A LINEAR ALGORITHM TO FIND A RECTANGULAR DUAL OF A PLANAR TRIANGULATED GRAPH *

Jayaram Bhasker+ and Sartaj Sahni

University of Minnesota

ABSTRACT

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

Keywords and Phrases

Rectangular dual, planar triangulated graph, algorithm, complexity, floor planning.

---

* This research was supported in part by the National Science Foundation under grant MCS-83-05567.

+ Present address: Computer Sciences Center, Honeywell Inc., 1000 Boone Ave North, Golden Valley, MN 55427.

1 INTRODUCTION

A *rectangular dual* of an *n*-vertex graph, G = (V, E), is comprised of *n* non-overlapping 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 ([BREB83], [HELL82], [MALI82]). 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 [BHAS85a], [BREB83], [HELL82], [KOZM84ab], and [MALI82]. 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).

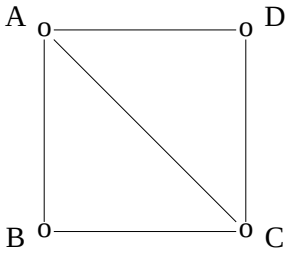
P2: All internal vertices have degree ≥ 4.

P3: All cycles that are not faces have length ≥ 4.

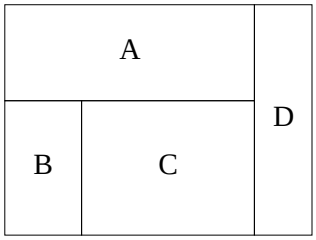
Figure 1.1 shows an example of a PTP graph, G, and two of its rectangular duals. In [BHAS85a], it is shown that every planar graph that satisfies P1 and P3 also satisfies P2.

Kozminski and Kinnen [KOZM84ab] 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 [KOZM84ab].

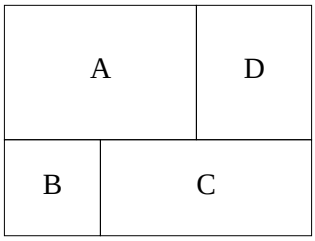
Definitions [KOZM84ab]: A *block* is a biconnected component. A *plane block* is a planar block. A *shortcut* in a plane block G, is an edge that is incident to two vertices on the outermost cycle of G and that is not a part of this cycle (see Figure 1.2). A *corner implying path* in a plane block G, is a segment $v_1, v_2, ..., v_k$ of the outermost cycle of G with the property that (v_1, v_k) is a shortcut and that $v_2, ..., v_{k-1}$ are not the endpoints of any shortcut. The *block neighborhood graph* (BNG) of a planar graph G, is a graph in which there is a



(a) Planar triangulated graph



(b) Rectangular dual



(c) Rectangular dual

Figure 1.1

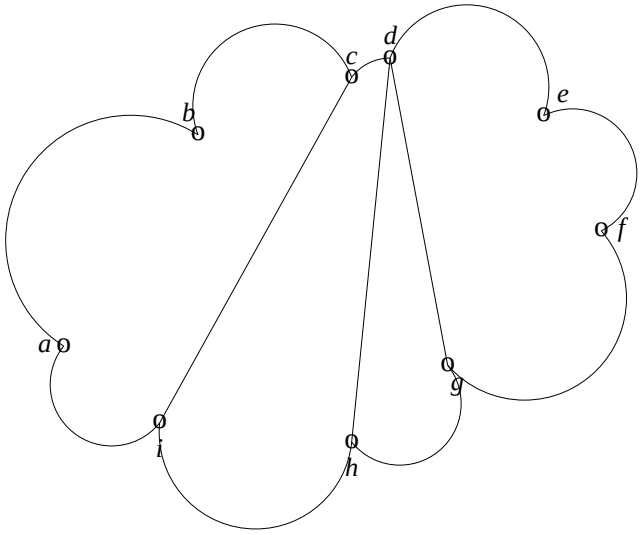
distinct vertex for each biconnected component of G. There is an edge between two vertices iff the two biconnected components they represent, have a vertex in common. A *critical corner implying path* in a biconnected component G_i of G is a corner implying path of G_i that does not contain cut vertices (articulation points) of G. □

Theorem 1.1 [KOZM84ab]: 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 [KOZM84ab] 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 [BHAS85a] have developed an $O(n)$ algorithm to determine if a given planar graph is properly triangulated. Since the conditions of Theorem 1.1 are



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

Figure 1.2

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 [BHAS85a] to actually construct a planar dual (whenever one exists) in O(*n*) time.

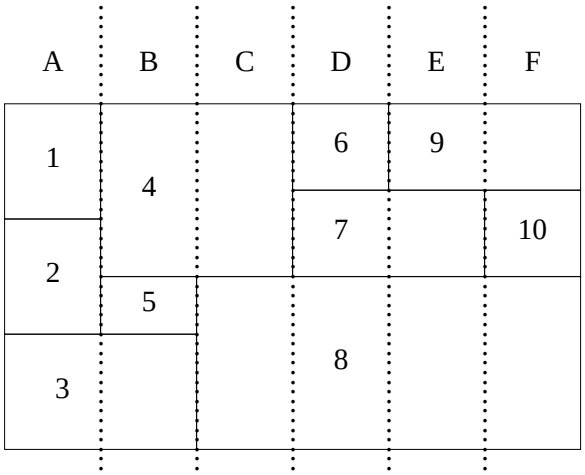
2 ALGORITHM OVERVIEW

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.

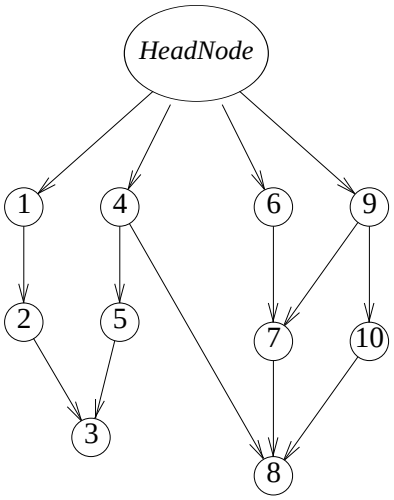
From this column partitioning of a rectangular dual, we can construct a directed graph called the *path digraph* (PDG). The PDG contains a distinguished vertex called the *HeadNode*. In addition, it contains one vertex for each rectangle in the dual. Since the dual of Figure 2.1(a) has 10 rectangles, 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.

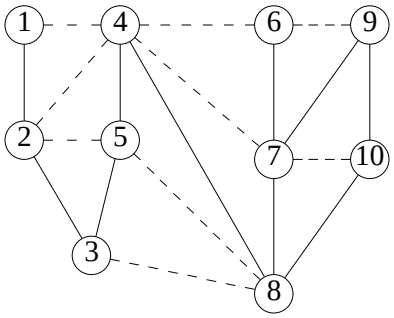
Traversing the six columns of Figure 2.1(a) yields the PDG of Figure 2.1(b). Each column of the dual corresponds to a distinct path from the *HeadNode* of the PDG to a leaf vertex (i.e., a vertex with no outgoing edges). For instance, the path (*HeadNode* $\rightarrow 4 \rightarrow 5 \rightarrow 3$)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



(a) Rectangular dual



(b) Path digraph



(c) Planar triangulated graph

Figure 2.1: Example of rectangular dual, PDG, and PTP graph.

A vertex *i* is a *parent* of another vertex *j* in the PDG iff <*i*, *j*> 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). 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.

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*.

Proof: Since *i* is a distant ancestor of *j*, the PDG contains a directed path $i, k_1, k_2, ..., k_p, j$ for some p, $p \geq 1$. Hence, in some column of the dual, *i* is on top of k_1, which is on top of k_2, which is on top of k_3, ..., which is on top of *j*. Since the dual is composed strictly of rectangles, it is not possible for *i* to be directly on top of *j* in some other column of the dual. Hence *i* cannot also be 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.

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.

Proof: Similar to that of Lemma 2.1. □

Our algorithm to obtain a rectangular dual begins with a planar triangulated graph that satisfies the necessary and sufficient conditions of [KOZM84ab], 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 somewhat simple graph 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.

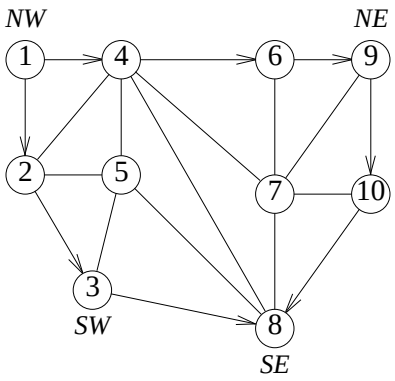


Figure 2.2: Boundary orientations.

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 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

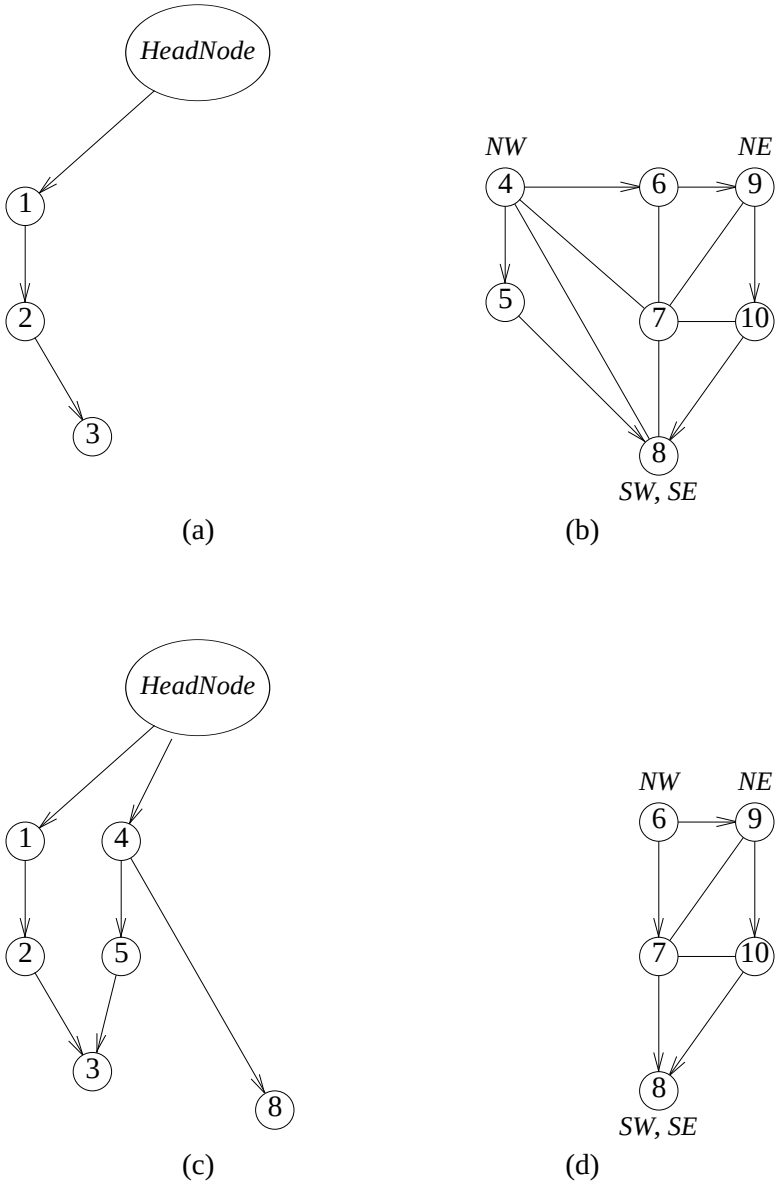
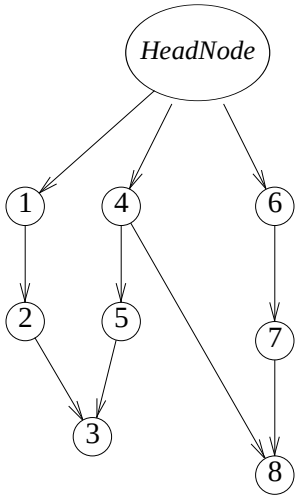


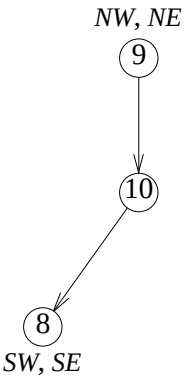
Figure 2.3 (Continued on next page)

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.

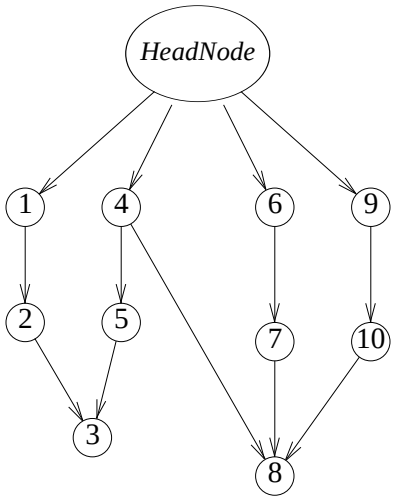
The right outer boundary is now (6,7,8). Adding this to the PDG yields the PDG of Figure 2.3(e). The planar graph that remains is Figure 2.3(f). Traversing this last path yields the PDG of Figure



(e)



(f)



(g)

Figure 2.3 (Contd.)

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 PDG´s 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).

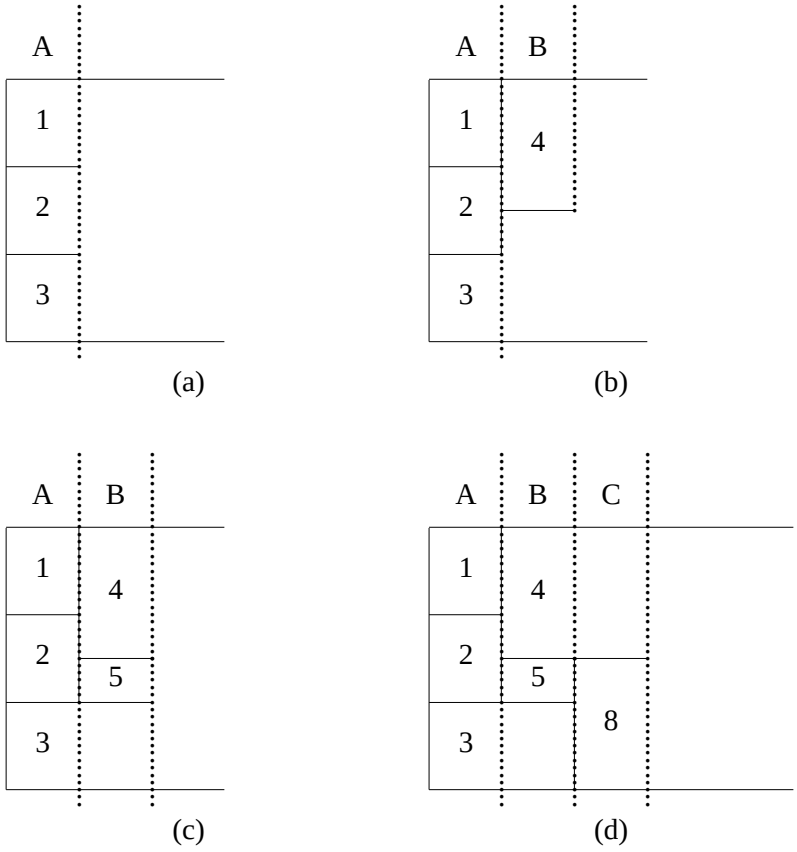


Figure 2.4: Rectangular dual construction. (Continued on next page)

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

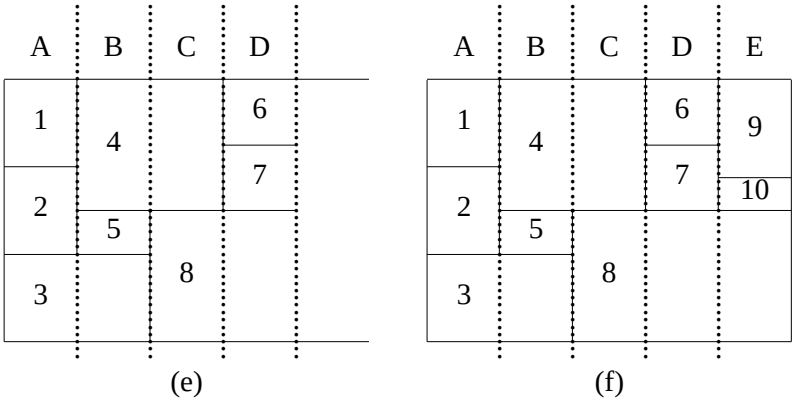


Figure 2.4: Rectangular dual construction. (Contd.)

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 non-overlapping 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

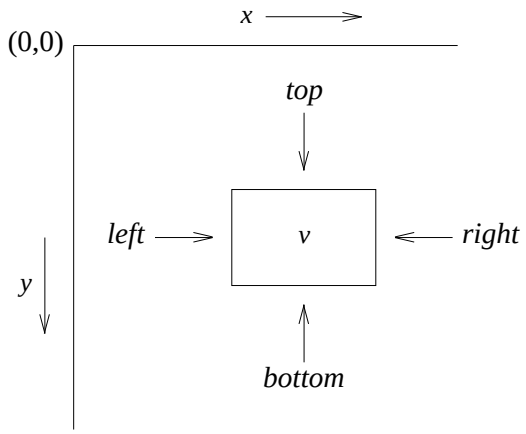


Figure 3.1: Coordinate system for the dual.

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.

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*. The variable *FirstPath* is Boolean valued. Its value is true initially (line 2) and remains true so 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).

```
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 ;
```

Algorithm 3.1

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 $x + 1$.

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*). On entry to *place*, *x* is

```
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 *)
```

Algorithm 3.2 (Continued on next page)

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* hasn´t been placed (i.e., *v* ≠ *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

```
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 ;
```

Algorithm 3.2 (Contd.)

$v.bottom = y$. *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 in C has already been visited, then its *u.top* is known and *v.bottom* must equal *u.top* (Figure 3.2(a)). If there is no vertex in C or if the C vertex has not been visited, then its *top* value is not known and *v.bottom* is determined by the vertices in D (Figure 3.2(b)).

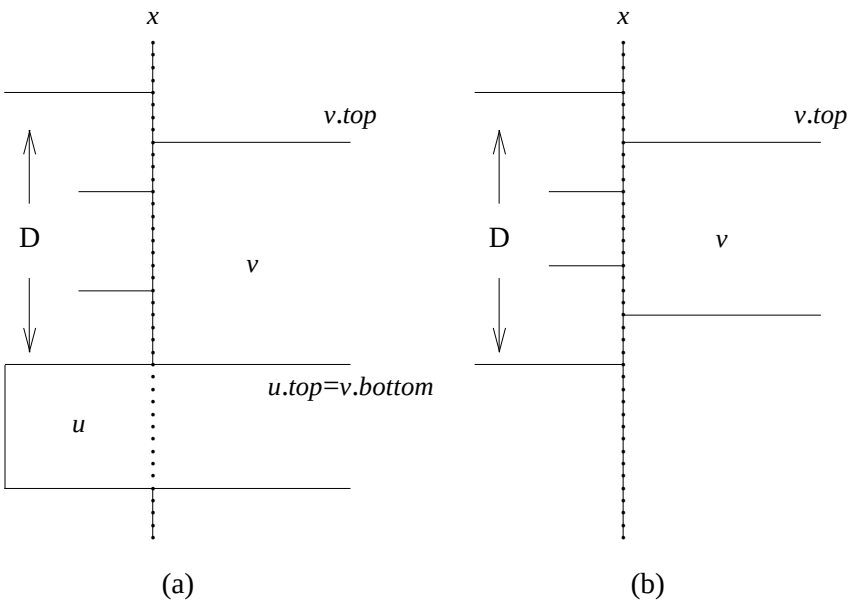


Figure 3.2: Determining *v.bottom*.

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 3.2(a)). Lines 10 - 14 are written as though Figure 3.2(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 3.2(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 3.2(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 3.2(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 3.2(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

Because of space limitations, this section has been omitted. It appears in its entirety in the report [BHAS85b].

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 [KOZM84a]. 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.

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

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

7 REFERENCES

BHAS85a Bhasker J. and S.Sahni, *A Linear Algorithm to Check for the Existence of a Rectangular Dual of a Planar Triangulated Graph*, Technical report, Computer Science Dept., University of Minnesota, Minneapolis, 1985.

BHAS85b Bhasker J. and S. Sahni, *A linear algorithm to find a rectangular dual of a planar grapg*, Technical Report TR 85-26, Computer Science Dept., University of Minnesota, Minneapolis, 1985.

BREB83 Brebner G. and D.Buchanan, *On Compiling Structural Descriptions to Floorplans*, Proc. IEEE ICCAD, Santa Clara, Sept 1983, pp 6-7.

HELL82 Heller W.R., G.Sorkin and K.Maling, *The Planar Package Planner for System Designers*, Proc. 19th DAC, Las Vegas, June 1982, pp 253-260.

HORO84 Horowitz E. and S.Sahni, *Fundamentals of Data Structures in Pascal*, Computer Science Press,Inc., Rockville, MD, 1984.

KOZM84a Kozminski K. and E.Kinnen, *An Algorithm for Finding a Rectangular Dual of a Planar Graph for Use in Area Planning for VLSI Integrated Circuits*, Proc. 21st DAC, Albuquerque, June 1984, pp 655-656.

KOZM84b Kozminski K. and E.Kinnen, *Rectangular Dual of Planar Graphs*, Networks (submitted for publication).

MALI82 Maling K., S.H.Mueller and W.R.Heller, *On Finding Most Optimal Rectangular Package Plans*, Proc. 19th DAC, Las Vegas, June 1982, pp 663-670.

--

--