A LINEAR TIME ALGORITHM TO CHECK FOR THE EXISTENCE OF A RECTANGULAR DUAL OF A PLANAR TRIANGULATED GRAPH *

Jayaram Bhasker and Sartaj Sahni

University of Minnesota

ABSTRACT

We develop a linear time algorithm to determine if a given planar triangulated graph has a rectangular dual.

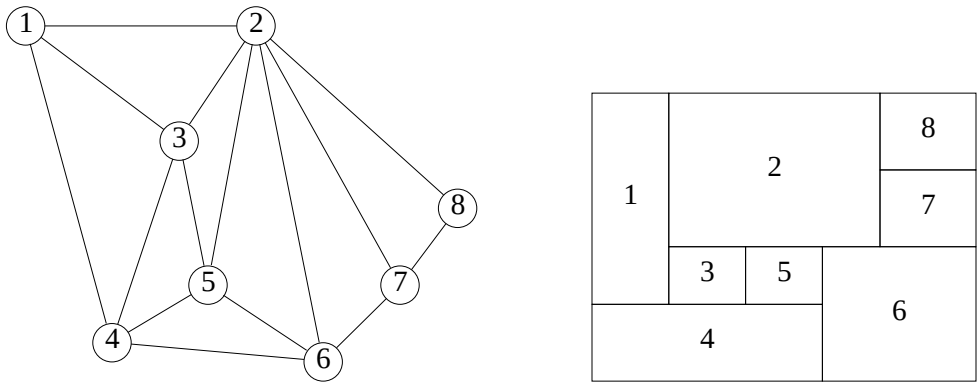
Keywords and Phrases

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

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

1 INTRODUCTION

In the floor planning for a rectangular chip, one begins with a graph, G = (V, E), in which the vertices represent circuit clusters or functional modules that are to be assigned a rectangular space on the chip. Edge (*i*, *j*) represents the requirement that modules *i* and *j* be adjacent (i.e. have an edge or a portion of one in common). A *rectangular dual* of G is an assignment of modules to non-overlapping rectangular areas on the chip such that the module adjacencies specified by the edges are satisfied. An example adjacency graph, G, and one of its rectangular duals is shown in Figure 1.1.



(a) Adjacency graph, G (b) Rectangular Dual

Figure 1.1: An example of a graph, G, and its rectangular dual.

The problem of obtaining a rectangular dual of an adjacency graph has been studied in [BREB83], [HELL82], [KOZM84a and b], [MALI82], etc. Kozminski and Kinnen [KOZM84a] have established necessary and sufficient conditions for the existence of a rectangular dual when G is planar and satisfies the following:

P1: Every face (except the exterior) is a triangle.

P2: All internal vertices have degree ≥ 4.

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

Using their necessary and sufficient conditions, Kozminski and Kinnen are able to obtain a rectangular dual (when one exists) in $O(n^2)$ time, where *n* is the number of vertices in G. In this paper, we develop an O(*n*) algorithm to determine if a planar graph G satisfies conditions P1 and P3 above. Lemma 1.1 shows that if P1 and P3 are satisfied, then so also is P2. Our algorithm

actually obtains a planar embedding that has properties P1 and P3, whenever such an embedding exists. This algorithm is used by us in [BHAS85] to obtain an O(n) algorithm to obtain the rectangular dual of a planar graph that satisfies P1 - P3 and the necessary and sufficient conditions of [KOZM84a].

Lemma 1.1: Let G be a planar graph that satisfies properties P1 and P3. Then G also satisfies P2.

Proof: If G does not satisfy P2, then it must contain an internal vertex, v, of degree 1, 2 or 3. If v is of degree 1 or 2, then G is not triangulated. If v is of degree 3, then since G is triangulated, G must contain the subgraph K_4 (K_4 is a complete graph with 4 vertices). However, this cannot be drawn so as to satisfy P1 and P3. □

In Section 2, we develop an algorithm to obtain a planar embedding of a biconnected component of G. This planar embedding satisfies P1 and P3, and has all articulation (or cut) points on the outer boundary of the embedding. The algorithm obtained for biconnected components may be used to obtain a planar embedding of an arbitrary planar graph, G, that satisfies P1 and P3.

2 THE ALGORITHM

Let P13 denote the set of all planar graphs that can be drawn such that properties P1 and P3 are satisfied. It is easy to see that G $\in$ P13 iff every connected component H of G is in P13. Hence, we may restrict ourselves to connected graphs, G. The following lemma shows that we need concern ourselves only with the biconnected components of G.

Lemma 2.1: G $\in$ P13 iff it has a planar embrdding that in addition to satisfying P1 and P3, also has all the articulation points of G on the outer boundary (Figure 2.1).

Proof: This follows from the observation that if an articulation point A is not on the outer boundary, then at least one of the internal faces must be non-triangular. □

As a result of the preceding discussion, our algorithm to obtain a planar embedding of G that satisfies P1 and P3 takes the form of Algorithm 2.1.

Steps 1 and 3 can be performed in O(n) time (see [HORO78] or [AHO74], for e.g., for an O(n) algorithm for Step 1) for planar graphs, G. The remainder of this section is therefore concerned with Step 2 alone.

Let H be a biconnected component of G and let *outer* denote the subset of vertices of H that are required to be on the outer boundary (*outer* = V(H) $\cap$ A(G), where V(H) is the vertex set of H and A(G) is the set of articulation points of G). Before attempting to construct the required embedding for H, we can carry out simple checks that will eliminate many H´s for which such an embedding does not exist.

Check 1: Let (i, j) be an edge in H. Vertex $k \in$ V(H) is a common vertex of (i, j) iff (i, k) and (j, k) $\in$ E(H) (E(H) is the edge set of H). The vertices i, j and k form a triangle in H.

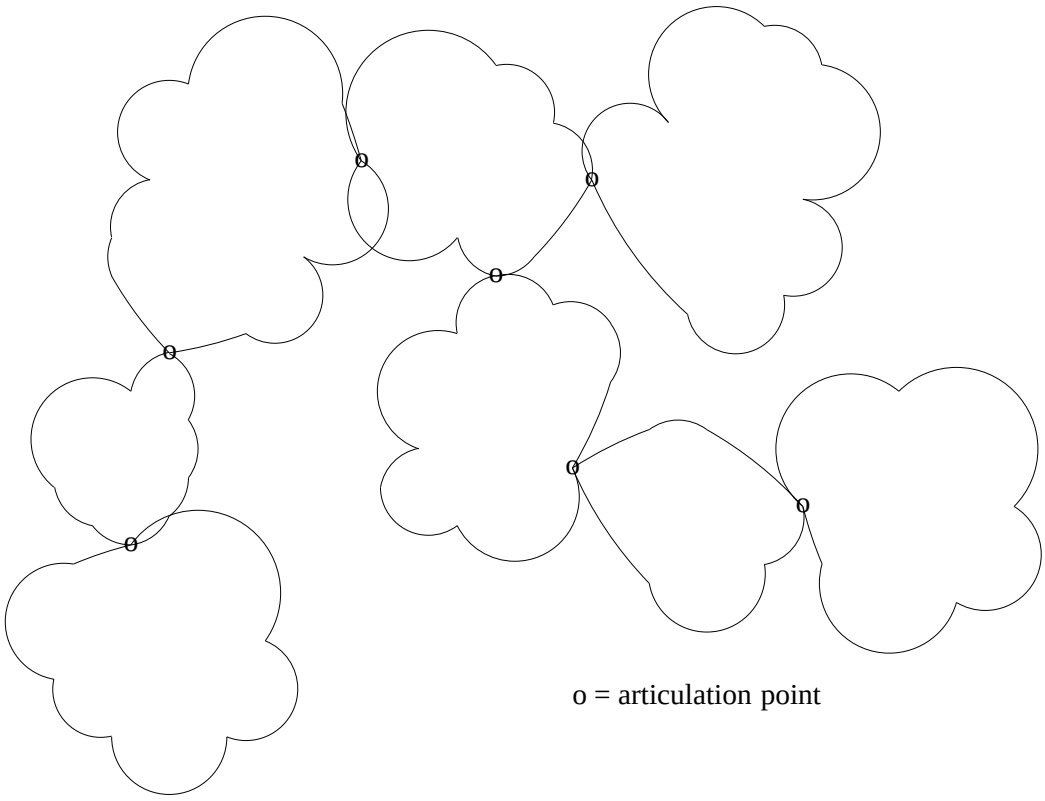


Figure 2.1: Planar embedding with only outer boundary shown.

In Check 1, for each edge (*i*, *j*), we determine the number of common vertices. If some edge has more than two common vertices, then every embedding of H will contain at least one cycle of length 3 that is not a face. So, P3 cannot be satisfied. If some edge has zero common vertices, then every embedding of H will contain a face that is not a triangle.

Check 2: The number of interior faces of H is *interior_faces* = $|E(H)| - |V(H)| + 1$. The number of triangles is *triangles* = (total number of common vertices)/3. The value of *triangles* is easily determined during Check 1. If *interior_faces* $\neq$ *triangles*, then H has no embedding that satisfies P1 and P3.

Checks 1 and 2 are easily performed in linear time. Assume that these checks have been performed and that H has passed both. Further, assume that *triangles* ≥ 1. Our algorithm to obtain

---

Step 1: Find the biconnected components of G.

Step 2: FOR each biconnected component of G, DO
    Try to find an embedding of this component that satisfies P1 and P3
    and for which all articulation points on the outer boundary;
    IF no such embedding exists, THEN terminate unsuccessfully;
ENDFOR.

Step 3: Place all the embeddings obtained so that common vertices of the biconnected components abut.

Algorithm 2.1

the required embedding of H works in a greedy way. It begins with any triangle of H. This triangle has a unique embedding (up to symmetries and rotations). This embedding is expanded in a systematic way by adding additional vertices and edges. During the expansion, we ensure properties P1 and P3 are always satisfied. Further, the vertices in *outer* remain on the outer boundary of the embedding. A small example will illustrate how the algorithm works.

Let V(H) = {1,2,3,4,5}. Assume that <1,2,3> is one of the triangles of H. Any two embeddings of this triangle are either rotations or mirror images of each other. We start with any embedding of <1,2,3>. Suppose we use the embedding of Figure 2.2(a). The outer boundary of this embedding is [2,1,3] (the boundary is traversed in anticlockwise order). At this time, *#_of_triangles_drawn* = 1; vertices {1,2,3} are *old*; the remaining vertices {4,5} are *new*; edges {(1,2), (1,3), (2,3)} are *covered*; and the remaining edges are *not_covered*.

To expand our triangulated embedding, we may pick any of the vertices in {1,2,3} as a *start_vertex*. Suppose we begin with *start_vertex* = 3.

Let *next_vertex* be the outer boundary vertex adjacent to *start_vertex* and anticlockwise from it. Since *start_vertex* = 3 and the outer boundary is [2,1,3], *next_vertex* = 2. We shall attempt to expand the current embedding at the edge (*start_vertex*, *next_vertex*) = (3,2). For this, we determine if (3,2) has a common vertex *i* such that (3,*i*) or (2,*i*) is an edge that is *not_covered*. Clearly, there can be at most one such *i*. (Note that from Check 1, we know that (*start_vertex*, *next_vertex*) has either one or two common vertices. However, one of these is connected to this edge by *covered* edges as our embedding is always triangulated.) There are three possibilities for vertex *i*:

1) There is no such *i*

2) *i* is a *new* vertex

3) *i* is an *old* vertex

For our example, suppose that *i* = 4. So, *i* is a *new* vertex. At this time, we have no choice other than to expand Figure 2.2(a) to Figure 2.2(b). All other expansions are equivalent to this in terms of accessibility of vertices from the outside. The outer boundary is now [2,1,3,4], vertices {1,2,3,4} are *old*, the edges {(1,2), (1,3), (2,3), (2,4), (4,3)} are *covered*, and *#_of_triangles_drawn* = 2.

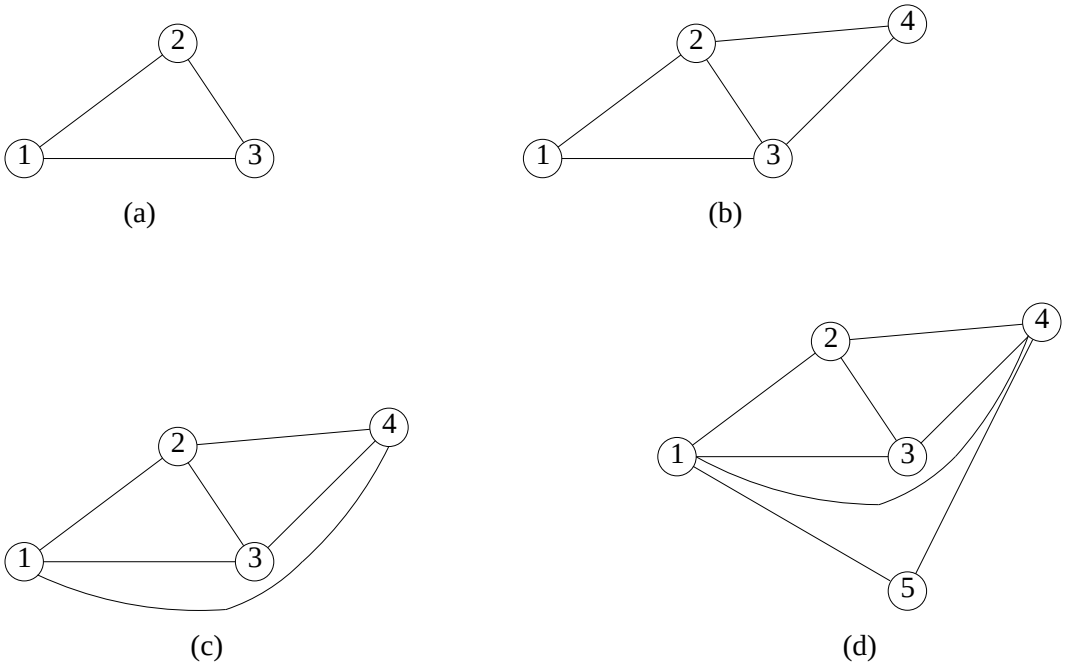


Figure 2.2

For further expansion, we consider the *start_vertex* again. The *next_vertex* is updated to 4 as this vertex is now the boundary vertex that is one position anticlockwise from *start_vertex*. Suppose that vertex *i* = 1 is a common vertex of (3,4). We are now in the third possibility mentioned above (vertex 1 is *old*). When vertex *i* is an *old* vertex, we have three subcases:

a) *i* is a boundary vertex that is one vertex anticlockwise from *next_vertex*.

b) *i* is a boundary vertex that is one vertex clockwise from *start_vertex*.

c) all other possibilities.

For the example of Figure 2.2(b), b) is the case. It is clear that (1,4) must be drawn to create the face <1,3,4> as in Figure 2.2(c). This can lead to infeasibility if any one of the following is true:

(i) Vertex 3 is an articulation point. In this case, we violate the requirement that articulation points be on the outer boundary.

(ii) The vertex *i* = 1 is two positions clockwise from *next_vertex* = 4. This is true in our case. At this time, <1,2,4> also forms a triangle. Both <1,2,4> and <1,3,4> cannot be drawn as faces. This condition is determined at the end of our algorithm when we detect that the number of triangles drawn is not equal to the number of triangles in the graph.

(iii) *Start_vertex* has additional edges incident to it. In this case, the graph has a face that is not a triangle. Note that edges (*i*, *start_vertex*) and (*start_vertex*, *next_vertex*) cannot have any *new* common vertices as each has two *old* common vertices.

Let us proceed with Figure 2.2(c). At this time, we have *#_of_triangles_drawn* = 3, vertices {1,2,3,4} are *old*, edges {(1,2), (1,3), (2,4), (2,3), (1,4), (3,4)} are *covered*, *start_vertex* = 3 and *next_vertex* = 4. The *start_vertex* is updated to be vertex 1 and we proceed with the expansion. Now, *next_vertex* becomes vertex 4 (as it is anticlockwise from 1 on the outer boundary). Suppose that vertex 5 is common to (1,4). This is a *new* vertex. It is drawn as in Figure 2.2(d). Vertex 5 becomes the new *next_vertex*. Suppose that no further expansion is possible. The algorithm will now check to see if the right number of triangle faces have been drawn. In the case of our example, the graph has 5 triangles but only 4 have been drawn. Since the algorithm never makes a choice that affects feasibility, we conclude that the graph has no embedding of the type desired.

A semi formal version of the algorithm that covers all the cases is presented below.

ALGORITHM *Draw_Biconnected* (H)

(* Obtain an embedding for H that satisfies P1 and P3 and has all vertices in *outer* on the outer boundary *)

Step 1: (* Initialize *)

- Initialize all vertices as *new* and all edges as *not_covered*.
- Pick an arbitrary edge and one of its common vertices. This is the starting triangle. Set the three vertices of the triangle to *old* and the three edges as *covered*. Set *#_of_triangles_drawn* to 1 and initialize the outer boundary to comprise the three vertices of this triangle in anticlockwise order. The outer boundary may be maintained as a doubly linked circular list [HORO84].

• Let *i* be any of the three *old* vertices. Set *start_vertex* = *i*.

Step 2: (* Repeatedly expand the embedding *)
Repeatedly execute Steps 3 and 4.

Step 3: (* Determine *next_vertex* and *i* *)
• *next_vertex* = (vertex that is anticlockwise from *start_vertex* on the outer boundary).
• Let *i* be a vertex common to (*start_vertex*, *next_vertex*) and such that at least one of the edges connecting *i* to this edge is *not_covered*. Let *i* = 0 if there is no such vertex.

Step 4: (* Expansion attempt *)
CASE *i* OF

: *i* = 0: (* No common vertex; Advance *start_vertex* anticlockwise *)
IF *next_vertex* has already been the *start_vertex* once, THEN go to Step 5.
start_vertex = *next_vertex*.

: *i* is a *new* vertex: (* Lemma 2.2 *)
• draw the triangle <*i*, *next_vertex*, *start_vertex*>.
• add *i* to the outer boundary between *start_vertex* and *next_vertex*.
• make *i old*.
• set edges (*i*, *start_vertex*) and (*i*, *next_vertex*) *covered*.
• *#_of_triangles_drawn* = *#_of_triangles_drawn* + 1.

: *i* is an *old* vertex: (* 3 subcases for *i* *)
CASE *i* OF

: *i* is anticlockwise from *next_vertex* on outer boundary:
(* Lemma 2.3 *)
IF (*next_vertex* ∈ *outer*) OR (*next_vertex* has additional *not_covered* edges incident to it)
THEN [embedding not possible; STOP]
ELSE [
draw the triangle <*i*, *next_vertex*, *start_vertex*>;
delete *next_vertex* from the outer boundary;
mark edge (*i*, *start_vertex*) used to draw new triangle as *covered*;
#_of_triangles_drawn = *#_of_triangles_drawn* + 1].

: *i* is clockwise from *start_vertex* on outer boundary:
(* Lemma 2.4 *)
IF (*start_vertex* ∈ *outer*) OR (*start_vertex* has additional

not_covered edges incident to it)
THEN [embedding not possible; STOP]
ELSE [
draw the triangle <*i*, *next_vertex*, *start_vertex*>;
delete *start_vertex* from the outer boundary;
mark edge (*i*, *next_vertex*) used to draw new triangle as *covered*;
#_of_triangles_drawn = *#_of_triangles_drawn* + 1;
start_vertex = *i*].

: ELSE: (* move forward; the triangle(s) involving *i* will be drawn later, if possible; see Lemma 2.5 *)
IF *next_vertex* has already been the *start_vertex* once, THEN go to Step 5.
start_vertex = *next_vertex*.

ENDCASE. (* end of subCASE of *i* *)

ENDCASE. (* end of CASE of *i* *)
(* End of repeat of Step 2 *)

Step 5: (* No further expansion possible; test for success *)
IF (there is a *new* vertex) OR (an edge that is *not_covered*) OR (*#_of_triangles_drawn* < *triangles*)
THEN [no embedding is possible]
ELSE [the desired embedding has been obtained].

ENDALGORITHM *Draw_Biconnected*.

Algorithm Correctness

The correctness of the above algorithm follows from the following observations:

1. If H has an embedding that satisfies P1 and P3 and in which all vertices in *outer* are on the outer boundary, then it has one in which the triangle selected in Step 1 has the chosen orientation and is a face. This statement is true as P3 implies that every triangle is a face. For each triangle <*i, j, k*>, there are only two different orientations: <*i, j, k*> and <*i, k, j*>. These are symmetric.
2. Each expansion step (iteration of Steps 3 and 4) preserves feasibility. I.e., if the embedding before an iteration of Steps 3 and 4 could be completed to satisfy the requirements, then the embedding after the iteration can. If the algorithm terminated during an expansion iteration, then no feasible embedding exists. The validity of this observation is established in

Lemmas 2.2 - 2.5.

3. If Step 5 is reached, the embedding cannot be expanded further without violating one of the requirements. This is true since at this time every vertex on the current outer boundary has been the *start_vertex* once and no expansion by a single triangular face is possible. Note that if expansion is not possible one triangle at a time, it is not possible by introducing many simultaneously. If some edge or vertex has not been included in the embedding by this time, then no embedding is possible. If the number of triangles in the graph is more than the triangles drawn, then there is a triangle that is not a face.

Lemma 2.2: Introducing a new vertex *i* as in Step 4 does not affect our ability to complete the embedding.

Proof: At this time, (*start_vertex*, *next_vertex*) is an exterior edge of the current embedding and is on one triangle of this embedding. The new vertex *i* cannot be placed inside this triangle as this will violate P3. So, the only possibility is to place *i* outside this triangle. This will cause the outer boundary to change from [...., *start_vertex*, *next_vertex*, ...] to [...., *start_vertex*, *i*, *next_vertex*, ...]. Since this is the only feasible way to introduce the triangle <*start_vertex*, *i*, *next_vertex*>, this choice doesn´t affect our ability to complete the embedding in a feasible manner. □

Lemma 2.3: If the common vertex *i* selected in Step 3 is anticlockwise from *next_vertex* on the current outer boundary, then the actions of Step 4 do not affect our ability to complete the embedding.

Proof: The configuration at this time is as in Figure 2.3(a). Because of P3, the triangle <*start_vertex*, *next_vertex*, *i*> must be present as a face. This edge cannot be placed inside the outer boundary as (*start_vertex*, *next_vertex*) and (*next_vertex*, *i*) are already edges on two different triangle faces. Hence, this edge must be drawn as in Figure 2.3(a). This does not guarantee that P3 is not violated unless *start_vertex* is more than one vertex anticlockwise from *i* (Figure 2.3(b)). In this case, <*start_vertex*, *j*, *i*> is another triangle but is not a face. This can be checked for in Step 4 but it is easier to detect this in Step 5.

If *next_vertex* $\in$ *outer*, then having drawn the new edge as in Figure 2.3(a), it is impossible to keep this vertex on the outer boundary. Also, if *next_vertex* has additional *not_covered* edges incident to it, these cannot be added to the new embedding without violating P3. So, in either case, we may terminate the algorithm. Note that (*start_vertex*, *next_vertex*) and (*next_vertex*, *i*) cannot be on any additional triangles as two have already been drawn for each. So, if there are any *not_covered* edges incident to *next_vertex*, the embedding cannot be completed. □

Lemma 2.4: If the common vertex *i* selected in Step 3 is clockwise from *start_vertex* on the current outer boundary, then the actions of Step 4 do not affect our ability to complete the embedding.

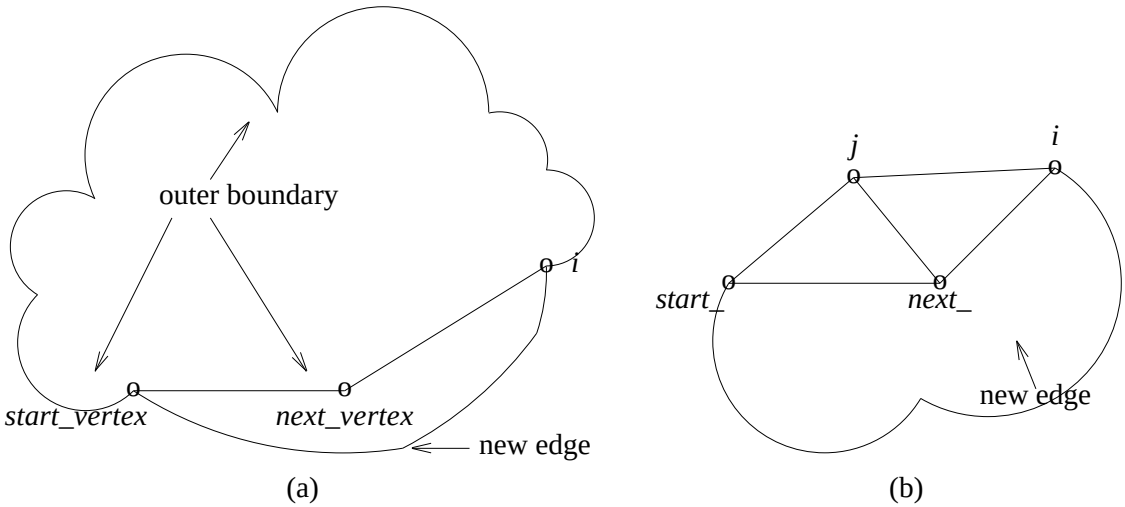


Figure 2.3

Proof: Similar to that for Lemma 2.3. □

Lemma 2.5: If the vertex *i* does not satisfy the conditions of Lemmas 2.3 - 2.4, then no expansion by a single triangle face is possible at this time.

Proof: First note that vertex *i* must be a boundary vertex. This follows from the remainder of Step 4. Whenever a boundary vertex becomes an interior vertex, we verify that there will be no remaining *not_covered* edges incident to it. So, we have the situation of Figure 2.4.

We have two choices for embedding the two new edges. Both create two faces (one is a triangle and the other may or may not be one). There is however no way to draw the two edges so that only a single face that is a triangle is introduced. We may postpone the treatment of the detected triangle to a later time. □

Complexity

By using appropriate data structures, the algorithm can be implemented to run in O(n) time (note that since the graph is planar, $|E|$ is O(n)). Also, observe that because of the way *start_vertex* is updated, an edge can play the role of (*start_vertex*, *next_vertex*) at most once. Each time Steps 3 and 4 are executed, the edge that plays this role changes. Further speedup is possible by eliminating both the IF-THEN parts of Step 4 and verifying in Step 5 that all

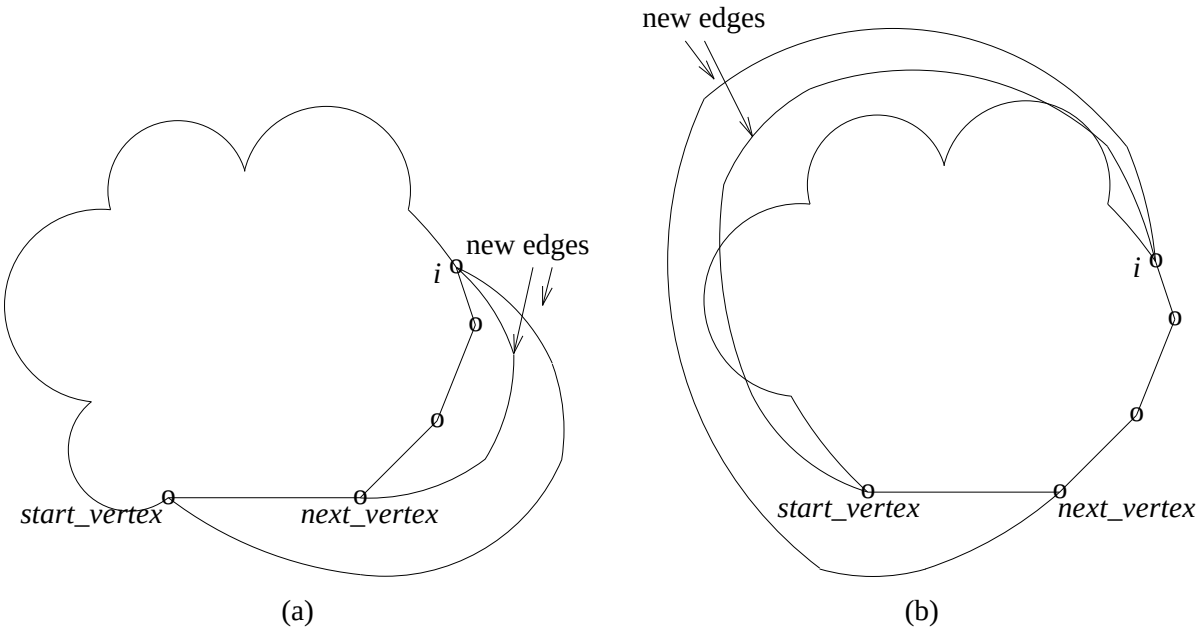


Figure 2.4

articulation points are on the outer boundary.

3 CONCLUSIONS

We have developed a linear time algorithm to check properties P1 and P3. This algorithm is useful in the floor planning of VLSI chips. We have programmed this algorithm in Pascal. The run times on an Apollo DN 320 workstation are given in Table 1. As can be seen, the algorithm is quite practical.

n	Time (in secs)
100	0.27
150	0.499
200	0.73

4 REFERENCES

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

BHAS85 Bhasker J. and S.Sahni, *A Linear Algorithm to Find the Rectangular Dual of a Planar Triangulated Graph*, Technical report, Computer Science Dept., University of Minnesota, Minneapolis, 1985.

BREB83 Brebner G. and D.Buchanan, *On Compiling Structural Descriptions to Floor-plans*, 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.

HORO78 Horowitz E. and S.Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD, 1978.

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.