Copyright © 1981, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

FAST ALGORITHMS FOR VLSI LAYOUT RULE CHECKING

by

Dale Skeen

Memorandum No. UCB/ERL M81/74

17 September 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Fast Algorithms for VLSI Layout Rule Checking

Dale Skeen

Computer Science Division
EECS Department
University of California, Berkeley

Abstract

This memo discusses two *offline* algorithms for VLSI layout rule checking, specifically for checking minimum separation and minimum feature size. In the first part a rule checker using a data structure similar to two-dimensional bins is presented. It is a (worst case) linear algorithm that is fairly restricted in scope. However, it can be generalized (with some loss in performance) to perform incremental checking. The second part presents a rule checker based on a *scan line* algorithm. Scan line algorithms constitute a powerful class of *O(e log e)* algorithms that are applicable to a wide variety of planar geometric problems. It is the purpose of the second part to serve as a tutorial introduction to this class of algorithms.

Fast Algorithms for VLSI *Dale Skeen*

The advent of VLSI technology with its 10 to 100 fold increase in circuit complexity has spurred the need for computationally efficient algorithms to assist in circuit design. Many of computer aided design problems encountered in VLSI can be formulated as planar geometric problems. In this memo we consider one such problem – the problem of ***layout rule checking***, which includes (1) checking that the internal dimensions of components satisfy minimum width rules and (2) checking that rules for minimum spacing between (unconnected) components are satisfied.

We assume that the geometries defining the components of the circuit are of the Mead and Conway variety ([MEAD80]) – geometries are rectangles oriented parallel to the x- and y-axes (often called "Manhattan rectangles"). Although, components are distributed on several layers (typically 5 to 12), we will present the algorithms as if only a single layer was involved. Extending them to multiple layers is straightforward.

The memo is logically divided into two self-contained parts, which may be read separately. In the first part we discuss an asymptotically optimal layout rule checker using two dimensional bins. The algorithm requires linear time and space, and it is easy to implement. However, it is rather limited in scope – in particular, it is not suitable for the more general problem of identifying intersecting rectangles.

The second part is a tutorial introduction to a simple but powerful class of algorithms: the ***scan line*** algorithms. Scan line algorithms are (worst-case) asymptotically optimal for many geometric problems involving oriented rectilinear polygons (i.e. ***rectagons***), and in particular, they are optimal for the intersecting rectangles problem. Many VLSI design problems – including layout rule checking, finding connected components, and (geometric) normalization of components – can be reduced to a special case of this problem. In addition to

introducing the scan line technique, we show how it can be tailored to the problem of layout rule checking.

The algorithms presented are *offline* algorithms: they require that the geometries be known a priori. The algorithm in the first section can be extended in a straightforward manner to perform *online* (or incremental) layout rule checking; although, it will lose its (worst case) linear performance. The scan line algorithms can not be generalized to perform *online* checking.

I. Linear Time Design Rule Checker

The algorithm described is tailored to the following types of layout rule checking:

(1) checking for minimum overlap of intersecting regions,
(2) checking for minimum separation of nonoverlapping regions,
(3) checking for minimum width of a region.

While this method can be extended to solve the general intersecting rectangles problem, it will lose its linear worst-case bound. (Although, from the statistics presented in [BENT80a], an average-case linear time bound is still expected for typical circuit layouts.)

To simplify the presentation we will discuss only minimum separation checking ((2) above) on Manhattan rectangles. We will present the algorithm using the simple layout rule: *(unconnected) components must be separated by a minimum width of* λ. (Having a single parameter denoting minimum separation is a simplification of the layout rule checking problem. We discuss using more complicated layout rules at the end of the section.)

The algorithm proceeds by partitioning the layout across one of its dimensions (e.g. horizontally). The partitions are then processed sequentially (e.g. from bottom to top). Within each partition, the contained vertices are examined. It is in this latter phase that this algorithm resembles the scan line approaches of the next section. However, since there are significant differences between the approaches, we will call the sequential processing of the partitions, a ***sweep***.

The major data structure used in the implementation of the algorithm is a ***bin***. A bin is an unordered list whose elements have values within a predefined interval [a, b]. The width of this interval is referred to as the width of the bin. Bin algorithms have previously been applied to VLSI problems, notably in

[BENT80b].

In contradistinction to other layout rule checkers, this algorithm never requires the vertices to be sorted; instead, only the classification of vertices with respect to partitions is required. This accounts for the speed of the algorithm.

While we discuss only Manhattan rectangles, the extension to rectagons is straightforward. However, the algorithm is not extensible to other geometries (e.g. circles or 45 degree angles).

The Algorithm

It is convenient to consider the circuit as being bounded by the y-axis on the left and by the x-axis along the bottom. Therefore, the lower left corner coincides with the origin.

The algorithm consists of two *sweeps*: one in the vertical direction (from bottom to top) to detect violations in the horizontal spacing of the geometries, and one in the horizontal direction to detect violations in the vertical spacing of the geometries. We will discuss only the vertical sweep; the horizontal sweep is symmetric to the vertical one.

For the vertical sweep, the layout must be decomposed into horizontal partitions of equal width, and a bin allocated for each partition. Then, each vertex is placed into the bin associated with the partition containing it. This is illustrated in Figure 1, where vertices are labelled and stored in their respective bins. The attributes stored with a vertex are its Cartesian coordinates and an indication of its orientation with respect to the rectangle containing it (e.g. lower-left, upper-right). After the completion of this step the *sweep* can begin.

Recall again that the purpose of a vertical sweep is to detect horizontal spacing violations, and this entails measuring the spacing between vertical

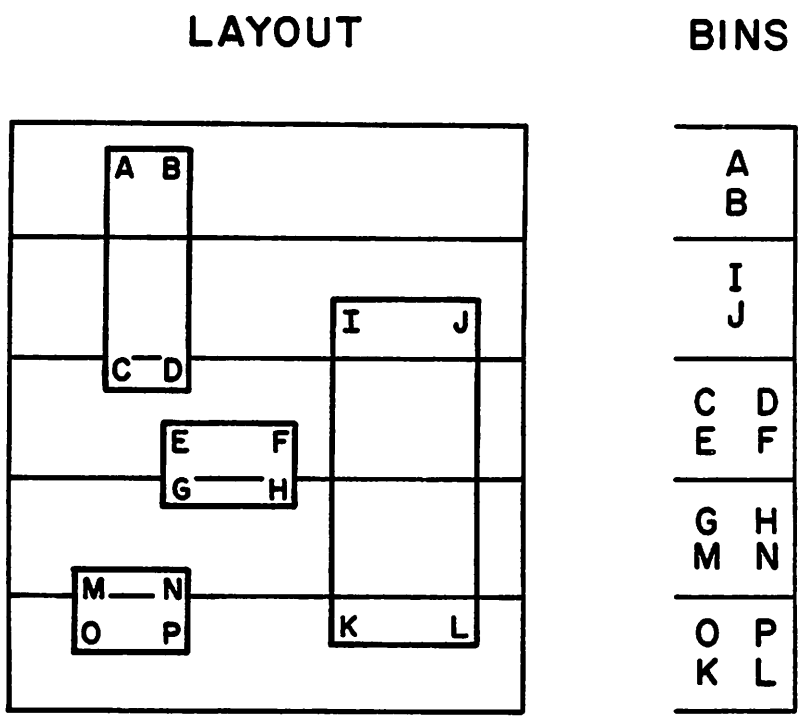


Figure 1. A horizontally partitioned layout and the grouping of labelled vertices into bins.

edges. Conceptually, the sweep proceeds as follows. For every partition, we will project all of the **vertical edges** intersecting the partition onto the x-axis. This is illustrated in Figure 2 for the highlighted partition. Now, if the distance between two vertical edges is less than λ, then the distance between their projections will also be less than λ. For example, in Figure 2 the edges FH and IK are too close. Of course, it is not always the case that a horizontal spacing violation occurs when the projections are too close. For example, edges EG and NP obey the layout rules, even though their projections indicate that they could actually overlap.

LAYOUT

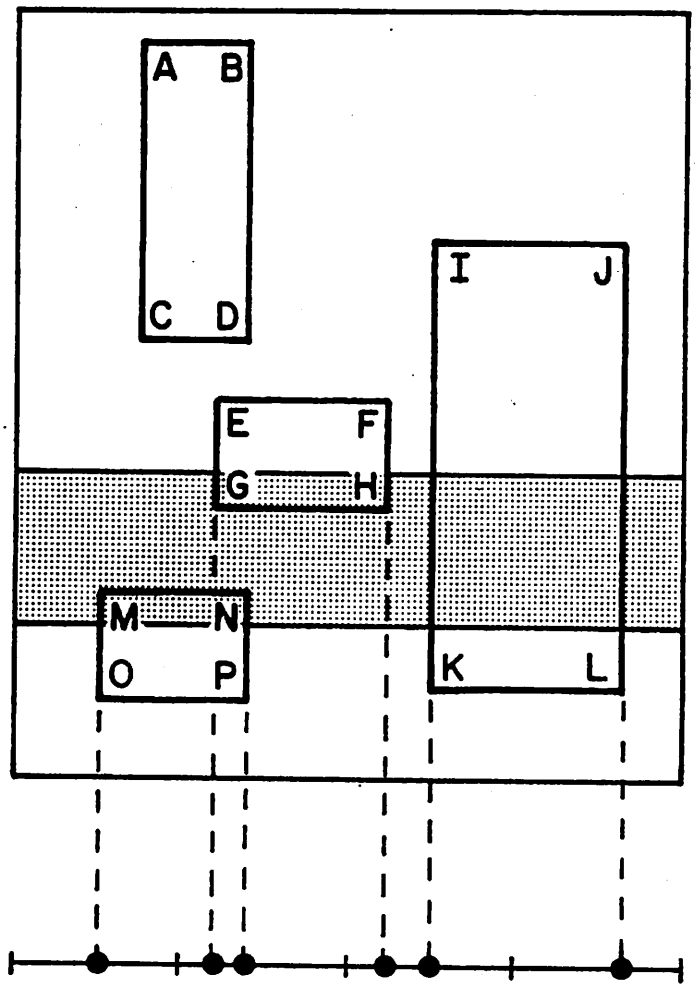


Figure 2. Projecting the vertical edges intersecting with the highlighted partition of the layout onto the x-axis.

A list of the projected edges, the *projections list* needs to be maintained by the algorithm. Again, bins can be useful in the implementation of this list with the bins holding (the representation of) the projected edges. The x-axis should be decomposed into equal length intervals, and a bin associated with each interval. Now, to check if a given vertical edge is too close to any other vertical

edge, one need only check those edges projected into the same or adjacent bins.

The algorithm for maintaining the projections list depends primarily on the sequential processing of partitions. A vertical edge is first encountered when the its *lower* vertex is found in the partition currently being processed. This lower vertex is then projected into correct bin (i.e. a marker for the edge is moved into the appropriate bin). Clearly, all subsequent partitions will intersect with that edge until its *upper* vertex is found. After processing of the partition containing the upper vertex, the edge (i.e. its marker) can then be removed from the projected bin. No subsequent partitions will encounter that edge.

```
Initially the bins holding projections are empty.

For each partition encountered:
    {
     For each lower vertex in the partition:
       {
        Add a marker for the vertical edge
        containing this vertex to the
        appropriate bin.

        Check for layout rule violations
        involving this edge. (This involves
        checking at most 2 adjacent bins.)
       }

     For each upper vertex in the partition:
       {
        Delete the marker for the vertical
        edge containing it.
       }
    }
```

Figure 3. The algorithm for performing a vertical sweep.

The complete algorithm for performing a *vertical sweep* is given in Figure 3.

Diagonal Violations

A vertical sweep can detect all violations between vertical edges, and similarly, a horizontal scan can detect all violations between horizontal edges. However, not all violations fall into these categories – Figure 4 illustrates a violation between diagonally positioned regions. Neither a horizontal nor a vertical sweep is guaranteed to detect this type of violation.

It is easy to augment the sweep algorithm to check for diagonal violations: during a sweep, maintain the projections of edges intersected by two adjacent partitions. This can be accomplished in the algorithm given in Figure 3 by changing the second nested loop (the loop for upper vertices). The revised loop

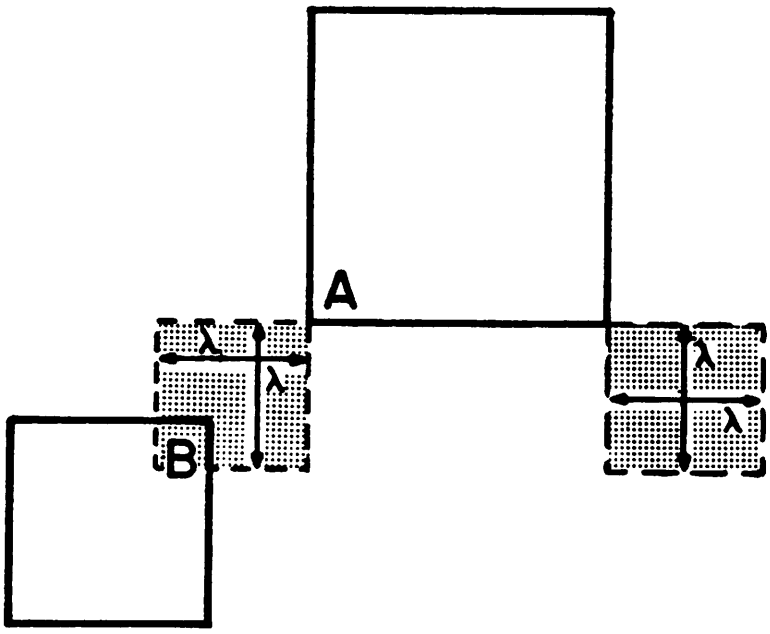


Figure 4. Example of a diagonal violation.

reads: *For each upper vertex in the previous partition....* Note that one modified sweep is guaranteed to detect all diagonal violations. (If both sweeps are modified, then all diagonal violations will be detected twice.)

The complete algorithm for layout rule checking, with diagonal checking, is summarized in Figure 5.

Analysis

The speed of the algorithm is determined by the width of the bins used in partitioning and the width of the bins used in representing the projections list. To achieve linear performance, both bin sizes must be equal to a small constant times λ.

Consider a vertical sweep, where there are no layout rule violations. Let $k_1\lambda$ be the width of a horizontal partition (i.e. bin), and let $k_2\lambda$ be the width of the bins representing the projections list. Recall that these bins hold the projections of the edges intersecting a $k_2\lambda$ wide section of a partitioned region. Now, the dimensions of one of these sections are $(k_1\lambda)(k_2\lambda)$. Since the minimum size of a legal component is λ^2, a section can be tiled with at most $\left\lceil \frac{k_1}{2} \right\rceil \left\lceil \frac{k_2}{2} \right\rceil$ legal

---

I. Partition the layout into horizontal regions of equal width, assigning vertices to their respective regions. (Using a *bin* to hold the vertices contained in a region).
Perform a *modified* vertical sweep (detecting all horizontal and diagonal violations).

II. Partition the layout into vertical regions of equal width, assigning vertices to their respective regions. (Again using *bins*.)
Perform a horizontal sweep (detecting all vertical violations).

Figure 5. The complete linear algorithm for layout rule checking.

---

components that obey the minimum separation rule. Therefore, the maximum number of points in a bin of the projections list is proportional to the constant k_1k_2. Since at most two bins are checked for each vertical edge encountered,[1]* the amount of work to process an edge is proportional to k_1k_2. We have shown that a sweep encountering no violations requires time proportional to k_1k_2e.

If we consider violations, then we observe that violations (between two edges) are detected once during a sweep except for some diagonal violations which are detected twice.[2] Therefore, the total running time of the algorithm is $O(e+V)$, where V is the number of violations.

More complicated layout rules can be handled with little degradation in speed. To handle several layers with a variety of separation rules, we can associate a color with each layer. Now, let the color of an edge be the color of the layer containing the edge. We can define a $C\backslash(mu\,C$ table, where C is the number of distinct colors. The table contains minimum separation rules between the colors. Each rule can be arbitrarily complex, but most rules simply contain a minimum separation constant. Now, checking for separation violations between two edges is a simple table look-up.

[1]The bin containing the vertical edge is checked, and one of the neighboring bins is also checked if the edge lies close to a boundary of its containing bin.

[2]Some diagonal violations are detected during both sweeps. In the algorithm presented in Figure 5, the (unmodified) horizontal sweep should suppress reporting all diagonal violations.

II. Scan Line Algorithms

In the first subsection of this part we define the terminology used in the remainder of the paper and introduce the scan line paradigm. One particular variant of the paradigm is examined in detail. Since all scan line algorithms require sorting, they exhibit $O(e \log e)$ worst case time complexity, where e is the number of edges in the components. In the variants that we present, $O(e \log e)$ is also the average case performance. $O(e)$ space is required.

In the second subsection we introduce the problem of layout rule checking and then present a scan line solution. This solution serves as an archetype for scan line algorithms because of its simplicity and its adherence to the general paradigm. It is a straightforward application of scan line techniques. The algorithm requires two passes over the circuit layout.

Scan line algorithms are attractive because they are conceptually simple and applicable to a wide variety of planar geometric problems. (Shamos discusses many such problems in his thesis [SHAM78].) Moreover, for many problems, scan line algorithms are asymptotically optimal in the worst case, and in particular, they are optimal for the problem of reporting all intersecting rectangles ([BENT80b]). Many VLSI design problems can be cast into the intersecting rectangles problem, including layout rule checking. However, in this memo, we will view layout rule checking as a distinct problem and present an algorithm that is specifically tailored to solve it. The algorithm presented is simpler than those for solving the more general intersection problems.

2.1. The Scan Line Paradigm

We assume that the area of interest is bounded by a rectangle lying in the first quadrant of a Cartesian co-ordinate system. For convenience let the x- and y-axes coincide with the bottom and left sides of this region. A legal geometry component) within this area has an exterior face defined by a rectagon. It

may also contain one or more "holes" (i.e. cutouts), and if it does then its interior face(s) are also defined by rectagon(s). An interior face for one region may be the exterior face of a contained region. A collection of legal geometries is illustrated in Figure 6.

Two regions are said to be *nonoverlapping* if their interiors do not intersect. Two regions are *adjacent* if they are nonoverlapping and if they share at least one common vertex or edge. We will also use the term *abutted* to describe adjacent regions sharing at least one edge. The *normalized form* used in some

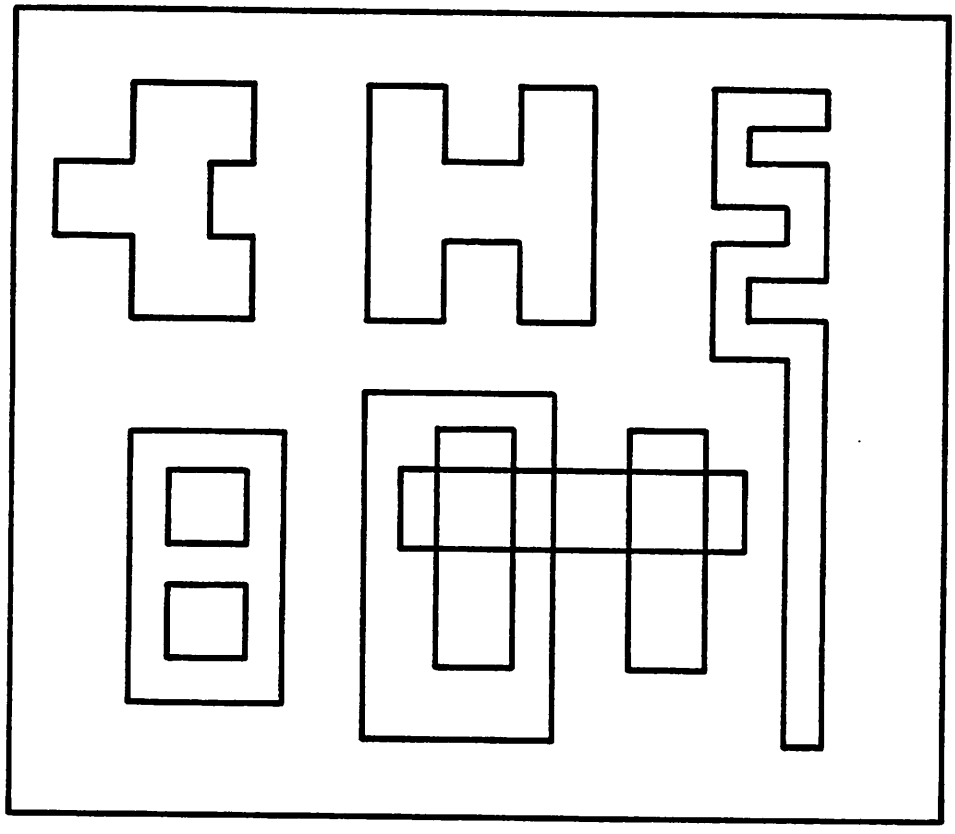


Figure 6. Examples of legal regions.

algorithms does not allow adjacent or overlapping regions.[3]

In a scan line algorithm, a line is systematically swept across the primary region in either the vertical or the horizontal direction. Figure 7a illustrates a horizontal scan line traversing a region in normal form. Since the line is moving in the vertical direction, the figure illustrates a *vertical scan*. The scan is frozen at the instant that the scan line is at position y on the y-axis. At this particular position, the scan line is intersecting all of the components defined in the figure.

The representation of the scan line is dependent upon the particular application and upon the restrictions placed on the regions (e.g. normalization). However, for most applications two standard representations prevail, and these are illustrated in Figures 7b and 7c. They are:

(1) An ordered list of line segments defined by the intersection of the scan line with the regions themselves.
(2) An ordered list of points defined by the intersection of the scan line with faces of the regions.

Representation (1) is suitable only for nonoverlapping regions (as in the case of Figure 7a).

The **scan list** (the representation of the scan line) is ordered on the x-coordinate for a vertical scan (on the y-coordinate for a horizontal scan). A node in the scan list of Figure 7b contains the x-coordinates of the endpoints defining a segment and an unique **region id.** The y-coordinate is not recorded since it is the same for all segments in the scan list.

The second representation is normally only used when the regions, and therefore the segments used in the first representation, overlap. Again, the intersection points are stored ordered on their x-coordinate (for a vertical

[3]However, scan line techniques can be used to decomposed arbitrarily overlapping rectangles into an equivalent set of nonoverlapping rectangles ([BENT79]). Therefore, using nonoverlapping rectangles does not diminish the significance of the algorithm.

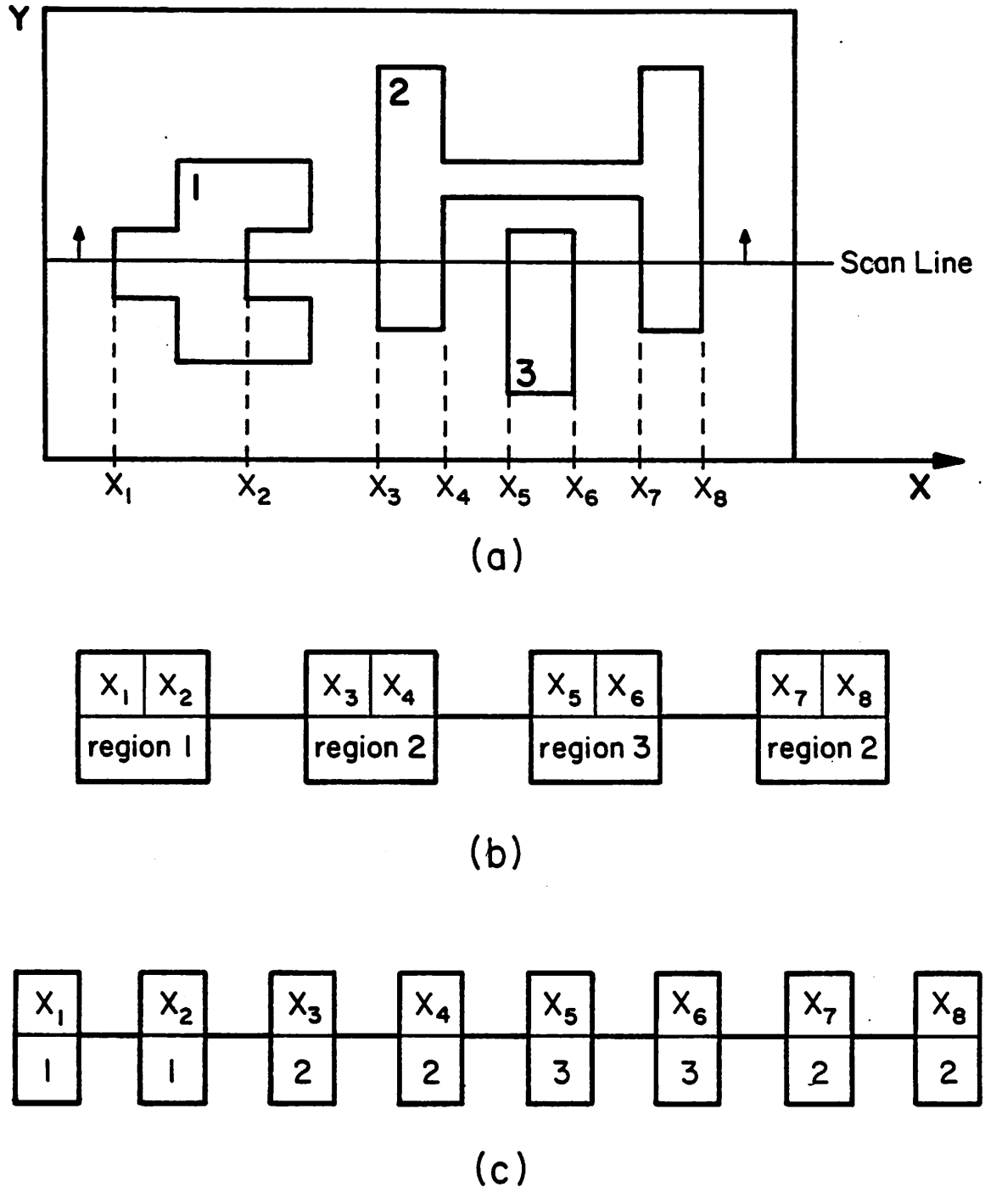


Figure 7. (a) a scan line, (b) a segmented scan list, (c) a point scan list

scan). Figure 7c illustrates such a scan list.

To implement a scan efficiently we require the following observation: during a vertical (horizontal) scan, the scan list changes only when a horizontal (vertical) edge is encountered. For a collection of rectilinear regions defined by e edges, half of the edges will be horizontal, and thus there will be exactly $\frac{e}{2}$ distinct instances of the scan list (this holds only for rectilinear regions).

A scan can be made to proceed efficiently because:

(1) The scan list is updated at discrete intervals along the scan (rather than continuously). For a vertical scan the list is updated only when a horizontal edge is encountered.
(2) Since the list is ordered, a segment (or point) in the list can be accessed quickly. If the list is stored as a balanced tree, then access requires $O(\log e)$ time at most.
(3) Updates to the list are localized to either one or two nodes (and possibly the nodes lying on the access path -- for rebalancing). The scan list need never be completely restructured.

The generalized algorithm for making a vertical scan is the following:

Step 1. Sort the horizontal edges in increasing order on the y-coordinate.

Step 2. For each edge, **e**, in the sorted list do:

a. Update the scan list to reflect **e**.
b. *(This step is application dependent.)*

Step 2a. generally requires a lookup into the scan list and modifying one or two nodes. The modifications performed in this step can be classified into several cases depending on the geometry currently being scanned. The cases for a scan list consisting of line segments (i.e. using the first representation) are given shortly.

An analysis of the algorithm is straightforward. Step 1 sorts $\frac{e}{2}$ edges, thus requiring $O(e \log e)$ time. Step 2 iterates $\frac{e}{2}$ times. Since the list can be stored as a balanced tree and since the list always contains less than $\frac{e}{2}$ entries, then

each iteration of Step 2a requires $O(\log e)$ time. This includes the time it takes to access and modify at most 2 nodes. This implies a worst case time bound of:

$$O(e \log e) + O(V)$$

where V is the total number of operations performed in Step 2b. In many applications V is dependent upon the "interactions" between the regions.

The details of Step 2a are dependent on the representation of the scan line. We now present an algorithm for the representation based on line segments. The algorithm for the *point* representation is very similar. (Again, for simplicity, we are assuming that the geometries in the layout are nonoverlapping and have no shared edges.)

Consider a vertical scan. Figure 8 repeats the regions given in the previous figure, labelling each horizontal line with a number identifying the case to which it belongs. Some cases, notably 2 and 5, have symmetric left and right variants. The cases are briefly described below (note that the line is traversing the layout from bottom to top):

1. *Enter* - a new region (or distinct leg thereof) is entered by the scan line.
2. *Expand* - the width of a region intersected by the scan line is increased.
3. *Join* - in a U-shaped region, the bridge that joins the two legs already scanned is encountered.
4. *Exit* - a region (or distinct leg thereof) is exited by the scan line.
5. *Reduce* - the width of a region intersected by the scan line is reduced.
6. *Split* - in a U-shaped region, the scan line has left the bridge of the region and is entering the legs.

Edges that fall in the first three categories are called ***leading edges***, because they indicate that the scan line is moving into a new region. Similarly, edges in the last three cases are called ***trailing edges***, because they indicate that the scan line is moving out of a region.

When an edge is encountered in the scan, it is first classified according to case, and then the scan list is transformed to record the presence of the edge.

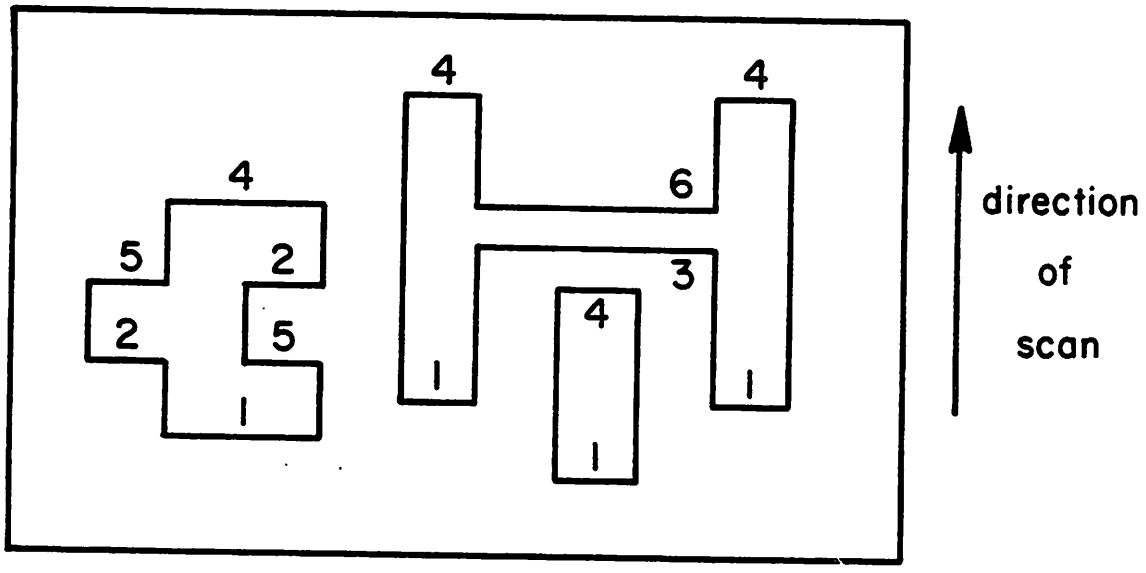


Figure 8. Horizontal edges labelled with their case number for a vertical scan.

As an example, consider encountering the edge with endpoints (x_1,y) and (x_2,y), where $x_1 < x_2$. We will assume that the scan list already contains entries for segments [a,b] and [c,d] and that the nodes for these are adjacent in the lexicographical ordering of the nodes in the list. Table 1 lists each case and gives its defining equations and the transformations required to update the scan list.

The sets of equations given in Table 1 are pairwise inconsistent; therefore, only one set can apply. Also the family of sets for the six cases (including the symmetric right variants that are not illustrated) is complete – in the sense that at least one set of equations must apply. The transformation rules give a before and after image of the scan list. (The solid connector between nodes

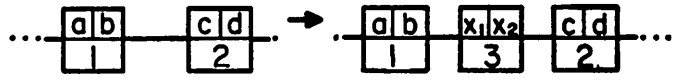
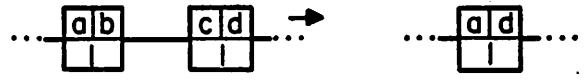
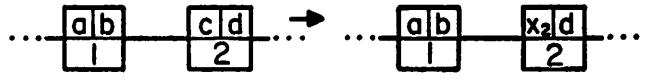
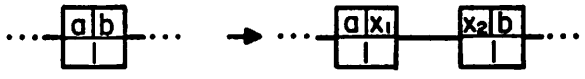
Case	Defining Equations	Scan List Transformation
(1) Enter	$b<x_1$, $x_2<c$	
(2) Expand (left variant)	$b<x_1$, $x_2=c$	
(3) Join	$b=x_1$, $x_2=c$	
(4) Exit	$a=x_1$, $x_2=b$	
(5) Reduce (left variant)	$c=x_1$ $c<x_2<d$	
(6) Split	$a<x_1<x_2<b$	

Table 1. Transformations required to update the *scan list* when edge with endpoints (x_1,y), (x_2,y) is encountered.

indicates adjacency within the scan list.)

The implementation of the above transformation rules in a balanced tree is not difficult. The primitive operations required are: search for a point, modify the contents of a given node, insert/delete a node with rebalancing, and find the successor/predecessor of a node. All of these operations can be performed in *O(log e)* time.

In this section we have defined the scan line paradigm and have given the details of a vertical scan. A horizontal scan is identical to a vertical scan except that vertical edges are used instead of horizontal edges. In the next section we give an example illustrating these techniques.

2.2. A Scan Line Algorithm for Design Rule Checking

In this section we adapt the general *scan line* algorithm of the previous section to the problem of checking layout rules. The adaptation is straightforward and serves to illustrate the power and generality of the scan line approach.

As in Part I, we will assume that the minimum feature size is λ. (Most likely, λ is composed of several components. See [MEAD80] for a complete discussion of layout rules). We will study two (simplified) design constraints:

external constraint – the minimum separation between two components is λ,

internal constraint – the minimum width of a component is λ.

In layout rule checking, it is convenient to measure distance as the maximum displacement in either the horizontal or vertical direction. Hence, the distance between two points (x_1,y_1) and (x_2,y_2) is given by the formula:

$$\max(|x_1-x_2|,|y_1-y_2|)$$

where $|x|$ denotes the absolute value of x.

The Algorithm

The algorithm uses a *segmented scan line* as described in detail in the previous section. Referring back to Figures 7a and 7b, we observe that checking for internal rule violations is tantamount to checking that line segments (i.e. nodes in the scan list) are at least λ wide. Also, we observe that the external layout rule requires that the minimum distance between any two line segments in the scan list is at least λ. This latter observation suggests an efficient way for

checking external layout rules. Given a segment $[x_1, x_2]$ in the scan list, we need only check segments whose endpoints lie in the interval $[x_1-\lambda, x_2+\lambda]$ to discover external layout rule violations involving the given segment.

Now external and internal layout rule checking can be performed at the time an edge is encountered in the scan. The checks are localized to the vicinity of the update, and the type (i.e. internal or external) and range of checking is dependent on which of the 6 cases is encountered. External checks need only be made when:

1. A new region is encountered in the scan (case 1).
2. A region is expanded (case 2).
3. A region is split (case 6) -- the two new regions must be separated by a distance of λ.

Internal checks need to be made when:

1. A new region is encountered in the scan (case 1).
2. A region is shrunk (case 5).
3. A region is split (case 6) -- the two new regions must have width λ.

Table 2 details the checking required whenever a vertical scan encounters an edge with x-coordinates x_1 and x_2. The cases are those defined in Table 1 and illustrated in Figure 8.

It should be apparent from the previous figures that a vertical scan can detect all violations between vertical edges, and similarly, a horizontal scan can detect all violations between horizontal edges. However, diagonal violations fall into neither category (as was illustrated in Figure 4 in Part I.). Clearly, neither a horizontal nor a vertical scan can detect diagonal violations, because at no position during the scan are both regions simultaneously intersected by the scan line.

We now propose an extension to the scan line algorithm that detects diagonal violations. This extension requires an additional constraint: the regions

	Case	Internal Check on	External Check (in the range(s) of)
(1)	*enter*	inserted node	$[x_1-\lambda, x_1]$ and $[x_2, x_2+\lambda]$
(2)	*expand*	–	*(left variant)* $[x_1-\lambda, x_1]$ *(right variant)* $[x_2, x_2+\lambda]$
(3)	*join*	–	–
(4)	*exit*	–	–
(5)	*reduce*	modified node	–
(6)	*split*	both modified nodes	–

Table 2. Design rule checking required when the horizontal edge with vertices (x_1,y), (x_2,y) is encountered.

must obey the internal layout rule.[4] This implies that external and internal rule checking must be performed separately.

Figure 4 illustrates that the minimum distance between two diagonally positioned regions are two corner points: one of them is an endpoint for a leading edge (point A in the figure), while the other is an endpoint for a trailing edge (point B). A procedure for finding diagonal violations is:

```
for each leading edge, e, encountered:
  {
   search for previously encountered
   trailing edges that are in the
   vicinity of either endpoint of e.
  }
```

The vicinity searched is a λ by λ square diagonally adjacent to an endpoint (refer to the shaded area of Figure 4).

This solution requires maintaining a second list of trailing edges. This list contains all trailing edges lying in the region bounded by the current scan line and a parallel line a distance of λ behind it. Notice that trailing edges outside

[4] This constraint can be relaxed at the expense of a slightly more complex representation of the scan line. We do not pursue that here because a more efficient layout rule checker was introduced in Part I.

of this region can not be involved in a diagonal violation with any edge on the scan line. We will call this list the *trailing edge list.*

If the constraint that regions must satisfy the internal layout rule (i.e have width λ) is added, then the projections of the edges in the trailing edge list onto the x-axis will not overlap. In this case, the trailing edge list can be implemented in a data structure similar to that used for the scan list. Specifically, a balanced binary tree storing line segments could be used.

In Table 3 we give the steps necessary to perform a scan with checking for diagonal violations. Again, the table is decomposed into the six cases described earlier. The table repeats the steps given in Table 2 for external layout rule checking and it adds the steps necessary to (1) maintain the trailing edge list and (2) check for diagonal violations. Essentially Table 3 states that, upon encountering a leading edge, a range within the trailing edge list must be searched for diagonal violations, and that, upon encountering a trailing edge, it must be inserted into the trailing edge list.

Case	Modifications to trailing edge list	Check in trailing edge list (in the range(s) of)
(1) *enter*	–	$[x_1-\lambda, x_1]$ and $[x_2, x_2+\lambda]$
(2) *expand*	–	*(left variant)* $[x_1-\lambda, x_1]$ *(right variant)* $[x_2, x_2+\lambda]$
(3) *join*	–	–
(4) *exit*	add segment $[x_1, x_2]$	–
(5) *reduce*	add segment $[x_1, x_2]$	–
(6) *split*	add segment $[x_1, x_2]$	–

Table 3. Design rule checking for *diagonal* violations and the required modifications to the *trailing edge list* when the horizontal edge with vertices (x_1,y), (x_2,y) is encountered.

Now, we are ready to state a completed version of the algorithm for external rule checking. The algorithm is summarized in Figure 9. Notice that two scans are required: one in the vertical direction and one in the horizontal direction. To detect diagonal violations, one of the scans must be augmented with a trailing edge list. Since one augmented scan can detect all diagonal violations, it is important that only one scan is so augmented; otherwise, diagonal violations will be detected twice. We have arbitrarily chosen the horizontal scan to check for diagonal violations.

Some implementation details are important in the complete algorithm. The first is the case where several edges are encountered simultaneously. Notice that in the case where a trailing edge and a leading edge are encountered simultaneously, the detection of a violation during a normal scan (i.e. without the trailing list) may depend on whether the trailing edge is deleted before the leading is inserted. To circumvent this problem we defer the deletion of trailing edges until all leading edges at the level of the scan line have been processed.

A second point is that the trailing edge list requires the removal of edges when they are more than distance λ behind the scan line. A couple of schemes for removing the edges are possible. One is to leave inactive edges in the tree

Step 1.	Perform a vertical scan, using Table 2 to check ***vertical*** spacing and using Table 3 to check ***diagonal*** spacing.
Step 2.	Perform a horizontal scan, using Table 2 to check only ***horizontal*** spacing.

Figure 9. The Design Rule Checking Algorithm.

until they are encountered during a search. They can be deleted at that time. A second scheme is maintain a queue of descriptors for the edges in the tree. Each time the scan line advances, descriptors are removed from the queue until the front descriptor is for an edge that belongs in the tree. Every time a descriptor is removed from the queue, the corresponding edge is removed from the trailing edge list.

2.3. Conclusions

We have illustrated the scan line algorithms using a single layer; however, more complicated layout rules can be handled with little degradation in speed. (We outlined a multilayer approach at the end of Part I.)

All scan line algorithms require sorting the edges; therefore, they have an $O(e \log e)$ worst case time complexity. For layout rule checking, the complexity is better described by $O((e \log e) + V)$, where V is the number of violations found (maximally, this number is quadratic in the number of edges).

These algorithms are conceptually attractive but tend to be hard to implement. While they exhibit the optimal asymptotic worst case performance for several important VLSI problems, in practice they sometimes lose to more naive algorithms, which have an $O(e^{\frac{3}{2}})$ average case time complexity, on reasonable sized and realistic layouts. (See [BAIR78] for a discussion of the naive algorithms currently in use.) This is due to the relatively large overhead for maintaining the data structures.

Nonetheless, the scan line algorithms constitute an important class of algorithms because of their generality. Other applications include decomposing rectagons into rectangles (required in manufacturing masks) and finding connected components.

Acknowledgements

I would like to thank Bernard Mont-Reynaud for introducing this problem in his Problem Solving Workshop, Richard Newton for many meaningful discussions, and Ken Keller for his comments on an early draft of this paper and for sharing his insight on the current issues of CAD-VLSI.

References

[BAIR78] Baird, H. S., "Fast Algorithms for LSI Artwork Analysis," *Journal of Design Automation and Fault-Tolerant Computing*, 2, 2, pp. 179-209.

[BENT79] Bentley, J., and D. Wood, "An Optimal Worst-Case Algorithm for Reporting Intersections of Rectangles," to appear in *IEEE Transactions on Computers*.

[BENT80a] Bentley, J., D. Haken, and R. Hon, "Statistics on VLSI Design," Carnegie-Mellon Technical Report No. CMU-CS-80-111, April 1980.

[BENT80b] Bentley, J., D. Haken, and R. Hon, "Fast Geometric Algorithms for VLSI Tasks," available from *IEEE* as reprint CH1491-0/80/0000-0088, 1980.

[MEAD80] Mead, C. A., and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980.

[SHAM78] Shamos, M. I., "Problems in Computational Geometry," Ph.D. Thesis, Yale University, 1978.