Fast Channel Graph Construction

James P. Cohoon
Department of Computer Science
University of Virginia

Computer Science Report No. TR-85-16
Revised April 15, 1986

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 methodology that requires the routing region be divided into rectangular regions or *channels*. An algorithm is proposed that rapidly constructs the channels and determines the adjacencies between the channels in $O(n \log n + m)$ time, where n is the number of segments to represent the circuit's modules and m is the number of channels created. The algorithm determines the number of channels through a parameter k. With k set to some value between 0 and $2n$, the algorithm constructs a channel graph with $O(n + k\,n)$ channels. No previous algorithm has allowed users such control over the size of the channel graph. Traditional algorithms can be considered to have k preset to either 0 or $2n$. With k set to 0, our algorithm constructs its $O(n)$ channels as fast as previous algorithms. And with k set to $2n$, our algorithm constructs its $O(n^2)$ channels faster than previous algorithms by a factor $O(n)$.

Keywords: Channel routing, channel graph, circuit layout

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 *rectangular* 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.

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 [OUST84a] performs better on channels that are square-like rather than simply rectangular [OUST84b]. A heuristic for improving 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 is more square-like. A similar strategy for Rivest's PI circuit layout system [RIVE82] was suggested by Ullman [ULLM84]. Here we propose here an algorithm, *CCG*, for the construction of the channels and the channel graph. *CCG* directly considers modules that are *rectilinear* polygons rather than first constructing bounding rectangles for the modules. The run-time of *CCG* is $O(n \log n + m)$, where m is the number of channels that are created and n is now the number of segments used to represent the modules' perimeters. *CCG* is therefore faster for channel graphs with $O(n^2)$ channels and as fast for graphs with $O(n)$ channels. Also, *CCG* through an input k allows the user to control the number of channels that are created. Thus, by combining *CCG* with a heuristic that selectively combines channels, a user can create channel graphs that are better tailored for their individual router.

Throughout our discussion, we assume that the rectangular circuit perimeter and rectilinear modules are represented by their horizontal and vertical boundary segments. As abutting modules can be combined into a single unit, we assume without loss of generality that modules do not abut. However, modules are permitted to abut the circuit perimeter. Furthermore, we assume that all boundary segments have nonzero length and that they are appropriately labeled upper, lower, right, or left. For example, if a vertical module segment is labeled left, then the routing region lies to left of the segment and the module's interior lies to the right of the segment. An additional label is associated with each endpoint a of each boundary segment s. The label indicates the relative position to s of the other boundary segment which terminates at a (i.e. whether its above, below, right, or left of s).

2. CHANNEL GRAPH CONSTRUCTION

To decompose the routing region into a collection of channels, the 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

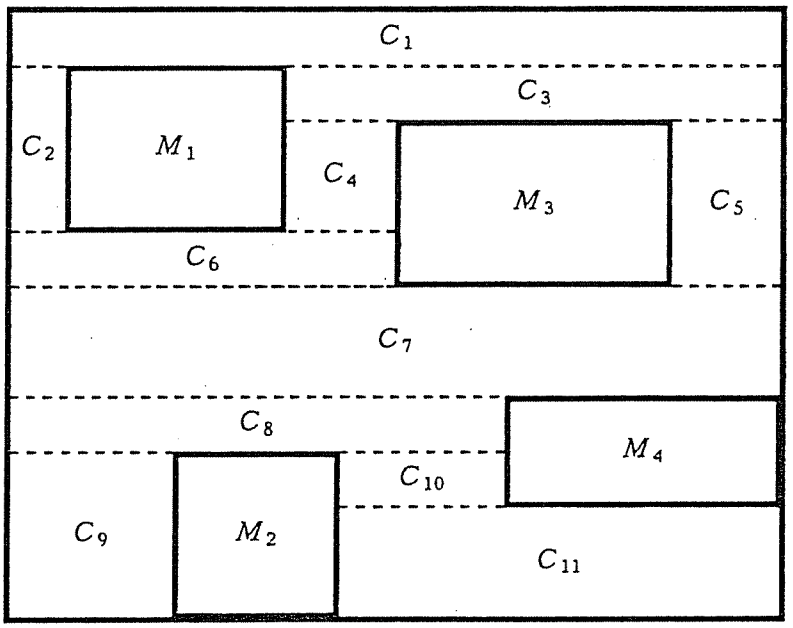

Figure 1 - Decomposing Routing Region into Channels with $k=0$

intersects another module segment or 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, either all horizontal or all vertical boundary segments are extended into the routing region until they intersect a boundary segment. To achieve the maximum number of channels, all horizontal and vertical boundary segments are extended into the routing region until they intersect a boundary segment. To achieve $O(kn)$ channels, all horizontal (vertical) boundary segments are extended into the routing region until they intersect a boundary segment and then all vertical (horizontal) boundary segments are extended until they intersect k extension 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 has two phases. The first phase constructs horizontal extension segments; the second phase constructs 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.

2.1. Horizontal Extension Segment Construction

An algorithm *HESC* is given in Figure 4 to construct the horizontal extensions segments. *HESC* determines the set of horizontal extensions segments H by performing a left-to-right line sweep [BENT79] of the vertical boundary segments V. 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 $O(n \log n)$, since $|V|$ 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).

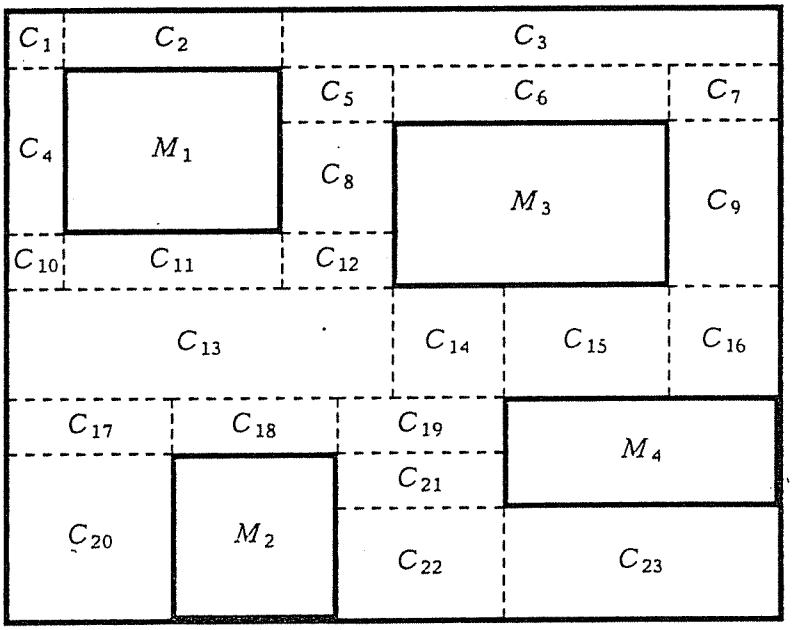

Figure 2 - Decomposing Routing Region into Channels with $k=1$

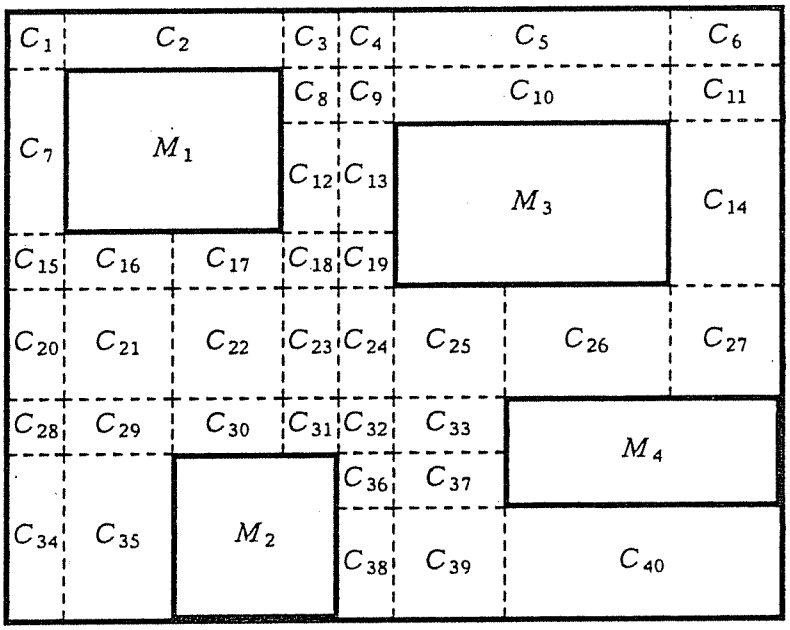

Figure 3 - Decomposing Routing Region into Channels with $k=n$

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 $v_l \leqslant b \leqslant v_u$.

The *interior* of a vertical segment consists of all its points but its endpoints.

A segment is *trivial* if it consists of a single point.

A horizontal extension segment is *leftward* if its right endpoint is a boundary segment endpoint. A horizontal extension segment is *rightward* if its left endpoint is a boundary segment endpoint and its right endpoint is not one.

The type of tree used by the algorithm is a B^+—tree [COME79]. Such a 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 a set of boundary segments currently involved in determining horizontal extension segments. The interior points of a boundary segment in the tree are possible endpoints of leftward extending horizontal extension segments. And the endpoints of a boundary segment in the tree can possibly originate rightward extending

---

```
1. H ← ∅
2. for each vertical segment v ∈ V (in line sweep order) do
3.    if v is a right module segment or a left perimeter segment then
4.       Insert v into tree
5.    else
6.       if (v_x,v_u) is black then
7.          a ← segment in tree covered by (v_x,v_u)
8.          H ← H ∪ {((a_x,v_u),(v_x,v_u))}
9.       end if
10.      if (v_x,v_l) is black then
11.         b ← segment in tree covered by (v_x,v_l)
12.         H ← H ∪ {((b_x,v_l),(v_x,v_l))}
13.      end if
14.      S ← segments covered by v's interior
15.      Q ← boundary endpoints of the segments in S
16.      for each boundary endpoint (x,y)∈S covered by v's interior do
17.         H ← H ∪ {((x,y),(v_x,y))}
18.      end for
19.      Delete from the tree and S those segments in S completely covered by v
20.      Truncate in the tree the remaining S segments to their uncovered v portion
21.   end if
22. end for
```

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

---

horizontal extension segments whose right endpoints lie on unswept boundary segments. If the endpoint of a boundary segment can possibly originate an extension segment, it is labeled *black* otherwise it is labeled *white*.

When a vertical boundary segment v is swept, the action taken depends on v's boundary type. If v is a right module or left perimeter segment then it is inserted in the tree since there may be horizontal extension segments abutting its right-hand side (step 4). An endpoint of v is marked white if the horizontal boundary segment adjacent to the endpoint is to the endpoint's right, otherwise the endpoint is marked black. For example in Figure 5, the upper endpoints of a and b and the lower endpoints of b and c are white. The other endpoints of a b, c, and d are black. If v is instead a left module segment or a right perimeter segment then the horizontal extension segments that abut v's left-hand side are determined. If the boundary endpoint is black then a leftward extending horizontal extension segment is computed for the point (steps 6-13). For example in Figure 5, a leftward horizontal extension is considered for the upper endpoints of e and f and the lower endpoints of f and g. The other endpoints of e, f, g, and h are white and do not originate leftward extending horizontal extension segments. The remaining horizontal extension segments computed from v are those horizontal extension segments, denoted T, that extend rightward into v's interior (steps 14-18). T is determined by computing the segments S in the tree that are at least partially covered by v. The left endpoints of the segments in T are exactly those black boundary endpoints of segments in S which are

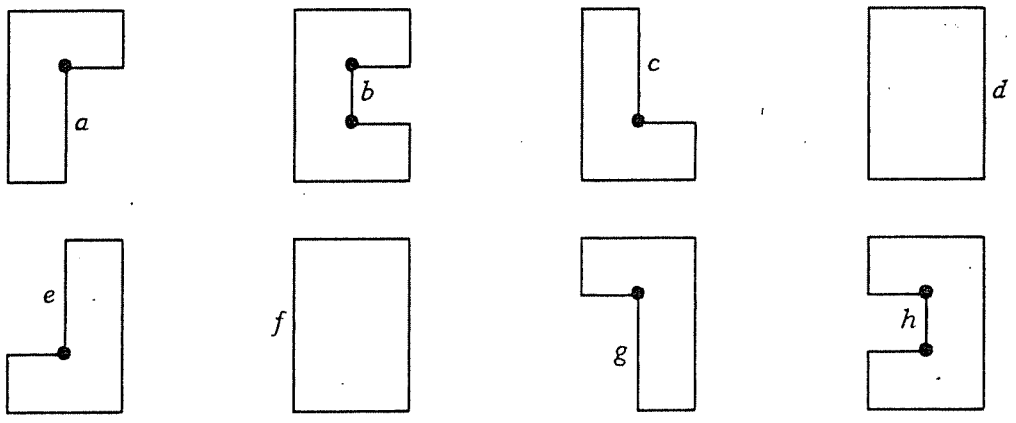

Figure 5 - Segments Whose Marked Endpoints Are White

themselves covered by v's interior. After determining T, the segments in S that are entirely covered by v are deleted from the tree (step 19). These segments are deleted since their associated rightward horizontal extension segments have been determined and their interiors cannot serve as the termination of any other leftward horizontal extension segments (v would lie in the way). After deleting these segments, up to two segments may remain in S. As only the portions of these remaining segments uncovered by v can possibly be used to determine other endpoints of horizontal extension segments, these remaining segments are truncated to their uncovered portions (step 20). This truncation may split a segment into two parts. If this is the case then the subsegments are both inserted into the tree. Note that in truncating a segment in S, the endpoint of an altered segment may not be a boundary endpoint. Such an endpoint does not originate a rightward extending horizontal extension segment and is therefore labeled white.

In $O(\log n)$ time, v can be inserted in the tree and the segments a and b covered by v's endpoints can be determined. The segments in S can be found in $O(|S|)$ time by following predecessor pointers from a. As there are at most two boundary endpoints for a segment in S, the total time to update H for a given v in step 13 is also $O(|S|)$. As the segments in S are contiguous values in the B^+—tree, they can all be deleted in $O(\log n)$ time [AHO74]. The truncation can be performed in either constant time or $O(\log n)$ time depending whether a segment is split in two. Thus an iteration of step 2 can be performed in $(\log n + |S|)$ time. As there are $O(n)$ iterations and as the sum of the individual $O(|S|)$'s is $O(n)$, the run-time of *HESC* is $O(n \log n)$.

2.2. Channel Determination

An algorithm, *CD*, that constructs the channels and channel graph, is given formally in Figure 6. This version of the algorithm assumes that no modules abut the circuit perimeter. A version without this restriction is subsequently discussed. Informally, *CD* operates in the following manner. A line sweep of the vertical boundary segments V is 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. (This grouping of segments in V is for pedagogic purposes).. 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. The right boundaries of channels are determined from the segments in T and the left boundaries of

```
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 T
7.    T ← segments in V with x-coordinate x
8.    r ← y-coordinate of lower perimeter segment
9.    for v ∈T (in line sweep order) do
10.      case
11.         v is left module segment:
               Create channels below and above v and terminate channels to v's left
12.         v is right module segment:
               Create channels below and above v
13.         v is right perimeter segment:
               Terminate channels to v's left
14.      end case
15.   end for
16.   S ← segments in H with left x-coordinate x
17.   Update tree with S
18.   V ← V − T
19. until V = ∅
```

Figure 6 - Algorithm *CD* to Construct Channels and Channel Graph

the channels are determined from the horizontal extension segments.

Before 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 17-18). 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 points in the tree are ordered by y-coordinate. The predecessor and successor of a point in a tree are determined respectively by functions *pred* and *succ*. In addition, each point in the tree has two labels. One label is either *passing* or *blocking*. A passing (blocking) label indicates that a vertical extension segment can (cannot) 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 label indicates that the point is the lower left corner of a channel whose right boundary has not yet been swept. A green label indicates that the point is not a

lower left corner of a channel. As the red points serve to identify the channels, associated with each red point will be an edge list of its channel's adjacencies.

The actions taken with the current vertical segment v depends on v's type (steps 10–14). If v is a left module segment then there are three possible actions: extend v downward to form the right boundary of channels; extend v upward to form the right boundary of channels; and recognize the channels whose right boundary is a subsegment of v (step 11). If instead v is a right module segment then only the first two actions must be attempted as there are no channels abutting its left-hand side (step 12). Finally, if v is a right perimeter segment then it is not possible to vertically extend its endpoints. Therefore, it suffices to determine which channels abut it (step 14).

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. Processing ceases after all vertical boundary segments have been considered (step 19).

A procedure, *PLMS*, to perform the step 11 processing of a left module segment is given in 7. Procedures to process the other segment types are analogous and are left to the reader. *PLMS* begins by searching the B^+—tree for the points α and β whose y-coordinates are respectively v_u and v_l (steps 1-2). 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 15 **while** loop after step 28 and by using the successor links).

After determining α and β, the principle actions of *PLMS* occur. First, the step 5 **while** loop attempts to construct a downward vertical extension. This may be done as long as the segment is in the routing region (p is passing), the number of intersected horizontal extensions segment i is less than k, there is another extension segment to intersect ($pred(p)$ exists), and the newly formed channel would not be a duplicate ($p_y \neq r$). Then, the step 15 **while** loop attempts to construct an upward vertical extension segment. Its iteration criteria is analogous to step 5's criteria. Finally, the step 24 **while** loop processes those channels that abut v's left-hand side.

Each point q considered in the **while** loops of steps 5 and 24 and each point p considered in the **while** loop of step 15 is a red point. Once the channel associated with the red point has been determined in the **while** loops of steps 5 and 15, the red point is

```
 1. α ← point in tree with y-coordinate v_u
 2. β ← point in tree with y-coordinate v_l
 3. p ← β
 4. i ← 0
 5. while p is passing and i < k and pred(p) exists and p_y ≠ r do
 6.     q ← pred(p)
 7.     C ← C ∪ {rectangle defined by q and (x, p_y)}
 8.     Update q in tree
 9.     p ← q
10.     i ← i + 1
11. end while
12. p ← α
13. s ← min(∞, lower y-coordinate of next T segment (if one exists))
14. i ← 0
15. while p is passing and i < k and succ(p) exists and p_y ≠ s do
16.     q ← succ(p)
17.     C ← C ∪ {rectangle defined by p and (x, q_y)}
18.     Update p in tree
19.     p ← q
20.     i ← i + 1
21. end while
22. r ← p
23. p ← α
24. while p ≠ β do
25.     q ← pred(p)
26.     C ← C ∪ {rectangle defined by q and (x, p_y)}
27.     p ← q
28. end while
29. Delete points between α and β from tree
30. Update α and β in tree
```

Figure 7 - Procedure *PLMS* to Process a Left Module Segment

updated (steps 8 and 18). The update of the red point changes its x-coordinate to x. As each newly shifted point (except α and β) lies on the same segment as its pre-shifted counterpart, its labels are not changed. As the updated α and β would not lie on the same segment as before, these points are separately handled (step 30). The updating of α and β is discussed below. Each point q considered in the step 24 **while** loop is deleted (step 29), since the channel identified with it has been determined and since the right boundary of the channel abuts a module.

After constructing the channels that abut the right-hand side of v and v's extension segments, α and β can be updated (step 30). Let h be the horizontal boundary segment attached to v's upper endpoint v_u. If h is to the right of v_u then the point α is shifted

over to v_u and is labeled red and blocking (Figure 8.a). If instead h is to the left of v_u then the update of α is in fact a deletion. α is deleted since its updated location at v_u would have the interior of a module to its above left and below left (Figure 8.b). As such, the new α would not introduce channels and further it would not be needed for blocking since both its predecessor and successor in the tree must also be blocking.

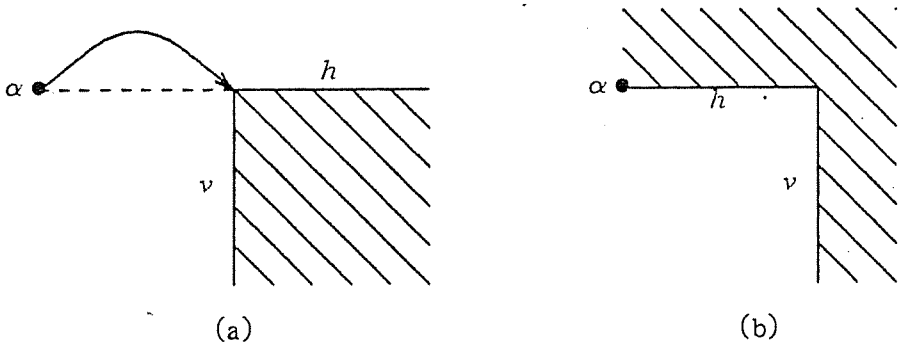

Figure 8 - Updating α and β

Similarly, if the horizontal boundary segment attached to v's lower endpoint v_l is to the right of v_l then β is shifted over to v_l and is labeled green and blocking, otherwise β is deleted.

Variable r keeps track of the y-coordinate of the highest point that has been reached while extending one of the current T's segments. As the segments in T are considered from bottom-to-top, r's use in the step 5 **while** loop ensures that no duplicate channels are issued when extending the next vertical boundary segment downward. r is re-initialized to the minimum y-coordinate value each time a new T is constructed (step 8 of CD) and is updated in *PLMS* after v has been extended upward (step 22). Variable s has a similar role (steps 13 and 15). It ensures that the current vertical segment is not extended above the next vertical segment in T. The channels to the left of that segment are processed when the segment is swept.

PLMS can process v 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 following remarks: for each vertical segment there are two 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 5, 15, and 24 determines the right boundary of a new channel; and the points

between α and β can be deleted in $O(\log n)$ time [AHO74]. Procedures to process a right module segment or a left perimeter segment can be performed in comparable time. Thus, V is processed in $O(n \log n + m)$ time. An individual horizontal segment h in S is processed by CD 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 CD constructs the channels in $O(n \log n + m)$ time.

2.3. Channel Adjacencies

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

As all insertions introduce points into the tree that lie either on a right module 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 and is thus not part of a channel. Similarly, when a point is deleted, the right-hand side of the channel associated with the point abuts a boundary segment. Thus, horizontal adjacencies can be determined strictly during updates. During an update operation, the point in question has its x-coordinate reset to x. The updated point now represents the lower left corner of a new channel and so a horizontal adjacency occurs 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 shown graphically in Figure 9. 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.

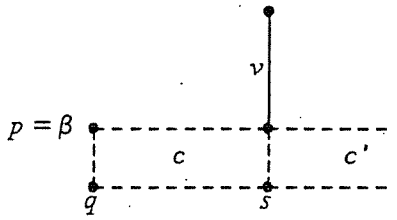

Figure 9 - Determining Horizontal Adjacencies

Vertical adjacencies are determined during insertions into the tree and during updates to C. Below we consider the case when a newly inserted endpoint p has both a predecessor s and a successor t in the tree. 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 associated with p. Let c' be the channel associated with s. Channels c and c' are adjacent. The adjacency is recorded in the linked-lists of both p and s in constant time. This is shown graphically in Figure 10. In Figure 10 p, s, and t have the same x-coordinate, however this is not always the case.

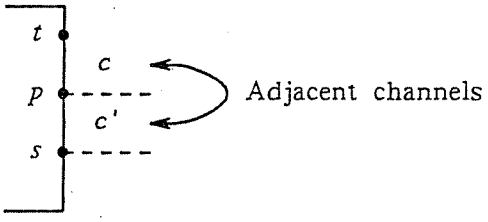

Figure 10 - Determining Vertical Adjacencies During an Insertion

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 associated 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 associated 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. The above actions can be performed in constant time for the given inserted point p. The two

remaining cases of determining vertical adjacencies—during an insertion where one or both of s and t does not exist and during an update to C—can be handled similarly and are left to the reader. The above discussion shows 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 *CD* is $O(n \log n + m)$.

As both *HESC* and *CD* and are $O(n \log n + m)$, *CCG* itself is $O(n \log n + m)$.

2.4. Modules Abutting Circuit Perimeter

Two minor modifications to *CD* are necessary to handle modules abutting the circuit perimeter. The two changes respectively handle when a module abuts the left perimeter and the upper or lower perimeter. No change is needed to permit modules to abut the right perimeter.

To allow modules to abut the left perimeter, some additional points, Q, must be inserted into the tree at step 3. The points in Q ensure that no extension segment is extended upward through a module that abuts the left perimeter. Q consists of the upper or lower endpoints of module segments that lie on the left circuit perimeter. By labeling the points in Q blocking, no extension segments can pass through the area reserved for a module abutting the left circuit perimeter. A point in Q is normally labeled red or green depending whether it is respectively an upper or lower endpoint. However, if a point in Q also corresponds to the upper left corner of the circuit perimeter then that point is labeled green as it cannot be the lower left corner of a channel.

To allow modules to abut the lower or upper circuit perimeter, the code for updating α and β must be modified. A check must be made during the update of α and β to see if either of the points lie on the perimeter. A point on the perimeter must be labeled blocking, since no channel lies outside the circuit perimeter.

3. SUMMARY

We have demonstrated an $O(n \log n + m)$ algorithm *CCG* that constructs channels when the modules are rectilinear polygons, where m is the number of channels created and n is the number of segments used to represent the module and circuit boundaries. This is an improvement over previous algorithms which constrained the modules to be rectangles. In addition, our algorithm controls the number of channels through a parameter k. The permissible values for k are between 0 and $2n$. With k set to 0, *CCG* constructs a

channel graph with $O(n)$ channels. With k set to $2n$, it constructs a channel graph with the maximum number of channels (i.e. $O(n^2)$ channels). And with k set to some value between 0 and $2n$, *CCG* constructs a channel graph with $O(n+kn)$ channels. We are unaware of any previous algorithm that gives the user such control over the number of channels in the channel graph. Traditional algorithms can be considered to have k preset to either 0 or $2n$. With k set to 0, our algorithm is as fast as previous algorithms. And with k set to $2n$, our algorithm is faster than previous algorithms by a factor $O(n)$.

4. ACKNOWLEDGEMENTS

This research was supported in part by NSF Grant DMC-8505354 and Virginia Center for Innovative Technology grant INF-86-001. As always, I thank Joanne Cohoon for her suggestions and comments.

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.

[OUST84a] 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.

[OUST84b] J. K. Ousterhout, Corner Stitching: A Data-Structure Technique for VLSI Layout Tools, *IEEE Transactions on Computer-Aided Design*, CAD-3(1), January 1984, pp. 87-100.

[RIVE82] R. Rivest, The 'PI' (Placement and Interconnect) System, *19th Design Automation Conference Proceedings*, Las Vegas, Nevada, 1982, pp. 475-481.

[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.