FAST CHANNEL GRAPH CONSTRUCTION

James P. Cohoon
Department of Computer Science
University of Virginia

CS Report No. TR-85-18
August 1, 1985

Fast Channel Graph Construction†

James P. Cohoon
Department of Computer Science
University of Virginia
Charlottesville, VA 22901

Abstract: Channel routing is a valuable VLSI routing technique that requires the routing region be divided into rectangular regions or *channels*. A new algorithm is proposed that constructs the channels and determines the adjacencies between the channels in $O(n \log n + m)$ time, where n is the number of modules in the circuit and m is the number of channels that are created. In addition to its fast run-time, this algorithm has a parameter k that permits control of the number of channels created.

Keywords: Channel routing, channel graph, circuit layout

† This research was supported in part by Virginia Center for Innovative Technology grant INF-86-001

1. INTRODUCTION

Channel Routing is an important technique for routing interconnections in VLSI layouts. Its importance stems from its principal operating characteristics:

- 100-percent completion when given acyclic constraints and adjustable heights;
- low order polynomial run-time.

First proposed by Hashimoto and Stevens [HASH71] for realizing the interconnections on the ILLIAC-IV computer, the channel routing method requires that the routing region be decomposed initially into a collection of rectangles or *channels*. From the channels, a *channel graph* (V,E) is constructed, where vertex set V is the set of all channels and edge set E is the set of all unordered pairs of channels (u,v) such that u and v are contiguous channels. Subsequent phases of the method perform a coarse routing that specifies the channel sequence for each net and a fine routing that specifies physical paths within the channels for each net. While there are numerous algorithms to determine the coarse and fine routings (e.g. [BURS84, RIVE83, YOSH82]), there are only a few informal algorithms to construct the channels and the channel graph. These informal algorithms are limited to constructing channel graphs with either $O(n)$ or $O(n^2)$ channels, where n is the number of modules (e.g. [CHIB81, SOUK80, ULLM84]). They require $O(n \log n)$ and $O(n^2 \log n)$ time to construct channel graphs with respectively $O(n)$ and $O(n^2)$ channels. The interest in channel graphs with more or less channels stems from characteristics of the particular channel router used to determine the coarse and fine routings. For example, the Magic router performs better on channels that are square rather than simply rectangular [OUST84]. A heuristic to improve the Magic router's performance initially constructs $O(n^2)$ channels and then applies a merging algorithm to selectively collapse some of the channels together so the resulting set of channels are more square [ULLM84].

We propose an $O(n \log n + m)$ algorithm for the construction of the channels and the channel graph, where m is the number of channels that are created. Thus, our algorithm is faster for channel graphs with $O(n^2)$ channels and as fast for graphs with $O(n)$ channels. Also, our algorithm through an input k allows the user to control the number

of channels that are created. Throughout our discussion we assume that the circuit perimeter and the modules are rectangular and represented by their horizontal and vertical boundary segments. We also assume that all boundary segments have nonzero length and are appropriately marked upper, lower, right, and left.

2. CHANNEL GRAPH CONSTRUCTION

To decompose the routing region into channels, the module boundary segments are extended into the routing region until they intersect another line segment. These new line segments are called *extension* segments. An extension segment is *always* ended when it intersects another module boundary segment or a circuit perimeter segment. However, depending on the number of desired channels, an extension segment can also be ended when it intersects another extension segment or more generally when it intersects k extension segments. To achieve $O(n)$ channels, simply extend into the routing region all horizontal (vertical) module boundary segments until they intersect a boundary segment. To achieve the

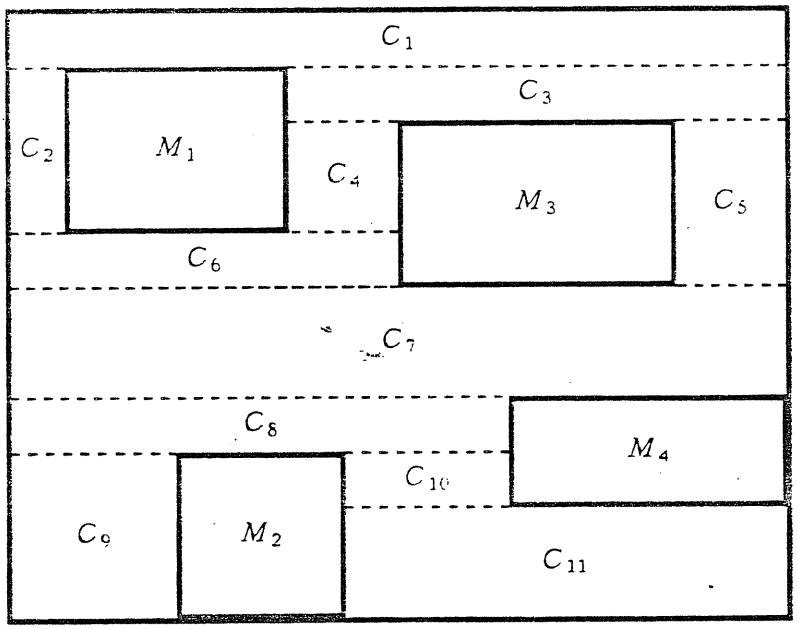

Figure 1 - Decomposing Routing Region into Channels with $k=0$

maximum number of channels, extend all horizontal and vertical module boundary segments into the routing region until they intersect a boundary segment. To achieve $O(k \cdot n)$ additional channels, extend all horizontal (vertical) module boundary segments into the routing region until they intersect a boundary segment and then extend all vertical (horizontal) module boundary segments until they intersect k segments or a boundary segment whichever occurs first. Figures 1-3 show a simple circuit with 4 modules and with respectively $k=0$, $k=1$, and $k=n$. As a result Figure 1 has 11 channels, Figure 2 has 23 channels, and Figure 3 has 40 channels.

Our algorithm for constructing the channels and channel graph is a two-phase algorithm. The first phase constructs the horizontal extension segments; the second phase constructs the channels and the channel graph. Without loss of generality, our description of the phases assumes that k controls the number of horizontal extension segments a vertical extension segment can intersect.

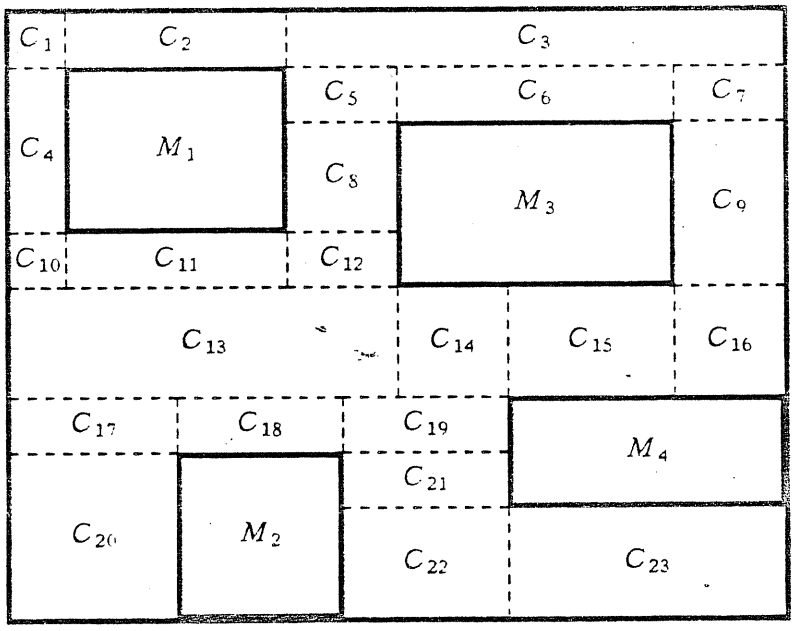

Figure 2 - Decomposing Routing Region into Channels with $k=1$

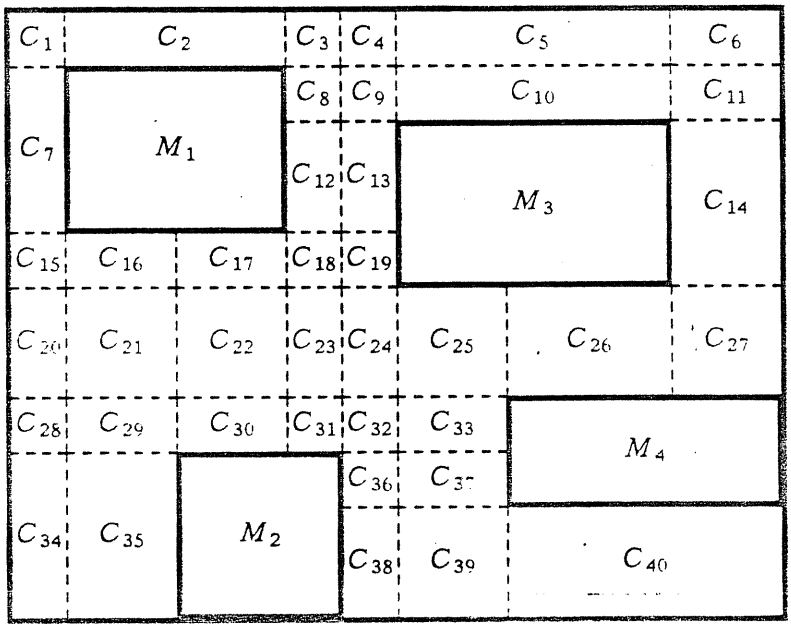

Figure 3 - Decomposing Routing Region into Channels with $k=n$

Horizontal Extension Segment Construction

An algorithm *HESC* is given in Figure 4 to construct the horizontal extensions segments. The set of horizontal extensions segments H is determined by performing a left-to-right line sweep of the vertical boundary segments V [BENT79]. The sweep is accomplished by accessing the set of vertical boundary segments V after they have been arranged in non-decreasing order with x-coordinate as the primary key and lower y-coordinate as the secondary key. Each vertical boundary segment is considered in turn as the termination point for horizontal extension segments extending left and as the initiation point of horizontal extension segments extending right. The time to arrange V in line sweep order is $O(n \log n)$, as $|V|$ is $O(n)$.

The following notation is used in the algorithm.

The upper endpoint of a vertical segment v is (v_x, v_u). The lower endpoint of a vertical segment v is (v_x, v_l).

A vertical boundary segment v *covers* point (a,b) iff v is the vertical boundary segment with the minimal x-coordinate greater than a such that b lies in the open interval (v_l, v_u). Note from the definition, if v covers point (a,b) then b is not equal to v_l or v_u.

The type of tree used by the algorithm is a B^+—tree [COME79]. A B^+—tree allows insertions and deletions in $O(\log p)$ time, where p is the number of elements in the tree. In addition, a B^+—tree allows the successor or predecessor of an element to be found in constant time. The tree maintains those endpoints of horizontal segments that have not yet been covered by a vertical boundary segment. The key used to place an endpoint is its y-coordinate. Note that a right vertical module boundary segment cannot cover any of the endpoints in the tree. Instead it introduces two new points to be inserted into the tree (step 4). However, a left vertical module boundary segment or the right circuit perimeter segment can cover points in the tree. When computing the set of points S covered by vertical segment v in step 6, there is an $O(\log n)$ cost to find the covered point with maximal y-coordinate. (Note that S can be empty and there can be no such point). The remaining points in S (if any) can be found in $O(|S|)$ time by following predecessor

---

```
1. H ← ∅
2. for each vertical segment v ∈ V (in line sweep order) do
3.     if v is a right module boundary segment or is a left circuit perimeter segment then
4.         Insert (v_x, v_l) and (v_x, v_u) into tree
5.     else
6.         S ← points covered by v (sorted by y-coordinate)
7.         for each (a,b) ∈ S do
8.             H ← H ∪ {horizontal segment from (a,b) to v}
9.         end for
10.        Delete S from tree
11.        if v is a left module boundary segment then
12.            (c,d) ← point in tree with minimal y-coordinate ≥ v_l
13.            (e,f) ← point in tree with maximal y-coordinate ≤ v_l
14.            H ← H ∪ {horizontal segment ((c,v_u),(v_x,v_u))}
15.            H ← H ∪ {horizontal segment ((e,v_l),(v_x,v_l))}
16.        end if
17.    end if
18. end for
```

Figure 4 - Algorithm *HESC* for Horizontal Extension Segments Construction

---

pointers. As the tree is a B^+—tree, the points in S can all be deleted from the tree in $O(\log n)$ time [AHO74].

In addition to the $O(|S|)$ horizontal extension segments extending right (step 8), if v is a left vertical boundary segment then there are two horizontal extension segments extending left from v's endpoints (steps 12-15). The points (c,d) and (e,f) can be found in constant time from S. If S is not empty then the predecessor of highest point in S is (a,b) and the successor of the lower point in S is (c,d). Otherwise, the search that determined S is empty can be made to terminate at (a,b) where (a,b)'s predecessor for this case is necessarily (c,d).

From the above discussions, we see that for each of the $O(n)$ iterations of step 2 there is a search with $O(\log n)$ cost. The time spent in processing an individual S is $O(|S|)$ which throughout the course of the algorithm sums to $O(n)$. Therefore, the run-time of *HESC* is $O(n \log n)$.

Channel Graph Construction

The construction of the channels and channel graph is similar to the construction of the horizontal extension segments. In Figure 5 an algorithm *CCG* is given formally that constructs the channels and channel graph from left-to-right. Informally, the algorithm operates in the following manner. A line sweep of the vertical boundary segments V is again performed. During the line sweep, the vertical boundary segments V are grouped by identical x-coordinates. The groups are considered left-to-right with the current group being denoted T. The segments within T are ordered from bottom-to-top. As V is arranged in line sweep order from algorithm *HESC*, each successive T is readily determined in $O(|T|)$ time. (This grouping of segments in V is for pedagogic purposes. An actual implementation does not require that T exists as an entity). The segments in T are used to determine the right boundaries of channels. The left boundaries of the channels are determined from the horizontal extension segments. The functions *pred* and *succ* used in the algorithm determine respectively the predecessor and successor of a point in the tree.

```
1. C ← ∅
2. S ← {Segments in H that abut left vertical perimeter boundary segment w}
        ∪ {upper and lower circuit perimeter segments}
3. Initialize tree with S
4. V ← V−{w}
5. repeat
6.     x ← minimal x-coordinate of segments in V
7.     T ← segments in V with x-coordinate x
8.     r ← (x, ymin)
9.     for v ∈ T (in line sweep order) do
10.        α ← point in tree with y-coordinate v_l
11.        β ← point in tree with y-coordinate v_h
12.        p ← β
13.        while number of segments intersected < k and pred(p) exists
                 and pred(p) is not below r do
14.            q ← pred(p)
15.            C ← C ∪ {rectangle defined by q and (x, p_y)}
16.            Update q in tree
17.            p ← q
18.            if p is a blocking point then exit while-loop
19.        end while
20.        p ← α
21.        while number of segments intersected < k and succ(p) exists do
22.            q ← succ(p)
23.            C ← C ∪ {rectangle defined by p and (x, q_y)}
24.            Update q in tree
25.            p ← q
26.            if p is a blocking point then exit while-loop
27.        end while
28.        r ← p
29.        if v is not a right vertical module boundary segment then
30.            p ← α
31.            while p ≠ β do
32.                q ← pred(p)
33.                C ← C ∪ {rectangle defined by q and (x, p_y)}
34.                p ← q
35.            end while
36.            Delete points between α and β from tree
37.        end if
38.        Update α and β in tree
39.    end for
40.    S ← segments in H with left x-coordinate x
41.    Update tree with S
42.    V ← V−T
43. until V=∅
```

Figure 5 - Algorithm *CCG* to Construct Channels and Channel Graph

Prior to sweeping the current group of vertical segments T with x-coordinate x, the algorithm processes those horizontal extension segments S with x as their left endpoint's x-coordinate (steps 2-3 and 40-41). By arranging H in line sweep order, S can be determined in $O(|S|)$ time. The processing of S requires that the left endpoint of the segments in S be inserted into a B^+—tree if it is not already there. The left endpoint is present already in the tree if its segment was constructed by extending a horizontal module boundary segment right. The points in the tree are ordered by their y-coordinate. Each point in the tree will have two labels. One label is either *blocking* or *passing*. A blocking point indicates that a vertical extension segment cannot extend beyond the y-coordinate of the point; a passing point indicates that a vertical extension segment can extend beyond the y-coordinate of the point. Except for the two endpoints introduced by the lower and upper circuit perimeter segments, all inserted points are initially labeled passing. The other label of a point is either *red* or *green*. A red point indicates that the point is the lower left corner of a channel whose right boundary is as yet undetermined. A green label indicates that the point is not a lower left corner of a channel. The red points thus serve as a way to identify or specify a particular channel. Each red point heads a linked-list that records its channel adjacencies. Any point in the tree that lies either on the upper circuit perimeter segment or on a lower module boundary segment cannot be the lower left corner of a channel. Hence all such points are labeled green. From the algorithms operation, if a point in the tree lies on an extension segment, on an upper module boundary segment, or on the lower circuit perimeter segment then the point is a lower left corner of a channel. Hence all such points are labeled red.

When considering the current vertical segment v from T there are three actions that must be attempted: extending v downward to form the right boundary of channels (steps 12-19); extending v upward to form the right boundary of channels (steps 20-27); and recognizing the channels whose right boundary is a subsegment of v (steps 30-36). The last action is performed if v is not a right module boundary segment. Prior to these three actions, two searches are made in B^+—tree to find the points α and β whose y-coordinates

are respectively v_u and v_l (steps 10-11). The points α and β are used to initialize the loop variables. The searches to find α and β can be done in $O(\log n)$ time. (One search of the tree suffices with rearrangement of the step 21 **while** statement after step 37 and by using the successor links).

Each point q considered in the body of the **while** loops of steps 13 and 31 and each point p considered in the body of the **while** loop of step 21 is a red point. Once the channel associated with the red point has been determined in the **while** loops of steps 13 and 21, q is updated (steps 16 and 24). The update of q resets its x-coordinate to x. Since the updated q lies on the same segment as before, it keeps its previous labels of passing or blocking and red or green. Each point q considered in the **while** loop of step 31 is not updated but, rather it is deleted (step 36). It is deleted since the channel identified with it has been determined and since the right boundary of the channel abuts a module boundary or the circuit perimeter.

The point r is used during the processing of T to prevent duplicate channels from occurring. Point r's y-coordinate is that of the highest segment that has been reached when extending one of T's vertical segments. As the segments in T are considered from bottom-to-top, r's use in the step 13 **while** loop ensures that no duplicate channels are issued when extending the next vertical boundary segment upward. Point r is initialized for the iteration as $(x, ymin)$, where x is the value of the x-coordinate of the segments in T and $ymin$ is the value of the y-coordinate for the lower circuit perimeter boundary segment (step 8). Point r is updated after processing the current vertical boundary segment v (step 28).

The points α and β are updated after extending the vertical boundary segment in both directions and after determining whether it forms the right boundary of a channel(s). Like the updates of q in steps 16 and 24, the x-coordinates of α and β are set to x. However, the updated α and β are labeled blocking if v is a left module boundary segment since

there must be horizontal boundary segments extending right from v's endpoints†. If v is not a left module boundary segment then the updated points are labeled passing. The updated α is labeled red and the updated β is labeled green since α and β now lie respectively on an upper and lower module boundary segments.

After considering the current vertical boundary segments T and processing the horizontal extension segments S, the **repeat** loop of step 5 is iterated so that the line sweep of the next group of vertical boundary segments can proceed. The processing ceases after all vertical boundary segments have been considered (step 43).

A vertical boundary segment v can be processed in $O(\log n + s)$ time, where s is the number of right channel boundaries that are determined with v. This run-time follows from previous discussion and the three remarks below.

> For each vertical segments there are 2 searches made in the B$^+$—tree each of which can be performed in $O(\log n)$ time.
>
> Each iteration of the **while** loops of steps 13, 21, and 31 determines the right boundary of a new channel.
>
> The points between α and β can be deleted in $O(\log n)$ time [AHO74].

Thus, V is processed in $O(n \log n + m)$ time.

An individual horizontal segment h in S is processed in $O(\log n)$ time. This follows as a search is performed to see if h is already in the tree. And if h is not in the tree, it is then inserted. Thus, the time spent processing H is $O(n \log n)$.

As the vertical segments are processed in $O(n \log n + m)$ time, and as the horizontal segments H are processed in $O(n \log n)$ time, algorithm *CCG* constructs the channels in $O(n \log n + m)$ time.

Construction of the graph can proceed simultaneously with construction of the channels by expanding the actions that are performed when inserting or updating points in the B$^+$—tree to include recording the channel adjacencies. We demonstrate below that expanding these actions does not alter algorithm *CCG*'s $O(n \log n + m)$ time complexity.

† There are no rectangles with sides of length 0.

Because all insertions introduce points in the tree that lie either on a right module boundary segment or on the left circuit perimeter segment, the region to the left of a newly inserted point is not part of the routing region. Thus, the channel associated with the newly inserted point has no horizontal adjacencies to its left. This allows horizontal adjacencies to be determined only during updates (steps 16, 24, and 38). During these update operations, the point in question has its x-coordinate reset to x. The updated point then represents the lower left corner of a new channel and so a horizontal adjacency does occur between the channel represented by the point's previous location and the channel represented by its current location. This adjacency can be recorded in the two lists in constant time.

The above discussion is demonstrated graphically in Figure 6. In the figure vertical boundary segment v is being extended downward. Point q is the predecessor of p in the tree with $p=\beta$. The channel C defined by q and (x, p_y) is adjacent to the channel C' that has s (updated value of q) as its lower left corner. If vertical segment (p,q) is a subsegment of a vertical boundary segment then there is no left adjacency from C. Otherwise there would be such an adjacency.

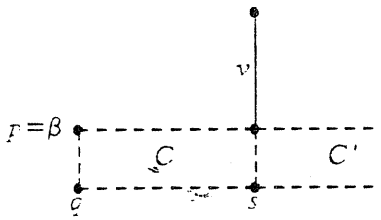

Figure 6 - Determining Horizontal Adjacencies

Vertical adjacencies are determined during both insertions (steps 3 and 41) and updates (steps 16, 24, and 38). Below we consider the case when a newly inserted endpoint p has both a predecessor and a successor in the tree. The cases where one or both of the predecessor and successor do not exist can be handled similarly.

Let the predecessor and successor of p be respectively s and t. The points p and s are readily confirmed to be red points. As such, they are both the lower left corner of a channel. Let C be the channel with lower left corner p. Let C' be the channel with lower left corner s. Channels C and C' are adjacent. The adjacency is recorded in the linked-lists of both p and s. This is demonstrated graphically in Figure 7. In Figure 7 p, s, and t have the same x-coordinate, however this is not always the case.

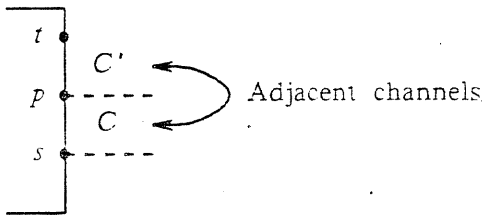

Figure 7 - Determining Vertical Adjacencies During an Insertion

As the left endpoints of the segments in S are inserted in line sweep order, the left endpoint q of the next extension segment in S (if one exists) may be inserted between t and p depending on its y-coordinate. If a point q is inserted between t and p then it is necessarily a red point and the channel identified with q is adjacent to channel C. (This adjacency is recorded when q is inserted in the tree). If t is a red point and if there is no point q that is to be inserted between t and p, then channel C is adjacent to the channel identified with t and is recorded in the linked-lists of p and t. Otherwise, there is no upper vertical adjacency to record between p and endpoints in the tree or in S. All of the above actions can be performed in constant time for the given inserted point p. Determining vertical adjacencies as a result of update are handled similarly and left to the reader.

The above discussion demonstrates that there is a constant amount of work during each update and insertion to record channel adjacencies. Thus, from previous remarks concerning its run-time, we conclude that algorithm *CCG* is $O(n \log n + m)$.

3. CONCLUSIONS

We have demonstrated an algorithm that rapidly constructs channel graphs. Through an input parameter k, the algorithm allows the user to control the number of channels in the channel graph. With k set to 0, the algorithm constructs a channel graph with $O(n)$ channels. With k set to n, the algorithm constructs a channel graph with $O(n^2)$ channels. For the former case the algorithm is as fast as previous algorithms. For the latter case the algorithm is faster than previous algorithms by a factor $O(n)$. With k set to some value between 0 and n, the algorithm constructs a channel graph with $O(k \cdot n)$ channels. We are unaware of any previous algorithm with this capability.

4. ACKNOWLEDGEMENTS

The author thanks D. S. Richards for several conversations.

5. REFERENCES

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

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

[BURS84] M. Burstein and R. Pelavin, Hierarchical Channel Router, *Computer-Aided Design*, **16**(4), July 1984, pp. 216-224.

[CHIB81] T. Chiba, N. Okuda, T. Kambe, I. Nishioka, T. Inufushi and S. Kimura, SHARPS: A Hierarchical Layout System for VLSI, *18th Design Automation Conference Proceedings*, Nashville, TN, 1981, pp. 820-827.

[COME79] D. Comer, The Ubiquitous B-tree, *Computing Surveys*, **11**(2), June 1979, pp. 121-137.

[HASH71] A. Hashimoto and J. Stevens, Wire Routing by Channel Design, *8th Design Automation Workshop Proceedings*, Atlantic City, NJ, 1971, pp. 155-169.

[OUST84] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott and G. S. Taylor, Magic: A VLSI Layout System, *21st Design Automation Conference Proceedings*, Albuquerque, NM, 1984, pp. 152-159.

[RIVE83] R. L. Rivest and C. M. Fiduccia, A 'Greedy' Channel Router, *Computer-Aided Design*, **15**(3), May 1983, pp. 135-140.

[SOUK80] J. Soukup and J. Royle, Cell Map Representation for Hierarchical Layout, *17th Design Automation Conference Proceedings*, Minneapolis, MN, 1980, pp. 591-594.

[ULLM84] J. D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Rockville, MD, 1984.

[YOSH82] T. Yoshimura and E. S. Kuh, Efficient Algorithms for Channel Routing, *IEEE Transactions on Computer-Aided Design*, **CAD-1**(1), January 1982, pp. 25-35.