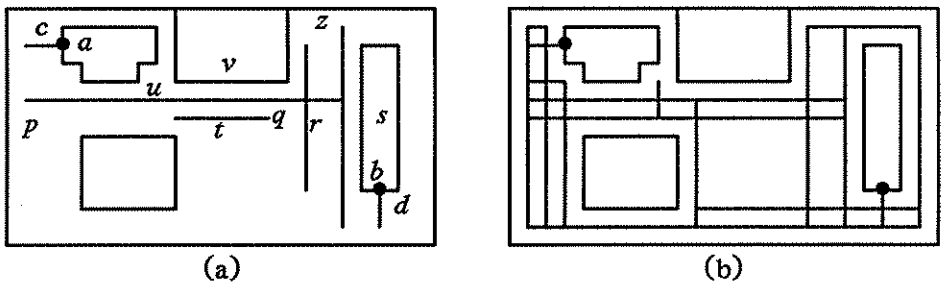

Figure 1 - Terminology Examples

segment for r.

A horizontal line segment $u=((x_u,y_u),(w_u,y_u))$ in the routing region is a *type I escape segment* for horizontal boundary segment $v=((x_v,y_v),(w_v,y_v))$ iff u is a maximal segment that covers v and the absolute difference between y_u and y_v is 1 unit of separation. A definition follows analogously for a vertical type I escape segment. In Figure 1.a, u is a horizontal type I escape segment and r is a vertical type I escape segment.

A horizontal segment $u=((x_u,y_u),(w_u,y_u))$ in the routing region is a *type II escape segment* iff u is a maximal segment that intersects terminals a or b. A definition follows analogously for a vertical type II escape segment in the routing region. Note, the terminals a and b are considered to be in the routing region. In Figure 1.a, c is a horizontal type II escape segment that intersects terminal a, and d is a vertical type II escape segment that intersects terminal b.

A segment is an *escape segment* iff it is either a type I or type II escape segment. In Figure 1.b, we give a problem instance and its set of escape segments.

Formally then the set of possible routing segments S is the set of a escape segments.

CONSTRUCTING THE ESCAPE LINES

The first step is to identify where we want escape lines. Generating the appropriate initial ranges of coordinates is straightforward. However, determining where the limiting boundary segments are for each escape line is not easy. If we had a data structure modeling the underlying grid it would be trivial, but costly. Not only would the time be independent of m but the data structure would occupy too much space. We must abandon such approaches since our data structures must consist only of the line segments of the constraints and derived information. There is a growing body of literature dealing with data structures for horizontal and vertical line segments and the algorithms that utilize them.

In this paper, we use the *line-sweep approach*, an approach which is widely used within the field of Computational Geometry. A notable example is [BENT79] where it is used to report intersecting line segments. Informally it allows our algorithm to view geometrical objects in the plane through a very long slit. The slit is swept across the (hypothetical) grid to view the sections of interest. The slit or *cross-section* has either a horizontal or vertical orientation; hence typically two disjoint orthogonal sweeps are performed. In Figure 2 a horizontal slit is shown. To simplify this exposition, we will assume that our line-sweep is horizontal and is as wide as necessary. The vertical sweep will be analogous. Most horizontal (vertical) line-sweep algorithms accomplish their goal in a single bottom to top (left to right) pass.

For such an algorithm we must preprocess our data. The modules are described by their boundary segments. We assume that they can be identified as being bottom, top, left, or right edges. Wire segments can be thought as segments or as degenerate rectangles. Here, we assume the latter (i.e. a wire segment is described by four segments, two of which coincide and two of which are points). In addition to these segments, our input data for the line-sweep algorithm contains the request for all (horizontal) escape segments.

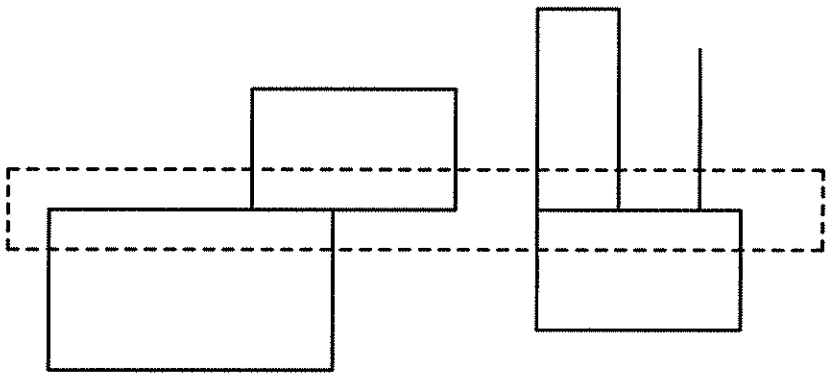

Figure 2 - Horizontal Slit

Every top or bottom edge corresponds to a request, denoted $ELR<i,j,y>$, for the horizontal escape line(s) with coordinate y with some abscissae in the range $[i,j]$. The coordinate y differs from the ordinate of the corresponding edges by ± 1. Terminals that lie on either right or left boundaries also request horizontal escape lines.

Preprocessing involves sorting the data. For the horizontal slit sweep phase we only use top and bottom edges (of modules and wire segments) and requests for horizontal escape lines. The former are denoted $TE<i,j,y>$ and $BE<i,j,y>$, where each corresponding edge has ordinate y and abscissae over $[i,j]$. These are all sorted by the ordinates in increasing order. When there are ties, we use as a secondary key the request type where $BE<ELR<TE$. In addition, if there are a series of TE and BE "requests" with the same ordinates then the BE series is repeated again after the TE series. (This redundancy is used to solve some problems introduced by abutting modules). The number of requests remains $O(m)$ and thus can be sorted in $O(m \log m)$ time. The sorted data is denoted as the *command list*. Any request from this list for escape lines above or below the routing region are removed.

Our algorithm for generating all of the (horizontal) escape line segments makes use of a data structure *GAPS* which reflects what can be seen through our hypothetical slit. What is seen, left to right are open portions of the routing region and portions occluded by modules or wires. Each gap or open portion is considered to be a single object (with two fields indicating its range of abscissae) and the gaps are totally ordered from left to right, since they cannot overlap. As the slit moves up two events can occur - either a bottom edge is encountered or a top edge is scrolled passed. If the former event occurs then some gap may be partially or even totally blotted out, or a gap may be split into two smaller gaps. If a top edge occurs then the range of some gap is extended. This extension may allow two gaps to coalesce.

GAPS can be implemented using data structures that can represent totally ordered sets. For our analysis, we assume that balanced search trees are used. Since there will be at

most $O(m)$ gaps, we can expect all operations to be done with $O(\log m)$ overhead time. One operation we need is $INSERT(GAPS,i,j)$ which will insert a new gap into *GAPS* with range $[i,j]$ where this new gap does not intersect any existing gaps. The other operation we need is $FETCH(GAPS,i,j,L,option)$ which returns an ordered (possibly empty) list L of gaps. A gap on L with range $[l,r]$, denoted $g<l,r>$, intersects the range $[i,j]$ at more than one point. The *option* parameter indicates whether the elements in L are to be deleted from *GAPS* or not. By using data structures that permit efficient splits and merges this operation can be performed in $O(log\ m+|L|)$ time [AHO74].

The advantages of *GAPS* over similar alternative line-sweep structures is that the requests $ELR<i,j,y>$ can be trivially satisfied. Simply produce (i.e. output) escape line segments across all the gaps that intersect the range $[i,j]$. The algorithm in Figure 3 ties the above comments together. Note that *GAPS* should be initialized to be one gap which is the maximum width of the routing region. Also, the algorithm makes little use of the current ordinate and is unaware, in fact, when the ordinate (of the slit) has increased. In particular, the slit does not take unit steps, but automatically *jumps* to the next segment. It is easy to show that the total length of all the lists returned by *FETCH* is $O(m)$, hence the algorithm runs in $O(m \log m)$ time.

EXTENSIONS TO ALGORITHMS

In the previous section we used a fairly generous model of the input. We allowed modules to abut, but not to overlap. Hence if the modules were known only by their constituent rectangles there would be no need to find the entire contour first. If the contour is known the algorithm still works of course. If the modules are only known as a collection of possibly intersecting rectangles then the contour must be determined first. This can be done in $O(m \log m)$ time [GUTI84].

With more specific knowledge about a particular problem instance redundancy in the code can be removed. This will improve the running time, but not the overall time complexity. For example, if we know all modules were separated by at least 2 units then

```
for each C⟨i,j,y⟩ in the sorted command list do
    case
        C=BE⟨i,j,y⟩:
            FETCH(GAPS,i−1,j+1,L,delete)
            for each g⟨l,r⟩ in L do begin
                if l<i then
                    INSERT(GAPS,l,i)
                if r>j then
                    INSERT(GAPS,j,r)
            end for
        C=ELR⟨i,j,y⟩:
            FETCH(GAPS,i,j,L,save)
            for each g⟨l,r⟩ in L do
                if l<r−2 then
                    output ((l+1,y),(r−1,y))
        C=TE⟨i,j,y⟩:
            FETCH(GAPS,i−1,j+1,L,delete)
            for each g⟨l,r⟩ in L do begin
                if r=i then
                    i:=l
                if j=l then
                    j:=r
            end for
            INSERT(GAPS,i,j)
    end case
```

Figure 3

we could conclude that each *ELR* would generate exactly one escape line. Further, the code for each *BE* and *TE* would be simplified and there would be no need to repeat the *BE*'s for each ordinate. Though the separation seems reasonable it usually violated by wires, which we treat as degenerate modules.

We might proceed by treating wires separately. We introduce the commands $BW\langle i,y\rangle$ and $TW\langle i,y\rangle$ to reflect that during a sweep we have encountered the bottom or the top, respectively, of a (vertical) wire segment at ordinate y with abscissa i. With these we could maintain in addition to *GAPS* (which will be ignorant of the effect of the wires) another simple data structure *WIRES* which would keep the abscissae (in order) of all wires that cross the slit in its current position (horizontal wire segments would still be handled as before). The cumulative cost of this maintenance is $O(m \log m)$ time if we

again use balanced tree techniques. *WIRES* would be consulted for each escape line segment (generated by GAPS) to see if it was *cut* by a wire or wires into smaller segments. This two-tiered scheme is still $O(m \log m)$ overall. If the separation assumption holds this scheme should run somewhat faster, since most redundant operations have been removed.

If during preprocessing we know whether the endpoints of wire segments were attached or not then we could return to a one-tiered scheme using just the *GAPS* data structure. Let $BW'<i,y>$ be the command for the (rare) event of encountering during a sweep, the bottom edge of a wire that does not connect to another (horizontal) wire segment or a module; we reserve $BW<i,y>$ for the complementary case. Define *TW* and *TW'* similarly. With this knowledge, during the initial sorting phase if ties in ordinate value occur than we use the command type as secondary key using the following ordering - $TW < BE < BW' < \text{ELR} < TW' < TE < \text{BW}$. Note that no commands are repeated and that we assume modules do not abut. It is straightforward to update Figure 3 for this case. Again this scheme runs in $O(m \log m)$ time.

Another incremental improvement involves detecting duplicated escape lines. The simplest, and sometimes best, way is to sort the (horizontal) escape lines by ordinate and then abscissa. Note two lines may coincide, but they will not properly overlap. Suppose we initially sort $ELR<i,j,y>$ before $ELR<i',j',y>$ if $i \leqslant i'$. In this case the order of the escape lines are output is very nearly sorted order, so a simple algorithm like *InsertionSort* would give good results. By using auxiliary data structures the sorted order could be generated on the fly. For realistic inputs, duplicates can be found in constant time. However, pathological cases could make a final *InsertionSort* preferable.

The improvements in this section have been intentionally vague since they rely on specific assumptions about the data. They are made to suggest avenues that can be exploited depending upon what assumptions can be made.

MINIMUM BEND ROUTING

We wish to find a path from one terminal to another which minimizes bends. We solve this problem with a shortest path algorithm on the graph induced by the escape line segments, called the *intersection graph*. Each vertex of this graph corresponds to a segment, and an edge (of length 1) corresponds to intersecting segments. Recall that only orthogonal segments intersect. So the minimum bend path corresponds to the shortest path between the type II escape segments corresponding to the two terminals. Let I be the number of intersections (i.e. the number of edges).

The standard algorithms in the literature [AHO74] run in $O(p^2)$ or $O(q \log p)$ time, for graphs with p vertices, q edges, and non-negative weights. This implies we could solve our problem in $O(m^2)$ or $O(I \log m)$ time. Since $I{=}O(m^2)$ the latter is $O(m^2 \log m)$ though in practice this bound is pessimistic. Using sophisticated data structures [FRED84b] we could have an $O(p \log p + q)$ time algorithm, which translates as $O(m \log m + I)$ time, which is $O(m^2)$ in the worst-case. The approach we recommend was discovered by Lipski [LIPS84] and is shown in Figure 4. It employs a simple breadth-first scheme to compute the single source solution (i.e. from segment s to all other segments). A breadth-first scheme is used since all the edges weights are 1. The algorithm runs in $O(m \log m)$ time which is remarkable since this implies that every edge of the intersection graph need not be examined. This is done by not representing the graph explicitly but working directly from the rectilinear segments. When a segment is encountered it is processed and "erased" so that it is not "intersected with" later. Hence only $O(m)$ intersections are encountered, even if I is not $O(m)$. (Essentially the same scheme was found by Imai and Asano [IMAI84], though they only give an explicit discussion of depth-first search).

The outline of the algorithm is same as all breadth-first search algorithms; the key here is understanding *NEIGHBORS*. The routine returns a list L of all segments in U that are adjacent to (i.e. intersect) segment p, where U is the set of all segments that have not yet been visited. The queue used by the algorithm contains visited segments that may

```
Back_s := nil
InitializeQueue
Enqueue(s)
U := {all segments}
DELETE(U,s)
while not EmptyQueue do
    Dequeue(p)
    NEIGHBORS(p,U,L)
    for each r in L do
        DELETE(U,r)
        Enqueue(r)
        Back_r := p
    end for
end while
```

Figure 4

intersect unvisited segments. The link structure $Back_r$ points backward along the shortest path from s to r.

If *NEIGHBORS* were implemented by sequentially searching U then we derive an $O(m^2)$ time algorithm. What is desired is that the call to *NEIGHBORS* takes $O(\log m+k)$ time where $k=|L|$, the number of reported intersections. This can be done [VAIS82], however, there is a substantial penalty when U can shrink over time (which is the case here), i.e. $O(\log^3 m + k)$ time. However, Lipski has effectively accomplished the same thing using amortized worst-case analysis. In particular, each call may not take $O(\log m+k)$ time but averaged over all the $O(m)$ calls to *NEIGHBORS*, the net result is the same as if it had. This is due to the use, internally, of a disjoint set algorithm [GABO83] that can do n_1 set finds and n_2 set unions in $O(n_1+n_2)$ total amortized time.

The primary data structure used by *NEIGHBORS* is a *segment tree*, first described in [BENT80]. A node of this tree corresponds to some range of abscissae $[a,b]$ and a complicated secondary data structure is associated with each such node. These secondary structures are used to remember all horizontal segments that completely span that range and all the vertical segments in that range. Since the ranges for various nodes (on a path of the tree) intersect there is much redundancy in the whole scheme. As a result the space

requirements are $O(m \log m)$. The *DELETE* procedure runs in $O(\log m)$ amortized time. Its task is complicated by the fact that to remove an escape segment from the whole segment tree requires that we must also remove all its redundant instances as well. This scheme is well-suited for this kind of application, however, the scheme does not readily admit an $INSERT(U,p)$ to its repertoire.

From the above remarks we can see that the running time of Lipski's algorithm is $O(m \log m)$. So in conjunction with the algorithms in the previous sections, we get an optimal algorithm for the single-layer minimal bend problem in the worst-case.

EMPIRICAL RESULTS

The minimum bend algorithm in Figure 4 was implemented in C. Beneath the surface simplicity of the breadth-first search is a sea of details. The data structures are quite complicated. The secondary data structures, mentioned above, make use of a set union algorithm. The internal details of this algorithm allows us to use a complicated approach with good amortized performance. We acknowledge that for many applications the additional complexity is not justified and a simpler set union algorithm should be used [FRED84b]. However, we did implement the complete algorithm. To gauge the benefit of the overhead we also coded a straightforward breadth-first scheme. Instead of using a complete linear search for *NEIGHBORS* we used a binary search into the list of segments (sorted by ordinate and then abscissa) to speed the linear search.

Operation	Simple BFS	Lipski's BFS
Routing	19.34	9.17
Sorting	3.25	3.60
Preprocessing	0.1	8.20

Figure 5

The two algorithms were run on the same data. The data consisted of many rows of randomly placed modules. The table of Figure 5 shows the run-times, in seconds, of a trial with 2041 modules.

The most important observation is that the time needed for actual routing is much less, indicating that the complex algorithms are more efficient for many modules. However for small m the situation is reversed, as expected. The other entries in the table are harder to compare. For example, preprocessing time for the segment trees is somewhat inflated, since we made no attempt at optimizing this code.

The sorting times represent an unfair but enlightening comparison. Given the original general problem statement it is clear that we should use a general comparison-based sorting algorithm. We were seduced into using a bucket sort due to its speed. However, the two entries are irrelevant since in an actual system the input to this program is either presorted or contains a degree of sortedness that may favor one algorithm or the other.

A classical Hightower-like line-search algorithm was also implemented. Its run-time performance was approximately 4 times slower than the simple BFS algorithm for instances with 1000 modules. Its relative performance to the BFS algorithms steadily degraded as the number of modules increased.

ROUTING ON SEVERAL LAYERS

We can easily make the transition from one to several layers. We assume all the vias are known during preprocessing. Hence we can create requests $ELR < i, i, y >$ for the (horizontal) escape line segments going through a via with coordinates (i, y). We assume that every escape segment will be labeled with the names of any vias it intersects by a simple modification of our original procedure. Further, we assume the *names* of vias are small integers that can be used as indices into other structures. We will maintain an auxiliary structure, called *via–lists*, to supply those escape segments from the different layers that intersect a given via.

Clearly, we require our original data to specify the layers on which the module boundaries and the wire segments occur. Let the number of layers be l which we assume to be a small constant. There will be $2 \cdot l$ passes of our line-sweep algorithm for escape segment generation. (A horizontal and vertical orientation for each layer). Note that a via maintains one global name for all of its appearances.

To discuss minimum bend routing we must define the effect of using a via on the number of bends. We adopt a simple and reasonable convention of counting the via as one more segment in the path. (If a penalty more arbitrary than some constant is used we need to abandon the breadth-first schema we intend to use). In that case, we can adapt the Lipski shortest path algorithm.

The principal change in the algorithm is that when a segment p is taken off the queue it puts on the queue the names of any vias that it intersects that have not been visited. Each such via is immediately marked as visited in some global data structure. When the via is later taken off the queue, it is processed differently from a segment. It uses its *via—list* to put each of its unvisited intersecting segments (i.e. those in U) onto the queue. It will be faster if we implement U as a partition $U_1, U_2, \cdots, U_l$, where the segments are grouped according to layer. This implies that *NEIGHBORS* and *DELETE* will be responsible for maintaining l disjoint sets of data structures.

There are two ways to find the neighbors of a *via—list* with respect to the set U. The first is to have all appearances of a segment on all the individual *via—lists* to be linked together. Then, when the segment is removed from U, it is immediately removed from the *via—lists*. Typically this would take constant time, but if vias are co-linear then it could take $O(m)$ time. Even so, the amortized time for all such removals is still $O(m)$. If it is desired that each *DELETE* take $O(\log m)$ time then we can leave the *via—lists* alone and maintain the set U explicitly in another balanced tree to be consulted in $O(\log m)$ time.

SINGLE LAYER MINIMUM LENGTH ROUTING

To determine the minimum length routing between two terminals we recast this as a shortest path problem on the *escape graph*. The vertices of this graph correspond to the points that lie at the intersection of escape segments and to the two terminals. An edge connects two vertices if they lie on the same escape segment and the length of the edge is the corresponding Euclidean distance. A shortest path between the terminals gives a solution to our original problem. The escape graph has $O(I)$ vertices and edges, where I is the number of intersections (e.g. the number of vertices is $I+2$). By modifying the procedure for generating the escape lines we can generate this graph in $O(m \log m + I)$ time.

Using known techniques [AHO74] discussed above we can solve this problem in $O(I \log I)$ or $O(I \log m)$ time. Recall, in the worst-case that I is $O(m^2)$, but typically I is $O(m)$ so we might expect $O(m \log m)$ performance in practice. In any event this step's time complexity subsumes the preprocessing time. The only major advance is a recent algorithm by Frederickson [FRED84a] for finding shortest paths in planar graphs with p vertices that runs in $O(p\sqrt{\log p})$ time. The escape graph is planar in the single layer case, therefore this algorithm runs in $O(I\sqrt{\log m})$ time. However, the algorithm in [FRED84a] is quite complicated. It makes extensive use of algorithms for finding separators of planar graphs that are prohibitively complex for small values of m. Hence it is only of theoretic interest. However it does point the way for improved shortest path problems on other restricted classes of graphs. We are concerned with Euclidean planar graphs, in particular rectilinear planar graphs.

Wu, *et al.* [WU85] consider the problem of rectilinear shortest paths betweens points in the presence of t modules (formed from m constraints) which are x—y *convex*. They allow routing along the boundaries of the modules but our problem statement can be mapped to theirs by enlarging all our modules by one unit in each direction. They construct a reduced version of the escape graph called the *track graph* which extends the four extreme edges of a module (i.e. those that lie on the least bounding rectangle). The

two terminals also have escape lines but they only extend to the first boundary or previous escape line. Finally all the other edges are included but *not* extended into escape lines. This final set of edges may include "staircases" of vertices of degree 2 but these are removed during preprocessing and are replaced in the track graph with a single artificial edge. Hence the track graph has just $O(I')$ vertices and edges, where I' is the number of intersections of those escape lines we retained (note I' is $O(t^2)$). They give an $O(m \log t = I')$ time algorithm to construct the track graph.

They prove that the shortest paths for the original problem are shortest paths on the track graph, with the exception of the case where the two terminals are near each other. This case, where the terminals are in the same "horizontal or vertical region", is solved in a preprocessing step in constant time with the track graph. Otherwise they revert to a standard shortest path algorithm [AHO74] that runs in $O(q \log p)$ time for q edges and p vertices. Hence their algorithm runs in $O(I' \log I')$ or $O(I' \log t)$ time. Since t can be $\Omega(m)$ this algorithm is no better in the worst case. However if the modules typically have many edges then t could be much smaller than m and their algorithm would be faster. Note that the track graph is planar. Using Frederickson's result we get $O(I'\sqrt{\log t})$ which is the best known result for $x-y$ convex modules.

Another problem similar to ours was addressed by de Rezende, *et al.* [DE 85]. However they assume that the obstacles are all rectangular modules and that they are all disjoint. While they do not necessarily route over integral grid lines, we could transform our problem instance, as described above, to use their algorithm. They give an $O(m \log m)$ time algorithm for finding the shortest rectilinear path, in this case.

REMARKS

In a typical system a router is expected to route K nets. We assume that these nets have been ordered and we will use our algorithm to route each successive net. After each wire is routed that wire becomes an obstacle(s) for the next net, i.e. the parameter m increases over time. In the worst case each routing will require $\Omega(m \log m)$ time. Therefore it

would be permissible to "waste" $O(m \log m)$ time to reconstruct the data structures from scratch.

It is an important benefit of a breadth-first search algorithm that short wires are routed quickly. In practice, since most wires are short, it would be extravagant to build new data structures after each routing. Suppose that there are k updates, i.e. insertions or deletions, to the set of escape segments. These would be due to new segments bordering the previous wire and old segments cut by the wire. These can be discovered in $O(k \log m)$ amortized time. (We would repeatedly use the procedure *right* in [IMAI84]). If the data structures could then be updated in the same time, we would be done. However, as mentioned above, the operation of insertion is not supported. This is due to the asymmetric nature of the underlying set union algorithm. (Interestingly, by using set splitting we could support insertion without deletion [IMAI84]). If we return to the original Lipski algorithm [LIPS83] we find a data structure that supports both insertion and deletion in $O(\log^2 m)$ time and fetches in $O(\log^2 m + |L|)$ time. He improved these to $O(\log m \log\log m)$ and $O(\log m \log\log m + |L|)$ respectively. However he assumes that the integer coordinates come from the range 0 to $N=O(m)$. A complicated data structure that does not have this restriction but has the same time bounds appears in [FRIE85]. We can perform the fetches in $O(\log m + |L|)$ time [GUTI85] but need $O(\log^3 m)$ time for the updates and $O(m \log^2 m)$ space ($O(\log^2 m)$ time if $N=O(m)$).

Therefore we can go from one routing problem to the next in $O(k \log m \log\log m)$ time. In practice, though, we would recommed the simpler $O(k \log^2 m)$ approach. Note that we must restore the data structures after the breadth-first search since it deleted many segments. If we stack the deleted segments and restore them with matching insertions the total execution time at most doubles.

Another approach is to generate positions of the escape graph as needed [COHO84a]. This would entail using some procedure akin to *NEIGHBORS*. The savings would come from those cases where the destination was fully processed before the entire graph was

explored. However, more intriguing is a possibility of using the partial escape graph for heuristics. It is an open area of research to combine these techniques with heuristics such as those used to guide current production routers.

The final generalization we consider is interconnecting a multi-terminal net. For this problem, we recommend the technique of multiplexing the two-terminal routing algorithms. Let t be the number of terminals in the net. Initially, there are t simultaneous active searches being conducted, where the i^{th} search is associated with the i^{th} terminal in the net. As escape segments are visited, they are marked with the terminal associated with the search. Whenever, a search visits a segment that was previously visited by another search, the search that did not perform the initial visit is halted. By establishing back pointers during the searches, a spanning tree interconnecting all terminals can be constructed from those segments that were visited by different searches. This spanning tree is not guaranteed to be optimal. However in practice it is quite good. For example, applying the multiplexed search to a net with two-terminal problem increases the number of bends in the solution by at most one bend. Variations of the searches are also possible to increase the quality of the solution. As the total number of segments visited is $O(m+t)$, the running time of the basic multiplexed search is comparable to the running time for a single search with a two terminal net.

ACKNOWLEDGEMENTS

The work of James Cohoon was supported in part by NSF grant DMC-8505354.

REFERENCES

[AHO74] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[BENT79] J. L. Bentley and T. Ottmann, Algorithms for Reporting and Counting Geometric Intersections, *IEEE Transactions on Computers*, C-28(9), September 1979, pp. 643-647.

[BENT80] J. L. Bentley and D. Wood, An Optimal Worst-Case Algorithm for Reporting Intersections of Rectangles, *IEEE Transactions on Computers*, C-29(7), July 1980, pp. 571-577.

[COHO84a] J. P. Cohoon, A Line Intersection Algorithm for the Routing Problem, DAMACS 84-02, University of Virginia, 1984.

[COHO84b] J. P. Cohoon, A Fast Line Intersection Routing Method for Optimal Wirings, *22nd Allerton Conference on Communication, Control, and Computing*, 1984.

[DE 85] P. J. de Rezende, D. T. Lee and Y. F. Wu, Rectilinear Shortest Paths with Rectangular Barriers, *Proceedings of Symposium on Computational Geometry*, Baltimore, MD, June 1985, pp. 204-213.

[FRED84a] G. N. Frederickson, Fast Algorithms for Shortest Paths with Applications, Computer Science Dept. Tech. Rep. 486, Department of Computer Science, Purdue University, July 1984.

[FRED84b] M. L. Fredman and R. E. Tarjan, Fibonacci Heaps and Their Uses in Improved Network Optimization, *24th Annual Symposium on Foundations of Computer Science*, October 1984, pp. 338-346.

[FRIE85] O. Fries, K. Mehlhorn and S. Naher, Dynamization of Geometric Data Structures, *Proceedings of Symposium on Computational Geometry*, June 1985, pp. 168-176.

[GABO83] H. N. Gabow and R. E. Tarjan, A Linear-Time Algorithm for a Special Case of Disjoint Set Union, *Proceedings 15th Annual ACM Symposium on Theory of Computing*, Boston, MA, 1983.

[GUTI84] R. H. Guting, An Optimal Contour Algorithm for Iso-oriented Rectangles, *Journal of Algorithms*, 5, 1984, pp. 303-326.

[GUTI85] R. H. Guting, Fast Dynamic Searching in a Set of Isothetic Line Segments, *Information Processing Letters*, 21, October 7, 1985, pp. 165-171.

[HEYN80] W. Heyns, W. Sansen and H. Beke, A Line-Expansion Algorithm for the General Routing Problem with a Guaranteed Solution, *17th Design Automation Conference Proceedings*, Minneapolis, MN, 1980, pp. 243-249.

[HIGH69] D. W. Hightower, A Solution to the Line-Routing Problem on the Continuous Plane, *6th Design Automation Workshop Proceedings*, Miami Beach, FL, 1969, pp. 1-24.

[HIGH74] D. W. Hightower, The Interconnection Problem: a Tutorial, *Computer*, 7(4), April 1974, pp. 18-32.

[HORO78] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD, 1978.

[IMAI84] H. Imai and T. Asano, Dynamic Segment Intersection Search with Applications, *25th Annual Symposium on Foundations of Computer Science*, October 1984, pp. 393-402.

[LEE61] C. Y. Lee, An Algorithm for Path Connections and Its Applications, *IRE Transactions on Electronic Computers*, EC-10(3), September 1961, pp. 346-365.

[LIPS83] W. Lipski, Jr., Finding a Manhattan Path and Related Problems, *Networks*, 13, 1983, pp. 399-409.

[LIPS84] W. Lipski, Jr., An $O(n \log n)$ Manhattan Path Algorithm, *Information Processing Letters*, **19**, August 31, 1984, pp. 99-102.

[RUBI74] F. Rubin, The Lee Path Connection Algorithm, *IEEE Transactions on Computers*, C-23(9), September 1974, pp. 907-914.

[SOUK81] J. Soukup, Circuit Layout, *Proceedings of the IEEE*, **69**(10), October 1981, pp. 1281-1304.

[VAIS82] V. K. Vaishnavi and D. Wood, Rectilinear Line Segment Intersection Layered Segment Trees, and Dynamization, *Journal of Algorithms*, 3(2), June 1982, pp. 160-176.

[WU85] Y. F. Wu, P. Widmayer, M. D. F. Schlag and C. K. Wong, Rectilinear Shortest Paths and Minimum Spanning Trees with Rectilinear Barriers, *Proceedings of the WG'85 (International Conference on Graphtheoretic Concepts in Computer Science)*, Wurzburg, Germany, 1985, pp. 409-420.