University of California
Santa Cruz

Geometric Transformations for a Rubber-band Sketch

A thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Engineering

by

David Joseph Staepelaere

September 1992

The thesis of David Joseph Staepelaere is approved:

Wayne Wei-Ming Dai

Tracy Larrabee

Martine Schlag

Dean of Graduate Studies and Research

Contents

List of Figures

List of Tables

Geometric Transformations for a Rubber-band Sketch

David Joseph Staepelaere

ABSTRACT

The flexible rubber-band sketch is a useful representation for routing interconnect. In addition to supporting an incremental design style, rubber-bands provide a flexible framework for generating layout under performance constraints. However, due to reasons of compatibility between CAD tools, it may be necessary at times to convert a rubber-band sketch to a more restricted geometry such as rectilinear or octilinear wiring. This paper presents an efficient method, based on the enhanced plane sweep, for converting a rubber-band sketch to a topologically equivalent rectilinear or octilinear wiring with minimum wire length. A sketch with n rubber-band segments can be converted to a restricted geometry with m segments in $O(n \log n + m)$ time. In addition to guanateeing minimum wire length, the technique uses heuristic methods to reducing the total number of jogs.

Acknowledgements

I would like to thank my advisor Dr. Wayne Dai. Not only did he introduce me to the field of CAD for VLSI design, but he suggested the topic for this thesis and provided a great deal guidance, encouragement, and support along the way. I also want to thank Dr. Tracy Larrabee and Dr. Martine Schlag for serving on my thesis reading committee.

Professor Masao Sato was invaluable in helping formulate many of the concepts and solution techniques used in the SURF system. I thank him for his comments and suggestions regarding the geometric wiring transformation problem.

I also thank the other students who have worked on the SURF project—without them there would be no application for this work. I would especially like to thank: Tal Dayan for providing many useful and insightful comments regarding this work; and Mark Fitzpatrick for his efforts in implementing the octilinear portions of the project.

Tom Hughes of North Carolina State University (now at IBM in Research Triangle Park, NC) has also studied the problem of transforming topological routing to precise geometry. I would like to thank him for his comments on my work and on related work.

I am also indebted to Advanced Packaging Systems of San Jose, CA for supporting this work during the summer of 1990 and providing some of the examples used in this thesis. Both Joe Wild and Sherri Cramphorn at APS were especially patient in helping me to understand some of the practical details of MCM packaging that influence the layout process.

1. Introduction

The Santa Cruz ULSI Routing Framework CAD tool (SURF) is an area router designed primarily for routing multi-chip module (MCM) substrates. MCMs are a high density packaging technology in which multiple devices are mounted and interconnected on a single multi-layer substrate. These devices usually consist of unpackaged integrated circuit chips (dice) and discrete components. In addition to signal routing layers, the substrate typically provides power and ground planes. The substrate routing problem consists of determining the configuration of the signal layers so as to provide the interconnect for the devices. Since the area under the devices is generally available for wiring, the problem is one of *area routing*.

1.1 Overview of the SURF System

The SURF routing system transforms a netlist description of an interconnect pattern to a final precise layout by a series of successive refinements. These refinements are performed in four phases: global routing, local routing, spoke creation, and geometric wiring. The final phase, geometric wiring, will be the subject of this thesis.

1.1.1 Topological Routing

The first two phases of the SURF routing process transform a netlist into a rough form of multi-layer routing called *topological routing* [CS84, LM85, DDS91]. In topological routing, wires are represented as zero-width flexible curves with fixed endpoints. These wires may be continuously moved and stretched as long as they do not cross other wires or objects. For this reason, topological routing is useful for specifying the paths that wires take relative to one another without giving the exact positions of the wires. Two topological routings are said to be *topologically equivalent* if one may be transformed to the other by a process of continuous deformation. In order to represent wirings uniquely, SURF uses a canonical form of topological wiring called a *rubber-band sketch* to represent each layer of the interconnect

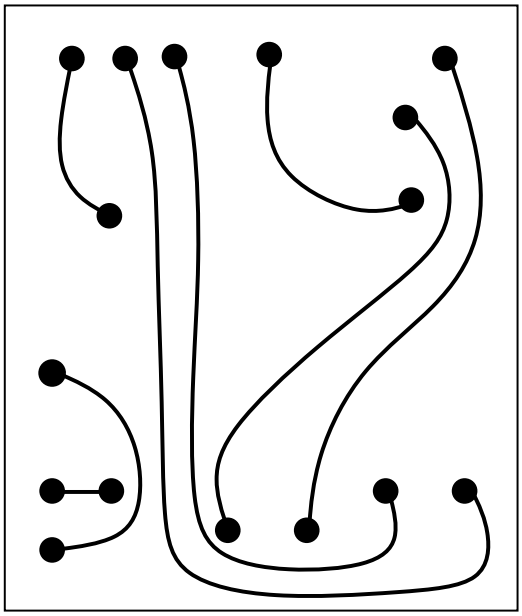
[CS84, LM85]. In rubber-band routing, individual wires are treated as elastic rubber-bands. Rubber-bands naturally contract to the shortest possible length that still maintains the same topology. Figure 1.1 (a) shows a planar wiring pattern and Figure 1.1 (b) shows the topologically equivalent rubber-band form.

Representing interconnect as rubber-bands instead of precise rectilinear or octilinear wiring (hereafter referred to collectively as *geometric wiring*) has many advantages. Since rubber-band routing has, in general, fewer wire segments than geometric wiring, it contains less unnecessary information. The flexible nature of rubber-bands allows designs to be modified incrementally supporting an iterative design process [DKJS90]. This flexibility also permits designs using variable-width traces or traces whose width is a function of length—this is especially important for low-power high-speed designs where line resistance can be used to damp reflections in place of terminating resistors [Dai91]. In addition, the rubber-band representation provides an easier and more intuitive wiring model for manual layout editing than does geometric wiring.

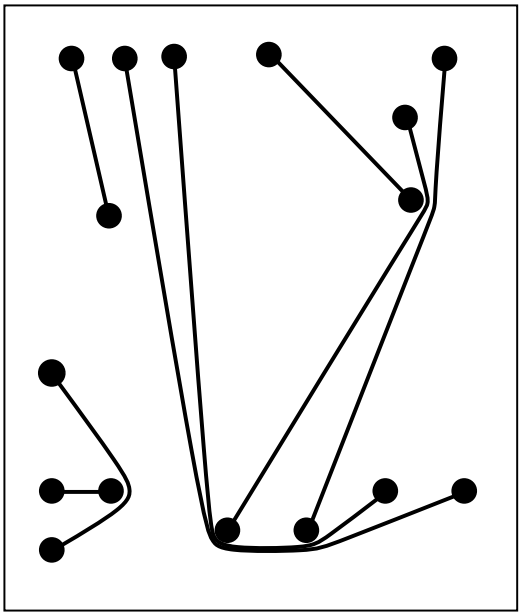
1.1.2 Spoke Creation

Once the global and local routing is complete, the topological wiring is transformed to a final geometric wiring in two steps: spoke creation and transformation to geometric wiring [LM85, DKS91, Kon92]. The goal of the spoke creation step is to transform the topological wiring represented by the rubber-band sketch into an equivalent topology in which all of the width and spacing constraints are met. Such a sketch is referred to as an *extended rubber-band sketch*. The spoke step does this by radiating *spokes* from the endpoints of wires in the rubber-band sketch. A spoke is an open-ended line segment used to "prop-up" the rubber-band routing in the neighborhood of these endpoints. See Figure 1.1(c) and Figure 1.2. These spokes push wiring away from the endpoints to allow proper spacing.

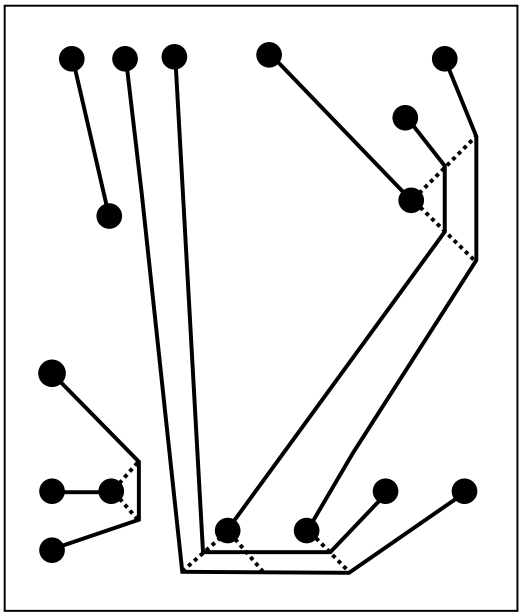
Before proceeding further, it is helpful to present some concepts used to determine the routability of a rubber-band sketch. A *cut* is a line segment between two points in the sketch that intersects with no other points or objects [CS84]. The *capacity* of a cut is the



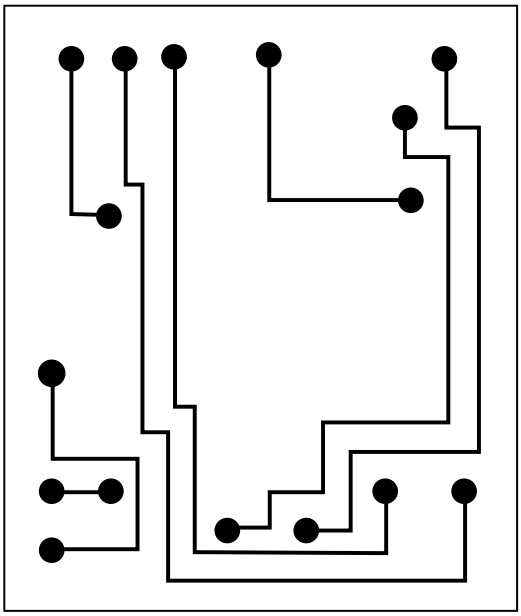
(a) Topological Wiring



(b) Rubber-band Equivalent



(c) Extended Rubber-band Sketch



(d) Geometrical Wiring

Figure 1.1: Four Views of a Sketch

This figure shows various wiring patterns that are topologically equivalent. These patterns correspond to different stages of the transformation from rubber-band sketch to precise geometric wiring. Figure (a) shows arbitrarily positioned paths, (b) shows the rubber-band equivalent, (c) shows the extended rubber-band sketch and (d) shows a rectilinear wiring.

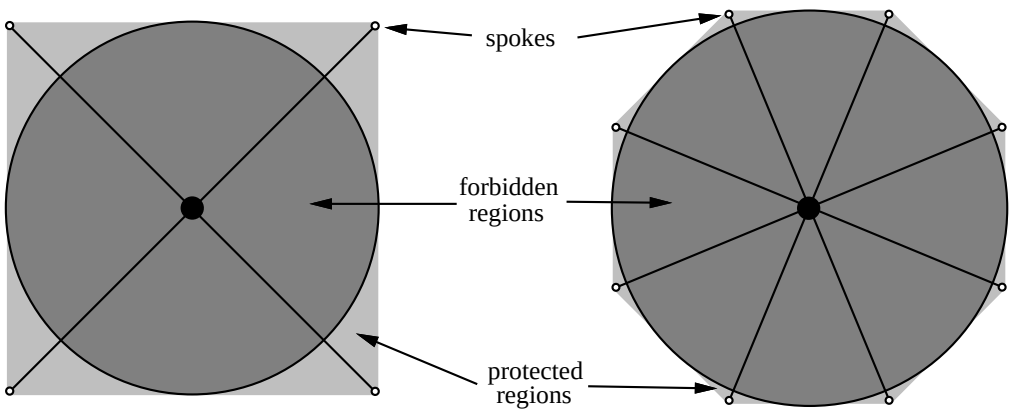


Figure 1.2: Potential Spokes of a Point

number of wires that may safely cross the cut without violating the spacing constraints. The capacity of a cut depends on the positions of its endpoints and the wiring pattern used. For a cut $\overline{pq}$ with enpoints $p = (x_1, y_1)$ and $q = (x_2, y_2)$ the rectilinear capacity is given in Equation 1.1 and the octilinear capacity is given in Equation 1.2. For simplicity, in these expressions wires are assumed to have zero width and require unit separation.

$$cap(\overline{pq}) = max\{|x_1 - x_2|, |y_1 - y_2|\} - 1 \tag{1.1}$$

$$cap(\overline{pq}) = \begin{cases} |x_1 - x_2| - 1 & \text{if } (\sqrt{2} - 1)|x_1 - x_2| \geq |y_1 - y_2| \\ |y_1 - y_2| - 1 & \text{if } (\sqrt{2} - 1)|y_1 - y_2| \geq |x_1 - x_2| \\ \frac{|x_1 - x_2| + |y_1 - y_2|}{\sqrt{2}} - 1 & \text{otherwise} \end{cases} \tag{1.2}$$

The *flow* of a cut is the number of wires crossing the cut. The routability of a sketch can be expressed in terms of the flow and capacity of its cuts: *a sketch is routable if and only if no cut has a flow that exceeds its capacity* [CS84]. Generating an extended rubber-band sketch is sufficient to determine the routability of a sketch. That is, *a sketch is routable if and only if it can be transformed into the corresponding extended rubber-band sketch* [DKS91, Kon92]. If the rubber-band sketch represents a routable topology, the result of the spoke creation

step will be a fully extended rubber-band sketch. Otherwise, if the sketch is not routable, the spoke creation step will locate all of the endpoints that participate in overflowing cuts.

If the spoke creation step succeeds, the resulting extended rubber-band sketch is a legal routing of the topology—all of the spatial constraints have been met. In general, the extended rubber-band sketch will have shorter wire lengths and fewer jogs than the corresponding geometric wiring. However, for reasons of compatibility between tools, it is often necessary to produce a final wiring in rectilinear or octilinear geometry.

1.1.3 Geometric Wiring

The final transformation to geometric wiring is the subject of this thesis. The geometric wiring problem is defined as follows. Given a sketch in which all of the spatial constraints have been met (extended rubber-band sketch) and a geometric wiring pattern (rectilinear or octilinear), produce a routing that:

- conforms to the specified geometry,
- is topologically equivalent to the input routing,
- meets the width and spacing constraints, and
- has minimum wire length.

Because the positions of the wire endpoints in the extended rubber-band sketch are fixed, the minimum wire length of the sketch is merely the sum of the individual rubber-band segment lengths. Let S be the set of straight-line rubber-band segments in the extended rubber-band sketch. For wiring pattern m, the total wire length of sketch S is given by

$$L_m(S) = \sum_{s \in S} ||s||_m \tag{1.3}$$

where $||s||_m$ is the length of segment s in the proper distance metric (Manhattan or octilinear distance).

In addition to the above constraints, it is also desirable to reduce the number of jogs in the final sketch. The reasons for reducing the number of jogs include:

- improving the yield,

- decreasing noise due to reflections,
- reducing the size of the output, and
- making the result simpler and easier to understand.

1.2 Organization

The rest of this thesis is organized as follows: A top level overview of the geometric wiring transformation process is given in Chapter 2. Chapters 3, 4, and 5 discuss the segment ordering, initial transformation, and jog removal phases of the algorithm in detail. Some results are presented in Chapter 6. Possible future extensions to the algorithm are described in Chapter 7. Chapter 8 presents some previous and related work. Finally a summary is given in Chapter 9.

2. Overview of Approach

The general approach used to transform an extended rubber-band sketch to a precise geometric wiring is to transform each rubber-band segment to a wire that conforms to the required geometry, has the minimum length, and maintains the necessary spacing to other wires in the design.

2.1 Basic Method

The wiring transformation described in this paper relies on an algorithmic technique known as the *enhanced plane sweep* [SO86]. The enhanced plane sweep is an extension of the traditional plane sweep. In addition to the normal scan-line used in the plane sweep, the enhanced plane sweep maintains a second scan-line known as the *previous boundary*. The previous boundary is composed of a set of line segments. It follows the first scan-line and records the boundaries of polygons visible from the first scan-line. This allows the algorithm to maintain a limited history about objects that the primary scan-line has passed.

The top-level algorithm for transforming an extended rubber-band sketch to geometric wiring is presented in Figure 2.1. Each layer of a multi-layer extended rubber-band sketch is transformed separately. Within each layer there are three steps. The first step, OrderSegments, determines the proper plane sweep order for the rubber-band segments of the layer. The next step, TransformLayer, produces an initial geometric wiring (see Figure 2.2 (b)). Finally, StraightenLayer improves the initial transformation by performing jog-removal (see Figure 2.2 (c)).

The approach used by TransformLayer to produce the initial geometric wiring is to sweep the extended rubber-band sketch from bottom to top, transforming the rubber-band segments (the individual straight-line segments that make up a rubber-band) as they are reached by the scan-line. The previous boundary maintains the portion of the previously transformed wiring that is visible from (below) the scan-line. Segments are transformed by pushing them down "on top" of the previously transformed segments. When segments

```
GeometricWiring(E : ERBS, w : Wiring type) {
    W      : Geometric Wiring
    S      : Ordered List of Segments
    T      : Initial Transformation

    for each layer E_i {
        S ← OrderSegments(E_i)
        T ← TransformLayer(S, w)
        W_i ← StraightenLayer(T, w)
    }
    return W
}
```

Figure 2.1: Pseudo-code for GeometricWiring

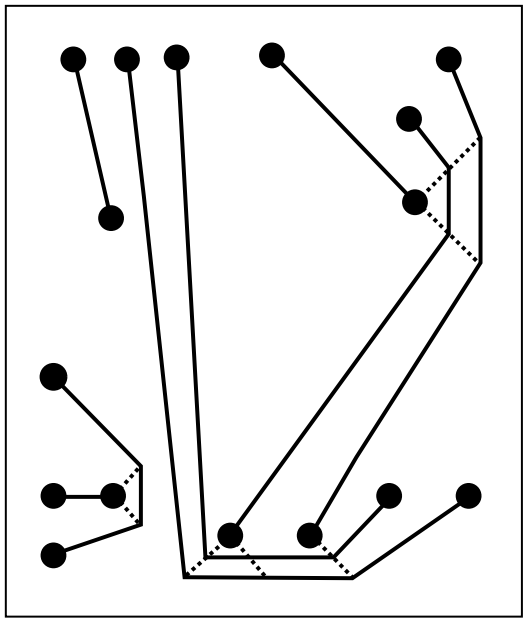
are pushed down, they are maintained within the lower triangle of a parallelogram defined by the positions of their endpoints and the current transformation method (rectilinear or octilinear). Figure 2.3 shows an example of a rubber-band segment being transformed within its lower triangle. The fact that the wiring may be completed within these triangles is guaranteed by the spoke creation step.

Once the initial bottom-up sweep is finished, StraightenLayer uses a second plane sweep to reduce the total number of jogs. The second sweep is very similar to the first. It sweeps the plane in the opposite direction (from top to bottom). At each step, the geometric sections corresponding to a single rubber-band segment are compared with the previous boundary. If a section can be slid upwards and aligned with a higher section without conflicting with the previous boundary, this operation is performed. Each such operation removes two jogs from the transformation.

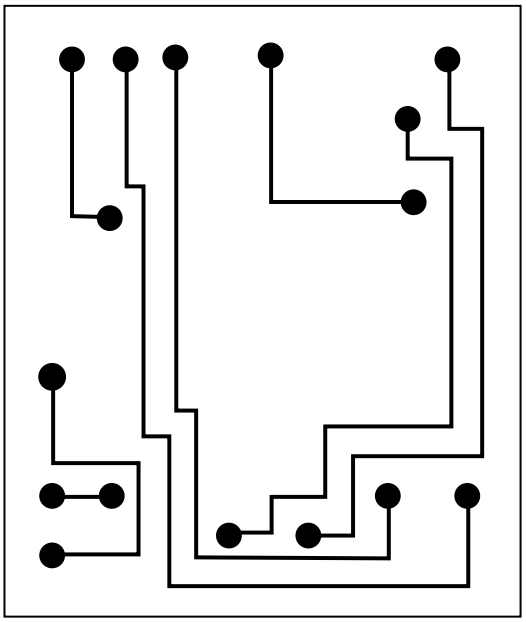
2.2 Why does it work?

In order for a geometric wiring to be correct, it must satisfy the criteria presented in Chapter 1. That is, it must:

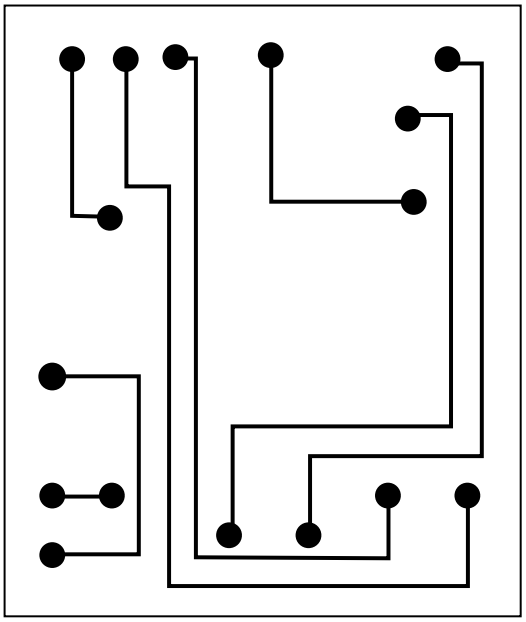
- be topologically equivalent to the input sketch



(a) Extended Rubber−band Sketch



(b) Initial Geometrical Wiring



(c) Straightened Geometrical Wiring

Figure 2.2: Simple Example of Geometric Wiring Process

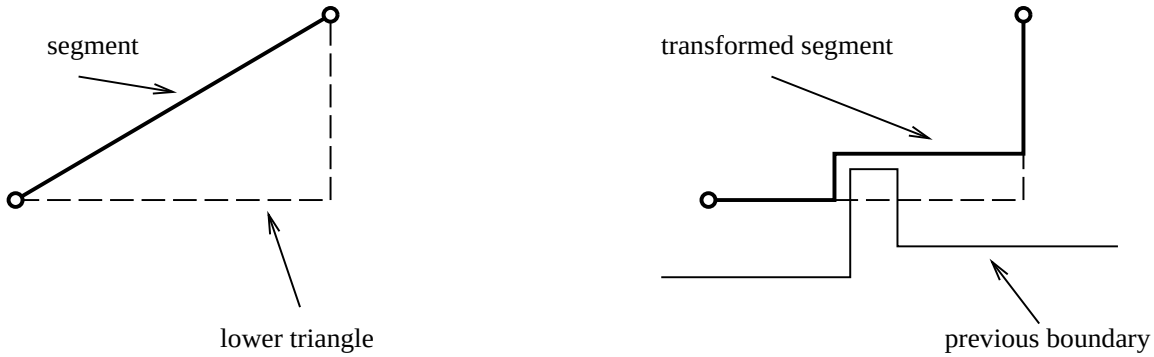


Figure 2.3: Segment Transformed within Lower Triangle

- conform to the proper geometry (rectilinear, octilinear)
- meet the width and spacing constraints, and
- have minimum wire length.

The first requirement (proper geometry) is maintained by the details of the specific algorithms that produce and straighten the transformation.

The reason that the final wiring is topologically equivalent to the initial extended rubber-band sketch is that each wiring transformation can be viewed as a continous deformation (homotopic transformation). When a segment is transformed, it is pushed down on top of the current previous boundary. Because the segment ordering guarantees that a segment will not be pushed down before the segments below it have been transformed, the push-down operation will not change the topology of the sketch. Similar reasoning applies to the jog-removal phase.

When a rubber-band segment is pushed down during the initial transformation, it always maintains minimum design rules from the segments below it in the previous boundary. If the wires in the input sketch were too tightly spaced, the result would be that during transformation, one or more segments would not be able to connect to their endpoints while maintaining the proper spacing from the previous boundary. However, the spoke creation phase guarantees that there is always enough space for such a transformation. Kong shows that all wires in an extended rubber-band sketch can be transformed to geometric wiring by replacing them with their *S-Route*'s [Kon92]. The S-Route of a rubber-band segment

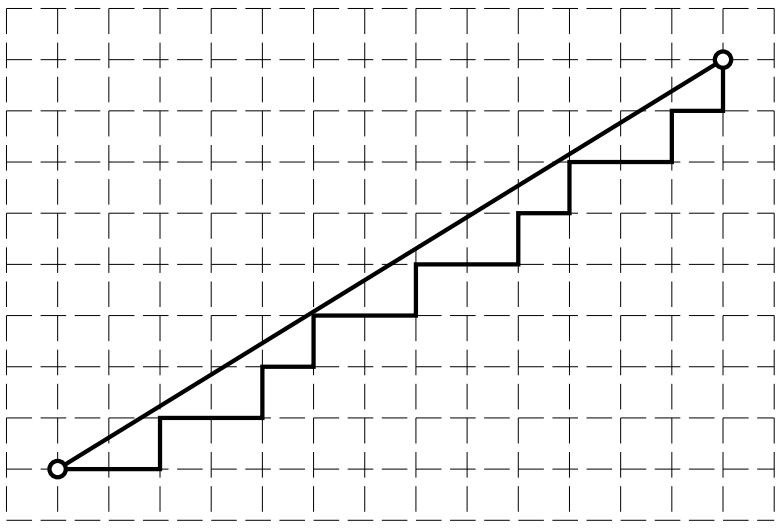


Figure 2.4: Rectilinear S-Route for a Rubber-band Segment

is the transformation that stays "closest" to the original segment without going above it (See Figure 2.4). Since the segments are transformed in bottom-up order, the portion of the previous boundary below the current segment will only contain transformations of segments that are below the current segment. Since all of these segments were transformed without exceeding their S-Routes, the current segment can be transformed without exceeding its own S-Route.

Since the straightening phase always has a design rule correct wiring and only repositions sections when the spacing allows, there is no danger of producing a spacing violation during this step.

In order to ensure that the minimum wire length for the segment is achieved, restrictions are placed on the types of wire segments used in the transformation. The first restriction is that the transformation will be monotonic. Specifically, if the transformation of an individual rubber-band segment is traced from its lower endpoint towards its upper endpoint, the path will be non-decreasing in y.

The second restriction used to ensure minimum wire length is to restrict the choice of slopes for the individual wire segments. There are three different cases—one for rectilinear and two for octilinear. Consider transforming a rubber-band segment with positive slope

(the left endpoint is the lower endpoint). The three cases are described below:

$\mathcal{R}$. The rectilinear case uses horizontal and vertical wire segments. See Figure 2.5(a). In this case, the transformation of the segment will lie entirely within the rectangle defined by its endpoints.

$\mathcal{O}1$. The second case is for the octilinear transformation of a rubber-band segment whose slope is greater than or equal to one. In this case, vertical and diagonal ($slope = 1$) segments are used as in Figure 2.5(b). The geometric transformation of such a segment will lie entirely within the octagonal parallelogram defined by its endpoints.

$\mathcal{O}2$. The third case is used to transform a rubber-band segment with slope less than one to octagonal geometry. This case uses horizontal and diagonal wire segments. The final wiring will be contained within the octagonal parallelogram as shown in Figure 2.5(c).

In all the cases mentioned above, the final wiring will be monotonic and restricted to the slopes of two adjacent sides of the proper "routing parallelogram". Clearly, the length of the transformation of a segment will be equal to the sum of the lengths of these two sides. Since this length is equal to the distance between the endpoints of the segment in the proper distance metric, the transformation is obviously of minimum length as defined above.

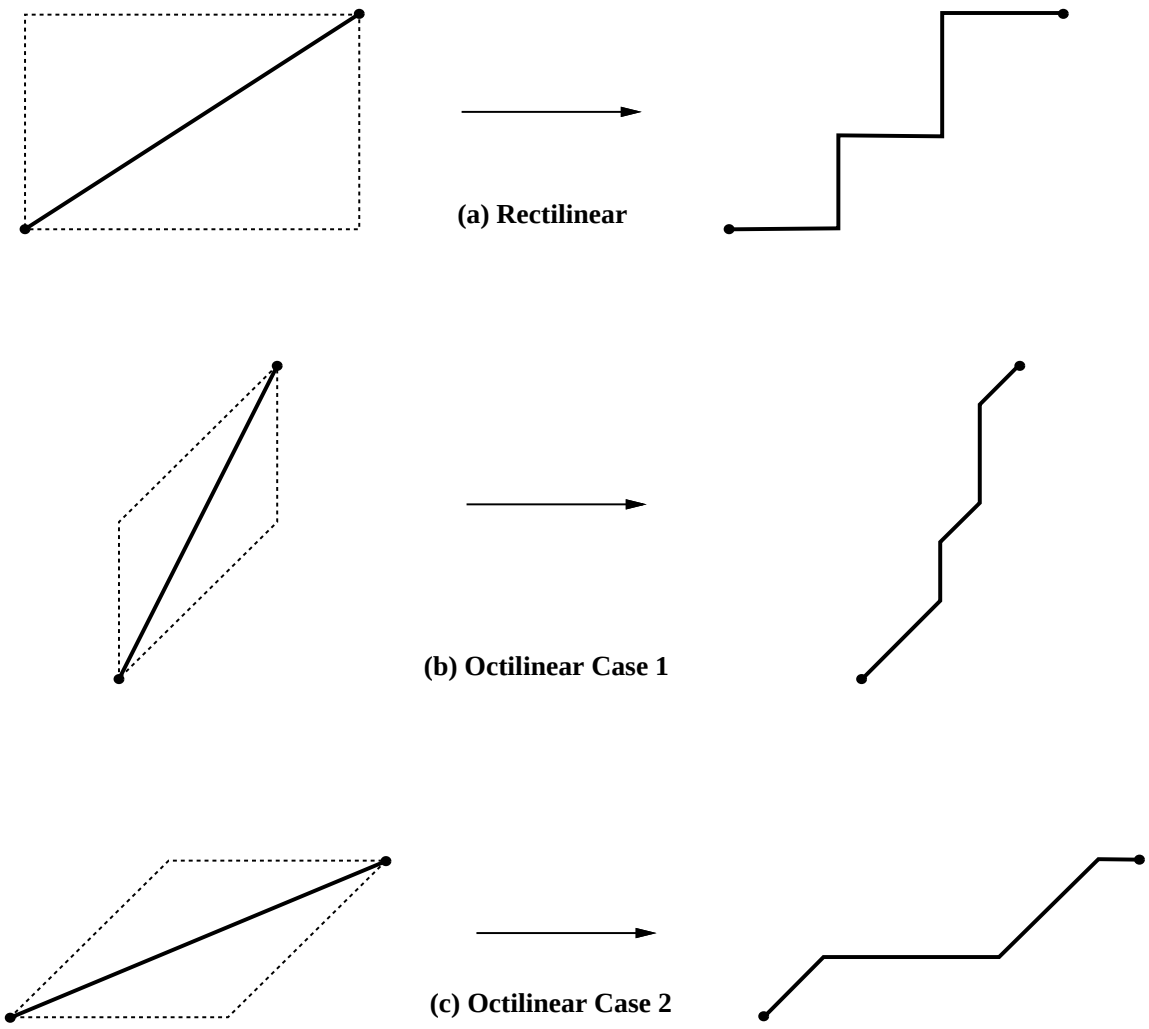


Figure 2.5: Segment Routing Parallelograms

3. Segment Transformation Order

In order for the transformation technique to succeed, the segments must be visited in "bottom-up" order. That is, the segments must be ordered so that when a given segment is transformed, all the segments "below" it have already been transformed. It is easy to see that merely sorting the segments by y-value is not sufficient. For example, in Figure 3.1 the y-projections of segments s_1 and s_3 are the same and the projections of s_2 and s_4 are also the same. However, s_1 must be after s_2, but s_3 must be before s_4. We will define the proper ordering of segments in terms of a relation called *precedes*.

Definition: Given two elements, a and b, from a set of non-intersecting, open-ended line segments S, we say a precedes b ($a \prec b$) if there exists a vertical line l that intersects segments a and b at points p_a and p_b and that p_a is below p_b ($p_a.y < p_b.y$). See Figure 3.2.

It is helpful to make a few observations about the precedes relationship:

- Precedes is anti-reflexive ($a \not\prec a$).
- Precedes is anti-symmetric ($a \prec b \Rightarrow b \not\prec a$). This can be seen by noting that for both $a \prec b$ and $b \prec a$ to hold, it would be necessary for the two segments to intersect. See Figure 3.3 (a).
- Transitivity does not hold for precedes. That is $a \prec b$ and $b \prec c$ do not imply that $a \prec c$. See Figure 3.3 (b).

If the precedes relationship is going to be used to order the segments of S, it must be shown that there are no "cycles" in the relation.

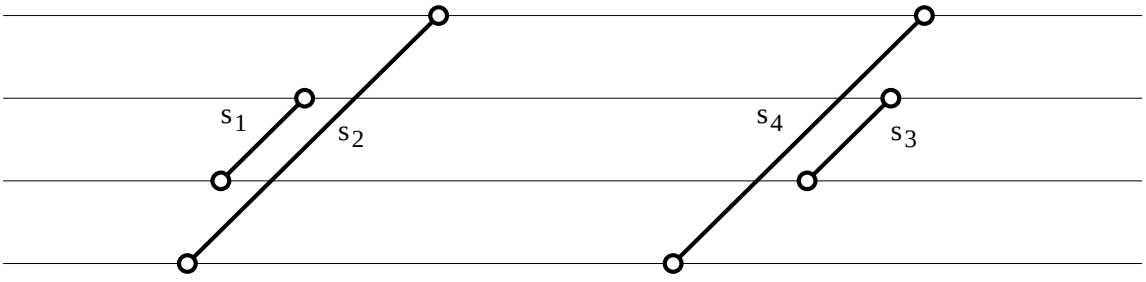
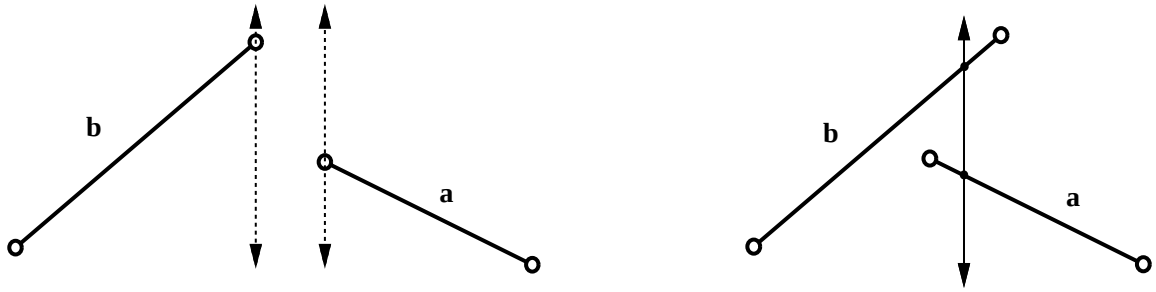
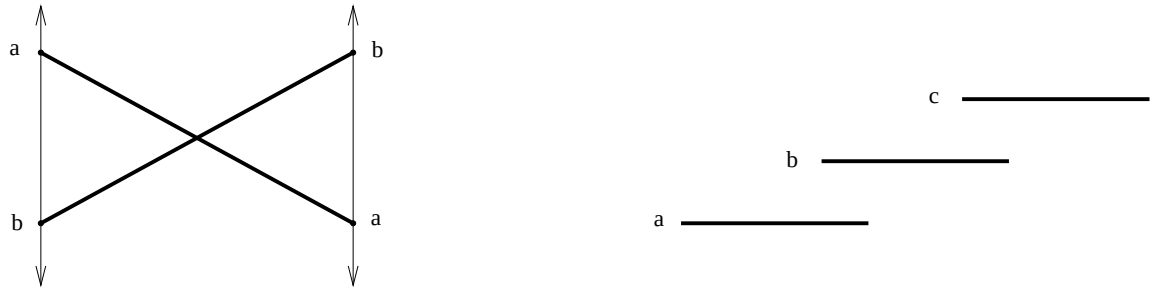


Figure 3.1: Sorting by y is not a sufficient ordering



(a) no preceeds relationship **(b) segment a preceeds b**

Figure 3.2: Precedes Relationship



(a) precedes is anti−symetric **(b) transitivity may not hold**

Figure 3.3: Properties of the precedes relation

Lemma 1: *There are no cycles in the precedes relation.*

Proof

Case I: cycles of length 2

There are no cycles of length 2 because precedes is anti-symmetric.

Case II: cycles of length > 2

The proof will be by contradiction. Let $C = \{x_0, x_1, \cdots, x_n\}$ be the smallest subset of S that forms a cycle under the precedes relation. Assume that $x_0 \prec x_1 \prec \ldots \prec x_n \prec x_0$. The vertical line l_i will be a line that intersects segments x_i and x_{i+1} (l_n intersects x_n and x_0). Assume, without loss of generality, that l_0 (crossing x_0 and x_1) is the leftmost vertical line (has the lowest x coordinate). The segments x_0 and x_1 each cross at least one more vertical line (l_n and l_1). If l_1 is left of l_n, then segment x_0 must also cross this line on its way to l_n.

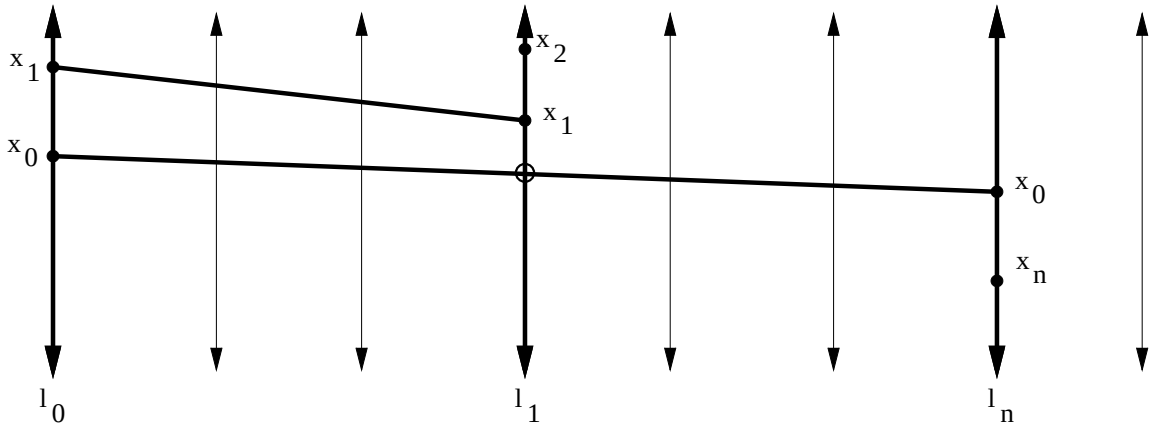


Figure 3.4: Precedes has no cycles

See Figure 3.4. Since x_0 and x_1 cannot intersect, x_0 must cross l_1 below x_1. This means that x_0 is also below x_2 on l_1. So $x_0 \prec x_2$. This contradicts the assumption that C is the smallest subset of S containing a cycle. The case for l_n left of l_1 is similar using x_1 above x_n at l_n ($x_n \prec x_1$). □

For an ordering to be legal, it must satisfy all of the precedes relationships. The sequence $(s_1, s_2, \ldots, s_n)$ is a legal ordering of the segments of S, if whenever s_i precedes s_j then $i < j$.

Between n line segments there are $O(n^2)$ precedes relationships. However, for the purposes of ordering, some of these relationships are redundant. For example, assume that three segments a, b, and c from S have the following precedes relationships: $a \prec b$, $a \prec c$, and $b \prec c$. Ordering a before b and b before c will ensure that a is before c. So, in this case, $a \prec c$ is redundant and does not need to be explicitly considered.

Our strategy will be to build a directed acyclic graph G whose vertices are the segments of S. An edge $(u, v) \in E(G)$ will mean that u precedes v. This graph will be constructed with a sufficient number of edges to ensure that for $a, b \in S$ if a precedes b, then a will be before b in a topological ordering of G.

In order to define G, it is helpful to define two functions that map points to segments. The function $above(p)$ will map to the segment immediately above the point p. If no such segment exists, $above(p)$ will map to an imaginary sink segment s_{n+1}. Similarly, $below(p)$

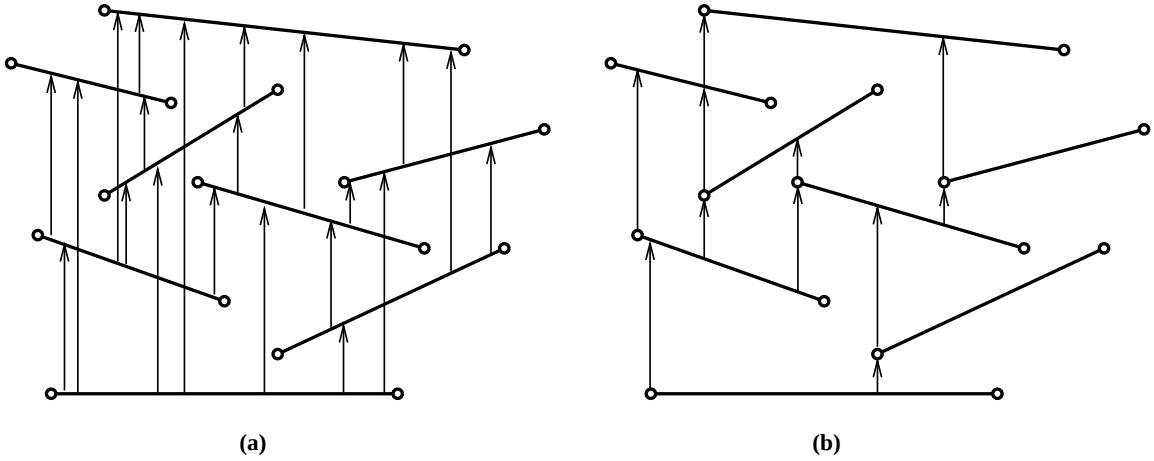


Figure 3.5: The set of precedes relationships and the graph G

map to the segment immediately below the point p (or, if none exists, an imaginary source segment s_0). Note that for segment $s \in S$ containing the point p, $s \prec above(p)$ and $below(p) \prec s$.

The graph G will be constructed to capture the precedes relationships around the left endpoints of the segments. Each segment will have an edge *from* the segment below its left endpoint; and an edge *to* the segment above its left endpoint. That is for all $s \in S$ with left endpoint (x, y),

$$(below(x + \epsilon, y), s) \in E(G)$$

and

$$(s, above(x + \epsilon, y)) \in E(G)$$

for $\epsilon \rightarrow 0^+$. The reason for the ϵ's is to ensure that the segments are treated as open-ended. In practice it is sufficient for ϵ to be smaller than the length of the smallest x-projection of all the segments in S. Figure 3.5(a) shows all of precedes relations for a set of segments. Figure 3.5(b) shows the relationships captured by the graph G.

Lemma 2: *The graph G is acyclic.*

Proof

Figure 3.6: Simple Cases of Left-endpoint Rule

This can be seen directly from Lemma 1. Each edge in G expresses some precedes relationship in S. Since G represents a subset of these relationships and there are no cycles in precedes, there are no cycles in G. □

Lemma 3: *For two segments $a, b \in S$, if $a \prec b$ then there will exist a path $a \rightsquigarrow b$ in the graph G.*

Proof

The proof will be by induction on the size of the set S.

Base: $|S| = 2$

Since $a \prec b$, either the left endpoint of a is below b or the left endpoint of b is above a. See Figure 3.6. In each of these cases, the "left endpoint rule" will capture the precedes relationship as an arc from a to b in G.

Inductive Step: $|S| > 2$

In this case we will assume that b is above the left endpoint of a. (The case of a below the left endpoint of b is similar).

Case I: There are no segments between the left endpoint of a and segment b.

In this case there is an explicit edge $(a, b) \in E(G)$.

Case II: Some segments are between the left endpoint of a and segment b.

Let x be the segment immediately above the left endpoint of a. See Figure 3.7. The definition of G guarantees that $(a, x) \in E(G)$. Since the vertical line drawn through the left endpoint of a intersects segment x below segment b it is clear that $x \prec b$.

Let $S' = S - a$ and let G' be the graph built from the segments of S'. Since $|S'| < |S|$ and $x \prec b$, the inductive hypothesis guarantees that there exists a path $x \rightsquigarrow b$ in G'. We

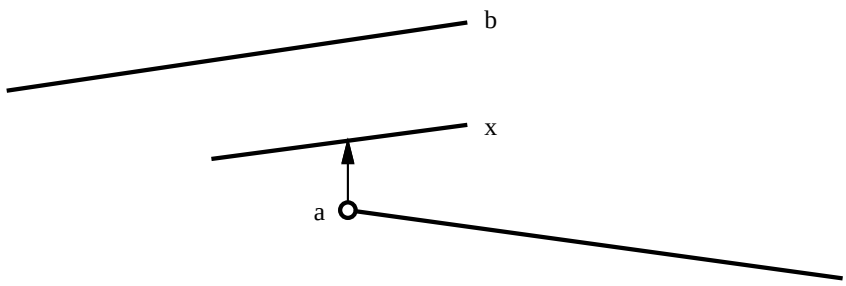


Figure 3.7: Illustration for inductive case II of Lemma 3

will now show that a similar path must also exist in G. The graph G' may be different from G in two ways:

1. edges incident on a in G will be absent from G', and
2. G' may contain additional edges that were "blocked" by the segment a in G.

The edges missing from G' do not prevent the path $x \rightsquigarrow b$ from existing in G. However, suppose the path $x \rightsquigarrow b$ in G' contains an edge (u, v) that was added as a result of removing a. The path $x \rightsquigarrow u$ in G' represents a sequence of precedes relationships $x \prec \cdots \prec u$. Since (u, v) was not in G, it was blocked by a and so $u \prec a$. Since $a \prec x$, we have a cycle of precedes relationships $x \prec \cdots \prec u \prec a \prec x$. This contradicts Lemma 1. So we know that the path $x \rightsquigarrow b$ must also exist in G.

So there is a path $a \rightarrow x \rightsquigarrow b$ in G. □

Lemmas 2 and 3 ensure that a topological ordering of the graph G is a legal transformation order for the set of segments S.

3.1 Producing the graph G

A left-to-right plane sweep may be used to generate the graph G for an extended rubber-band sketch. The pseudo-code for this function is shown in Figure 3.8. First, the endpoints of the line segments are sorted into scan-lines by increasing x-order. This list is then traversed and an *active edge list* (AEL) is maintained in the following fashion: if the endpoint is a left endpoint, its segment is placed on an *insert* list, if the endpoint is a right endpoint,

```
BuildArcs(S) {

    G = (S, ∅)
    AEL ← ∅ /* active edge list */
    sort endpoints of S by x into scan-lines
    for each scan-line {
        delete ← segments with right endpoint on scan-line
        insert ← segments with left endpoint on scan-line
        remove segments of delete from AEL
        add segments of insert to AEL
        for each segment s of insert {
            add (pred(s), s) to E(G)
            add (s, succ(s)) to E(G)
        }
    }
    return G
}
```

Figure 3.8: Algorithm for building graph G

its associated segment placed on a *delete* list. At each scan-line, the segments on the delete list are removed from the AEL, the segments on the insert list are added to the AEL, and then precedes arcs are generated for each segment in the insert list. By first deleting, then inserting, and then building the arcs, we ensure that: (a) no arcs are built between segments for which a vertical line can intersect only at the endpoints, and (b) the case of multiple insertions at the same scan-line position is handled correctly.

The active edge list may be implemented as a balanced binary search tree sorted by y-position. Since the y-position of the intersection between each segment and the scan-line changes as it moves along the x-axis, a "comparison function" can be used to insert into the tree. This function would compare segments by comparing their y-positions at the current scan line x-position. Since there are no intersecting segments in an extended rubber-band sketch, segments will never have to be reordered once they are inserted in the tree.

3.2 Time complexity

The total time required to run BUILDARCS on n segments is $O(n \log n)$. Sorting the endpoints takes $O(n \log n)$ time. For each of the n segments, the following operations are performed: inserting into the active segment list ($O(\log n)$), deleting from the list ($O(\log n)$), and building up to two arcs ($O(1)$). So the total time is $O(n \log n)$.

Once G is constructed, a topological sort can be used to produce a total order for the segments. The running time of the topological sort on a directed acyclic graph $G(V, E)$ is $O(|V| + |E|)$. Since the total number of edges in G is bounded by $2n$, the running time of the topological sort is $O(n)$. So, the total time to produce a legal transformation order is $O(n \log n)$.

3.3 Implementation Details

Until now, we have assumed that the input to the segment ordering problem was a set of segments that represent the centerlines of rubber-band segments in an extended rubber-band sketch. This is a slight simplification. Since width and spacing information is considered at later stages of the geometric wiring process (TRANSFORMLAYER and STRAIGHTENLAYER), it cannot be ignored here. For example, consider the two centerline segments in Figure 3.9 (a). Since they do not overlap at all in the x-dimension, there would be no ordering dependency between these two segments. However, if we consider the widths of the wires (Figure 3.9 (b)) it is clear that segment a must be pushed down before segment b.

The solution to this problem will be to use a modified version of the extended rubber-band sketch for the vertices of the graph G. Two types of transformations will be used to create the modified sketch:

1. reposition *one-sided* endpoints, and
2. create extra *overlap segments* in the area of *two-sided* endpoints.

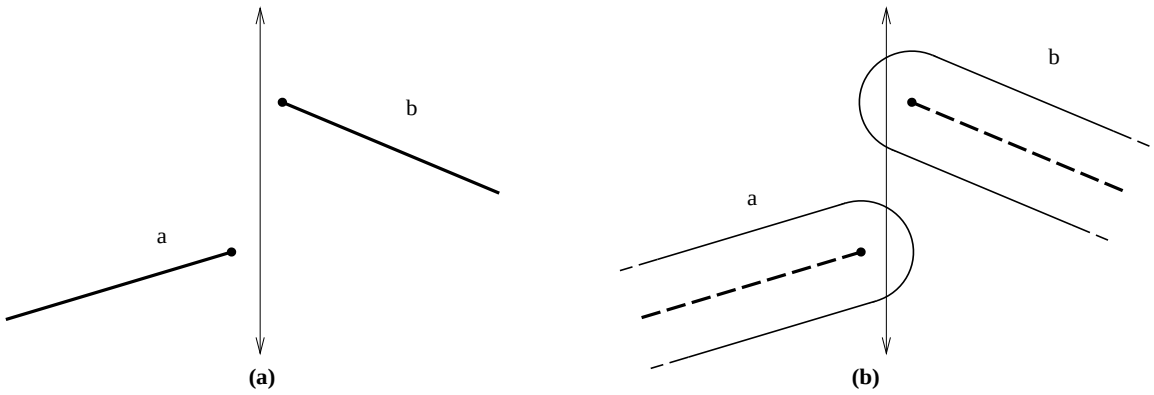


Figure 3.9: Effect of Width and Spacing on Ordering

If all of the segments incident on an endpoint are to one side of the vertical line through the endpoint, it will be called "one-sided". Endpoints with segments on both sides are "two-sided".

One-sided endpoints will be repositioned to account for the actual width and spacing requirements. Specifically, the endpoint will be slid either left or right by a distance of $r = \frac{width+spacing}{2}$. The direction of the slide will depend upon which side of the endpoint the segments are on—left-sided endpoints will be slid to the right and right-sided endpoints will be slid left. See Figure 3.10. Note that there is no danger of crossing segments as a result of this sliding operation because the spoke phase guarantees the necessary spacing.

Since a two-sided endpoint cannot be slid in both directions, the solution in this case will be to create a dummy segment that represents the overlap area common to all the segments incident on the endpoint. Any arc to or from this overlap segment will represent an ordering dependency with all of the common segments. For example, in Figure 3.11, since segment c is above the overlap area of segments a and b, an arc will be built from both a and b to c. Note that some care must be taken with constructing the overlap segments because they may overlap with one another. When two overlap segments overlap, a new segment will be created for this region. It will represent the union of segments from the original two segments. See Figure 3.12.

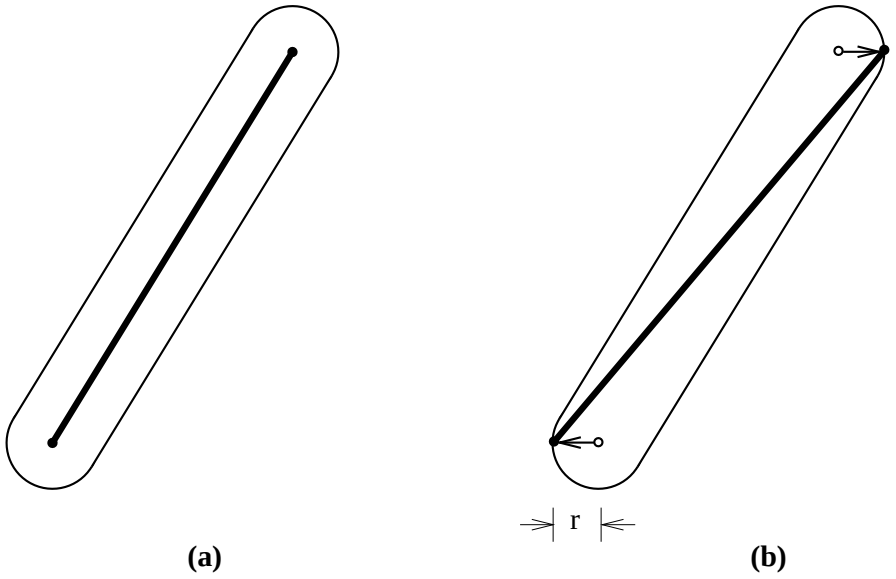


Figure 3.10: Sliding one-sided endpoints

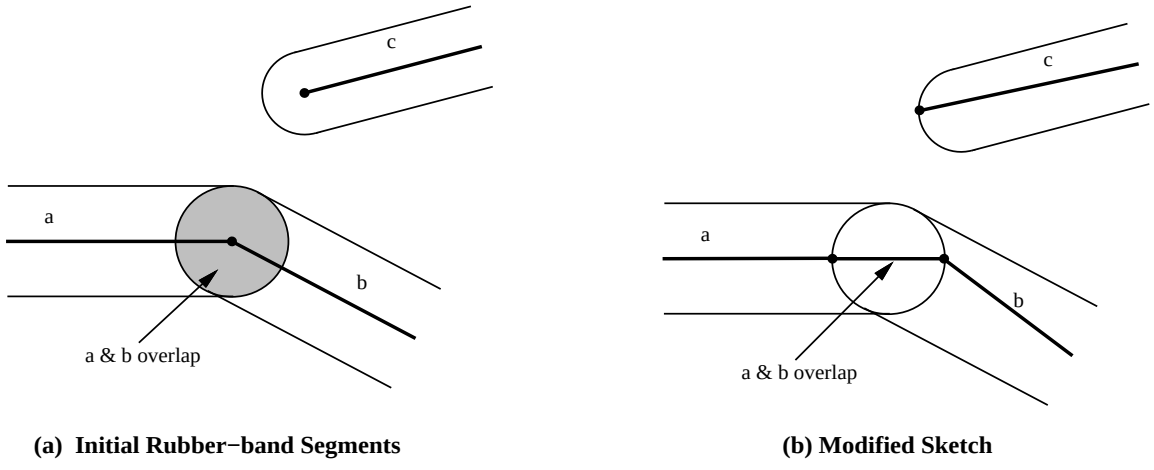
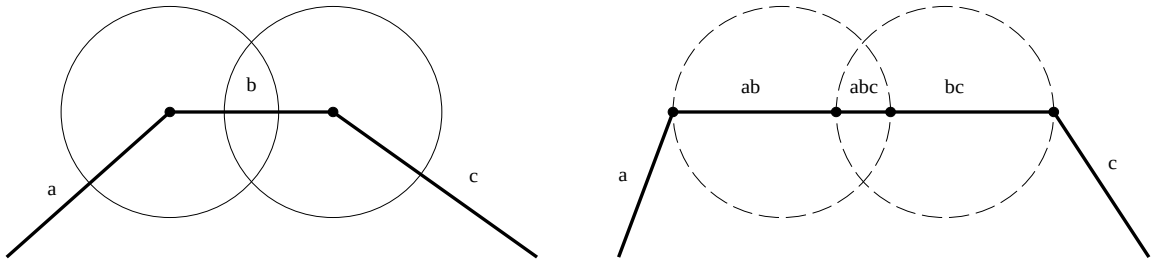


Figure 3.11: Overlap segment for two-sided endpoint



(a) Initial Rubber−band Segments **(b) Modified Sketch**

Figure 3.12: Overlapping overlap segments

4. Producing an Initial Transformation

This section describes the process used to produce a valid initial geometric wiring for a multi-layer topological routing. The input consists of a wiring type and an extended rubber-band sketch. The output will be a legal geometric routing, conforming to the wiring type, that is topologically equivalent to the input sketch and has minimum wire length. The geometric transformation outlined in this section is described as "initial" because it may contain extra unnecessary jogs. Some of these extra jogs may be removed in the subsequent straightening phase.

The basic strategy will be to transform each layer of the input sketch independently. The pseudo-code for the transformation of a single layer is given in Figure 4.1. TransformLayer makes use of a "previous boundary" to maintain a limited history about previously transformed segments. It initially consists of a single section of infinite length positioned at $y = -\infty$. The body of the *for*-loop transforms each of the rubber-band segments in order. The first step is to produce a minimum length geometric transformation, t_i, of the segment s_i that maintains the proper spacing from the sections of the previous boundary. This step is performed in TransformSegment. The transformation t_i is then added to the current transformation T. The last step of the loop is a call to UpdateBoundary which incorporates the transformation t_i into the previous boundary.

4.1 Transforming a Segment

The main step in TransformLayer is to transform a single rubber-band segment to geometric wiring. A segment is transformed by pushing it down within the lower triangle of the proper routing parallelogram (Figure 2.5). The shape of the routing parallelogram and the possible slopes for transformed segments depend on the geometry and the slope of the rubber-band segment. There are three such cases: $\mathcal{R}$, $\mathcal{O}1$, and $\mathcal{O}2$ as outlined in Chapter 2. These cases are summarized in Table 4.1. Each case uses two types of line segments to complete the routing.

```
TransformLayer(S : Ordered List of Segments, w : Wiring type) {
    B : List of previous boundary sections
    T : Ordered list of transformed segments

    B ← initial boundary segment at y = −∞
    T ← ∅
    for each segment s_i ∈ S in order {
        t_i ← TransformSegment(s_i,w,B)
        Append(T, t_i)
        UpdateBoundary(t_i,B)
    }
    return T
}
```

Figure 4.1: Pseudocode for Transforming a Layer

Case	*Condition*	*Geometry Used*
$\mathcal{R}$: Rectilinear	any slope	horizontal, vertical
$\mathcal{O}1$: Steep Octilinear	$\|slope\| \geq 1$	diagonal, vertical
$\mathcal{O}2$: Shallow Octilinear	$\|slope\| < 1$	horizontal, diagonal

Table 4.1: Summary of Transformation Cases

The easiest case is transforming a rubber-band segment to rectilinear wiring ($\mathcal{R}$). The pseudo-code for this transformation is presented in Figure 4.2. The basic approach is to start at the lower endpoint and extend a horizontal wire section until the x-position of the upper endpoint is reached or an obstruction in the previous boundary is encountered. Vertical jogs are introduced to clear any obstructions and to reach the proper y-position at the end of the transformation. Figure 4.3 shows an example of this process. The algorithm works by iterating through the sections of the previous boundary below the segment being transformed (s). The position of the last jog point is maintained in p. At any step, if a horizontal line from p ($\mathcal{H}(p)$) will not clear the current section, b, of the boundary, a horizontal wire extending from p until just before b is added to the transformation and a vertical jog is introduced. When all of the previous boundary sections have been examined,

```
TransformSegmentR(s : Rubber-band segment, B : Previous boundary) {
    P   : Point list
    p, j : Point
    b   : Boundary section
    d   : Spacing

    d ← width(s)/2 + interwire
    p ← s_l, P ← (p)
    b ← segment of B below s_l
    while b_l < s_rx + d {                  /* while b is under s */
        j ← (b_lx − d, b_y + d)             /* determine jog point j */
        if j_y > p_y {                      /* if H(p) won't clear b */
            Append( P, (j_x, p_y) )         /* extend horizontally and jog up */
            Append( P, (j_x, j_y) )
            p ← j
        }
        b ← next(b)
    }
    if p_x < s_rx {                         /* extend horizontally to s_r */
        p_x ← s_rx
        Append( P, p )
    }
    if p_y < s_ry {                         /* jog vertically to s_r */
        Append( P, s_r )
    }
}
```

Figure 4.2: Pseudocode for TransformSegmentR

final horizontal and vertical wire segments are added to the transformation as needed.

The steep-slope octilinear case, $\mathcal{O}1$, is very similar to the rectilinear case. The pseudocode for this case is presented in Figure 4.4. The main difference between these two cases is that diagonal wire segments are used in TransformSegmentO1 instead of horizontal ones. An example of this transformation is shown in Figure 4.5. At each step, the diagonal line $\mathcal{D}(p)$ extended from the last jog point p is compared to the current boundary section b. If it will not clear b, a diagonal wire segment extending as far as possible is added to the transformation and a vertical jog is introduced. As in the previous case, the final step is to add final diagonal and vertical wire segments as required to complete the transformation.

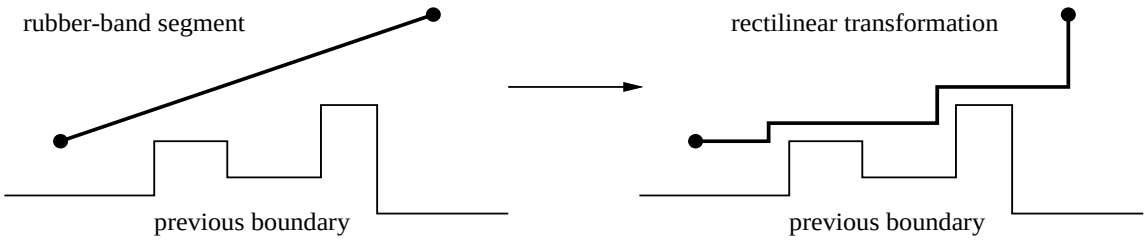


Figure 4.3: Rectilinear wire transformation for a segment

```
TransformSegmentO1(s : Rubber-band segment, B : Previous boundary) {
    P   : Point list
    p, j : Point
    b   : Boundary section
    d   : Spacing

    d ← width(s)/2 + interwire
    p ← s_l, P ← (p)
    b ← segment of B below s_l
    while b_l < s_rx + d {              /* while b is under s */
        Δ ← b_l − p_x
        if p_y + Δ < b_ly + d√2 {       /* if D(p) cannot clear b */
            p ← (p_x + Δ − d, p_y + Δ − d)
            Append( P, p )              /* route diagonally as far as possible */
            p_y ← b_ly + d√2
            Append( P, p )              /* jog vertically to avoid obstruction */
        }
        b ← next(b)
    }
    if p_x < s_rx {                     /* if a final diagonal is needed */
        Δ ← s_rx − p_x
        p ← (p_x + Δ, p_y + Δ)
        Append( P, p )
    }
    if p_y < s_ry {                     /* if a final vertical is needed */
        Append( P, s_r )
    }
}
```

Figure 4.4: Pseudocode for TransformSegmentO1

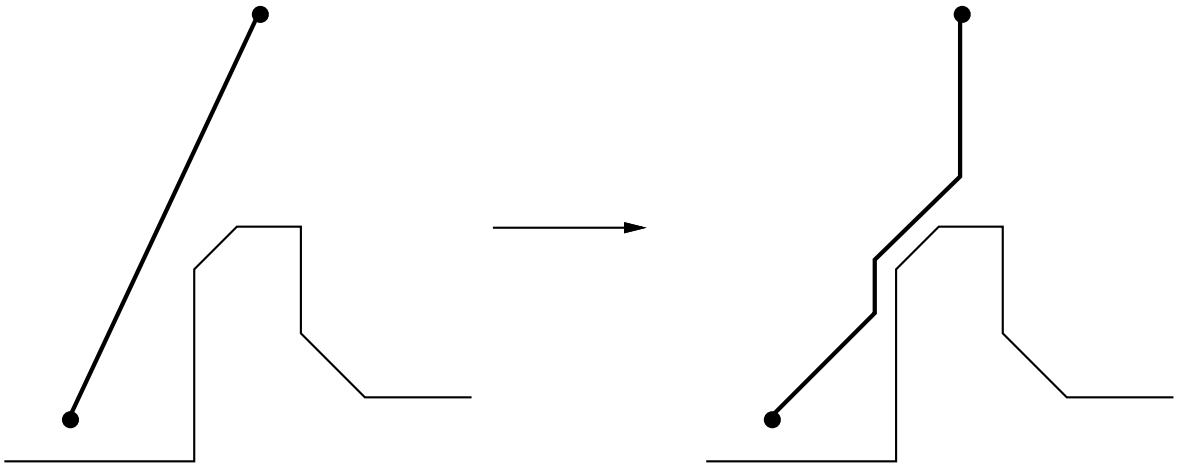


Figure 4.5: Transforming a steep segment to steep octilinear

The final case for transforming a shallow ($|slope| < 1$) segments to octilinear wiring is more complex than the previous two cases. Since this case does not use vertical wire segments, it is not possible to jog upwards "at the last minute" to clear obstructions presented by the previous boundary. To remedy this, some limited backtracking may be required. Figure 4.6 presents an example where a tall section of the previous boundary requires that an earlier decision regarding the position of a jog point be changed. The pseudo-code for the $\mathcal{O}2$ case is presented in Figure 4.7. As in the previous two cases, TransformSegmentO2 iterates through sections of the previous boundary in the direction from low endpoint to high endpoint. In contrast to the previous cases, TransformSegmentO2 maintains a *mode* to determine whether the segment extended from the last jog point p is meant to be horizontal or vertical. In both cases, if a segment routed from p will not clear the current boundary section b, wire segments are removed from the transformation P until a suitable jog point is determined. Once all of the previous boundary sections have been examined, the transformation is completed. It may require one final backtrack if it is not possible to reach the upper endpoint from the last jog position.

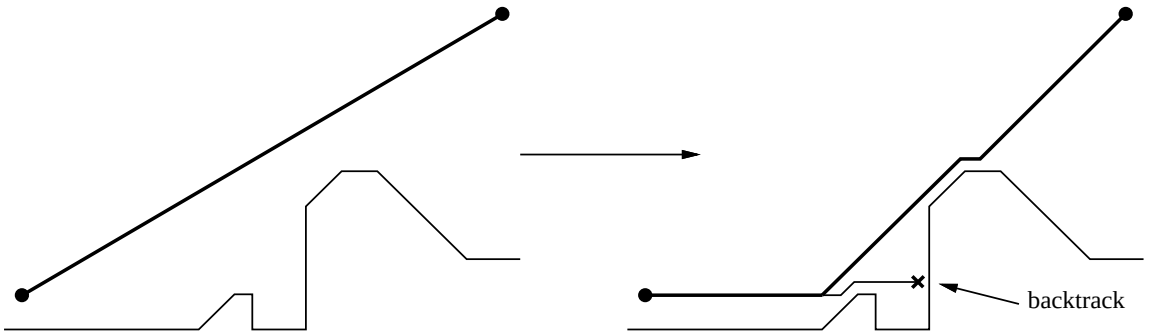


Figure 4.6: Transforming a shallow segment to octilinear

4.2 Updating the Previous Boundary

Once the rubber-band segment currently under consideration has been transformed to geometric wiring, the next step in the TransformLayer algorithm is to incorporate it into the previous boundary. This is done with the function UpdateBoundary. Pseudo-code for this function is presented in Figure 4.8. UpdateBoundary takes two parameters. The first, t, is the transformation of the current rubber-band segment s. The transformation represents the centerline of the geometric transformation of s. The second parameter, B, is the current previous boundary. The first step in UpdateBoundary is to determine t', the top contour of t. This can be accomplished by scanning each of the segments of t. Once this is done, the portion of B covered by t' is replaced with t'.

4.3 Time Complexity

4.3.1 Previous Boundary

Before discussing the running times of the individual portions of the TransformLayer algorithm, it is useful to make some observations about the previous boundary. The first observation is that the size of the boundary during the execution of TransformLayer is $O(n)$. This can be seen by noting that each of the n segments can add only a constant number of sections to the boundary. This fact also provides us with the worst-case output

```
TransformSegmentO2(s : Rubber-band segment, B : Previous boundary) {
    P       : Point list
    p, j    : Point
    b       : Boundary section
    mode    : {diagonal, horizontal}

    p ← s_l, P ← (p)
    b ← segment of B below s_l
    while b is below s {
        if mode = diagonal {
            if D(p) won't clear b {
                (p, P) = Backtrack(p, P, b)
            } else if D(p) clear b with excess space or b does not slope up {
                j ← point of D(p) s.t. H(j) will just clear b
                Append(P, j)
                p ← j
                mode ← horizontal
            }
        }
        if mode = horizontal {
            if H(p) won't clear b {
                j ← point of H(p) s.t. D(j) will just clear b
                if j is before p {
                    (p, P) ← Backtrack(p, P, b)
                } else if j is after p {
                    Append(P, j)
                    p ← j
                }
                if b slopes up {
                    mode ← diagonal
                } else {
                    j ← point of D(p) s.t. H(j) will just clear b
                    Append(P, j)
                    p ← j
                }
            }
        }
        b ← next(b)
    }
    if s_r above D(p) {
        (p, P) = Backtrack(p, P, s_r)
    } else if s_r below D(p) {
        j ← point of H(p) s.t. D(j) contains s_r
        Append(P, j)
    }
    Append(P, s_r)
}
```

Figure 4.7: Pseudocode for TransformSegmentO2

```
UpdateBoundary(t : Transformation, B : Previous Boundary) {
    t' ← top contour of t
    B' ← B with portion covered by t' replaced by t'
    return B'
}
```

Figure 4.8: Pseudocode for UpdateBoundary

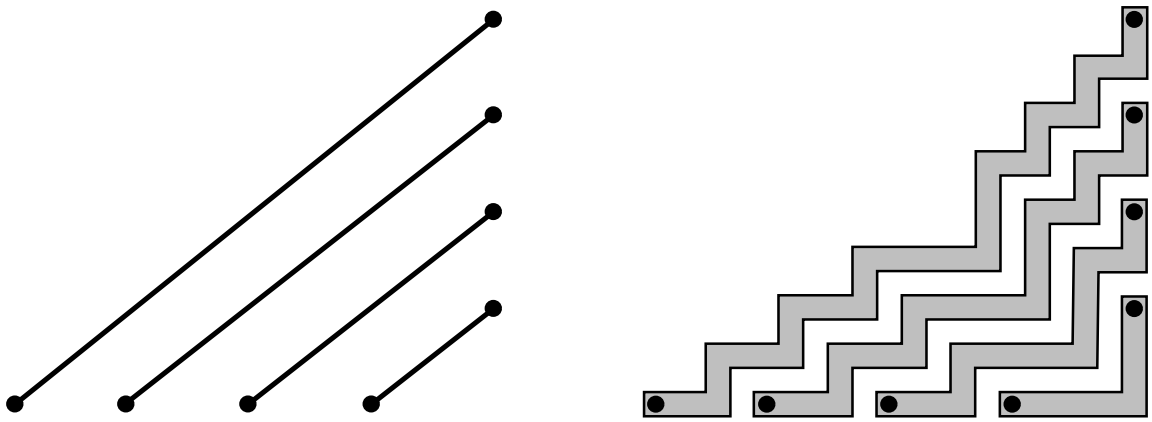


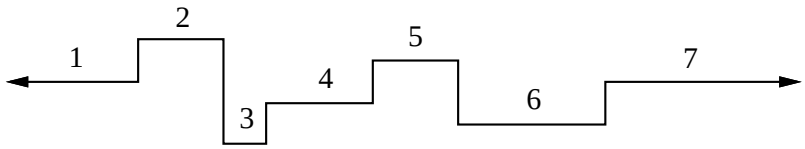
Figure 4.9: Worst-case relationship between input and output size

complexity. Since the transformation of a given segment can be no more complex than the boundary below it, the worst-case complexity is $\Theta(n^2)$. See Figure 4.9.

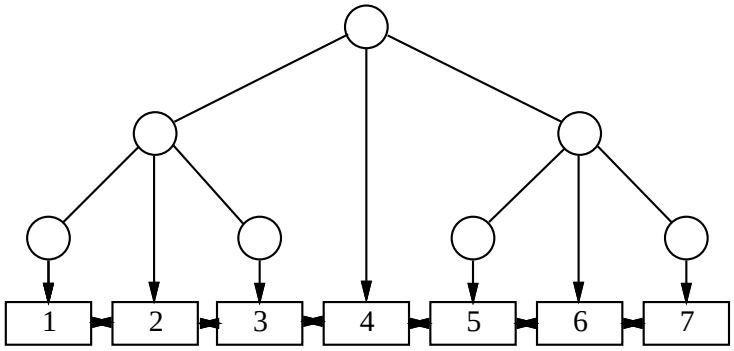
The previous boundary must be maintained in such a way as to efficiently support the following operations:

1. locate the first section of the boundary that lies below the segment being processed,
2. traverse the previous boundary in the proper direction until the segment has been completely transformed, and
3. replace the portion of the boundary below the segment with the updated boundary.

The solution used in my implementation is a hybrid balanced binary search tree. In this tree, the leaf nodes are linked together to form a linked list (see Figure 4.10). This allows the first query to be performed in $O(\log n)$ time and the second and third operations in $O(k)$ time where k is the number of segments below the segment being transformed.



(a) Previous Boundary



(b) Hybrid BST Representation

Figure 4.10: Hybrid List-Binary Search Tree Representation of Previous Boundary

4.3.2 Time Analysis of TRANSFORMSEGMENT

The following lemma presents the time complexity of transforming a single segment to geometric wiring.

Lemma 4: *A rubber-band segment s covering k out of n sections of a previous boundary can be transformed to geometric wiring in $O(\log n + k)$ time.*

Proof

Case $\mathcal{R}$ (TRANSFORMSEGMENTR):

The first step in TRANSFORMSEGMENTR is to locate the section of the previous boundary under the lower endpoint of segment s. Since the previous boundary contains n sections and is stored in a balanced binary search structure, this obviously requires $O(\log n)$ time. Once the initial section is located, the segment is transformed to rectilinear wiring by visiting all of the sections it covers. In this step, each of the k sections is visited exactly once and

adds at most two wire sections to the transformation of s. Since each of these operations takes constant time, the total time complexity of TransformSegmentR is $O(\log n + k)$.

Case $\mathcal{O}_1$ (TransformSegmentO1):

The proof for this case is essentially the same as for *Case* $\mathcal{R}$.

Case $\mathcal{O}_2$ (TransformSegmentO2):

As in the previous two algorithms, the first step of TransformSegmentO2 requires $O(\log n)$ time to locate the initial section of the previous boundary. Since TransformSegmentO2 employs backtracking during the transformation phase (*while*-loop), the segments visited during backtracking must also be counted as operations in the complexity analysis.

Since each of the k sections may cause a backtrack to occur, there are potentially $O(k)$ backtrack operations. When a section of s is visited in a backtrack, it is either "passed over" and removed from the transformation or it is the last in the backtrack sequence. So, a segment may be passed over and deleted only once by a backtrack during the transformation of s. So, the following costs may be assigned to each of the covered k sections from the previous boundary:

- $O(1)$ for visiting the section in the previous boundary
- $O(1)$ for the new section added to the transformation of s
- $O(1)$ for passing over and removing the segment during backtracking

An amortized analysis yields a total time complexity of $O(k)$ for the transformation phase and $O(\log n + k)$ for the entire TransformSegmentO2 algorithm. □

4.3.3 Time Analysis of UpdateBoundary

Suppose that the top contour t' consists of j sections and spans k sections of the previous boundary B. The naive approach of first removing the sections spanned by t' from B and then inserting t' would require $O(j + k)$ inserts and deletes. If the boundary is stored in a balanced structure, each of these operations will require $O(\log n)$ time. If UpdateBoundary rewrote key values where possible instead of always performing inserts

and deletes, updating the previous boundary could by done in $O(\Delta \log n + k)$ time where Δ is the amount the previous boundary changes size ($\Delta = |j - k|$).

4.3.4 Time Analysis of TransformLayer

In this section we will examine the time complexity of transforming an extended rubber-band sketch to geometric wiring using the TransformLayer algorithm described above. The time complexity will be expressed in terms of both the input and output size of the problem, specifically the number of rubber-band segments, n, in the extended rubber-band sketch and the number of geometric wire segments, m, in the resulting transformation. As we have seen, in the worst case, m is $\Theta(n^2)$.

The *for*-loop that transforms each segment is executed n times. From Lemma 4 we know that each TransformSegment takes $O(\log n + k)$ time where k is the number of previous boundary sections covered by the segment s_i. Since each section in the previous boundary corresponds to a wire section in the final output, the total time contribution of TransformSegment is $O(n \log n + m)$.

As shown in Section 4.3.3, the total time complexity contributed by UpdateBoundary when updating the boundary after transforming segment s_i is $O(\Delta \log n + k)$ where Δ is the change in the boundary's size due to the update, and k is the number of sections spanned by the newly transformed segment. So, in order to determine the total time complexity of the UpdateBoundary operation, we have to know how much the boundary's size changes during the entire process. At each step, UpdateBoundary can only increase the size of the previous boundary be a small constant. So, by charging for a section's removal at the time of insertion, we get a total contribution of $O(n \log n + m)$ for UpdateBoundary.

This gives a total time complexity of $O(n \log n + m)$ for the TransformLayer operation.

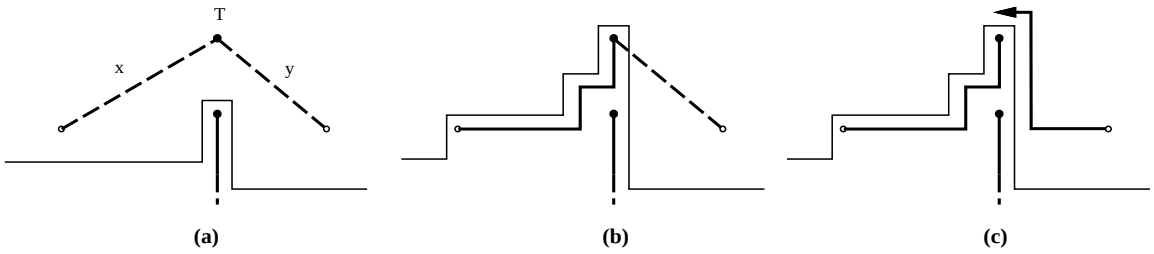


Figure 4.11: Problem with simple previous boundary

4.4 Implementation Details

This section addresses some details regarding the implementation of the previous boundary. Until now, it has been tacitly assumed that the previous boundary is represented as a series of zero-width horizontal and diagonal line segments. As we shall see, such a previous boundary is not sufficient when dealing with segments that meet at a point. Two or more segments may be connected at a point where a wire bends around an object or with terminals that are incident on more than one segment. Consider the case of terminal T shown in Figure 4.11(a). This terminal is incident on two segments, x and y. In this case, there is no precedes relationship between the two segments so they may be transformed in either order. Suppose x is transformed first. This transformation and the updated previous boundary are shown in Figure 4.11(b). If segment y is transformed normally, it will not be able to connect with T while maintaining proper spacing from the boundary. Since x and y belong to the same net, they should not be required to remain separated. It is not feasible for y to ignore the obstruction in the neighborhood of T because it has no knowledge of what other obstacles may be hiding below the previous boundary (e.g. the vertical segment shown in the figure). One solution to this problem is to incorporate some additional information into the previous boundary so that wire sections belonging to the same net may be transformed independently of one another.

At any point, each section of the previous boundary is the result of some past rubberband segment transformation. Suppose that a previous boundary section for segment s contains the following fields:

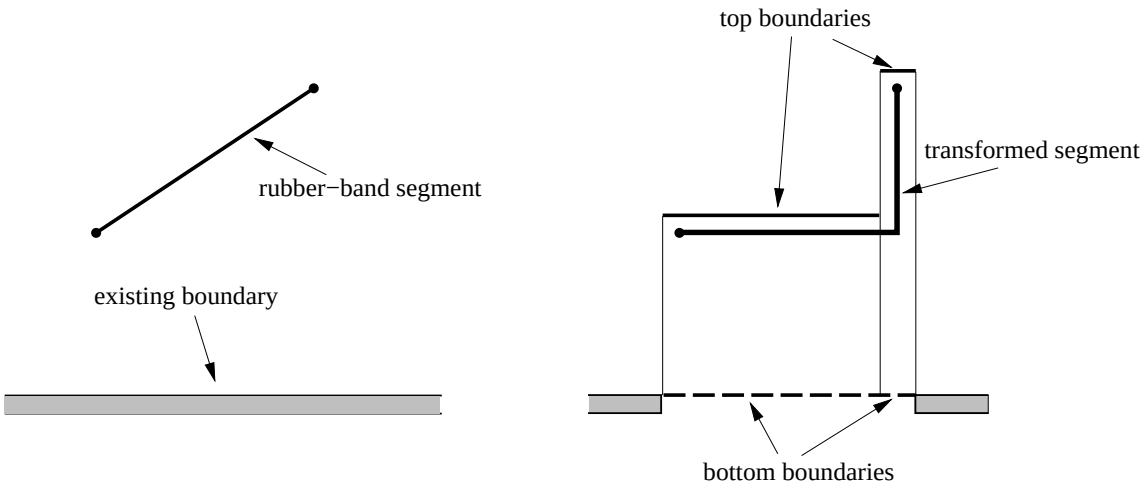


Figure 4.12: The double-width previous boundary.

- the net associated with segment s
- the x-interval that the section covers
- the boundary *before* s was transformed (lower boundary)
- the boundary *after* s is transformed (upper boundary)

The bottom boundary reflects the state of the previous boundary *prior* to the transformation of segment s—a *previous* previous boundary in some sense. Figure 4.12 shows an example of this previous boundary. With this scheme, when a segment is transformed, it can use the boundary that is appropriate. If the segment belongs to the same net as a previous boundary section, it would use the *lower* boundary. If the segment is part of a different net, if would use the top boundary. Figure 4.13 demonstrates the use of this previous boundary scheme in the transformation of the segments presented in Figure 4.11.

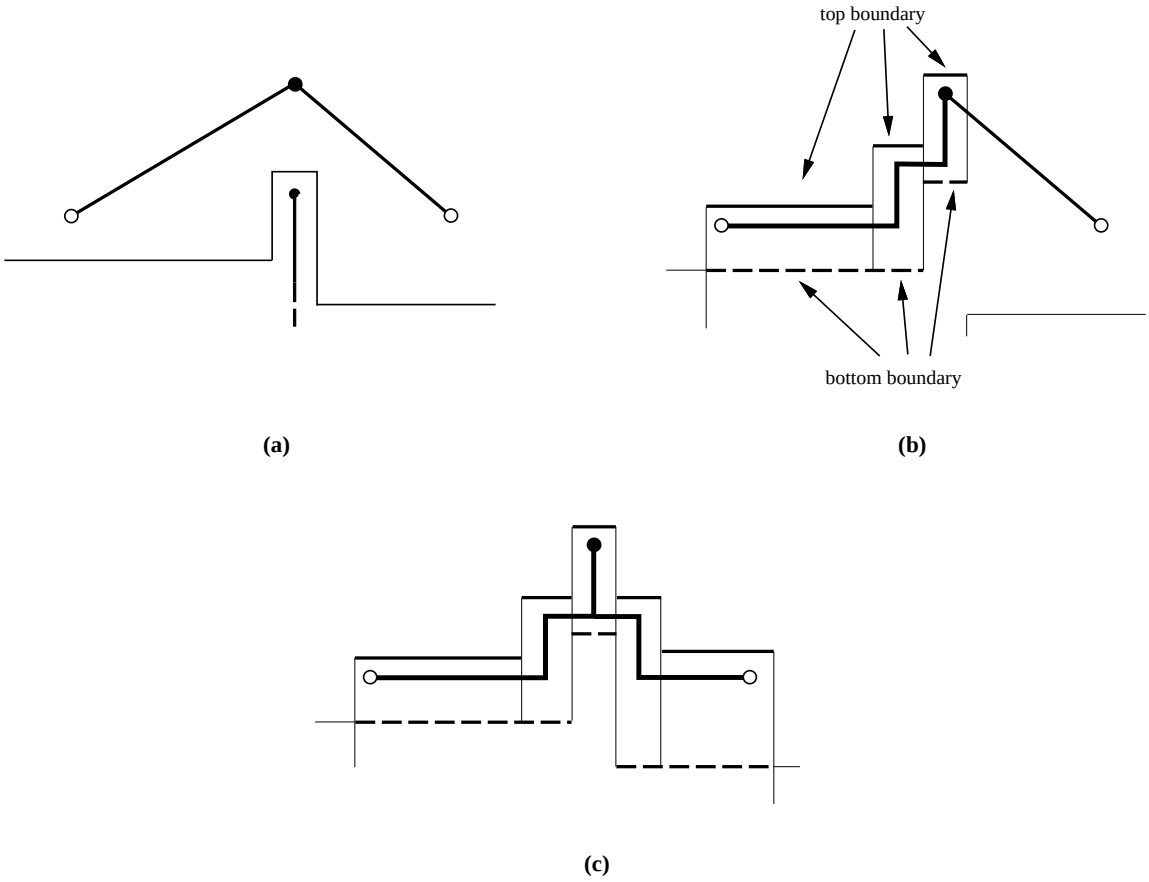


Figure 4.13: Example of the double-width previous boundary.

Figure (a) shows two untransformed segments incident on the same terminal. Figure (b) shows how the double-width previous boundary is updated after the segment on the left is transformed. Figure (c) shows the result after transforming the right-hand segment.

5. Straightening the Initial Geometric Transformation

The initial geometric transformation produced by TransformLayer may contain some unnecessary jogs (see Figure 5.1 (b)). The reason for this is that it takes the conservative approach of pushing each segment down as far as possible. Although the initial transformation has minimum wire length and satisfies the necessary width and spacing rules, it is also important to try to reduce the number of jogs. Some reasons for reducing the number of jogs are:

- **Better yield:** Bends are more likely than straight wires to be produce defects when the design is fabricated.
- **Better noise margin:** Each jog represents an electrical discontinuity in the wire. These discontinuities produce reflections as signals propagate along the traces. Reducing the number of jogs results in less reflection noise and a better noise margin.
- **Simpler designs:** Designs with a lot of jogs are confusing and harder to understand visually. This confusion may result in longer design times and increased chance of error.
- **Less data:** Each jog in a wire has to be stored in the database. More jogs result in higher memory and disk requirements as well as slower data access times.

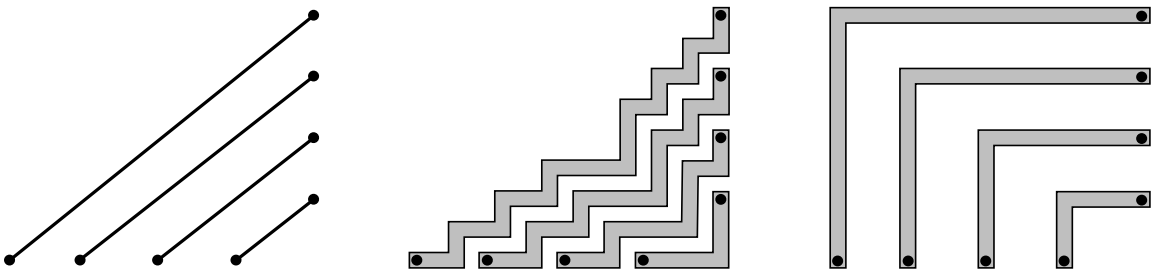


Figure 5.1: Jog Removal

The initial geometric transformation (b) may contain unnecessary jogs. A second plane sweep in the opposite direction to the first can be used to clean up some of these jogs (c).

```
StraightenLayer(T : Ordered list of transformed segments, w : Wiring type) {
    B : List of previous boundary sections
    T' : List of straightened segments

    B ← initial boundary segment
    T' ← ∅
    for each transformed segment t_i ∈ T in (reverse) order {
        t'_i ← StraightenSegment(t_i,w,B)
        Append(T', t'_i)
        UpdateBoundary(t'_i,B)
    }
    return T'
}
```

Figure 5.2: Pseudocode for Straightening a Layer

A second plane sweep can be used to reduce the total number of jogs in the layout. This plane sweep is very similar to the sweep used to produce the initial transformation. The main differences are that it sweeps in the opposite direction (top to bottom) and instead of transforming segments, it tries to reduce the number of jogs by sliding sections together. The pseudo-code for this step is presented in Figure 5.2. As mentioned above, the plane sweep in StraightenLayer processes segments in the opposite direction as TransformLayer. At this stage, it is not necessary to recompute the ordering. The straighten ordering can be obtained by reversing the transform ordering.

5.1 Straightening a Segment

Straightening a segment is quite similar to transforming a segment. The basic strategy is to work from the upper endpoint towards the lower endpoint and examine each of the sections of the current transformation. If there is enough room to slide a section upwards and align it with the previous section, up to two jogs can be removed from the transformation. If however, there is not enough room to align the current section with the previous one there are at least two obvious choices: leave it where it is, or slide it up as far as possible. The former heuristic increases the chance that the *next* section in the current transformation

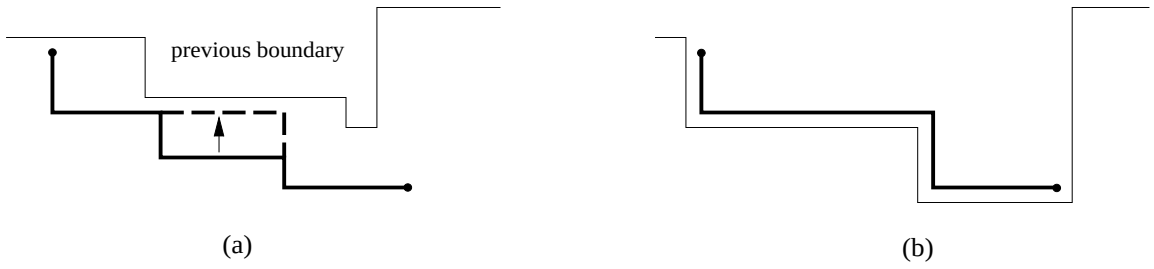


Figure 5.3: Straightening a rectilinear segment

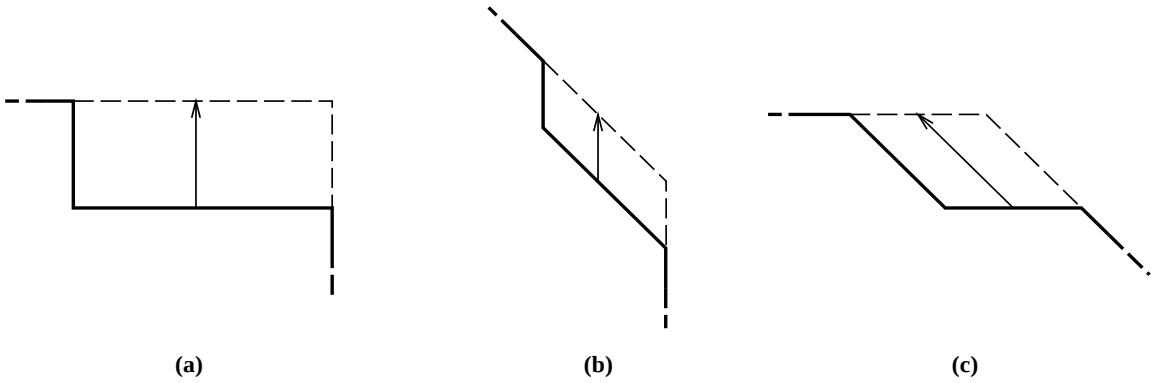


Figure 5.4: Sliding sections during straightening (3 cases)

will be able to merge with the current one. The latter gives more room for straightening future segments below the current one. An example of the straightening process is shown in Figure 5.3.

Each of the three different cases ($\mathcal{R}$, $\mathcal{O}1$, and $\mathcal{O}2$) experience a slightly different type of sliding during the straightening process. Examples of these cases are shown in Figure 5.4.

The pseudo-code for straightening rectilinear segments whose left endpoints are higher than their right endpoints (negative slopes) is presented in Figure 5.5. The basic strategy is to work from left to right examining the horizontal sections of the initial transformation and produce the straightened transformation, t', on the fly. At any point in the process, the point c is the left endpoint of the next horizontal section to be added to t'. At each step, the algorithm tries to merge a horizontal section h by sliding it upwards to the same level as the one extending from c. The validity of this merge is determined by comparing h

```
StraightenSegmentR(t : transformed segment, B : Previous boundary) {
    t'       : Straightened segment
    (c_x, c_y) : Current point
    b        : Boundary section
    h        : Horizontal section (x_1, x_2, y)

    (c_x, c_y) ← first point of s
    for each horizontal segment h = (x_1, x_2, y) in t from high to low {
        for each section b of B above h {
            if b prevents sliding h vertically to c_y {
                add horizontal section (c_x, x_1, c_y) to t'
                (c_x, c_y) ← (x_1, y)
            }
        }
    }
    if c_x < x_2 {
        add horizontal section (c_x, x_2, c_y) to t'
    }
    if c_y < y {
        add point (x_2, y) to t'
    }
}
```

Figure 5.5: Pseudo-code for StraightenSegmentR

against the previous boundary. When a section is found that cannot be merged, the section from c is added to t' and c is advanced to the left endpoint of section h.

The approach used to straighten steep octilinear transformations (case $\mathcal{O}_1$) is essentially the same as that for the rectilinear case. The main difference is that it slides and merges diagonal (slope ± 1) sections instead of horizontal ones.

As in the case of producing the initial transformation, the shallow-slope octilinear case ($\mathcal{O}_2$) presents some difficulties for straightening. The main difficulty is that a naive implementation may revisit portions of the previous boundary in a way that increases the time complexity of the algorithm.

Consider the case of straightening the initial transformation $(p_1, p_2, p_3, p_4, p_5)$ given in Figure 5.6. For this example, we will assume that the previous boundary does not conflict with any of the sections in the initial transformation (they all can be merged). If we

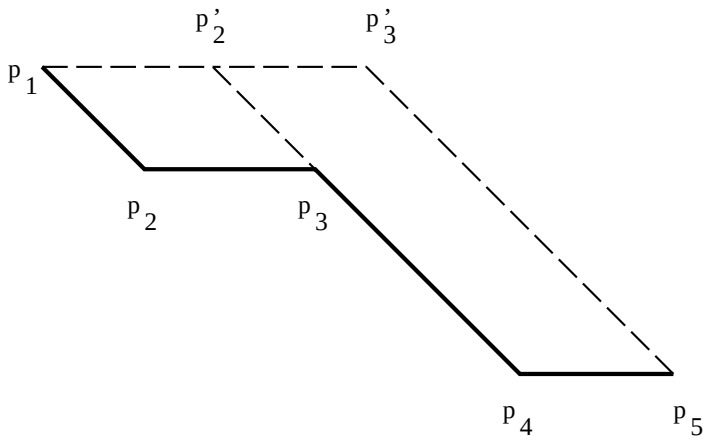


Figure 5.6: Straightening a shallow octilinear segment

work from top to bottom, sliding the first horizontal section $\overline{p_2p_3}$ upwards to $\overline{p_1p_2'}$ requires checking that the previous boundary is entirely above the contour (p_1, p_2', p_3). Straightening the next section $\overline{p_4p_5}$, will require checking the previous boundary against the contour (p_2', p_3', p_5). Since the two contours that were checked in this example overlap in the x dimension (in the range $(p_2'.x, p_3.x)$), and do not have the same y limit in the overlap area, a naive approach would require that the overlap region be visited twice. In general, a given region may be have to be visited $O(k)$ times for an initial transformation with k sections. This multiple visiting would increase the time complexity of the overall algorithm.

The difficulty with multiple visiting only occurs in cases where sections are merged. So, in order to find a solution to this problem, we will examine the criteria for sliding a single section and see how information can be maintained in a way that does not require revisiting while merging sections.

Consider the first portion of the transformation shown in Figure 5.6, (p_1, p_2, p_3). In the x-range $(p_1.x, p_2'.x)$, the previous boundary must be above the horizontal line $\overline{p_1p_2'}$. In the range $(p_2'.x, p_3.x)$, the boundary must be above the diagonal line $\overline{p_2'p_3}$. Determining if a point (x_0, y_0) is above or below the diagonal line (with slope -1) through another point (x_1, y_1) can be done by comparing $x_0 + y_0$ to $x_1 + y_1$.

So, to check whether $\overline{p_2p_3}$ can be slid to $\overline{p_1p_2'}$, we can scan the boundary in the x-range $(p_1.x, p_3.x)$ and find the boundary point (x, y) with minimum $x + y$ such that $y < p_1.y$. If this section is slid, the next task will be to perform a similar calculation for sliding $\overline{p_4p_5}$ to $\overline{p_2'p_3'}$. But since $p_1.y = p_2'.y$, the minimum $x + y$ value computed for the overlap region will be the same for both sections. So, there is no neeed to recompute it, we can merely continue examining the previous boundary from $p_3.x$ onwards.

Pseudo-code for case $\mathcal{O}_2$ straightening is presented in Figure 5.7. This algorithm explicitly visits only the "upper corner points" of the initial transformation (p_1, p_3, and p_5 in the example in Figure 5.6). The lower corners (c's) and the reflected lower corners (c''s) are implied from the upper corners. For example

$$p_2 = c(p_1, p_3) = (p_1.x + p_3.y - p_1.y, p_3.y)$$

and

$$p_2' = c'(p_1, p_3) = (p_3.x - p_3.y + p_1.y, p_1.y)$$

5.2 Running Time

The time complexity analysis for StraightenLayer is essentially the same as that for TransformLayer. From this we can see that StraightenLayer runs in $O(n \log n + m)$ time where n is the number of segments in the extended rubber-band sketch and m is the number of jogs in the initial transformation.

```
StraightenSegmentO2(t : transformed segment, B : Previous boundary) {
    t'      : Straightened segment
    P       : List of points
    p1, p2  : current jog
    b       : Obstruction point from B
    m       : Minimum x + y value

    P ← upper corners of t
    p1 ← DelHead(P)
    b ← point of B covering p1
    add p1 to t'
    m ← ∞
    while P is not empty {
        p2 ← DelHead(P)
        while b is before p2.x {/* scan boundary for min x + y */
            if b.y < p1.y and b.x + b.y < m {
                m ← b.x + b.y
            }
            b ← next(b)
        }
        if m ≥ p2.x + p2.y {    /* merge */
            p1 ← c'(p1, p3)
        } else {                /* don't merge */
            add p1 to t'
            add c(p1, p3) to t'
            p1 ← p2
            m ← ∞
        }
    }
    if first two points in t' are the same {
        remove the first point from t'
    }
    add p1 to t'
    if p1 ≠ p2 {
        add p2 to t'
    }
    return t'
}
```

Figure 5.7: Pseudo-code for StraightenSegmentO2

6. Results

The algorithms described in this thesis were implemented in C and integrated into the SURF layout system [KR78]. This section presents some sample results taken from both real and artificial examples. These results are summarized in Table 6.1. The results are divided into three sections: Euclidean, Rectilinear, and Octilinear. The column labeled "Euclidean" gives the total wire length of the input sketch before geometric wiring. The results of the transformation are shown in the columns labeled "Rectilinear" and "Octilinear". Each of these columns is divided into two subcolumns. The first gives the total wire length after transformation. The second reports figures on *intra-segment* jogs. [1] The first number is the final number of jogs. The second number (in parenthesis) is the jog count after the initial transformation.

The entry labeled `test5` is an artificial hand-prepared example that was devised to test the geometric wiring algorithms before the "spokes" module was completely debugged. It contains 28 simple multi-terminal nets. Even though it is fairly small, it contains enough interesting cases to be useful for debugging.

[1] Intra-segment jogs reflect only those jogs that occur *between* the two endpoints of some segment in the initial input—jogs at segment endpoints are ignored. The number of intra-segment jogs is the difference between the total number of jogs in the transformed layout and the number of jogs in the input sketch.

Example	*Euclidean*	*Rectilinear*		*Octilinear*	
	length	length	jogs	length	jogs
`test5`	10,763	14,120	81 (137)	11,349	61 (82)
`layer0`	39,820	51,635	391 (698)	41,841	322 (357)
`layer1`	29,187	35,864	243 (402)	30,734	207 (239)
`phone0`	236,458	297,341	687 (928)	249,255	645 (742)
`phone1`	224,804	279,077	605 (803)	236,613	591 (659)

Table 6.1: Sample results

The examples labeled `layer0` and `layer1` represent the two layers taken from a portion of a digital design. These examples contain only two-terminal nets. They were generated through the normal SURF design flow (global routing, local routing, spoke creation). It should be noted that both the rectilinear and octilinear results were generated from the same (rectilinear) extended rubber-band sketch.

The entries labeled `phone0` and `phone1` are two layers from an analog MCM design. This example contains multi-terminal nets.

Window dumps from the geometric wiring process for the `layer0` example can be seen in Figures 6.1–6.5. Figure 6.1 shows the input extended rubber-band sketch. The initial rectilinear transformation is presented in Figure 6.2 and Figure 6.3 shows the results after the jog removal phase. Figures 6.4 and 6.5 show the initial and straightened octilinear transformations.

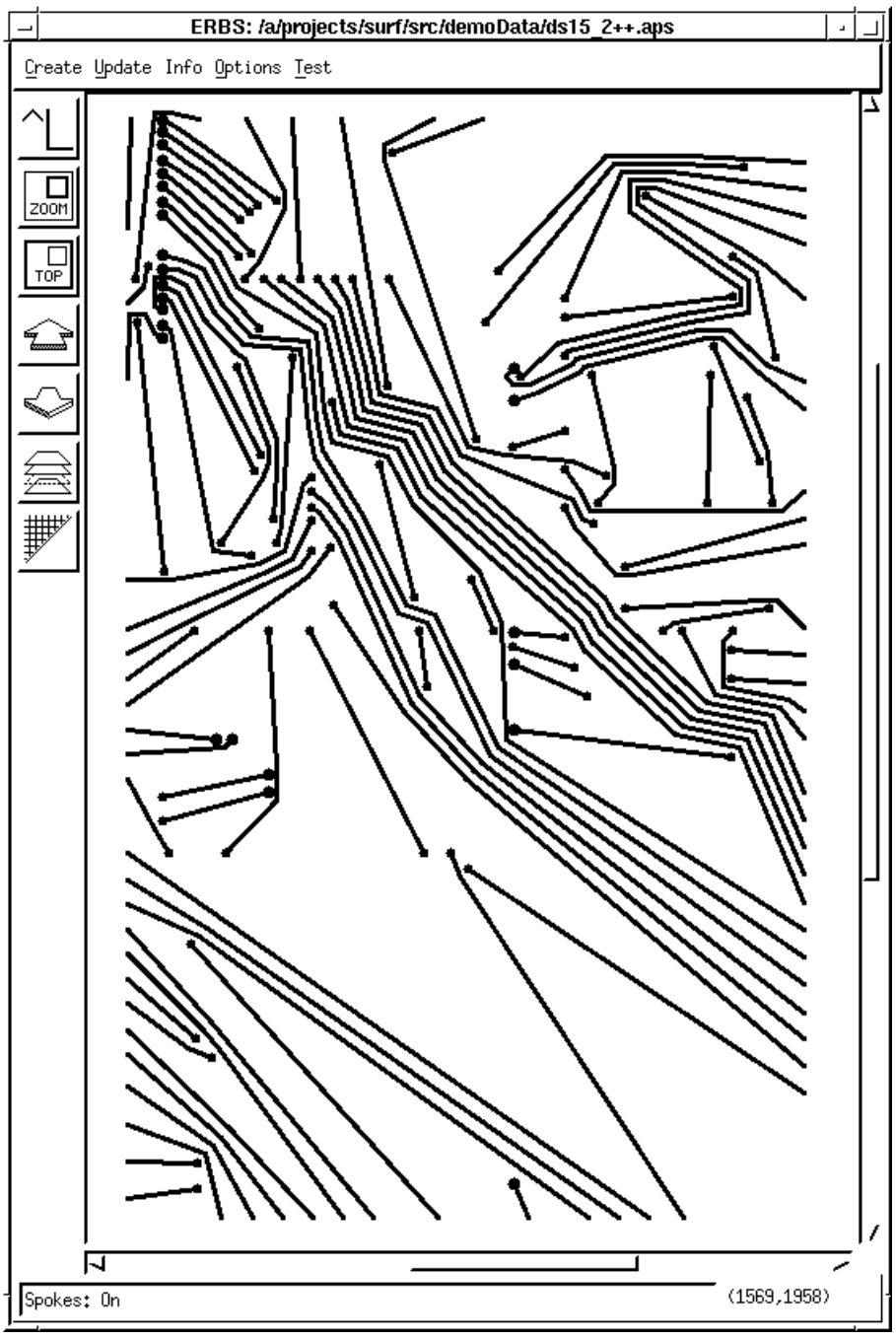


Figure 6.1: Initial input for geometric wiring

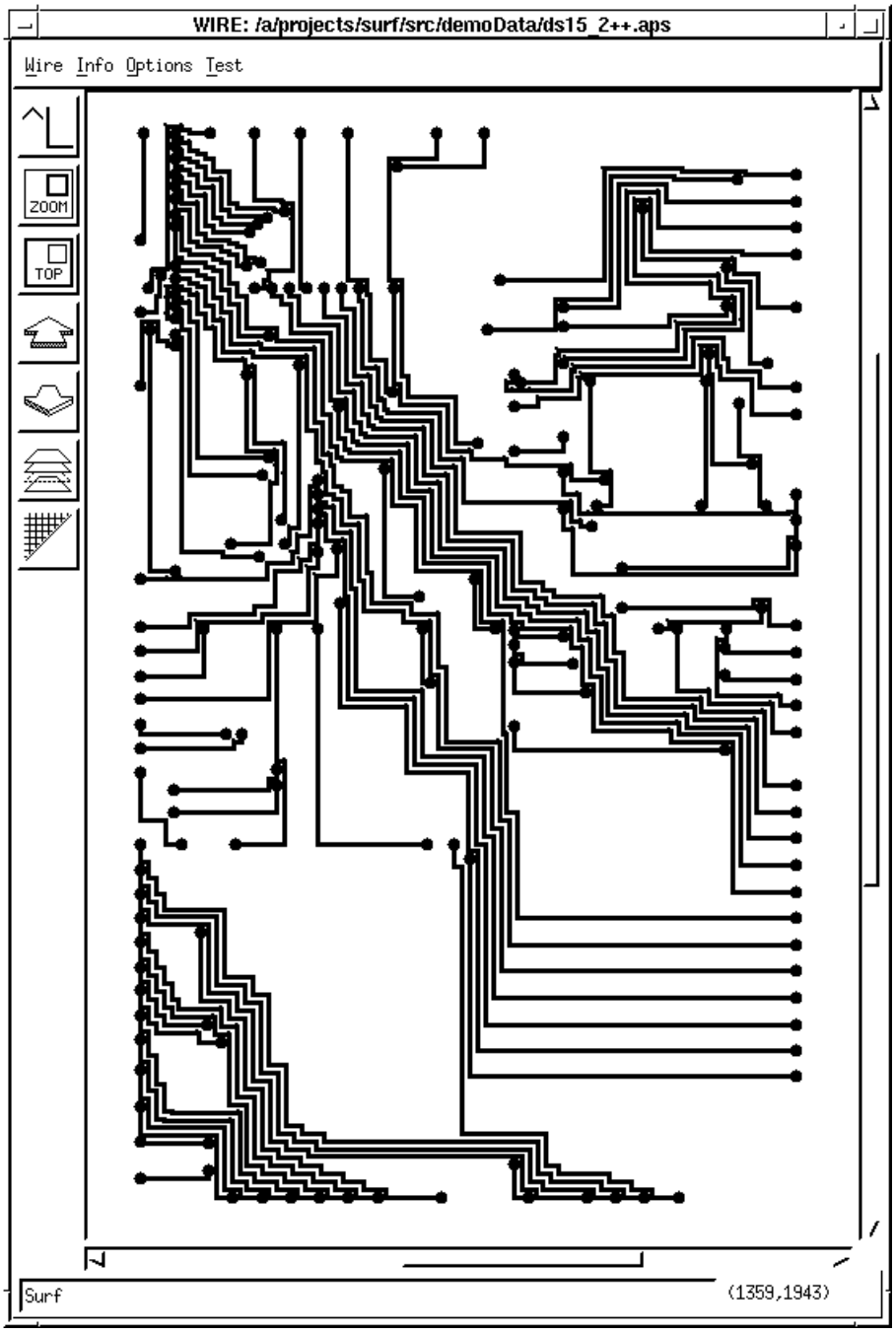


Figure 6.2: Initial rectilinear transformation with 698 jogs.

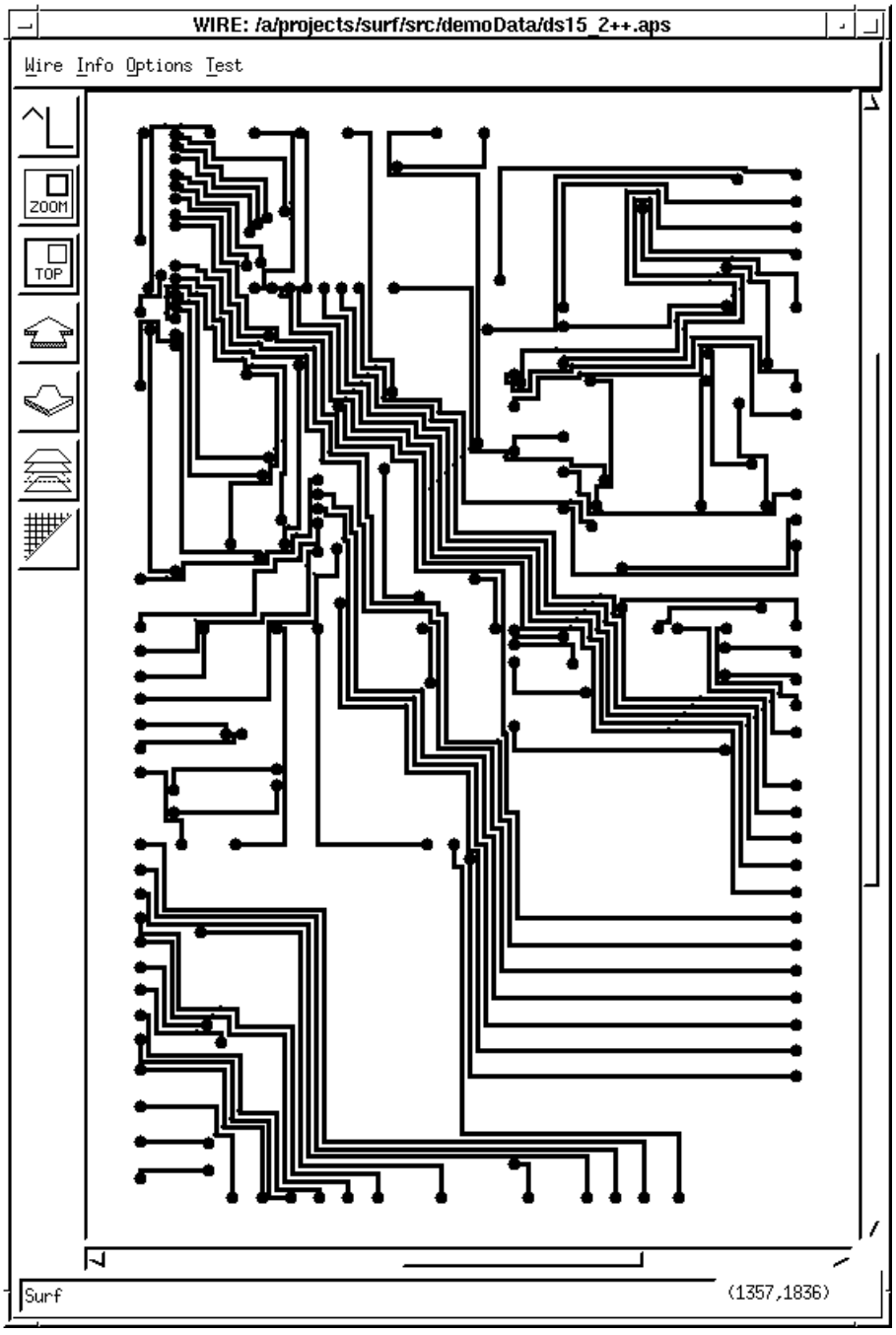


Figure 6.3: Straightened rectilinear transformation with 391 jogs.

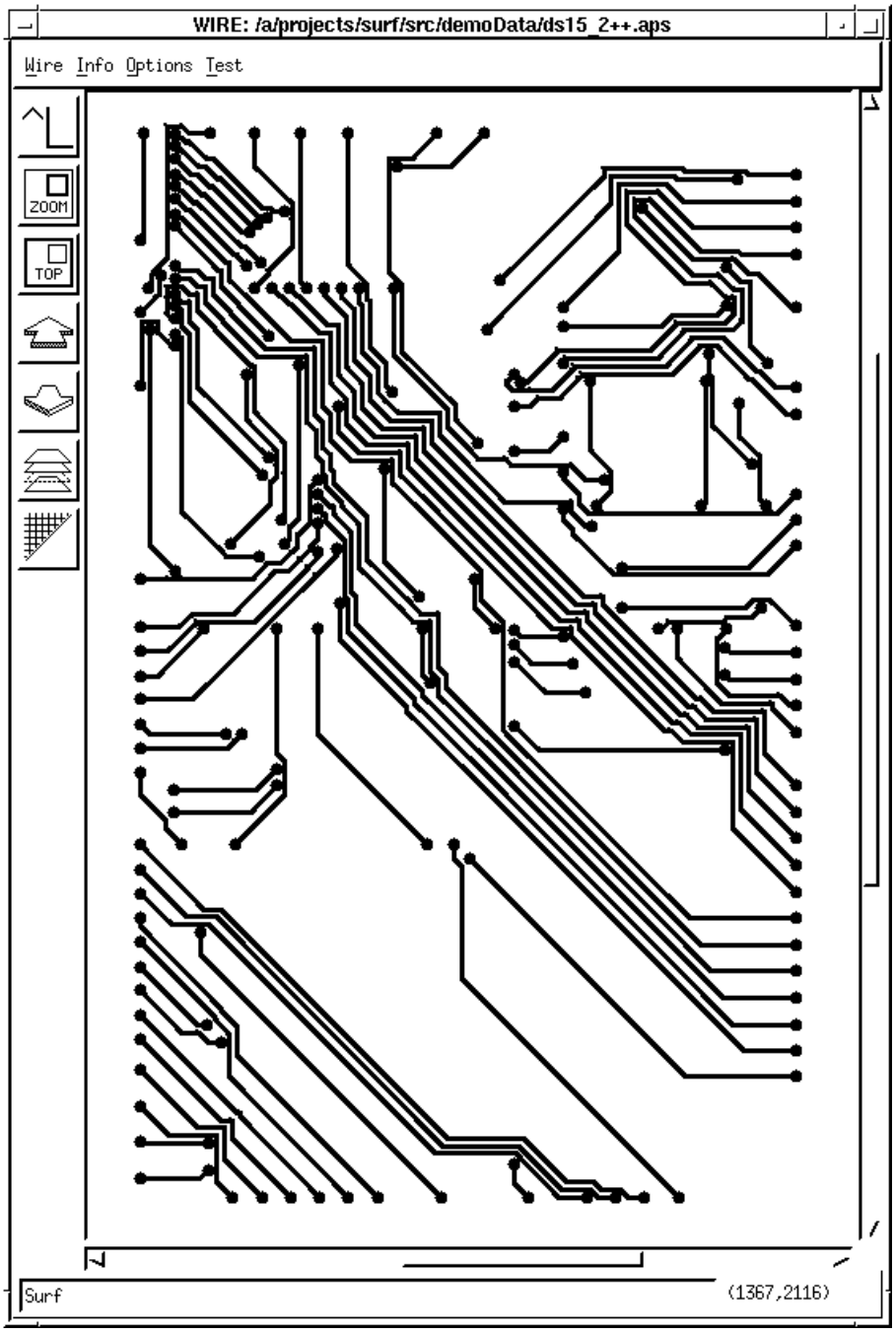


Figure 6.4: Initial octilinear transformation with 357 jogs.

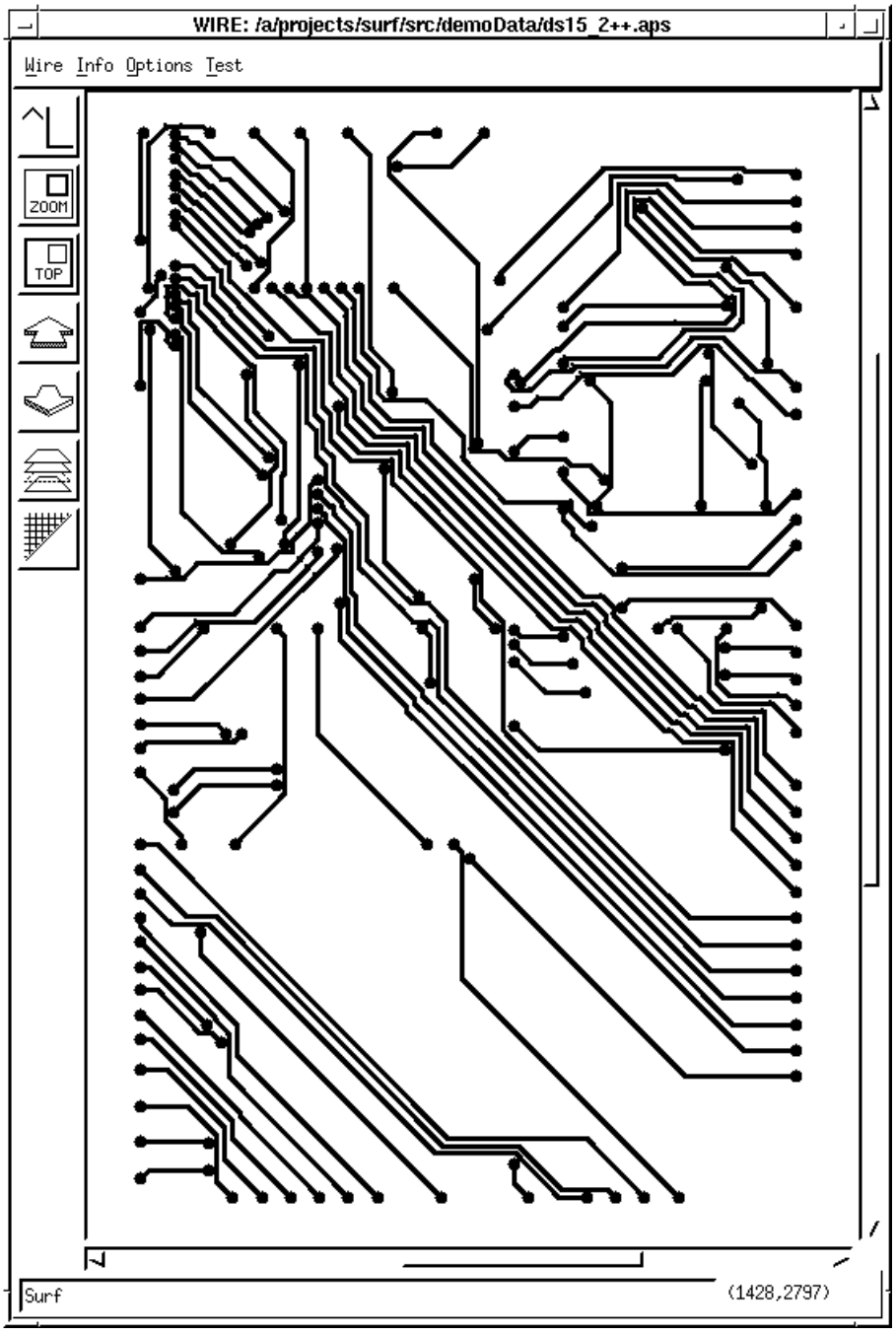


Figure 6.5: Straightened octilinear transformation with 322 jogs.

7. Future Work

There are a number of possible extensions to the work described in this thesis. A few of them are described in this section.

7.1 Endpoint Sizes

Both the algorithm described here and the corresponding implementation assume that the width of a rubber-band segment is uniform along its entire length. However, in many technologies this is not necessarily true—especially at the ends of the segment. The ends of wires may represent chip IO pins, vias, connections to integrated resistors or capacitors, etc. It is quite common for these objects to be larger than the wires that connect them.

Although it may complicate the implementation slightly, extending the current approach to allow for variable width endpoints would provide no conceptual difficulty. The segment ordering phase would have to be modified to take this into account when generating the modified sketch. Also, the UpdateBoundary routine used by TransformLayer and StraightenLayer would have to be modified to incorporate the larger endpoints into the previous boundary. It is interesting that neither the TransformSegment nor the StraightenSegment routines would have to be modified. This is because these routines have "no choice" as to the locations of the segment endpoints—so modifying endpoint sizes should not affect these algorithms.

7.2 Reducing Jogs

More work could also be done on the jog removal phase that follows the initial transformation. The current approach uses a simple heuristic for deciding when to slide a section: a section is slid only if it can be completely aligned with the previous section of the current segment. Other techniques may be useful here.

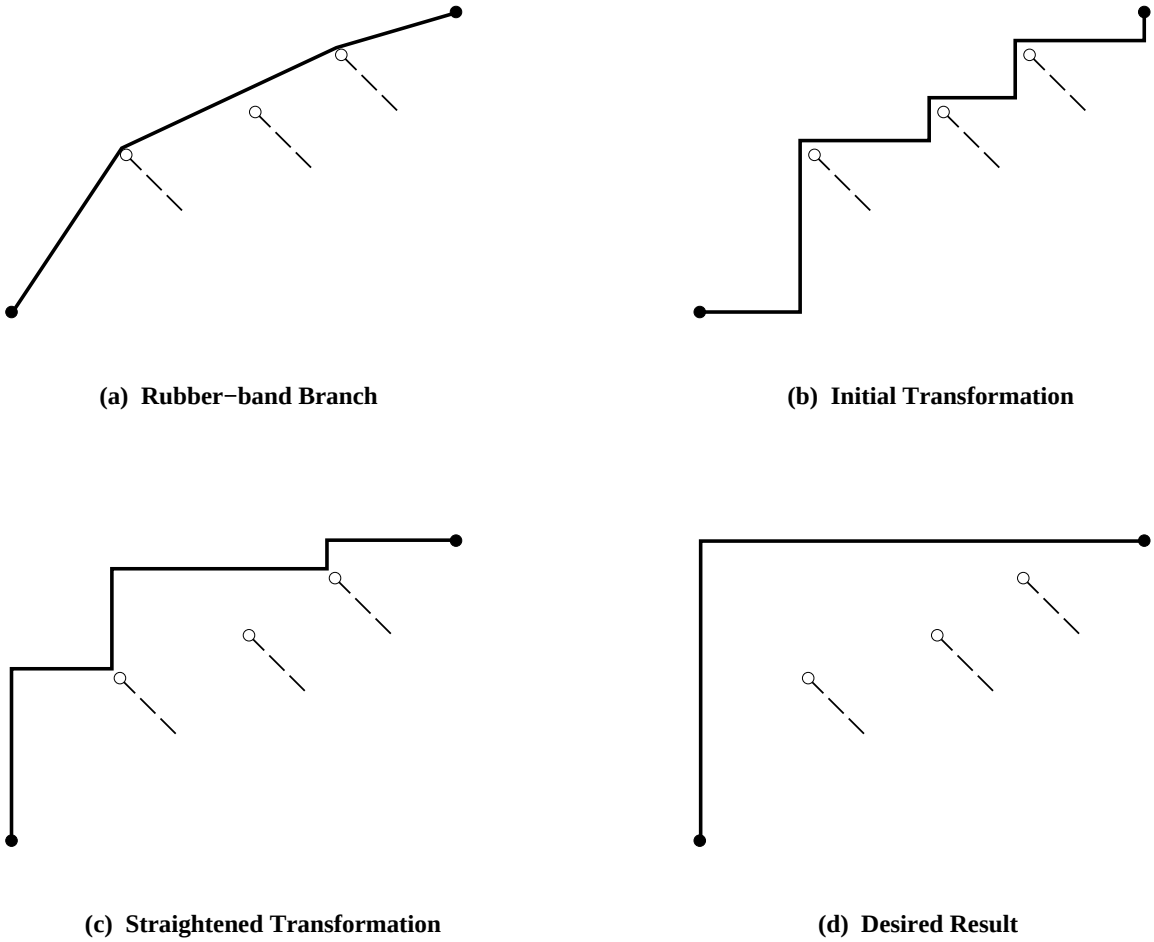
Another drawback of the current straightening approach is that it never repositions the endpoints of segments. In some cases, these endpoints should be fixed (pins, vias, etc.).

However, in some cases segment endpoints represent places that a wire bends around spoke obstacles. There is no reason that the endpoints should be fixed in these cases. Consider the rubber-band branch shown in Figure 7.1 (a). The initial transformation of this branch may look something like Figure 7.1 (b). Assuming that there is plenty of room above this branch, the best possible straightening is shown in Figure 7.1 (c). Because this solution was constrained to pass through the two spoke locations, it must contain at least five jogs. A solution not constrained in this way could be realized with only a single jog (Figure 7.1 (d)). Several examples of these unnecessary jogs can be seen in Figures 6.3 and 6.5. A possible solution to this problem would be to group segments into *chains* and straighten all the sections of an entire chain at once. In the rectilinear case, it is probably sufficient that a chain be monotone in y. However, because of the two different octilinear cases, all the segments of an octilinear chain may also have to be of similar (steep or shallow) slope.

7.3 Variable Interwire Spacing

The approach described in this thesis uses a metal-to-metal spacing that is constant for all wires on a given layer. There may be some advantage to supporting a variable spacing. For example, it may be necessary in order to generate evenly distributed wiring. One possible solution is to make spacing a function of the segments being separated. The TransformLayer and StraightenLayer routines could be modified fairly easily to support this extension. However, some additional thought will have to be given to the OrderSegments routine.

Currently, OrderSegments works by building "precedes" relationships on a modified version of the input sketch. The sketch is modified by sliding endpoints or creating "overlap segments" to account for the width and spacing requirements of the sketch. This approach requires that the amount of resources consumed by a given rubber-band segment be a function solely of that segment. A variable segment-to-segment spacing would violate this assumption.



(a) Rubber−band Branch

(b) Initial Transformation

(c) Straightened Transformation

(d) Desired Result

Figure 7.1: Further Jog Removal

7.4 Localized Update

One of the main advantages of using the rubber-band wiring model is the ability to support an incremental design style. That is, a small modification to the problem should result in a small modification of the final result; it should not be necessary to recalculate the entire solution "from scratch". Using the technique described in this paper, the entire geometric wiring would have to be recomputed for even a small change to the sketch. If the internal rubber-band representation were to be hidden from the user, this recomputation may be too slow to support interactive editing in the geometry domain.

To support these types of small changes, it may be useful to develop a *localized* version

of the geometric wiring algorithm. Such an algorithm would take an initial sketch, its transformation, and some localized changes as input. The output would be the updated transformation. Hopefully, the updated transformation could be calculated with less work than transforming the entire sketch.

8. Previous and Related Work

8.1 Rubber-band Routing

The concept of representing wires without specifying their precise geometry has received a fair amount of attention in the field of VLSI layout. There are many terms used to describe this type of model. Some of these include: rough routing, homotopic routing, and rubber-band routing. This section describes some of the work done in this area.

8.1.1 Routability Checking

Cole and Siegel address the problem of finding a detailed routing given a homotopy (DRH) [CS84]. That is, given a placement of modules, terminals along module boundaries, and a rough routing for each net, determine if there is a single layer detailed routing conforming to the homotopy. They showed how to solve the DRH problem in polynomial time and proved that a DRH problem was routable if and only if it is *safe*.

Leiserson and Maley formalized many of the concepts and ideas used by the SURF system [LM85]. These include the rubber-band equivalent, the rubber-band *sketch*, and *spokes*. They also proved, independently of Cole and Seigel, a number of theorems relating the routability of a sketch to the flow and capacity of its cuts. They presented a polynomial time algorithm for testing the routability of a sketch by transforming it to a legal routing. This algorithm provided the starting point for the SURF spokes algorithms presented [DKS91, Kon92].

8.1.2 Compaction

Maley presented an algorithm for performing one-dimensional VLSI layout compaction with automatic jog insertion [Mal86]. This algorithm compacts rigid circuit elements (modules) by treating wires as flexible objects which indicate the topology of the layout. Maley's work in this area is extended by others [VKLS90, LdDSW91, dDWLS91].

8.2 Enhanced Plane Sweep

Sato and Ohtsuki present the *enhanced plane sweep* [SO86]. In addition to formalizing this technique, they apply it to several layout problems including: geometrical design rule checking, mask generation, and layout pattern spacing.

8.3 Channel Compaction

The process described in this thesis of producing a geometric wiring in two steps is similar to the approaches used by many channel compactors. Channel compactors often use an initial sweep to determine minimum channel height and then a second sweep to embed the wiring. However, the channel compaction problem is slightly different than the geometric wiring problem in that channel compactors treat terminals as movable objects.

Xiong and Kuh present an algorithm for a one-dimensional channel spacer [XK87]. The algorithm uses two plane sweeps to transform an initial channel layout to one with minimal channel height and minimal wire length. The first sweep pushes the channel down as far as possible and creates "all possible jogs". The second sweep works in the opposite direction and generates the actual embedding of the channel. In this sweep, objects are allowed to "slide up" to reduce the wire length and number of jogs, if doing so does not increase the channel height.

A one-dimensional geometrical channel compactor is presented by [RPV+87] by Royle, *et al.* [RPV+87]. This compactor starts with a symbolic channel routing and uses a two-sweep process to determine the final layout for the horizontal routing trunks. The first sweep works from bottom to top and determines the minimum channel height. At each step, the current trunk is compacted to follow the contour of those below it. Sufficient spacing is provided to allow for wire width and interwire spacing. In the second step, a straightening sweep processes the trunks in the reverse order. The purpose of this step is to reduce the wire length without increasing the channel height. Each trunk is straightened between an upper and lower contour by removing U-turns and by "corner-flipping".

Van Ginneken and Jess describe a channel compactor as part of a gridless system for custom building block layout [vGJ87]. First, the channel is compacted. In this step, the minimum contour of the current trunk is incorporated into the contour of the already-compacted trunks by using a recursive algorithm similar to mergesort. The second wire straightening phase processes trunks in the reverse order of compaction. It uses a left to right sweep between an upper and lower contour to realize the current trunk using the fewest number of jogs.

Hughes studies a number of topological routing problems including unconstrained via minimization, and one-dimensional channel compaction with automatic jog insertion [Hug92]. He solves the channel compaction problem in two phases: determining the minimum channel height and then producing a routing that minimizes jogs and wire length. By basing compaction constraints on topological routability and careful use of balanced search trees, he is able to improve on previous results.

8.4 Jog Reduction

Liao, Sarrafzadeh, and Wong present a solution for the single-layer global routing problem [LSW91]. After constructing an initial geometric routing, homotopic post-processing steps are applied to minimize wire length and remove jogs. A topologically equivalent layout with minimum wire length is first constructed by a process of removing empty-U's. Once this is done, a *Greedy-Slide* algorithm is applied to reduce the total number of jogs. The Greedy-Slide algorithm produces a minimum bend routing for a single monotone chain at a time by treating all other wiring as hard constraints (obstacles). The Greedy-Slide algorithm makes no mention of straightening order for the chains. Since it treats other wires as obstacles, the final result may be far from a global (or even local) optimum in terms of jogs.

Tuan and Teo present a solution to river routing with minimum number of jogs [TT91]. This solution requires that the number of horizontal tracks be known in advance. It uses

dynamic programming to compute the minimum jog river routing in $O(n \times s)$ time where n is the number of nets and s is the number of horizontal tracks.

9. Summary

The contribution of this thesis was to present a method for the final transformation stage of the SURF topological routing process. Algorithms have been presented that can be used to transform an arbitrary-angle routing in which all of the width and spacing requirements have been met (extended rubber-band sketch) to a topologically equivalent form satisfying the requirements of a more restricted geometry such as rectilinear or octilinear wiring.

This process is divided into three major steps: ordering the segments, producing an initial transformation, and straightening the transformation. Combining the individual components yields a total algorithm that runs in $O(n \log n + m)$ time where n is the number of rubber-band segments and m is the number of wire sections in the initial transformation. The algorithm has been implemented in C and incorporated into the SURF system. It has been applied to several examples. Finally, several future modifications and extensions have been suggested.

References

[CDS+92] Chung-Kuan Cheng, David N. Deutsch, Craig Shohara, Mark Taparauskas, and Mark Bubien. Geometric compaction on channel routing. *IEEE Transactions on Computer Aided Design*, 11(1), January 1992.

[CS84] R. Cole and A. Siegel. River routing every which way, but loose. In *Proceeding of 25th Annual Symposium on Foundations of Computer Science*, pages 65–73, 1984.

[Dai91] Wayne Wei-Ming Dai. Performance driven layout of thin-film substrates for multichip modules. In *Multichip Module Workshop Extended Abstract Volume*, pages 114–121. IEEE Computer Society, March 1991.

[DDS91] Wayne Wei-Ming Dai, Tal Dayan, and David Staepelaere. Topological routing in SURF: Generating a rubber-band sketch. In *Proceedings of the 28th Design Automation Conference*, 1991.

[dDWLS91] Paul de Dood, John Wawrzynek, Erwin Liu, and Robert Suaya. A two-dimensional topological compactor with octagonal geometry. In *Proceedings of the 28th Design Automation Conference*, pages 727–731, June 1991.

[DKJS90] Wayne Wei-Ming Dai, Raymond Kong, Jeffrey Jue, and Masao Sato. Rubber band routing and dynamic data representation. In *IEEE International Conference on Computer Aided Design*, 1990.

[DKS91] Wayne Wei-Ming Dai, Raymond Kong, and Masao Sato. Routability of a rubber-band sketch. In *Proceedings of the 28th Design Automation Conference*, 1991.

[Hug92] Thomas Hughes. *Topological Routing Problems*. PhD thesis, North Carolina State University, 1992.

[Kon92] Raymond Kong. Incremental routability test for planar topological routing. Master's thesis, University of California, Santa Cruz, 1992.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., 1978.

[LdDSW91] Erwin Liu, Paul de Dood, Robert Suaya, and John Wawrzynek. A topological framework for compaction and routing. In *Advanced Research in VLSI: Proceedings of the 1991 University of California, Santa Cruz Conference*, pages 212–228. The MIT Press, March 1991.

[LM85] C.E. Leiserson and F.M. Maley. Algorithms for routing and testing routability of planar VLSI layouts. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 69–78, 1985.

[LSW91] K. F. Liao, M. Sarrafzadeh, and C. K. Wong. "Single-Layer Global Routing". In Proceedings of Fourth Annual IEEE International ASIC Conference and Exhibit, pages p14–4.1 – p14–4.4. IEEE, September 1991.

[Mal86] F. Miller Maley. Compaction with automatic jog introduction. Master's thesis, Massachusetts Institute of Technology, May 1986.

[RPV+87] J. Royle, H. Palczewski, N. VerHeyen, N. Naccache, and J. Soukup. Geometrical compaction in one dimension for channel routing. In *Proceedings for the 24th Design Automation Conference*, pages 140–144, 1987.

[SO86] Masao Sato and Tatsuo Ohtsuki. Enhanced plane-sweep method for LSI pattern design problems. *Institute of Electronics and Communication Engineering of Japan, Technical Report on Circuits and Systems*, CAS86-199, 1986.

[Tom80] Martin Tompa. An optimal solution to a wire-routing problem. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 161–171, 1980.

[TT91] T.C. Tuan and K.H Teo. On river routing with minimum number of jogs. *IEEE Transactions on Computer Aided Design*, 10(3):270–273, February 1991.

[van75] W. M. vanCleemput. On the topological aspects of the circuit layout problem. In *Proceedings of the 11th Design Automation Workshop*, pages 441–450, 1975.

[vGJ87] L.P.P.P. van Ginneken and J.A.G Jess. Gridless routing of general floor plans. In *IEEE International Conference on Computer Aided Design*, pages 30–32, 1987.

[VKLS90] John Valainis, Sinan Kaptanoglu, Erwin Liu, and Robert Suaya. Two-dimensional IC layout compaction based on topological design rule checking. *IEEE Transactions on CAD*, 9-3:260–275, March 1990.

[XK87] Xiao-Ming Xiong and Ernest S. Kuh. Nutcracker: An efficient and intelligent channel router. In *24th Design Automation Conference*, pages 298–304, 1987.