Algorithmica (1988) 3: 247–278

Algorithmica

A Linear Algorithm to Find a Rectangular Dual of a Planar Triangulated Graph[1]

Jayaram Bhasker[2,3] and Sartaj Sahni[2]

Abstract. We develop an $O(n)$ algorithm to construct a rectangular dual of an n-vertex planar triangulated graph.

Key Words. Rectangular dual, Planar triangulated graph, Algorithm, Complexity, Floor planning.

1. Introduction. A *rectangular dual* of an n-vertex graph, $G=(V, E)$, is comprised of n nonoverlapping rectangles with the following properties:

(a) Each vertex $i \in V$, corresponds to a distinct rectangle i in the rectangular dual.
(b) If (i, j) is an edge in E, then rectangles i and j are adjacent in the rectangular dual.

It is easily seen that some graphs do not have a rectangular dual and that for yet others, the rectangular dual is not unique. Further, whenever a graph has a rectangular dual, it has one whose outer boundary is rectangular. In this paper we are interested only in such duals.

The rectangular dual of a graph finds application in the floor planning of electronic chips and in architectural design [2], [3], [4], [8]. Each vertex of the graph G represents a circuit module and the edges represent module adjacencies. A rectangular dual provides a placement of the circuit modules that preserves the required adjacencies.

The problem of finding a rectangular dual has been studied in [1], [2], [4], [6], [7], and [8]. In all of these studies, the input graph is either assumed to be planar or is planarized by the addition of vertices during the early stages of the dual construction algorithm.

In this paper we assume that the graph, G, is a *properly triangulated planar* (PTP) *graph.* A PTP graph G, is a connected planar graph that satisfies the following properties:

P1. Every face (except the exterior) is a triangle (i.e., bounded by three edges).
P2. All internal vertices have degree ≥ 4.
P3. All cycles that are not faces have length ≥ 4.

[1] This research was supported in part by the National Science Foundation under Grant MCS-83-05567.
[2] University of Minnesota, Minneapolis, Minnesota, USA.
[3] Present address: Corporate Systems Development Division, Honeywell Inc., 1000 Boone Ave North, Golden Valley, MN 55427, USA.

Received August 1, 1986; revised November 10, 1986. Communicated by David S. Johnson.

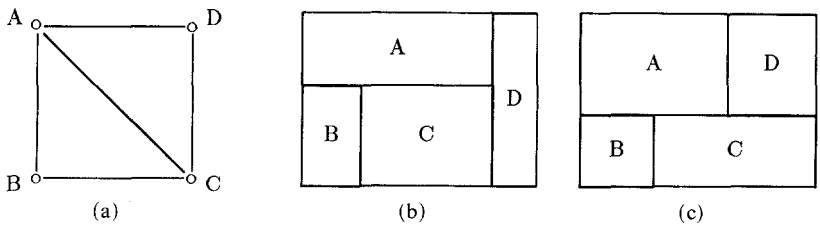
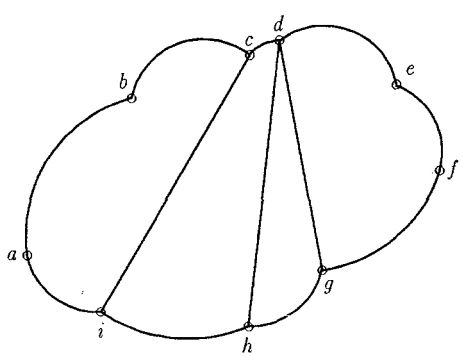


Fig. 1.1. (a) Planar triangulated graph. (b) and (c) Rectangular dual.

Figure 1.1 shows an example of a PTP graph, G, and two of its rectangular duals. In [1] it is shown that every planar graph that satisfies P1 and P3 also satisfies P2.

Kozminski and Kinnen [6], [7] have developed necessary and sufficient conditions under which a PTP graph has a rectangular dual. In order to state these conditions, we restate the following terminology from [6] and [7]:

DEFINITIONS [6], [7]. A *block* is a biconnected component. A *plane block* is a planar block. The *block neighborhood graph* (BNG) of a planar graph G, is a graph in which there is a distinct vertex for each biconnected component of G. There is an edge between two vertices iff the two biconnected components they represent have a vertex in common. The remaining definitions are with respect to an embedding of the planar graph. A *shortcut* in a plane block G, is an edge that is incident to two vertices on the outermost cycle (this is defined in a natural way with respect to a given embedding of the plane block) of G and that is not a part of this cycle (see Figure 1.2). A *corner implying path* in a plane block G, is a segment $v_1, v_2, \ldots, v_k$ of the outermost cycle of G with the property that (v_1, v_k) is a shortcut and that $v_2, \ldots, v_{k-1}$ are not the endpoints of any shortcut. A *critical corner implying path* in a biconnected component G_i of G is a corner implying path of G_i that does not contain cut vertices (articulation points) of G.



Outermost cycle: *abcdefghi*
Shortcuts: *ci*, *dh*, *dg*
Corner implying paths: *cbai*, *defg*

Fig. 1.2

THEOREM 1.1 [6], [7]. *A* PTP *graph, G, has a rectangular dual iff one of the following is true*:

(a) *G is biconnected and has no more than four corner implying paths.*

(b) *G has k, k* > 1, *biconnected components; the* BNG *of G is a path; the biconnected components that correspond to the ends of this path have at most two critical corner implying paths; and no other biconnected component contains a critical corner implying path.*

Kozminski and Kinnen [6], [7] have developed an $O(n^2)$ algorithm to construct the rectangular dual of an n-vertex PTP graph, G, that satisfies the necessary and sufficient conditions of Theorem 1.1. This algorithm simultaneously verifies that the given planar graph is properly triangulated. Bhasker and Sahni [1] have developed an $O(n)$ algorithm to determine if a given planar graph is properly triangulated. Since the conditions of Theorem 1.1 are testable in $O(n)$ time, their algorithm leads to an $O(n)$ algorithm to test the existence of a rectangular dual for a planar graph.

In this paper we extend our work reported in [1] to construct a planar dual (whenever one exists) in $O(n)$ time.

2. Algorithm Overview. The strategy adopted by our algorithm is best explained by examining a rectangular dual (Figure 2.1(a)). There are 10 rectangles in this figure. This figure may be partitioned into six columns, A–F. Each column has the property that it contains no vertical edge. Clearly, every rectangular dual can be so partitioned into a finite number of columns.

From this column partitioning of a rectangular dual, we can construct a directed graph called the *path digraph* (PDG). The PDG contains a distinguished vertex called the *HeadNode.* In addition, it contains one vertex for each rectangle in the dual. Since the dual of Figure 2.1(a) has 10 rectangles, its PDG will consist of 11 vertices. The directed edges of the PDG reflect the "*on top of*" relation defined by the dual. For example, rectangle 1 is on top of rectangle 2 which is on top of rectangle 3. This relation is completely specified by traversing the columns of the dual top to bottom. The *HeadNode* is, by definition, on top of all the rectangles.

Traversing the six columns of Figure 2.1(a) yields the PDG of Figure 2.1(b). Each column of the dual corresponds to a distinct path from the *HeadNode* of the PDG to a leaf vertex (i.e., a vertex with no outgoing edges). For instance, the path (*HeadNode* → 4 → 5 → 3) corresponds to column B while the path (*HeadNode* → 9 → 7 → 8) corresponds to column E.

A vertex i is a *parent* of another vertex j in the PDG iff $\langle i, j\rangle$ is a directed edge of the PDG. If i is a parent of j, then j is a *child* of i. In the PDG of Figure 2.1(b), 1 is a parent of 2 which in turn is a parent of 3; 7 has the parents 6 and 9; 8 has the parents 4, 7, and 10; 8 is a child of 4; 5 is a child of 4; etc. The children of any vertex of a PDG are ordered left to right. This ordering corresponds to the order in which the children appear in the dual. So, for example, 1 is to the left of 4 which is to the left of 6 which is to the left of 9 in the dual. Hence, as children of the *HeadNode*, they appear in the order 1, 4, 6, 9 (left to right).

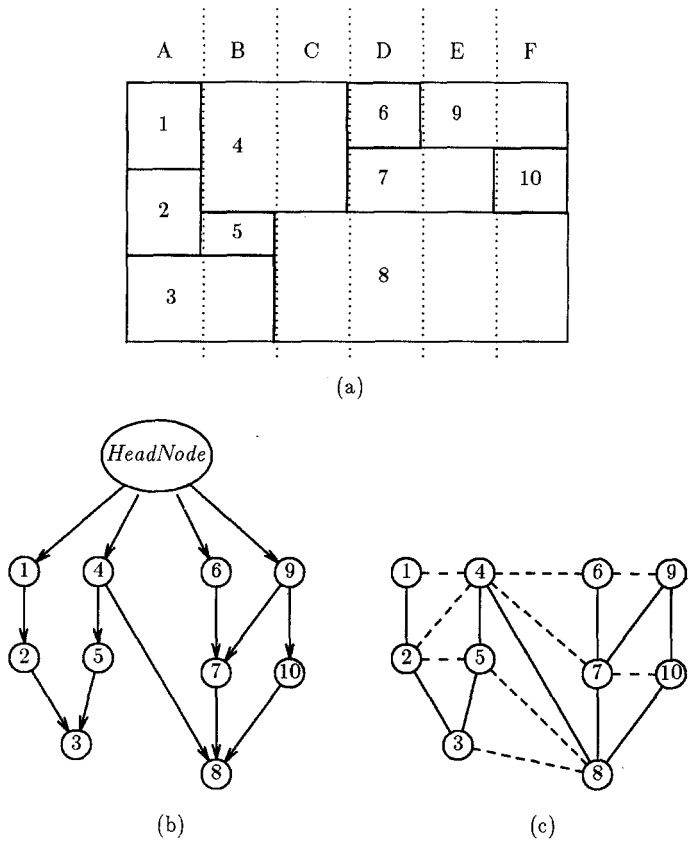


Fig. 2.1. Example of (a) a rectangular dual, (b) a PDG, and (c) a PTP graph.

Similarly, 5 is to the left of 8; so 5 is drawn to the left of 8 as children of 4. As a result of this ordering of the children of each vertex in the PDG, we can order the paths from the *HeadNode* to the leaves. When this is done, the first path in the PDG corresponds to the leftmost column of the dual, the second path to the next column, and so on.

Let i and j be two vertices in a PDG. We shall say that i is a *distant ancestor* of j iff the PDG contains a directed path from i to j that has length at least 2. For the example of Figure 2.1(b), 1, 4, and the *HeadNode* are the distant ancestors of vertex 3; 6, 9, and the *HeadNode* are the distant ancestors of vertex 8; the *HeadNode* is the only distant ancestor of vertices 2, 5, 7, and 10. No other vertex has a distant ancestor.

The following lemma is a direct consequence of the definition of a PDG.

LEMMA 2.1. *Let G be a* PDG *for some rectangular dual. Let i and j be two vertices in G. If i is a distant ancestor of j, then i is not a parent of j.*

Next, let us examine a planar triangulated graph for which Figure 2.1(a) is a rectangular dual. This is shown in Figure 2.1(c). The solid edges in this figure

represent edges contained in the PDG (though in the PDG these are directed). The broken edges represent edges not in the PDG.

The next lemma states two important properties of planar triangulated graphs and PDGs.

LEMMA 2.2. *If* (i, j) *is an edge in a planar triangulated graph, then*:

(a) i *is not a distant ancestor of* j *in the* PDG.
(b) j *is not a distant ancestor of* i *in the* PDG.

Our algorithm to obtain a rectangular dual begins with a planar triangulated graph that satisfies the necessary and sufficient conditions of [6] and [7], and first obtains a PDG that satisfies Lemma 2.2. From this PDG, a rectangular dual is obtained by traversing the *HeadNode* to leaf paths left to right.

We illustrate the basic mechanics of the process stated above on the planar embedding of Figure 2.1(c). To obtain a PDG, we first identify four (not necessarily distinct) vertices. These are called the northwest (NW), northeast (NE), southwest (SW), and southeast (SE) vertices of the planar graph. For the graph of Figure 2.1(c) we have $NW = 1$, $NE = 9$, $SW = 3$, and $SE = 8$.

Vertices 1, 4, 6, 9, 10, 8, 3, and 2 define the *outer boundary* of the graph. The outer boundary may be decomposed into four segments: top, right, bottom, and left. For the example of Figure 2.1(c), the top outer boundary is defined by the vertices 1, 4, 6, and 9; the right outer boundary by the vertices 9, 10, and 8; the bottom outer boundary by 3, and 8; and the left outer boundary by the vertices 1, 2, and 3. Figure 2.2 shows the boundary orientations that are used by us.

To obtain the PDG we begin with a *HeadNode* that has no children. The left outer boundary ($1 \rightarrow 2 \rightarrow 3$) of the PTP graph is traversed. This becomes the leftmost *HeadNode* to leaf path of the PDG. We will later see that by a proper choice of NW and SW, we can guarantee that the planar triangulated graph contains no edges that violate Lemma 2.2 with respect to the PDG so far constructed (Figure 2.3(a)). As the left outer boundary is traversed, these vertices are eliminated from the graph and the next left outer boundary identified. This elimination and boundary identification yields the graph of Figure 2.3(b).

The new left outer boundary cannot be included as a path in the PDG under construction because of the presence of the edge (4, 8). If the path (4, 5, 8) is

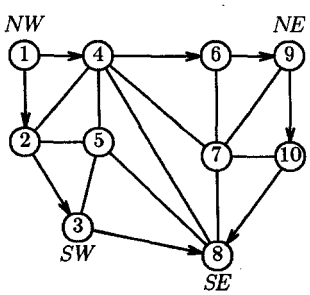


Fig. 2.2. Boundary orientations.

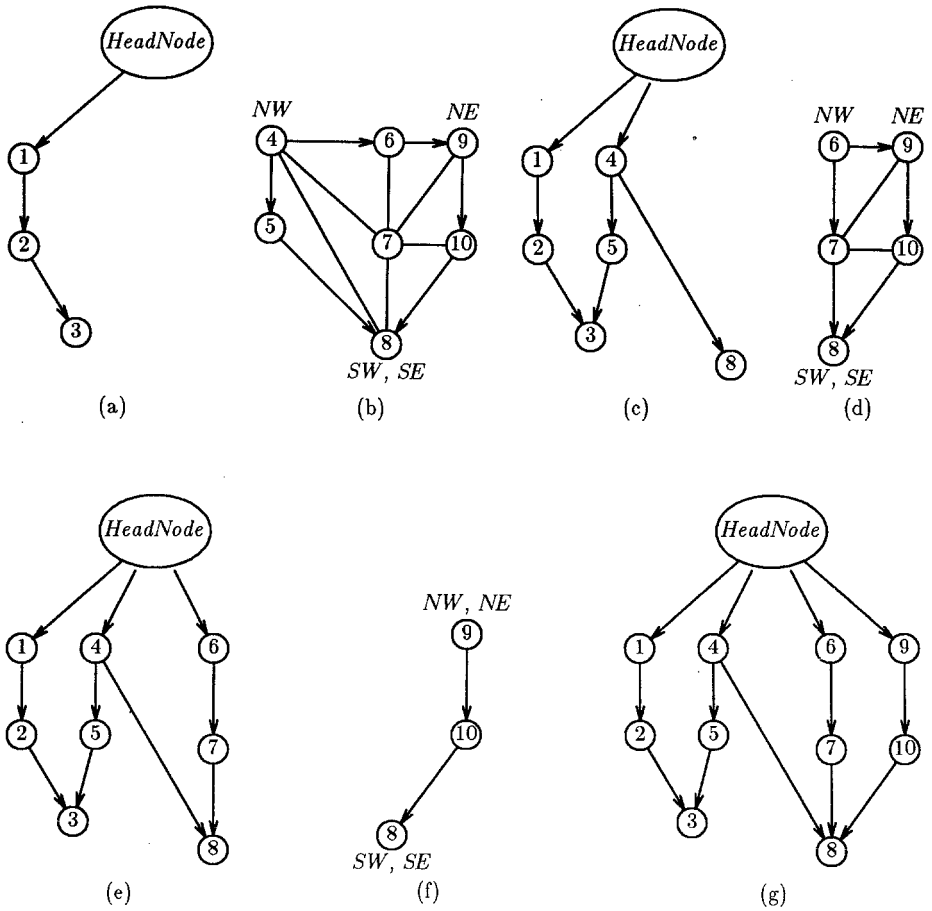


Fig. 2.3

added to the PDG, 4 will become a distant ancestor of 8 and the edge (4, 8) will violate Lemma 2.2. We can get around this difficulty by not using the edge (5, 8). Rather, the path is completed by using the edge (5, 3) in place of (5, 8). The offending edge (4, 8) is used to complete another path. This yields the PDG of Figure 2.3(c) and the graph of Figure 2.3(d). Note that in going from Figure 2.3(b) to 2.3(d), the *SW* vertex has not changed. This is because $SW = SE$ and all *HeadNode* to leaf paths must end at a bottom outer vertex.

The left outer boundary is now (6, 7, 8). Adding this to the PDG yields the PDG of Figure 2.3(e). The planar graph that remains is Figure 2.3(f). Traversing this last path yields the PDG of Figure 2.3(g). One may easily verify that the PDG of Figure 2.3(g) and the planar triangulated graph of Figure 2.1(c) satisfy Lemma 2.2. Further, observe that the PDGs of Figure 2.1(b) and 2.3(g) are not identical. This is no cause for concern as a planar triangulated graph can have many rectangular duals. The rectangular dual we shall obtain from the PDG of

Figure 2.3(g) is different from the one our algorithm would obtain from Figure 2.1(b).

The actual process of obtaining a PDG is far more complex than suggested by this simple example. This example does, however, illustrate the basic strategy. The complexities involved in obtaining the PDG are examined, in detail, in Section 4.

Obtaining a rectangular dual is now a relatively straightforward task. We traverse the leftmost *HeadNode* to leaf path and place rectangles of unit height in column A (Figure 2.4(a)). While the column is of unit width, it is possible for some of the rectangles in this column to be wider. The next *HeadNode* to leaf path is $4 \rightarrow 5 \rightarrow 3$. Since 4 is adjacent to the already placed rectangles 1 and 2 (see Figure 2.1(c)), we close off rectangles 1 and 2 and obtain the placement of Figure 2.4(b). This placement of rectangle 4 allows the next rectangle, 5, to be adjacent to rectangle 2. Rectangle 5 is to be adjacent to 2 and on top of the already placed rectangle 3. This leads to Figure 2.4(c). The next path is $4 \rightarrow 8$. Since 4 is already placed, it is extended into column C. Since 8 is adjacent to 3, 4, and 5, it is placed as in Figure 2.4(d) and rectangles 3 and 5 closed on their right. When the path $6 \rightarrow 7 \rightarrow 8$ is traversed, 6 is begun at the top. 6 is adjacent to the placed block 4. So, 4 is closed and 6 placed as in Figure 2.4(e). 7 is adjacent to the placed blocks 4 and 8 and so is placed as in Figure 2.4(e). At this time, the three blocks 6, 7, and 8 are open. Traversing the final path $9 \rightarrow 10 \rightarrow 8$ results in the placement of Figure 2.4(f) and the closing of all rectangles.

The details of the algorithm to obtain the dual from the PDG are provided in the next section.

3. From PDG to Rectangular Dual. The rectangular dual is obtained from the PDG by carrying out an ordered depth first traversal of the PDG. In this traversal, the children of each node are examined left to right. The basic principles underlying our algorithm for this task were described in Section 2. Here, we consider the details of our implementation.

The rectangular dual is a collection of nonoverlapping rectangles placed in a two-dimensional space. Any position in this space is defined by providing two coordinates: x and y. Figure 3.1 shows the origin of our coordinate system. Notice that y increases as we go down. Corresponding to each vertex v (other than the *HeadNode*) in the PDG, there will be a rectangle v in the dual. The position of this rectangle is uniquely characterized by providing the x positions of the two vertical edges and the y positions of the two horizontal edges (Figure 3.1).

With each vertex/rectangle v we associate the following values:

visit. This is a boolean field that is initially FALSE. It is set to TRUE the first time vertex v is reached during the depth first traversal. At this time, the rectangle for v is placed on the dual.

left. The x-coordinate of the left vertical edge of the rectangle v.

right. The x-coordinate of the right vertical edge of this rectangle.

top. The y-coordinate of the top horizontal edge.

bottom. The y-coordinate of the bottom horizontal edge.

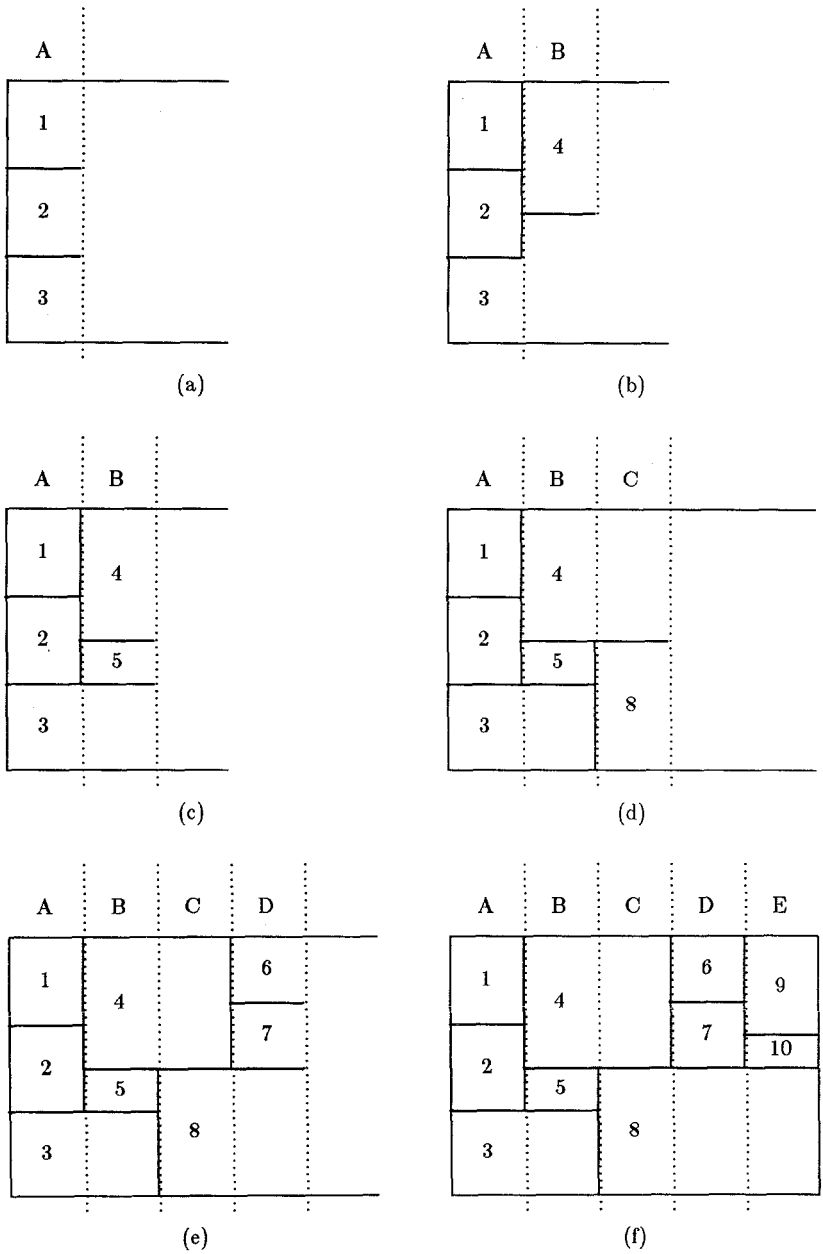


Fig. 2.4. Rectangular dual construction.

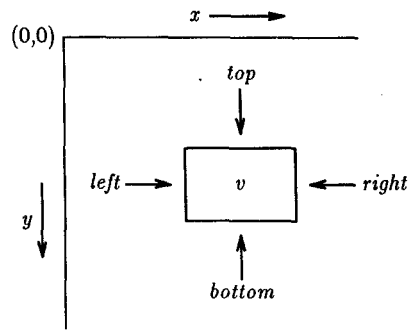


Fig. 3.1. Coordinate system for the dual.

We use the notation *a.b* to mean "the *b* value of *a*." For example, *v.visit* denotes the *visit* value of vertex *v*, etc. Notice that for each rectangle *v*, $v.bottom > v.top$ and $v.left < v.right$.

Our algorithm to obtain the rectangular dual from the PDG consists of two procedures: *ConstructDual* and *place*. *place* is a recursive procedure that does the actual placement of rectangles onto the two-dimensional space. This placement is carried out in a columnar fashion as suggested by the column decomposition of Figure 2.1(a). This procedure is invoked by *ConstructDual* after it has done some initialization. *x*, *y*, and *FirstPath* are variables that are global to procedure *place*. *x* records the *x*-coordinate of the left edge of the column that is currently being placed. All columns are assumed to be of unit width. So, for example, when we are placing the rectangles 1, 2, and 3 of column A of Figure 2.1(a), $x = 0$. When we are placing rectangles 4 and 5, $x = 1$ and when rectangles 6 and 7 are being placed, $x = 3$. *y* gives us the last *y*-coordinate in the current column to which a rectangle has been assigned (or equivalently, the next *y*-coordinate that is free). When procedure *place* is initially invoked, no rectangles have been placed. Hence, *x* and *y* are initialized to 0 in line 1 of *ConstructDual* (Figure 3.2). The variable *FirstPath* is boolean valued. Its value is true initially (line 2) and remains true as long as we are placing rectangles in the first column. This is the case as long as we are traversing the leftmost path of the PDG (e.g., *HeadNode* $\rightarrow 1 \rightarrow 2 \rightarrow 3$ in Figure 2.1(b)). Procedure *place* uses a boolean variable, *visit*, that is associated with each vertex *v* in the PDG. *v.visit* is FALSE initially and becomes TRUE when the vertex *v* is reached for the first time during the traversal of the PDG. The only other initialization that is done by *ConstructDual* is the rectangle for the *HeadNode*. This is defined to be of zero height and width and located at (0, 0) (line 4).

The invokation of procedure *place* from line 5 results in the placement of rectangles for all vertices in the PDG, except for the *HeadNode*. However, the position of the right vertical edges of the rightmost rectangles is not done. This is completed in lines 6–8. When the invokation of procedure *place* (line 5) is completed, *x* gives us the *x*-coordinate of the left edge of the last vertical column. The right *x*-coordinate of this column is $x + 1$.

```
line   PROCEDURE ConstructDual(HeadNode);
       (* use the path digraph with root HeadNode to obtain the rectangular dual *)
1        x = 0; y = 0; (* begin at coordinates (0, 0) *)
2        FirstPath = TRUE;
3        Set v.visit = FALSE for all vertices v in the PDG;
4        Set the left, right, top and bottom fields of HeadNode to 0;
5        place (HeadNode); (* procedure to place rectangles *)
6        FOR each vertex v on the rightmost digraph path, DO
         (* set right boundaries of rightmost rectangles *)
7          v.right = x + 1;
8        ENDFOR;
9      END ConstructDual;
```

Fig. 3.2

place (v) is a recursive procedure that results in the placement of all rectangles for all vertices in the PDG that are descendants of v (this includes vertex v, in case $v \neq HeadNode$) (Figure 3.3). On entry to *place*, x is the left boundary of the column we are to work in and y its top boundary. Furthermore, $v.visit$ = FALSE. So, except for the initial invokation when $v = HeadNode$, v always corresponds to a vertex whose rectangle has yet to be placed.

If the rectangle for v has not been placed (i.e., $v \neq HeadNode$), then this rectangle is placed in lines 1-18. The *left* and *top* values for this rectangle are clearly x and y, respectively. If v is on the leftmost path (i.e., *FirstPath* = TRUE), then its rectangle is arbitrarily given a height of 1. Hence, y is incremented by 1 (line 4) and $v.bottom = y$. *max_y* is a global variable used to record the overall height of the rectangular dual. It will become evident, later, that this is simply the number of vertices on the leftmost path in the PDG (excluding the *HeadNode*). In case v is not on the leftmost path of the PDG, then its bottom coordinate $v.bottom$ is determined by examining all the rectangles that are to be adjacent to it and on its left. These are obtained by examining the original adjacency list of v. This list contains four categories of vertices:

A. Vertices whose rectangles will begin in columns to the right of the one we are currently working on. These vertices have their *visit* value FALSE.
B. At most one vertex, u, whose rectangle is immediately above v in the current column. Note that by definition of a column partition, there are no vertical edges in a column. Hence, at most one vertex can have its rectangle immediately above v's rectangle in the current column. If v is the first rectangle in the column (i.e., v is a child of *HeadNode*), then no such u exists. If u exists, then $u.visit$ = TRUE and $u.bottom = v.top$.
C. At most one vertex u whose rectangle is immediately below v in the current column. This vertex u is a child of v in the PDG.
D. All other vertices. These vertices have *visit* = TRUE and occupy a contiguous segment of the previous column (i.e., the one with left boundary $x - 1$).

$v.bottom$ is determined by the vertices in C and D. If the vertex u on C has already been visited, then its $u.top$ is known and $v.bottom$ must equal $u.top$ (Figure 3.4(a)). If there is no vertex in C or if the C vertex has not been visited,

```
line   PROCEDURE place (v);
       (* place the rectangles corresponding to vertices in the digraph with root v *)
  1      IF (v ≠ HeadNode) THEN
  2        [ v.visit = TRUE; v.left = x; v.top = y;
  3        IF FirstPath THEN (* rectangle height = 1 *)
  4          [ y = y + 1; v.bottom = y; max_y = y ]
  5        ELSE
  6          [ v.bottom = y;
  7          FOR all vertices u on the adjacency list of v, DO
             (* u's are obtained from the original graph, not the path digraph *)
  8            IF (u.visit) AND (u.bottom ≠ v.top) THEN
  9              [(* u must be to the left of v *)
 10              u.right = x;
 11              IF (u.bottom > v.bottom) THEN
 12                [ y = (u.bottom + max{u.top, v.top})/2;
 13                v.bottom = y ]
 14              ENDIF ]
 15            ENDIF;
 16          ENDFOR ]
 17        ENDIF (* FirstPath *)]
 18     ENDIF; (* v ≠ HeadNode *)
 19     IF (v has no children) THEN
 20       [ v.bottom = max_y ]
 21     ELSE
 22       [ FirstChild = TRUE; (* local variable *)
 23       FOR all children w of v, DO
          (* children are obtained from the path digraph via an anticlockwise traversal *)
 24         CASE
 25           : w.visit: [ v.bottom = w.top ]
 26           : FirstChild: [ place (w); FirstChild = FALSE ]
 27           : ELSE: [ FirstPath = FALSE; x = x + 1; y = v.bottom; place (w)]
 28         ENDCASE;
 29       ENDFOR ]
 30     ENDIF; (* v has no children *)
 31   END place;
```

Fig. 3.3

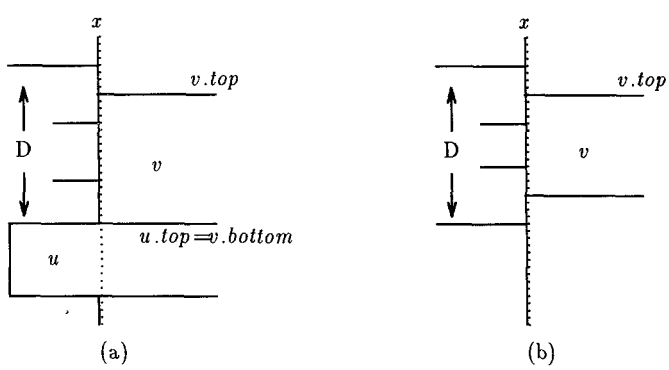


Fig. 3.4. Determining *v.bottom*.

then its *top* value is not known and *v.bottom* is determined by the vertices in D (Figure 3.4(b)).

In either case, for each vertex u in D, we can set $u.right = x$ as $u.right = v.left$. Note that ($u.visit$ AND $u.bottom \neq v.top$) iff $u \in$ D or ($u \in$ C and u satisfies Figure 3.4(a)). Lines 10–14 are written as though Figure 3.4(a) is not the case. Under this assumption, these lines are executed only for vertices u, $u \in$ D. Their right boundaries are set and $v.bottom$ is set so as to allow the child (if any) of v to share an adjacency with the lowermost rectangle of D (lines 12 and 13). In case Figure 3.4(a) is the case, then the only useful work done in lines 10–14 is the setting of $u.right$ for all u, $u \in$ D.

Lines 19–30 handle the placement of the children of v (if any), the case of Figure 3.4(a), and the case that C is empty. If C is empty, then v has no children. Hence, v is the bottommost rectangle in the current column and so should extend to max_y. This is handled in line 20. If v has children, then these are examined left to right (this is more clearly defined by following the edges in the PDG that leave vertex v in an anticlockwise order). Let w be a child of v. We have three cases:

1. w has already been visited. This is the case represented by Figure 3.4(a) with $w = u$. In this case, we need to set $v.bottom$ to $w.top$ (line 25).
2. w is the leftmost (or first) child of v. At this time, a recursive call is made to place w and all its descendants (line 26).
3. w has not been visited and is not the leftmost child. In this case, w is to be placed in the next column. So x is incremented and w and its descendants are placed by the recursive call of line 27. Note that since y is a global variable, its value could get changed as a result of an earlier call to *place* (w) (for example, from line 26 or even 27 itself). So, it is necessary to initialize y each time as in line 27.

The variables w and *FirstChild* are local variables of procedure *place*. The correctness of the placement procedure follows from the fact that the PDG is obtained from a planar triangulated graph and from our handling of the case of Figure 3.4(b). The complexity of the placement step is readily seen to be $O(n)$, where n is the number of vertices in the PDG (note that since the original graph is planar and connected, it contains $O(n)$ edges).

When our placement algorithm is used on the PDG of Figure 2.1(c), the rectangular dual of Figure 2.4 is obtained.

4. Obtaining the PDG

4.1. Input. The input to our algorithm is a connected planar triangulated graph described by its adjacency lists. Such a graph may or may not have a rectangular dual. The necessary and sufficient conditions of [6] and [7] that are described in Section 1 of this paper may be tested for in $O(n)$ time using the algorithm of

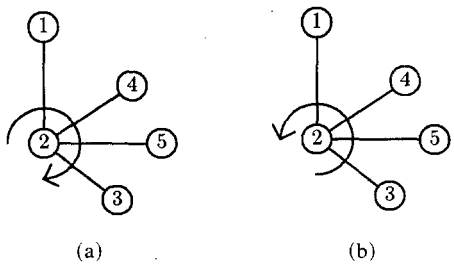


Fig. 4.1. (a) Clockwise ordering and (b) anticlockwise ordering.

[1]. This algorithm also determines whether the input graph is a PTP (properly triangulated planar) graph. If the input graph fails these tests, then no rectangular dual exists and we need proceed no further. So, assume that the tests are passed. Hence, a rectangular dual exists.

To find a PDG we can begin with the PTP graph drawing obtained by the algorithm of [1]. This drawing, quite naturally, satisfies the properties P1–P3 of a PTP graph as stated in Section 1. From this drawing, a new linked list representation of the graph is obtained. This is described below.

Every drawing of a graph imposes a natural cyclical ordering on the vertices that are adjacent to another vertex. For example, consider the drawing of Figure 2.1(c). Vertices 1, 3, 4, and 5 are adjacent to vertex 2. A *clockwise* ordering is obtained by following the incident edges in a clockwise direction (Figure 4.1(a)). This gives us the cycle (1, 4, 5, 3, 1). An *anticlockwise* ordering is obtained by following the incident edges in an anticlockwise direction (Figure 4.1(b)). This gives us the cycle (3, 5, 4, 1, 3). In either case, the starting point is irrelevant as the cycles (1, 4, 5, 3, 1), (4, 5, 3, 1, 4), (5, 3, 1, 4, 5), etc., are the same.

The starting point for our algorithm to obtain a PDG is the drawing obtained by the algorithm of [1]. This drawing is represented in the form of doubly linked circular adjacency lists (see [5] for a definition of a doubly linked circular list). When a circular adjacency list is traversed in one direction, the vertices appear in clockwise order. When it is traversed in the opposite direction, they appear in anticlockwise order. Our algorithm actually requires two copies of the circular adjacency list of each vertex. One copy gets modified as the algorithm progresses while the other is unchanged. The latter copy is referred to as the *original adjacency list* while the former is simply called the *adjacency list.*

4.2. Initialization and Variables Used. In addition to utilizing two sets of adjacency lists, our algorithm to construct a PDG uses several variables. These variables may be divided into the categories:

A. Variables associated with each vertex of the planar graph.
B. Variables associated with the PDG.
C. Variables associated with the planar graph (other than those in A).
D. Program variables.

We list, below, all the variables used by us. A description of the significance of each of these is also provided.

Category A: variables associated with each vertex of the planar graph

The variables are denoted using the notation *v.f.* This means variable *f* associated with vertex *v* (and read "*v* dot *f*").

A.1. *NotInPDG*... This is a boolean-valued variable that is initially TRUE for all vertices *v*. When a vertex *v* is added to the PDG, *v.NotInPDG* is set to FALSE.

A.2. *RightOuter*... This is a boolean valued variable that is TRUE for vertices on the right outer boundary of the subgraph being processed. For example, when the planar graph of Figure 2.2 is being processed, *RightOuter* is TRUE for vertices 8, 9, and 10, and FALSE for the remaining vertices. During the course of the algorithm, other vertices will have their *RightOuter* field set to TRUE (i.e., when they are on the right outer boundary of the subgraph being processed). However, the value of this variable never changes from TRUE to FALSE.

A.3. *TopOuter*... Similar to *RightOuter* except that it is true for vertices on the top outer boundary (e.g., vertices 1, 4, 6, 9 (initially) of Figure 2.2).

A.4. *BotOuter*... Similar to *TopOuter* except that it is true for vertices on the bottom outer boundary (e.g., vertices 3 and 8 (initially) of Figure 2.2).

A.5. *TopNext*... This is a link variable that links together the *TopOuter* vertices from left to right. Thus in Figure 2.2, the *TopNext* variable is used to record the chain $1 \rightarrow 4 \rightarrow 6 \rightarrow 9$ as the top outer boundary of the initial graph being processed. As our algorithm proceeds, the top outer boundary will change and the variable *TopNext* associated with other vertices will be initialized.

A.6. *BottomNext*... Similar to *TopNext* except that the bottom boundary is chained left to right. For the example graph of Figure 2.2, *BottomNext* is initially used to maintain the chain $3 \rightarrow 8$.

A.7. *LeftBoundary*... This variable may have one of the three values $\{0, 1, 2\}$. $v.LeftBoundary = 2$ iff v has never been a left boundary vertex. Otherwise, *v.LeftBoundary* may be 0 or 1. Recall from our informal discussion of Section 2 that the PDG is constructed by traversing the left boundary of the graph, deleting this left boundary, traversing the new left boundary, deleting this, and so on. The new left boundary is constructed while the present one is being traversed. To distinguish between vertices on the present left boundary and those on the next, two values 0 and 1 are used. If 0 (1) denotes vertices on the present left boundary, then 1 (0) denotes those on the next left boundary.

A.8. *RightOuterReached*... This variable is initially NIL for all vertices. During the course of the algorithm it will be used to record the fact that certain vertices are adjacent to certain right outer vertices. As will be seen later, this variable is introduced to ensure that the algorithm runs in $O(n)$ time.

A.9. *next*... This is used to chain together vertices on the left boundary. The last vertex, v, on the chain has $v.next = \text{NIL}$. For the graph of Figure 2.2, the initial left boundary chain $1 \rightarrow 2 \rightarrow 3$ is maintained using this variable.

Category B: variables associated with the PDG

B.1. *StartNode*... This is a vertex in the PDG. When a subgraph is being processed, all paths that are to be be added to the PDG begin at *StartNode*.

B.2. *EndNode*... All paths added to the PDG during the processing of a subgraph end at *EndNode*.

B.3. *VerticesRemaining*... This is a boolean-valued variable that is true as long as there are vertices yet to be added to the PDG.

B.4. *parent*... The next vertex added to the PDG is to be a child of this PDG vertex.

Category C: variables associated with the planar graph (other than those in A)

C.1. NW... Denotes the northwest vertex of the subgraph being processed. For the example of Figure 2.2, $NW = 1$ initially.

C.2. NE... The northeast vertex. Initially $NE = 9$ for the example of Figure 2.2.

C.3. SW... The southwest vertex. Initially $SW = 3$ for the example of Figure 2.2.

C.4. SE... The southeast vertex. Initially $SE = 8$ for the example of Figure 2.2.

C.5. *next_NW*... The vertex that will become the NW vertex after the present left boundary has been processed. *next_NW* $= 4$ when $NW = 1$ in the graph of Figure 2.2.

C.6. *next_SW*... The next southwest vertex. This is vertex 8 when $SW = 3$ in Figure 2.2.

Category D: program variables

D.1. *LeftBound*... This is a 0/1 valued variable used to tell us whether *LeftBoundary* $= 0$ or *LeftBoundary* $= 1$ signifies a vertex on the present left boundary.

D.2. p... Variable used to traverse the chain of left boundary vertices. Note that this chain begins at the vertex NW and ends at the vertex SW.

D.3. *pred_p*... The predecessor of p on the left boundary chain.

D.4. q... Variable used in the construction of the next left boundary chain. It denotes the last vertex added to this chain. Note that this chain will begin at *next_NW* and end at *next_SW*.

There are several other variables in Category D. The use of these is very localized and their significance will become apparent from the context in which they are used.

The initialization process requires us to do the following:

I1. Identify the NW, NE, SW, and SE vertices of the input PTP graph, G. This is done by identifying the corner (or critical corner) implying paths of G. There can be at most four such paths (Theorem 1.1). Let the number of such paths be k. Pick a vertex (any) from the interior of each of these k paths. Now pick an additional $4-k$ vertices from the outer boundary. Note that the outer boundary must be comprised of at least four vertices. If it has only three vertices, then G is either a triangle or has a triangle that is not a face. The former case can be handled as a special case and the latter case violates condition P1 (Section 1) of a PTP graph. The four vertices selected are labeled

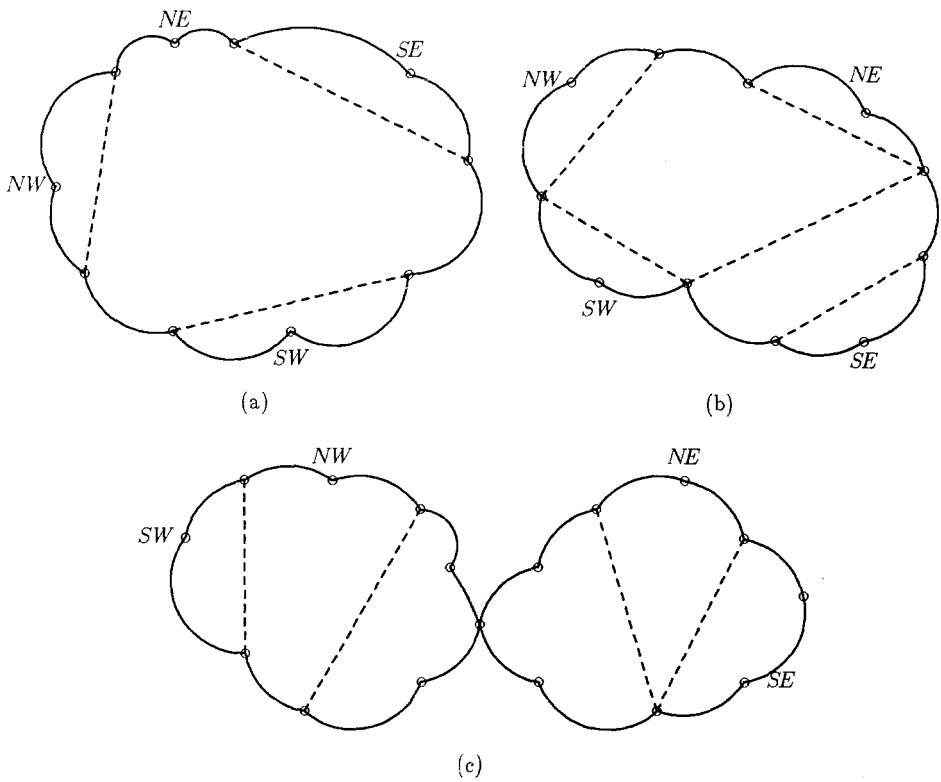


Fig. 4.2. Examples showing corner implying paths: (a) three corner, (b) four corner, and (c) two critical corner implying paths.

NW, *SW*, *SE*, and *NE* in such a manner that these are encountered in this (cyclic) order when the outer boundary is traversed in the anticlockwise direction. Some examples are shown in Figure 4.2. Broken edges denote corner (or critical corner) implying paths.

I2. Initialize the following boundary chains:
   (a) Left boundary from *NW* to *SW* using *next*.
   (b) Top boundary from *NW* to *NE* using *TopNext*.
   (c) Bottom boundary from *SW* to *SE* using *BottomNext*.

I3. Set *LeftBoundary* = 0 for all vertices on the left boundary chain and *LeftBoundary* = 2 for all other vertices. Set *TopOuter* and *BotOuter* = TRUE for all vertices on the chains (b) and (c) of I2, respectively.

I4. Set *RightOuter* = TRUE for all vertices on the outer boundary that are between *NE* and *SE* (inclusive).

I5. Set *RightOuterReached* = NIL for each vertex in *G*.

I6. Initialize the PDG to contain just the *HeadNode*.

Since the algorithm of [1] identifies the outer boundary of the obtained drawing, it is possible to carry out the initializations of I1–I6 in $O(n)$ time.

4.3. PDG Construction. While the basic idea introduced in Section 2 for the

construction of the PDG is quite simple, the actual construction is complicated by the need to handle several special cases that arise. We shall describe these special cases and how they are handled as we describe the working of the general PDG construction algorithm. This algorithm is comprised of the nine procedures: *set_next_NWSW*, *add_to_PDG*, *paths*, *process_chord*, *process_BottomLeft*, *process_hanging_component*, *process_TopLeft*, *process_RightOuter*, and *process_NotRightOuter*. After the initialization steps I1-I6 described in the previous section are complete, procedure *paths* is invoked. This in turn invokes procedures *set_next_NWSW*, *add_to_PDG*, *process_RightOuter*, and *process_NotRightOuter*. Procedures *process_RightOuter*, *process_NotRightOuter* in turn invoke the remaining four procedures. The details of each of these nine procedures are provided in the following subsections.

4.3.1. set_next_NWSW. This algorithm determines the values of *next_NW* and *next_SW*. As processing proceeds from one left boundary to the next, *NW* and *SW* move toward the right along the top and bottom outer chains. This continues until the end of the chain is reached. At this time, the variable *NW* or *SW* (or both) that has reached the end of its chain remains stationary.

4.3.2. add_to_PDG(p). This procedure is used to add the vertex *p* or just an edge to the PDG. If *p* is not already in the PDG, then it is introduced as a child of the vertex *parent*. Furthermore, if *p* is the last vertex on the left boundary, then it completes a newly added path in the PDG. Since all such paths must end at vertex *EndNode*, the edge $\langle p, EndNode\rangle$ is also added to the PDG.

If *p* is already in the PDG, there are two possibilities for *p*: $p = NW$ or $p = SW$. This is so, as it is only the *NW* or *SW* vertices of the subgraph currently being processed that can be on several left boundary paths (see the description of *set_next_NWSW*). If $p = NW$, the edge $\langle StartNode, p\rangle$ was added to the PDG during an earlier left boundary traversal. So nothing is to be done now. If $p = SW$, then the edge $\langle parent, p\rangle$ has to be added. Note that *parent* must be a newly added vertex as the current left boundary must contain at least one new vertex. Finally, note that since $p = SW$ is already in the PDG, the edge $\langle p, EndNode\rangle$ must have been added to the PDG at some earlier time.

4.3.3. paths. This is described formally by the Pascal-like code of Figure 4.3. It adds all the vertices in the subgraph bounded by the vertices *NW*, *NE*, *SW*, and *SE* to the PDG. This is done by successively traversing and deleting the left boundary of the subgraph. Each such traversal of a left boundary results in the addition of a path to the PDG. This path begins at *StartNode*, goes through the left boundary, and ends at *EndNode*. Procedure *paths* is a recursive procedure. When it is invoked initially, *NW*, *NE*, *SW*, and *SE* have the values specified in Section 4.2; $StartNode = HeadNode$; $EndNode = \text{NIL}$; and $LeftBound = 0$.

The repeat loop of lines 1-19 essentially traverses the graph by advancing from one left boundary to the next. This terminates when all vertices have been added to the PDG. The variable *parent* is set to *StartNode* in line 2 as all paths added to the PDG must begin at this node. *VerticesRemaining* is set to FALSE in line 3 as there may be no vertices left after we traverse the present left boundary. The

```
line   PROCEDURE paths (NW, NE, SW, SE, StartNode, EndNode, LeftBound);
       (* cover all vertices in the graph bounded by vertices NW, NE, SW, and SE by paths.
       These paths are added to the path digraph between StartNode and EndNode *)
 1       REPEAT (* obtain path cover *)
 2         parent = StartNode;
 3         VerticesRemaining = FALSE;
 4         set_next_NWSW;
 5         p = NW; (* vertex on present left boundary *)
 6         pred_p = NIL; (* predecessor of p on left boundary *)
 7         q = next_NW; (* vertex on next left boundary *)
 8         q.LeftBoundary = 1 - LeftBound;
 9         REPEAT (* travel down present left boundary *)
10           add_to_PDG (p); parent = p;
11           IF p.RightOuter
12           THEN process_RightOuter
13           ELSE process_NotRightOuter
14           ENDIF;
15           pred_p = p;
16           p = p.next; (* advance p *)
17         UNTIL (p = NIL);
18         NW = next_NW; SW = next_SW; LeftBound = 1 - LeftBound;
19       UNTIL (NOT VerticesRemaining);
20     END paths;
```

Fig. 4.3

values for *next_NW* and *next_SW* are determined by the procedure call of line 4. *p*, *pred_p*, and *q* are initialized in lines 5–8 to perform the traversal of the left boundary and to construct the next left boundary.

The repeat loop of lines 9–17 implements the traversal of the left boundary. *p* denotes the left boundary vertex presently being examined. This vertex is added to the PDG (line 10) and then processed in line 11. The nature of this processing depends on whether or not *p* is on the right boundary. Following this, *p* and *pred_p* are advanced to the next vertex on the left boundary (lines 15 and 16). The traversal of the left boundary ends when *p* falls off the left boundary ($p = \text{NIL}$).

During the processing of the left boundary, the graph is modified and the new *NW*, *SW*, *LeftBound* values are as indicated in line 18.

4.3.4. process_chord. The normal left boundary to left boundary advance of our algorithm is interrupted by the occurrence of special cases. There are five special cases that we need to handle. Four of these are handled by procedures *process_chord*, *process_BottomLeft*, *process_hanging_component*, and *process_TopLeft* and the fifth by the mechanism of the *RightOuterReached* variable that is associated with each vertex. The first special case arises when a *chord* is detected. A chord is an edge (u, v) that satisfies the following properties:

S1. Both u and v are on the present left boundary.
S2. (u, v) is not an edge of the left boundary.
S3. u is a predecessor of v on the left boundary chain and v is not in the PDG.

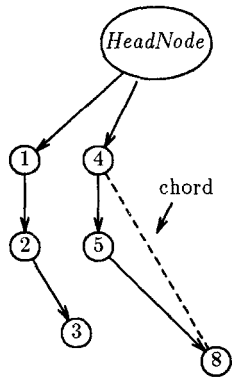


Fig. 4.4. Edge (4, 8) is a chord.

Suppose we start with the PTP graph of Figure 2.2. After the first left boundary has been processed, the PDG is as in Figure 2.3(a) and the PTP graph is as in Figure 2.3(b). The new left boundary is $4 \rightarrow 5 \rightarrow 8$. If this left boundary is added to the PDG to get the PDG shown in Figure 4.4, then because of the presence of the edge (4, 8) in the PTP graph, condition (a) of Lemma 2.2 is violated and no rectangular dual can be obtained from the PDG.

Hence, the presence of a chord requires us to deviate from our normal way of building the PDG. If (u, v) is a chord, then u must not be a distant ancestor of v in the PDG. Otherwise, no rectangular dual can be obtained from the PDG. The situation of Figure 4.4 is corrected by eliminating the edge $\langle 5, 8\rangle$ and adding the edge $\langle 5, 3\rangle$ to the PDG. Now, 4 is not a distant ancestor of 8 and the presence of the edge (4, 8) in the PTP graph does not create any difficulties with Lemma 2.2.

The general circumstance surrounding a chord is shown in Figure 4.5(a). Edge (ch_p, ch_r) denotes the chord and ch_p is a predecessor of ch_r on the left boundary chain. At the time a chord is detected by our algorithm, ch_p will be in the PDG while ch_r will not.

The planar region bounded by the cycle $ch\_p \rightarrow \cdots \rightarrow \cdots \rightarrow ch\_r \rightarrow ch\_p$ may contain additional vertices (such as a, b, c, etc., of Figure 4.5(a)). The edge (ch_p, ch_r) will be a chord with respect to all left boundaries in this region. So it is necessary to handle the entire region separately. This is done by first isolating this region from the rest of the remaining subgraph. Specifically, we traverse the adjacency list of vertex ch_r in the anticlockwise direction beginning at the vertex ch_p (step 1 of procedure *process_chord*, Figure 4.6). This traversal stops when the immediate predecessor, ch_SE, of ch_r in the left boundary chain is reached. Note that $ch\_SE \neq ch\_p$ as (ch_p, ch_r) is not a left boundary edge. Deleting the edges between ch_r and all the vertices encountered during this traversal (including ch_p and ch_SE), results in isolating the subgraph of Figure 4.5(b).

To process this subgraph, the NW, NE, SW, and SE vertices need to be identified. The first step in this identification is to traverse the original adjacency list of ch_SE clockwise beginning at ch_r. ch_end is the first vertex encountered. This vertex must be to the left of the current left boundary. To see this, observe

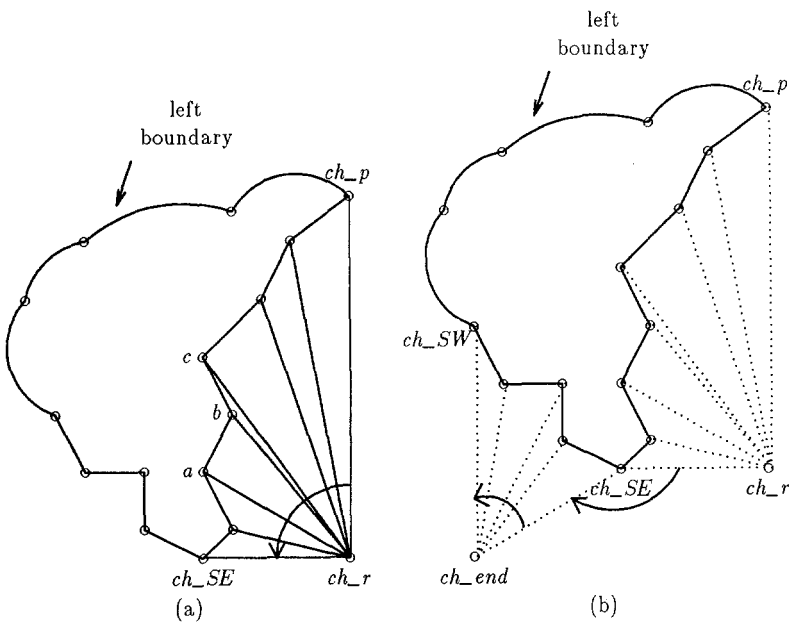


Fig. 4.5. Chord (*ch_p*, *ch_r*).

step PROCEDURE *process_chord* (*ch_p*, *ch_r*, *LeftBound*);
1 traverse the adjacency list for vertex *ch_r* anticlockwise beginning at vertex *ch_p* and ending when the vertex *ch_SE* such that *ch_SE.next* = *ch_r* (i.e., the predecessor of *ch_r* on the current left boundary) is reached
1.1 for all vertices (including *ch_p* and *ch_SE*) encountered during this traversal, set their *RightOuter* field to TRUE and delete edges between these vertices and *ch_r* from the graph;
2 set *ch_end* to be the vertex that is on the original adjacency list of *ch_SE* and clockwise from *ch_r*;
3 traverse the original adjacency list of *ch_end* anticlockwise beginning at *ch_SE*, until the last vertex *ch_SW* such that *ch_SW.NotInPDG* = TRUE and *ch_SW.LeftBoundary* = *LeftBound* is reached (this traversal essentially follows vertices on the current left boundary)
3.1 for vertices between *ch_SW* and *ch_SE* that are encountered during this traversal, set *BottomNext* field, set *BotOuter* to TRUE and *LeftBoundary* = 2;
4 {at this time, a subgraph bounded by *ch_p*, the newly marked *RightOuter* vertices, *ch_SE*, *ch_SW*, and vertices on the current left boundary between *ch_p* and *ch_SW* has been isolated.}
Set *ch_SW.next* to NIL;
5 *paths* (*ch_p*, *ch_p*, *ch_SW*, *ch_SE*, *ch_p*, *ch_end*, *LeftBound*);
END *process_chord*;

Fig. 4.6

that by the choice of the initial *NW*, *NE*, *SW*, and *SE*, the first left boundary cannot have a chord. So, the current left boundary cannot be the first left boundary. Hence, *ch_SE* has an adjacent vertex clockwise from *ch_r* and to its left.

Next, traverse the original adjacency list of *ch_end* anticlockwise beginning at the vertex *ch_SE*. This traversal terminates at the last vertex *ch_SW* that is both on the current left boundary and that is not in the PDG. Hence, this traversal essentially moves backward on the left boundary chain. Note that $ch\_SW \neq ch\_p$ as *ch_p* is already in the PDG (furthermore, it may not be adjacent to *ch_end*). However, *ch_SW* may be the same vertex as *ch_SE*.

The isolated subgraph of Figure 4.5(b) will be processed using procedure *paths* recursively. *ch_p* will be the northwest and northeast vertex, *ch_SW* the southwest, and *ch_SE* the southeast. Before the recursion can begin, the right outer boundary, bottom outer boundary, etc., need to be initialized. This is done in step 3.1 of *process_chord*. Finally, all the paths added to the PDG during the recursive call of step 5 should begin at *ch_p* and end at *ch_end*. This ensures that *ch_p* is not a distant ancestor of *ch_r* in the PDG.

4.3.5. process_BottomLeft. This is the second of the four special cases alluded to in Section 4.3.4. This arises when the current version of the PTP graph contains an edge (p, r) with the properties:

B1. p is on the current left boundary and $p \neq SW$.
B2. r is on the current bottom boundary and $r \neq SW$.

The situation is depicted in Figure 4.7. Because of the way in which we construct the next left boundary, it is necessary to handle the bottom left corner bounded by the edge (p, r) and the left and bottom boundaries in a special way.

The special processing of the bottom left corner is carried out by procedure *process_BottomLeft* (Figure 4.8). Step 1 isolates the affected region. This is done by traversing the adjacency list of vertex r anticlockwise beginning at the vertex p and terminating at the first vertex *bl_SE* that is on the bottom boundary.

We shall show later that the subgraph, H, being processed always satisfies the following condition:

C1. The subgraph H contains no edge (u, v) such that both u and v are on the bottom boundary and u and v are not adjacent on the bottom boundary chain.

As a result of C1, *bl_SE* and r must be adjacent on the bottom boundary chain and $bl\_SE.BottomNext = r$ (as in Figure 4.7). The vertices encountered during the anticlockwise traversal of r's adjacency list (including p and *bl_SE*) form the right outer boundary of the region to be specially handled. Deleting the edges that connect r to these vertices isolates the bottom left region from the remainder of the graph. The vertices in the isolated region get added to the PDG as a result of the recursive call of *paths* from step 2. Note that p is both the *NW* and *NE* vertex and that all paths added to the PDG will begin at p and end at the current *EndNode*. Observe that the bottom and top outer boundaries do not need to be initialized.

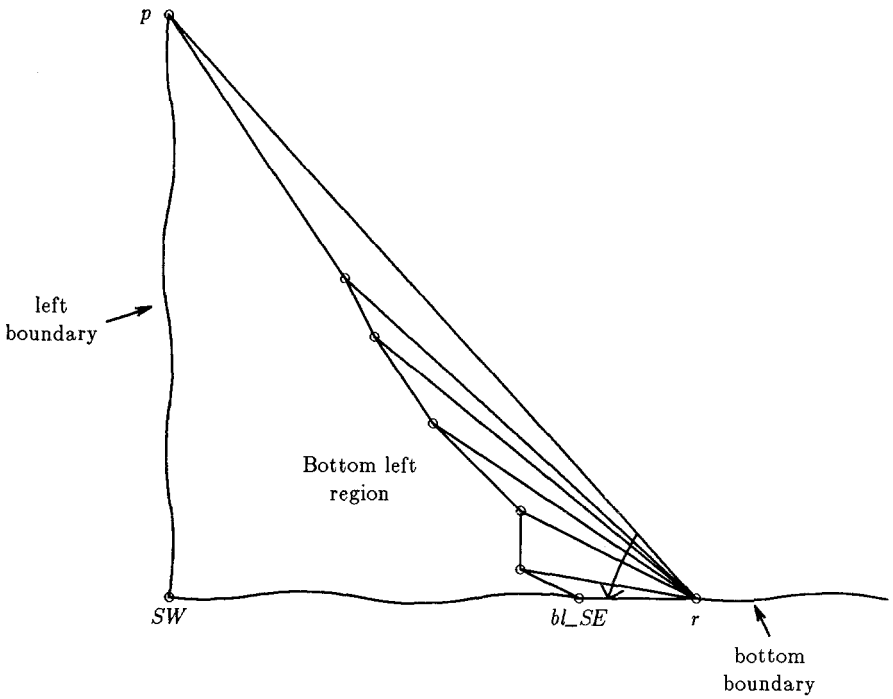


Fig. 4.7. Bottom left corner.

4.3.6. process_hanging_component. The third special case is also a consequence of the manner in which we construct the next left boundary. There will be times when the present and next left boundaries have the form given in Figure 4.9 and the region inbetween (marked hanging component) contains vertices that have yet to be added to the PDG. Before advancing to the next left boundary, it is necessary to handle the hanging component. This is done by Figure 4.10 (procedure *process_hanging_component*).

At the time *process_hanging_component* is invoked, vertices *hg_p* and *hg_r* are known. Furthermore, it is known that there is at least one vertex in the hanging component (as we shall see, (*hg_start*, *hg_p*) is not an edge of the current left

```
step   PROCEDURE process_BottomLeft;
1        traverse the adjacency list for vertex r anticlockwise beginning at vertex p and ending
         when the vertex bl_SE such that bl_SE.BotOuter = TRUE is reached
1.1        for all vertices (including p and bl_SE) encountered during this traversal, set their
           RightOuter field to TRUE and delete edges between these vertices and r from the
           graph;
         {at this time, a subgraph bounded by p, the newly marked RightOuter vertices, bl_SE,
         SW, and vertices on the current left boundary between p and SW has been isolated}
2        paths (p, p, SW, bl_SE, p, EndNode, LeftBound);
       END process_BottomLeft;
```

Fig. 4.8

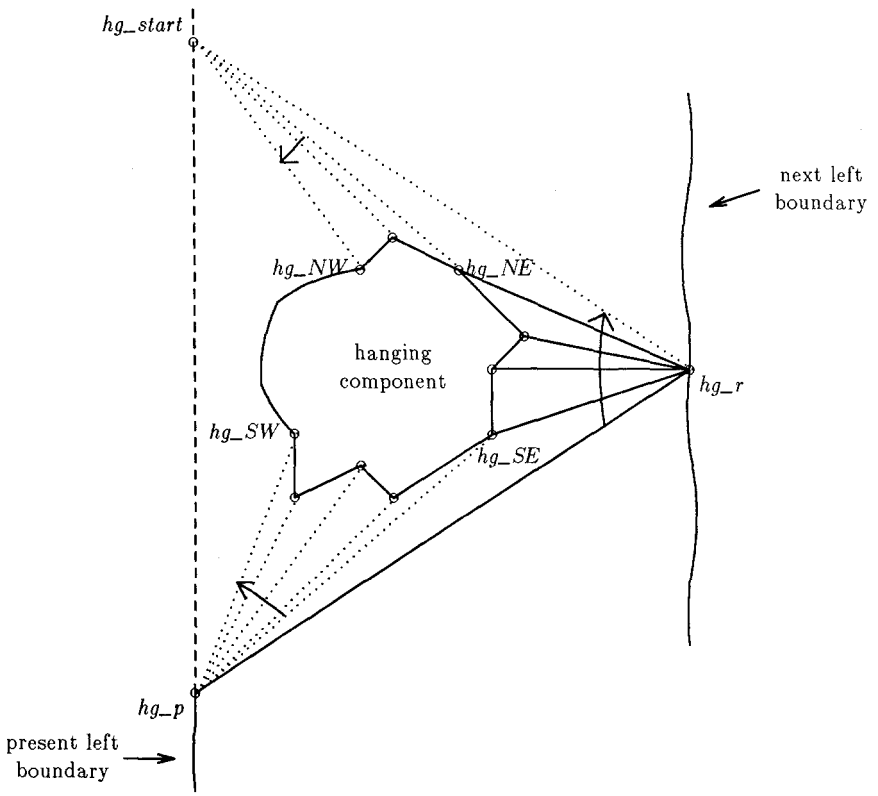


Fig. 4.9. Hanging component.

path. So, if the hanging component is empty, the graph has a face that is not a triangle).

As in the preceding two special cases, we first isolate the hanging component and then process it by using procedure *paths* recursively. The broken part of the present left boundary has been processed by the time *process_hanging_component* is invoked. Hence, all vertices on this segment and all edges incident to these vertices have been deleted from the modified copy of the PTP graph. To determine the vertex *hg_start*, it is necessary to traverse the original adjacency list of *hg_r* in clockwise order beginning at vertex *hg_p*. *hg_start* is the first vertex encountered (excluding *hg_p*) that is on the current left boundary (step 1 of Figure 4.10). Let *hg_SE* be the first vertex encountered after *hg_p* during this traversal. This vertex must be part of the hanging component. In particular, $hg\_SE \neq hg\_start$. This is so as the graph is a PTP graph and the hanging component is not empty.

This traversal of the adjacency list of *hg_r* also enables us to identify the vertex *hg_NE* (which may be the same as *hg_SE*) and label the right outer boundary of the hanging component (steps 1.1 and 2). The vertex *hg_NW* is obtained by traversing the original adjacency list of vertex *hg_start* clockwise beginning at the vertex *hg_NE*. During the processing of the broken segment (*hg_start* → *hg_p*) of the current left boundary, all vertices of the hanging component that are

```
step  PROCEDURE process_hanging_component (hg_p, hg_r, LeftBound);
1       traverse the original adjacency list of hg_r clockwise beginning at the vertex hg_p
        and ending when a vertex hg_start with hg_start.LeftBoundary = LeftBound is
        reached. Let hg_SE be the first vertex encountered after vertex hg_p
1.1       for all vertices (excluding hg_p and hg_start) encountered during this traversal,
          set their RightOuter field to TRUE and delete edges between them and vertex hg_r;
2       let hg_NE be the vertex encountered just before vertex hg_start is reached in the
        above traversal;
3       traverse the original adjacency list of hg_start clockwise starting at vertex hg_NE
        until a last vertex hg_NW such that hg_NW.LeftBoundary = 1 - LeftBound and
        hg_NW.NotInPDG = TRUE is reached (this traversal essentially follows vertices on
        the next left boundary)
3.1       for all vertices from hg_NE to hg_NW that are encountered in this traversal, set
          TopNext field, set TopOuter to TRUE and LeftBoundary = 2 (i.e., as new vertices);
4       traverse the original adjacency list of hg_p anticlockwise starting at vertex hg_SE
        until a last vertex hg_SW such that hg_SW.LeftBoundary = 1 - LeftBound and
        hg_SW.NotInPDG = TRUE is reached (this traversal also follows vertices on the
        next left boundary)
4.1       for all vertices from hg_SE to hg_SW that are encountered in this traversal, set
          BottomNext field, set BotOuter to TRUE and LeftBoundary = 2;
5       {the subgraph bounded by hg_NW, hg_NE, hg_SE, hg_SW has now been isolated
        from the rest of the graph.}
        Set hg_SW.next to NIL;
6       paths (hg_NW, hg_NE, hg_SW, hg_SE, hg_start, hg_p, 1 - LeftBound);
      END process_hanging_component;
```

Fig. 4.10

adjacent to vertices between *hg_start* and *hg_p* are labeled as candidates for the next left boundary. *hg_NW* is the last vertex encountered in the aforementioned traversal of the original adjacency list of *hg_start* that has been labeled in this way (step 3). Step 3.1 initializes the top boundary of the hanging component.

Step 4 identifies the vertex *hg_SW* and step 4.1 initializes the bottom outer boundary of the hanging component. To isolate the hanging component, it is necessary to delete the edges between its right outer boundary and *hg_r*. This is done in step 1.1. All other edges (i.e., those incident to the broken segment of the current left path) were deleted during the processing of this segment.

The left boundary chain was constructed while the broken segment of the current left path was processed. All that is required is to put in the end of chain terminator (NIL). This is done in step 5.

All paths added to the PDG because of the recursive invokation of step 6 begin at *hg_start* and end at *hg_p*.

4.3.7. process_TopLeft. The last algorithm that handles a special case is procedure *process_TopLeft* (Figure 4.11). This handles the situation when the modified graph being processed currently has an edge (p, r) such that:

TL1. p is on the current left boundary.
TL2. r is on the current top boundary and $r \neq NW$.
TL3. (p, r) is not a top boundary edge.

```
step  PROCEDURE process_TopLeft;
1       traverse the adjacency list of r clockwise beginning at the vertex p and ending when
        a vertex tl_NE with tl_NE.TopOuter = TRUE is reached. Let tl_SE be the first vertex
        encountered after vertex p
1.1       for all vertices (excluding p) encountered during the traversal, set their RightOuter
          field to TRUE and delete edges between them and vertex r;
2       traverse the adjacency list of p anticlockwise starting at vertex tl_SE until a last
        vertex tl_SW such that tl_SW.LeftBoundary = 1 - LeftBound and tl_SW.NotInPDG =
        TRUE (this traversal essentially follows vertices on the next left boundary)
2.1.      for all vertices from tl_SE to tl_SW that are encountered in this traversal, set
          BottomNext field, set BorOuter to TRUE and LeftBoundary = 2;
3       {the subgraph bounded by next_NW, tl_NE, tl_SE, tl_SW has now been isolated
        from the rest of the graph.}
        Set tl_SW.next to NIL;
4       paths (next_NW, tl_NE, tl_SW, tl_SE, StartNode, p, 1 - LeftBound);
      END process_TopLeft;
```

Fig. 4.11

The situation created by such an edge is shown in Figure 4.12. By the time the edge (p, r) is detected by our algorithm, the broken segment of the current left boundary has been processed and deleted from the graph. Further, the vertices from *next_NW* to *tl_SW* to *tl_SE* to r on the outer boundary of the top left region have been chained together as part of the next left boundary. When the edge (p, r) is detected, we realize that this next left boundary is incorrect and process the top left region recursively.

To process the top left region, this region is first isolated from the working PTP graph and the vertices *tl_SW*, *tl_SE*, and *tl_NE* identified. To begin with, we may assume that $r \neq next\_NW$. (If $r = next\_NW$, then the conditions for a hanging component are satisfied with $p = hg\_p$ and $r = hg\_r$. So, this case may be handled by *process_hanging_component.*) The code of Figure 4.11 makes this assumption. *tl_NE* and *tl_SE* are obtained as in step 1. *tl_NE* obtained in this way must be adjacent to r on the top boundary because of the following condition that the subgraph H being processed always satisfies (we shall prove this later):

C2. The subgraph H contains no edge (u, v) such that both u and v are on the top boundary and u and v are not adjacent on the top boundary chain.

Step 1.1 isolates the top left region; step 2 identifies the vertex *tl_SW*; and step 2.1 sets the bottom boundary of the top left region. The recursive call of step 4 results in all vertices of the top left region being added to the PDG by the inclusion of paths that begin at the present *StartNode* and end at the node p which is already in the PDG.

4.3.8. process_RightOuter. With the discussion of the special procedures complete, we can resume our discussion of procedure *paths*. As pointed out in Section 4.3.3, the processing of the current left boundary is done one vertex at a time, top to bottom. The processing associated with any vertex p depends on whether or not it is a right outer vertex. In this section we consider the case when p is a

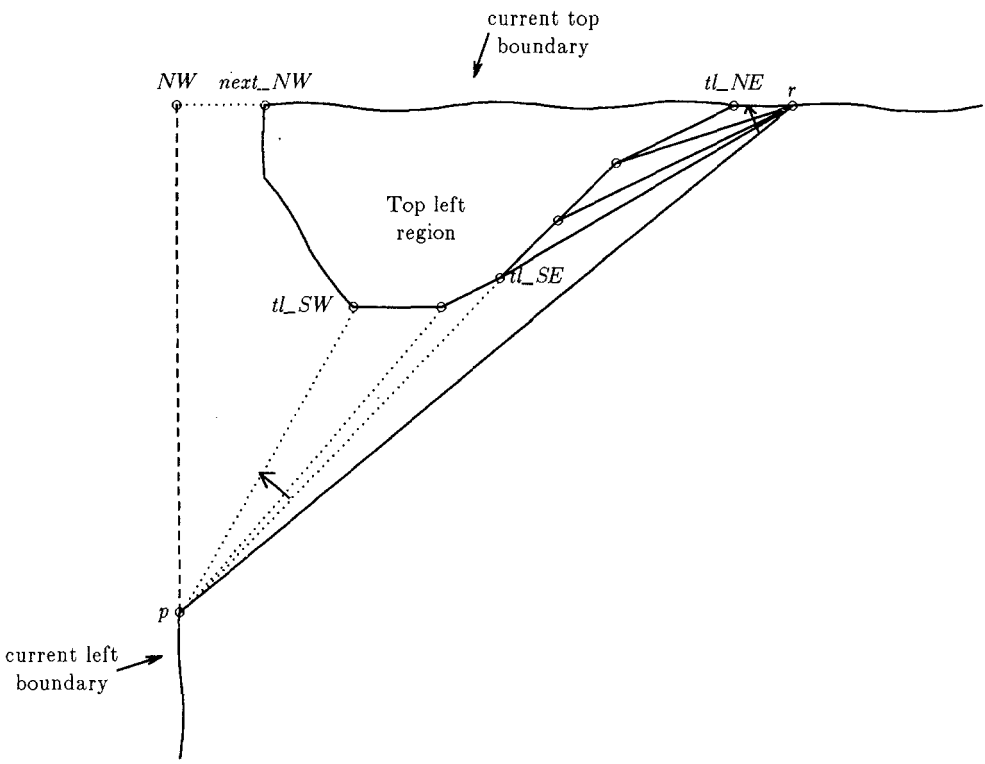


Fig. 4.12. Top left region.

right outer vertex. This is handled by procedure *process_RightOuter* (Figure 4.13).

This procedure performs the following functions:

Task 1. In case the graph has a nonempty region above p, this region is processed (Figure 4.14).

Task 2. If p is reached for the first time and there is an edge (p, t) such that t is on the left boundary and t is not adjacent to p on the left boundary chain, the region bounded by the left boundary and the edge (p, t) is processed (Figure 4.15).

At the time *process_RightOuter* is invoked, all predecessors of p on the left boundary chain have been processed. This means that these vertices and edges incident to them have been deleted from the working version of the PTP graph. If $pred\_p = \text{NIL}$, then $p = NW = NE$ and the region above p is empty. If $pred\_p$ is a right outer vertex, then the region (if any) above $pred\_p$ was processed when *process_RightOuter* was invoked with $pred\_p$ as the p vertex. If $pred\_p$ is not a right outer vertex, then there is at least one vertex on the right of the left boundary that is above p. Hence, the conditional of line 1 correctly identifies the conditions under which the region above p is not empty.

The broken line of Figure 4.14 signifies the segment of the left boundary that

```
line  PROCEDURE process_RightOuter;
 1      IF ((pred_p ≠ NIL) AND (NOT pred_p.RightOuter)) THEN
 2        [ q.next = p; p.LeftBoundary = 1 − LeftBound;
 3        m = p.next; p.next = NIL; (* save p.next in m *)
 4        paths (next_NW, NE, p, p, StartNode, NIL, 1 − LeftBound);
 5        p.next = m (* reset p.next *)]
 6      ENDIF;
 7      IF (p = SW) THEN RETURN;
        (* finished traversing current path and graph *)
 8      Set NW, next_NW, NE and q to p; p.LeftBoundary = 1 − LeftBound;
 9      StartNode = p;
10      IF (p is being visited for the first time) THEN
11        [ Starting from the vertex t that is adjacent to p and such that t.RightOuter = TRUE,
          traverse the adjacency list of p clockwise and do:
12          IF (t.LeftBoundary = LeftBound AND t ≠ p.next) THEN
13            [ IF (t.NotInPDG) THEN
                [ (* vertex not added to digraph *)
14              process_chord (p, t, LeftBound)]
15            ELSE
                [(* t must be SW *)
16              process_hanging_component (t, p, 1 − LeftBound)
                (* treat previous left boundary as current *)]
17            ENDIF;
18            p.next = t ]
19          ELSE
20            [ set t.RightOuterReached = p ]
21          ENDIF;
22        END traversal]
23      ENDIF;
24      deleted edge (p, p.next) from graph;
25    END process_RightOuter;
```

Fig. 4.13

has been processed prior to this invokation of *process_RightOuter*. When this segment was processed, the next left boundary chain from *next_NW* to q was created. To process the region above p, we merely complete the left boundary chain of this region as in lines 2 and 3 and invoke *paths* as in line 4.

When line 7 is reached, the region (if any) above p has been processed and deleted from the working copy of the graph. If $p = SW$, then since p is a right outer vertex, $p = SE$ also. Hence the processing of the entire graph (or subgraph) is complete. Note that at this time, *VerticesRemaining* = FALSE and $p.next$ = NIL. So, procedure *paths* terminates.

If $p \neq SW$, then we are not at the end of the left boundary and the situation is as depicted in Figure 4.15. If vertex p is being examined at this point of this procedure for the first time, then its adjacency list is traversed in the anticlockwise direction, beginning at a right outer vertex, t''. Because of the following condition, the vertex t'' is unique.

C3. The subgraph H being processed at any time has no edge (u, v) such that both u and v are on the right boundary and u and v are not next to one another on this right boundary.

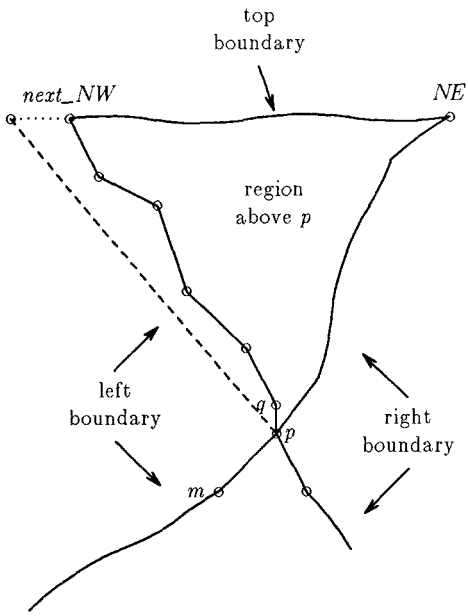


Fig. 4.14. Region above p.

Each vertex that is on p's adjacency list is examined. If t is on the left boundary and is not adjacent to p on the left boundary chain, then t is the t' of Figure 4.15. The left boundary from p to t' cannot be processed in the normal manner as this would result in a PDG in which p is a distant ancestor of t'. In case t' is not in the PDG, then the conditions for a chord are satisfied and procedure *process_chord* invoked to process the region between the left boundary and the

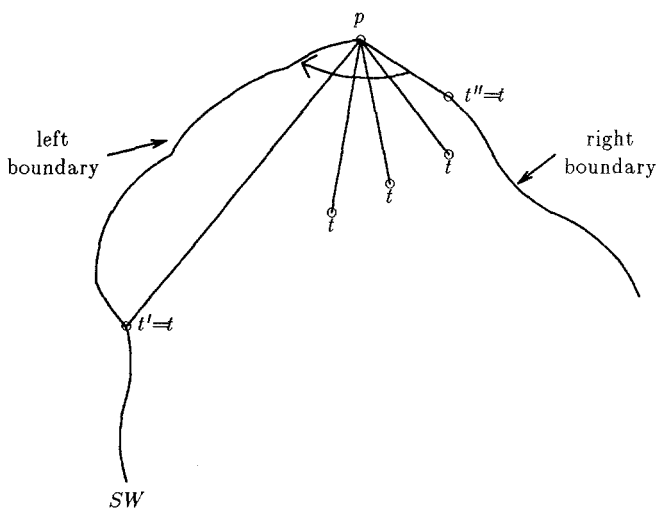


Fig. 4.15. Region below p.

edge (p, t'). During this processing, the entire region is deleted from the working copy of the graph and t'' is now the vertex that is anticlockwise adjacent to t'. So the traversal of p's adjacency list terminates. In case t' is already in the PDG, t' must equal SW as the NW and SW vertices are the only ones that can survive after being added to the PDG. The region bounded by the current left boundary and the edge (p, t') can be handled as a hanging component with $hg\_p = t'$ and $hg\_r = p$. Once again, this region is eliminated during its processing as a hanging component and t'' becomes the next vertex anticlockwise adjacent to p. So the traversal of p's adjacency list is complete.

If the vertex t does not satisfy the conditions of line 12, then its *RightOuterReached* variable is set to p to indicate that it is reachable by a single edge from the right outer vertex p. This is used in the next procedure to detect a chord. By setting this variable now, we avoid having to traverse p's adjacency list several times. This prevents the computing time of our algorithm from becoming $O(n^2)$.

4.3.9. process_NotRightOuter. The Pascal-like code for this procedure is given in Figure 4.16. This procedure is charged with the task of adding vertices to the next left boundary. On entry, this left boundary begins at *next_NW* and ends at q. Since the graph we are dealing with is a PTP graph, q and p are adjacent vertices. The adjacency list of vertex p is traversed in the clockwise direction in the loop of lines 2–30. This traversal begins just after the vertex q.

Let the adjacent vertex currently being examined be r. The case of lines 6–12 is the same as that of lines 12–18 of Figure 4.13 and is handled in an identical manner. Lines 13–21 examine the case when r is a vertex that has not been seen by the algorithm earlier. r is added to the next left boundary chain in lines 14 and 15. The special cases of a top left and bottom left corner created by the edge (p, r) are detected and processed in lines 16–21.

The last case for r that requires us to do anything is that of line 22. In this case the vertex r has been reached from two nonadjacent vertices on the current left boundary chain. This means that there is a hanging component that is to be processed.

Following the processing of vertex r in lines 5–24, this vertex is on the next left boundary chain. At this time we perform Task 2 of Section 4.3.8 for the case that p is not reached the first time. This is done in lines 26–29. Performing this task is somewhat simplified as vertex r cannot be in the PDG when the conditionals of line 26 are true. Because of the code of lines 26–29 in Figure 4.16, executing the code of lines 11 and 12 of Figure 4.13 when p is being visited other than the first time cannot result in any useful work. The adjacency list of p will be traversed and line 20 is the only one that can be reached. However *t.RightOuterReached* is already p!

4.3.10. Proofs for C1–C3

LEMMA 4.1. *Condition* C1 *of Section* 4.3.5 *is always true.*

PROOF. This is true initially when H is the whole original PTP graph. To see this, note that by the initial choice of NW, NE, SW, and SE, the bottom boundary

```
line   PROCEDURE process_NotRightOuter;
       (* p is not on the right outer boundary. So, q is adjacent to p. Examine all other vertices
       adjacent to p *)
 1       delete edge (p, q) from the graph;
 2       LOOP
 3         IF (adjacency list of p is empty) THEN RETURN; (* finished *)
 4         r = vertex on adjacency list of p that is clockwise from q;
 5         CASE
 6           : (r.LeftBoundary = LeftBound AND r ≠ p.next):
7-12         These lines are lines 13-18 of Figure 4.13 with t replaced by r
13           : (r.LeftBoundary = 2): (* r is a new vertex *)
14             q.next = r; r.LeftBoundary = 1 − LeftBound;
15             q = r; VerticesRemaining = TRUE;
16             IF (r.TopOuter) THEN
17               [ process_TopLeft; next_NW = r ]
18             ELSEIF (r.BotOuter AND p ≠ SW) THEN
19               [ process_BottomLeft; next_SW = r;
20               p.next = NIL; RETURN (* exit left boundary traversal *)
                 ]
21             ENDIF;
22           : (r.LeftBoundary = 1 − LeftBound):
23             q = r; process_hanging_component (p, r, LeftBound);
24         ENDCASE;
25         delete edge (p, r) from the graph;
26         IF ((r.RightOuterReached = NW) AND (NW.next ≠ r)) THEN
27           [ process_chord (NW, r, 1 − LeftBound);
             (* process subgraph marked by next left boundary vertices *)
28           NW.next = r; q = r ]
29         ENDIF;
30       REPEAT;
31     END process_NotRightOuter;
```

Fig. 4.16

cannot have a shortcut. We need to establish that all subsequent invokations of procedure *paths* preserve this condition. Let us examine each of these.

(a) From step 5 of procedure *process_chord* (Figure 4.6)

From Figure 4.5(b) we see that the bottom boundary consists of left boundary vertices between *ch_SW* and *ch_SE*. Each of these is adjacent to *ch_end*. So, if there is an edge (u, v) that violates C1, then vertices u, v, and *ch_end* form a triangle that is not a face. This violates the requirement that the initial graph is a PTP graph.

(b) From step 2 of procedure *process_BottomLeft* (Figure 4.8)

The new bottom boundary is a left segment of the previous bottom boundary (Figure 4.7). So, if C1 is true for the previous boundary, it must be for the new one.

(c) From step 6 of procedure *process_hanging_component* (Figure 4.10)

All the new bottom boundary vertices are adjacent to vertex *hg_p* (see Figure 4.9). So, if there is an edge (u, v) that violates C1, then the graph has a triangle $(u, v,$ *hg_p*$)$ that is not a face. So such an edge cannot exist.

Table 1. Typical run times on an Apollo DN320 workstation.

	Time (seconds)	
Number of nodes	PDG from PTP graph	Dual from PDG
25	0.179	0.0127
50	0.311	0.0237
100	0.693	0.0495

(d) From step 4 of procedure *process_TopLeft* (Figure 4.11)
All vertices on the new bottom boundary are adjacent to vertex p (see Figure 4.12). So C1 must be satisfied. □

The proofs of the following lemmas are similar to that of Lemma 4.1.

LEMMA 4.2. *Condition* C2 *stated in Section* 4.3.7 *is always true.*

LEMMA 4.3. *Condition* C3 *of Section* 4.3.8 *is always true.*

4.4. Complexity Analysis. The algorithm we have described is easily implemented to run in $O(n)$ time. The only cause for concern is that we need to begin the traversal of various adjacency lists from seemingly random points. A closer examination reveals that this is not so. The node at which the traversal starts fits into one of the following categories:

(a) It is one node clockwise or anticlockwise from a left boundary node.
(b) Let (u, v) be an edge. This edge is represented by two nodes: one (say A) on the adjacency list for u and the other (say B) on the adjacency list for v. If the edge (u, v) is detected from vertex u, then we may wish to traverse the list for v beginning at the node for u. This can be done, easily, by keeping a pointer from A to B and another from B to A.

5. Experimental Results. Our algorithm to find a rectangular dual was programmed in Pascal and run on an Apollo DN320 workstation. The typical time taken for PTP graphs with 25, 50, and 100 nodes is shown in Table 1. As can be seen, the algorithm is very practical. We also ran the 36-node example of [6]. This required only 0.232 seconds.

6. Conclusions. We have developed a linear-time algorithm to obtain a rectangular dual of a planar triangulated graph. This algorithm has been programmed in Pascal. Experimental results indicate that it is a very practical algorithm.

References

[1] J. Bhasker and S. Sahni, A linear algorithm to check for the existence of a rectangular dual of a planar triangulated graph, *Networks*, **17** (1987), 307–317.

[2] G. Brebner and D. Buchanan, On compiling structural descriptions to floorplans, *Proceedings of the IEEE International Conference on Computer-Aided Design*, Santa Clara, 1983, pp. 6-7.
[3] J. Grason, A dual linear graph representation for space filling problem of the floor plan type, in *Emerging Methods in Environmental Design and Planning* (G. T. Moore, ed.), Proceedings of the design methods group, First International Conference, Cambridge, MA, 1968, pp. 170-178.
[4] W. R. Heller, G. Sorkin, and K. Maling, The planar package planner for system designers, *Proceedings of the 19th Design Automation Conference*, Las Vegas, 1982, pp. 253-260.
[5] E. Horowitz and S. Sahni, *Fundamentals of Data Structures in Pascal*, Computer Science Press, Rockville, MD, 1984.
[6] K. Kozminski and E. Kinnen, An algorithm for finding a rectangular dual of a planar graph for use in area planning for VLSI integrated circuits, *Proceedings of the 21st Design Automation Conference*, Alburquerque, 1984, pp. 655-656.
[7] K. Kozminski and E. Kinnen, Rectangular dual of planar graphs, *Networks* (submitted).
[8] K. Maling, S. H. Mueller, and W. R. Heller, On finding most optimal rectangular package plans, *Proceedings of the 19th Design Automation Conference*, Las Vegas, 1982, pp. 663-670.