

An Algorithm for Straight-Line Drawing of Planar Graphs

David Harel* and Meir Sardas†

Dept. of Applied Mathematics and Computer Science
The Weizmann Institute of Science, Rehovot, Israel

Abstract: We present a new algorithm for drawing planar graphs on the plane. It can be viewed as a generalization of the algorithm of Chrobak and Payne, which in turn, is based on an algorithm by de Fraysseix, Pach and Pollack. Our algorithm improves the previous ones in that it does not require a preliminary triangulation step; triangulation proves problematic in drawing graphs “nicely”, as it has the tendency to ruin the structure of the input graph. The new algorithm retains the positive features of the previous algorithms: It embeds a biconnected graph of n vertices on a grid of size $(2n - 4) \times (n - 2)$ in linear time. We have implemented the algorithm as part of a software system for drawing graphs nicely.

1 Introduction

In this paper we describe a new drawing algorithm for planar graphs. The algorithm is a central component of a software system we have developed for drawing graphs “nicely” [HS94], and was especially designed for that purpose. (The other main component of the system described in [HS94] is the simulated annealing algorithm of [DH89], which is used in the second stage to “massage” a rough solution into a “nice” one.) The new algorithm was inspired by an algorithm of Chrobak and Payne [CP90], which is a linear time variant of an algorithm of de Fraysseix, Pach and Pollack [FPP88].

The algorithm of [CP90] draws a graph with n vertices on a grid of size $(2n - 4) \times (n - 2)$ in time $O(n)$, and is quite easy to implement. Vertices are

*Email: harel@wisdom.weizmann.ac.il. Part of this author’s work was carried out during a sabbatical stay at Cornell University in Ithaca, NY, and was partially supported by grants AF #F49620-94-1-0198 (to F. Schneider), NSF #CCR-9223183 (to B. Bloom), and NSF #CDA-9024600 (to K. Birman), and ARO #DAAL03-91-C-0027 (to A. Nerode).

†Email: meir@orbot-instr.co.il. Current address: Orbot Instruments LTD, Yavneh Industrial Zone, P.O.B 601, Yavneh, Israel

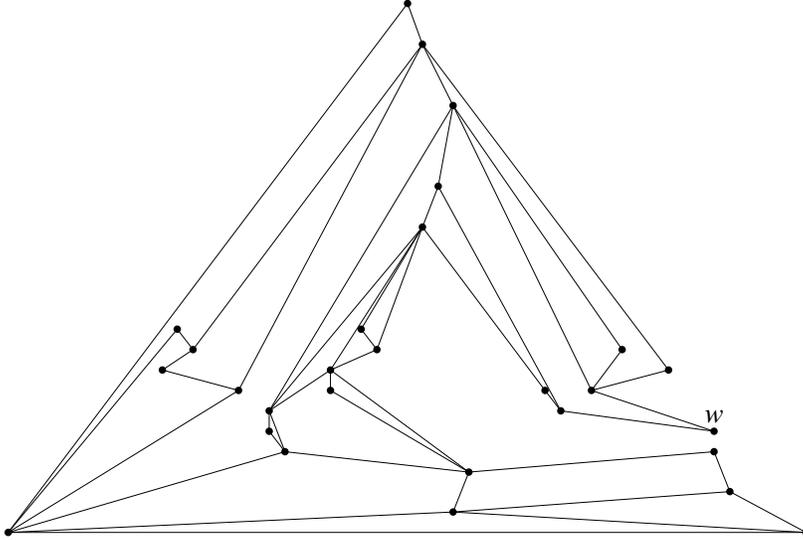


Figure 1: Sample output of the algorithm of [CP90] on a non-triangulated graph.

placed on grid points and edges are crossing-free straight lines.

The bound on the grid size required for the drawing becomes significant when the goal is aesthetics: If we know that the vertices of the graph will be located on (the grid-points of) a grid of size $O(n) \times O(n)$, we are guaranteed that the minimal distance between pairs of vertices will be no smaller than $\frac{1}{O(n)}$ of the entire drawing size, so they will not appear too close to each other. However, if vertices can be drawn at arbitrary locations, nothing can be guaranteed about the distance between them, which is a limitation. Another advantage of a small grid is that high precision operations are not necessary for calculating vertex positions; all numbers involved are on the order of n .

These features, coupled with its linear running time and relative simplicity, made the algorithm of [CP90] an excellent candidate for us to use when we were designing the system described in [HS94]. However, [CP90] has one serious drawback: It requires the input graph to be *maximal* planar, and we wanted to integrate it in a system that would handle planar graphs that are not necessarily maximal. The simplest way to achieve maximality is suggested in [FPP88], namely *triangulation*. If the graph is not maximal, dummy edges are added to make each face triangular. The result is then subjected to a drawing algorithm, and before producing the final drawing the edges added in the triangulation stage are deleted.

We tested this scheme and found it unsatisfactory for our purposes. For example, Figure 1 shows a typical output of the algorithm (after deleting the

added edges). The external face is drawn unacceptably concave. Standard “beautification” techniques, such as the simulated annealing scheme of [DH89] that we use in our system, are unable in many cases to overcome this kind of distortion, which can become much worse for larger graphs. The problem stems from the triangulation step, whose dummy edges often ruin the structure of the original graph, yielding totally unacceptable results.

To overcome this difficulty, we have developed a new algorithm, which is in the spirit of that of Chrobak and Payne. The new algorithm does not require a maximal triangulated graph, but works instead directly from the original input graph, which need only be biconnected and planar. It deals with the graph in steps, making the placement decisions for vertices in increasingly larger subgraphs. A vertex v appears in G_k , the graph constructed in the k 'th step, only if at least one of its neighbors appears in G_{k-1} . This avoids the situation of vertex w in Fig. 1, which was drawn based on dummy edges that were removed in the final drawing. The two main advantages of the original algorithm are retained: Ours also runs in linear time, and it employs a small bounded grid.

Before getting into the details of our work, we should mention that another drawing algorithm based on that of Chrobak and Payne has been developed by Kant [Kan92]. This algorithm is also aimed at producing aesthetic drawings, but it does so by drawing all the faces convex, with the external face being drawn as a triangle. Guaranteeing a convex drawing is stronger than what our approach guarantees, but it requires the input graph to be tri-connected, which is considerably harder to fulfill than our requirement of biconnectivity.

Section 2 describes the original algorithm of [CP90]. Section 3 contains a detailed description of our new algorithm, and Section 4 proves its correctness, discusses some issues of implementation, and analyzes its time complexity.

2 The original algorithm

This section describes the algorithm of [CP90], which is the starting point for our algorithm. It works in two steps. The first calculates the *canonical ordering*, which is the order in which the vertices will be processed, and the second (sometimes called the *placement step*) then constructs the drawing incrementally, adding vertices to the current drawing one by one, according to the canonical ordering.

2.1 The canonical ordering

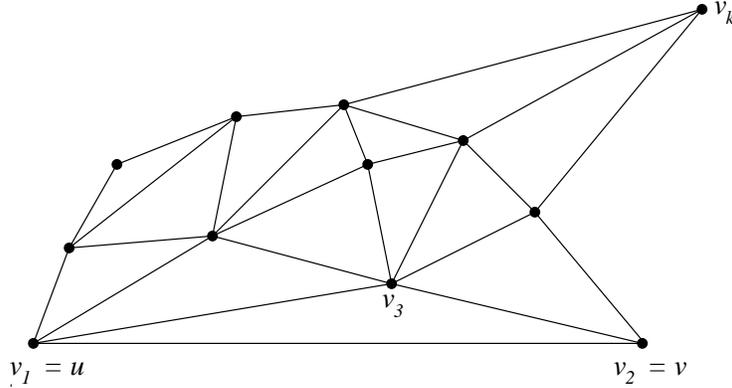


Figure 2: Illustration for the canonical ordering.

Definition 2.1 Let G be a maximal planar graph drawn in the plane, and let u, v, w be the vertices on the boundary of its exterior face. The canonical ordering is a labeling of the vertices of G in a sequence v_1, \dots, v_n such that $v_1 = u, v_2 = v$ and $v_n = w$, and for every $3 \leq k \leq n$ the following hold:

- (a) The subgraph G_{k-1} of G induced by v_1, \dots, v_{k-1} is biconnected, and the boundary of its exterior face is a cycle C_{k-1} containing the edge (u, v) .
- (b) The vertex v_k is on the exterior face of G_{k-1} , and has at least two neighbors in G_{k-1} . Moreover, all of its neighbors in G_{k-1} are consecutive on the path $C_{k-1} - (u, v)$. (See Fig. 2.)

The existence of such an ordering is proved in [FPP88], which also provides a linear time algorithm to compute one. We now describe this algorithm, which is adopted in [CP90].

A prerequisite is the availability of a *planar embedding*. A planar embedding is not an actual drawing of the graph, but is a data structure that describes the circular ordering of neighbors of each vertex in some planar drawing. A linear time algorithm for constructing a planar embedding for a planar graph is described in [CNAO85] (see also [Mel96]), and it can be used as a preliminary step to the canonical ordering algorithm now described.

The ordering algorithm works by processing each of the vertices (in an order explained below) and visiting its neighbors. Vertices are labeled, and when visiting neighbors the labels can be updated. The labels used are as follows: -1 , meaning ‘not yet visited’; 0 , meaning ‘visited once’; and $i > 0$, meaning ‘visited more than once and those of its neighbors already visited form i intervals in the circular order around the vertex given by the planar embedding’.

We start by choosing two vertices, calling them v_1 and v_2 , and assigning the label -1 to all the other $n - 2$ vertices. We then process v_1 and v_2 , as we now describe. Processing a vertex v_k is carried out by visiting each of its neighbors and updating the labels of those neighbors not yet processed. Let v be a neighbor of v_k . Then:

case (i) : v is labeled -1 , in which case relabel it with 0 .

case (ii) : v is labeled 0 . This means that v has one neighbor that has already been processed; call it u . Check if v_k is adjacent to u in the circular ordering of neighbors around v (given by the planar embedding). If so, label v with 1 , otherwise label it with 2 .

case (iii) : v is labeled $i > 0$. Check the two vertices adjacent to v_k in the circular ordering around v . If both have already been processed, label v with $i - 1$ (i.e., two intervals have now been merged). If one has been processed and the other not, v 's label remains i . If neither have been processed, label v with $i + 1$.

After processing v_k (for $k \geq 2$), a vertex with label 1 is chosen to be v_{k+1} in the canonical ordering (any such vertex can be chosen), and is thereby processed. This continues until no such v_{k+1} is found. However, the existence of a canonical ordering guarantees that this procedure will indeed continue until all vertices of G are processed.

2.2 The placement step

The second step in the algorithm of [CP90] places the vertices on grid points, to produce a planar drawing of the graph. To describe it, we shall need some notation.

Given two grid points $Q = (x_1, y_1)$, $R = (x_2, y_2)$, we denote by $\mu(Q, R)$ the intersection of the line with slope $+1$ from Q and the line with slope -1 from R . That is:

$$\mu(Q, R) = \frac{1}{2}(x_1 - y_1 + x_2 + y_2, -x_1 + y_1 + x_2 + y_2).$$

The *Manhattan distance* between two grid points Q and R is defined to be

$$MD(Q, R) = |x_2 - x_1| + |y_2 - y_1|.$$

Note that if this value is even, then so are the values $x_1 - y_1 + x_2 + y_2$ and $-x_1 + y_1 + x_2 + y_2$ that appear in the definition of μ . This means that if Q and R are grid points with even Manhattan distance, $\mu(Q, R)$ must be a grid point too.

In the second part of the algorithm, which we now describe, $P(v) = (x(v), y(v))$ denotes the current position of vertex v on the grid. To each vertex w we assign a set of vertices $L(w)$, whose meaning is explained below. The algorithm starts by placing v_1, v_2, v_3 on a triangle, as follows:

$$\begin{aligned} P(v_1) &:= (0, 0); & L(v_1) &:= \{v_1\}; \\ P(v_2) &:= (2, 0); & L(v_2) &:= \{v_2\}; \\ P(v_3) &:= (1, 1); & L(v_3) &:= \{v_3\}. \end{aligned}$$

The vertex v_k is added to the already placed vertices v_1, \dots, v_{k-1} at each step, forming the graph G_k . In order to better understand the iterative processing, it is worth noting the invariant claim, which captures the fact that at the k th step of the algorithm, the contour of G_k , termed C_k , is of triangle-like shape, and its top portion looks a little like a hilly landscape. More specifically, the following hold:

- (h1) $P(v_1) = (0, 0)$ and $P(v_2) = (2k - 4, 0)$.
- (h2) $C_k = w_1, w_2, \dots, w_m$, for some m , where $w_1 = v_1$, $w_m = v_2$, and $x(w_1) < x(w_2) < \dots < x(w_m)$.
- (h3) The slope of each segment $(P(w_i), P(w_{i+1}))$, for $i = 1, \dots, m - 1$, is either $+1$ or -1 .

Assume we have carried out $k - 1$ steps, and (h1)–(h3) hold. We now want to add v_k to the drawing. By the canonical ordering, we can assume that v_k is such that its neighbors on C_{k-1} are consecutive, and we can therefore denote them by w_p, \dots, w_q . Here is how to add v_k :

for each $v \in \bigcup_{i=p}^q L(w_i)$ **do**
 $x(v) := x(v) + 2;$ (i.e., move these points by 2 to the right)
for each $v \in \bigcup_{i=p+1}^{q-1} L(w_i)$ **do**
 $x(v) := x(v) + 1;$ (i.e., move these points by 1 to the right)
 $P(v_k) := \mu(P(w_p), P(w_q));$
 $L(v_k) := \{v_k\} \cup \bigcup_{i=p+1}^{q-1} L(w_i).$

Now, by (h3) we know that if w_i and w_j are any two vertices on the contour, and $I = P(w_i)$ and $J = P(w_j)$ are their current positions on the grid, then $MD(I, J)$ is even. As a result of this and the remark made earlier, $\mu(P(w_p), P(w_q))$ is always a grid point.

By moving some of the points $P(w_i)$ to the right, we ensure that all v_k 's neighbors will be visible from $P(v_k)$. With each vertex v that moves we also

move the set $L(v)$, consisting of the vertices that reside “below” it. This is needed to keep the part that has already been drawn without crossings from having crossings inadvertently introduced. For details, see Lemma 2 of [CP90].

A linear time implementation of this part of the algorithm is described in [CP90]. The basic idea is to maintain the sets $L(v)$ as trees rooted at v . At step k , the offspring of v_k are the vertices w_{p+1}, \dots, w_{q-1} , which are the roots of the trees $L(w_{p+1}), \dots, L(w_{q-1})$. The trees are implemented as binary trees, using *leftson* to hold the first offspring of a vertex, and *rightson* to hold the first sibling to the right of the vertex; see Section 2.3.2 of [Knu68]. To achieve constant time for updating this structure at step k , the contour chain is kept in the *rightson* array. We then update as follows:

```

if  $w_{q-1} \neq w_p$ 
    then  $leftson(v_k) := w_{p+1}$ 
    else  $leftson(v_k) := nil$ 
if  $w_{q-1} \neq w_p$ 
    then  $rightson(w_{q-1}) := nil;$ 

```

All other *rightson* connections are inherited automatically from the contour.

The contour chain is updated as follows:

```

 $rightson(w_p) := v_k;$ 
 $rightson(v_k) := w_q;$ 

```

The calculation of the x coordinate of v_k is carried out relative to that of w_p , and at the end of the algorithm these relative coordinates are translated into real ones by a single traversal of the binary tree. Since the vertices of the graph are processed according to their canonical ordering, a planar drawing is guaranteed.

3 The new algorithm

The main difference between our drawing algorithm and the original one is in the canonical ordering step; the placement step is much the same.

First, we note that the two requirements of the canonical ordering (see Def. 2.1) cannot be fulfilled when the graph is not triangulated. The first of these, clause 2.1(a), requires G_k to be biconnected. However, if we take a cycle on n vertices as input, any possible G_{n-1} will be a path, which is

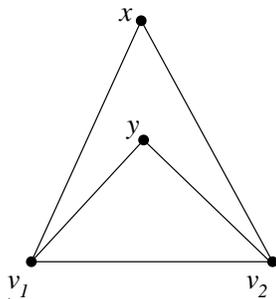


Figure 3: Example of a non-triangulated graph.

not biconnected. The second, clause 2.1(b), requires v_{k+1} to have consecutive neighbors on the path $C_k - (v_1, v_2)$ (the contour). However, consider Fig. 3, for example. The exterior face is v_1, v_2, x , and in this case the canonical ordering must have $v_4 = x$, which implies $C_3 = v_1, y, v_2$. But the neighbors of v_4 do not form a consecutive interval on C_3 , since y is not a neighbor of x .

What this means is that we need a new definition for the canonical ordering, which will enable us to draw the graph vertex by vertex in the placement step. We call the result of our new definition the *biconnected canonical ordering*

3.1 The right hand walk

The new algorithm works on the level of faces of the input graph, and uses a “right hand walk” around faces.

For the rest of the paper, G will be regarded as a directed graph, by viewing each undirected edge (u, v) as the pair of directed edges (u, v) and (v, u) , both of which are represented in the drawings by the same straight line segment uv connecting u and v .

Given an orientation for the edge (u, v) , we may speak of the *right* face and the *left* face of (u, v) . Note that these might be the same, as in the case where v has no incident edges other than (u, v) .

Let G be a planar connected graph drawn on the plane. Assuming G is connected, the boundary of each face in the drawing consists of a single connected polygonal line. The algorithm below produces a *boundary list* for each face, containing these polygonal lines in order.

The right hand walk:

mark all edges of G as unvisited;

while there are unvisited edges **do**

choose any unvisited edge (u, v) and initialize a new list b with

```

     $v_0 = u$ , and  $v_1 = v$ ;
  set  $i = 1$ ;
  repeat
    take as  $v_{i+1}$  the vertex immediately following  $v_{i-1}$  in the
      counter-clockwise circular ordering of neighbors around  $v_i$ ;
    add  $v_{i+1}$  to the list  $b$ ;
    mark the edge  $(v_i, v_{i+1})$  as visited;
    set  $i = i + 1$ ;
  until  $(v_i, v_{i+1}) = (v_0, v_1)$ ;
  close the list  $b$ ;
end-while

```

Execution of this algorithm can be viewed as a person walking along the edges of the graph, continuously choosing the rightmost option at every vertex. Thus, the resulting list $b(f)$ represents the boundary of a face f in a clockwise direction.

Clearly the right face of a directed edge (u, v) is also the left face of the dual edge (v, u) . Thus, if f is a face and $b(f) = v_0, v_1, \dots, v_m$ is the list produced by the right hand walk, the reversed list is another representation of the boundary of f , traversing it in a counter-clockwise fashion, and f is the left face of each of the edges (v_i, v_{i-1}) . We refer to the reversed lists as *counter-clockwise boundary lists*, while the lists produced by the algorithm are *clockwise boundary lists*, or just *boundary lists* for short. Obviously, the counter-clockwise lists can be obtained by a *left* hand walk algorithm, taking v_{i+1} to be the vertex following v_{i-1} in the *clockwise* circular ordering around v_i .

Each directed edge appears in exactly one boundary list. An undirected edge might appear in two different boundary lists, once in each direction, or it might appear in the same boundary list in both directions. As far as vertices go, unless v is a cut-vertex of G , it appears at most once in each boundary list. If v is a cut-vertex, each one of the boundary lists corresponding to the components of G that include v will contain v more than once.

Note that to construct the boundary lists we do not need the planar drawing itself; all we need is a planar embedding, as we only use the circular ordering of neighbors around each vertex. As mentioned earlier, a planar embedding can be found by the linear time algorithm of [CNAO85].

3.2 The biconnected canonical ordering

Let G be a biconnected planar graph drawn in the plane. Let G_k be a connected subgraph of G , and let $C_k = w_1, w_2, \dots, w_m$ be the counter-clockwise boundary

list of the exterior face of G_k (we call C_k the *contour* of G_k). Let v be a vertex in $G - G_k$ that lies in the exterior face of G_k , and which has exactly one neighbor in G_k . Note that, by planarity, that neighbor must lie on C_k (the contour of G_k), and we can thus assume it is w_i for some $i, 1 \leq i \leq m$.

Definition 3.1

- (a) We say that v has a right support if v immediately follows w_{i+1} in the counter-clockwise circular ordering around w_i ; it has a left support if v immediately precedes w_{i-1} in the counter-clockwise circular ordering around w_i .
- (b) We say that v has a legal support on C_k if: $i = 1$ and v has a right support, or $i = m$ and v has a left support, or $1 < i < m$ and v has a left support or a right support.

Note that since C_k is cyclic in nature, the starting point of the list, w_1 , can be fixed arbitrarily along C_k .

We now define the biconnected canonical ordering, as follows:

Definition 3.2 Let G be a biconnected planar graph drawn in the plane, and let (u, v) be an edge on the clockwise boundary list of its exterior face. A biconnected canonical ordering is a labeling of the vertices of G in a sequence v_1, \dots, v_n , such that $v_1 = u$ and $v_2 = v$, and for every $2 \leq k \leq n$ the following hold:

- (a) Let G_k be the subgraph of G induced by v_1, \dots, v_k . Then G_k is connected, and the edge (v_2, v_1) is on C_k , the contour of G_k . Fix w_1 to be v_1 , so that we write C_k as $w_1, w_2, \dots, w_m = v_2$.
- (b) All vertices in $G - G_k$ lie within the exterior face of G_k .
- (c) For $k > 2$, the vertex v_k has one or more neighbors in G_{k-1} . If v_k has exactly one neighbor in G_{k-1} , then it has a legal support on C_k .

This definition of the canonical ordering is a generalization of the original version to the case of non-triangulated graphs. For triangulated graphs it can be seen to be equivalent to that of Def. 2.1.

We now describe an algorithm that finds a biconnected canonical ordering. The algorithm can be regarded as a generalization of the original algorithm for canonical orderings described in Section 2; for triangulated graphs they perform similar steps. However, its proof of correctness, given in Section 4.1, is quite different.

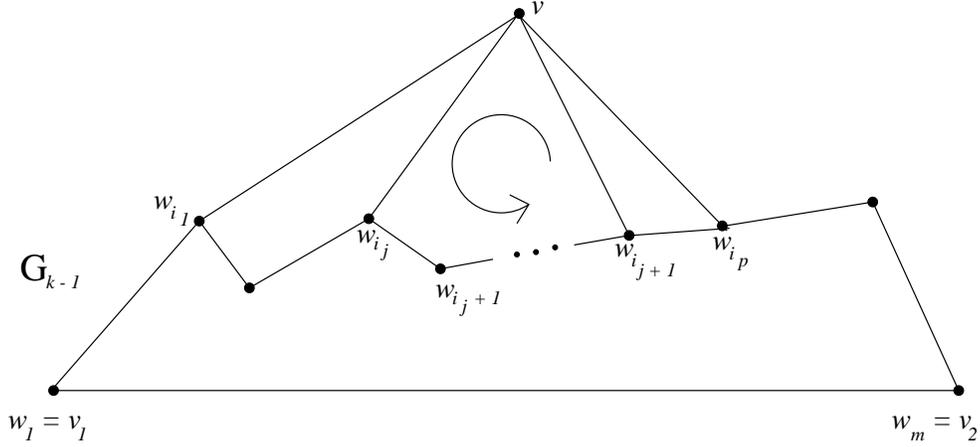


Figure 4: Illustration for Proposition 3.3.

3.3 Preliminaries

Before we proceed, we need to establish some facts. From now on, we will refer only to counter-clockwise boundary lists, and often omit their direction. Assume we have carried out $k - 1$ steps, and have obtained an ordering on $k - 1$ vertices satisfying the conditions of the biconnected canonical ordering. By induction on the k of Def. 3.2(c), since for every $j = 2, \dots, k - 1$, v_j has a neighbor in G_{j-1} , we know that G_{k-1} is connected.

Let C_{k-1} be the contour of G_{k-1} , and let us list it as $v_1 = w_1, w_2, \dots, w_m = v_2$ as described in Def 3.2(a). Now, let v be a vertex outside G_{k-1} , which has at least one neighbor in G_{k-1} . Then, by Def. 3.2(b), v is in the exterior face of G_{k-1} , and therefore lies on the exterior boundary of the larger graph $G_{k-1} \cup v$.¹ Since G_{k-1} is connected, v is not a cut-vertex of $G_{k-1} \cup v$, and it therefore appears exactly once along the boundary list of the exterior face of $G_{k-1} \cup v$. Let u be the vertex preceding v in the boundary list of the exterior face of $G_{k-1} \cup v$. By the planarity of G , the neighbors of v in G_{k-1} must all reside on C_{k-1} , and thus u is really w_{i_1} for some appropriate $1 \leq i_1 \leq m$. If v has p neighbors on C_{k-1} , we can list them similarly in their counter-clockwise circular ordering around v , as $w_{i_1}, w_{i_2}, \dots, w_{i_p}$.

Note that a vertex x may appear more than once in the list C_{k-1} . This could happen if it is a cut-vertex of the graph G_{k-1} , and C_{k-1} ‘goes around’ a component attached to x . Thus, if x is one of the neighbors of v , we should be more precise in defining the index i_j that satisfies $x = w_{i_j}$. We shall do this by considering the circular ordering around x . Obviously, there is exactly one index q that satisfies $x = w_q$, and such that on the clockwise circular ordering

¹We write $G_{k-1} \cup v$ for the subgraph of G induced by the vertices v_1, \dots, v_k, v .

around x the order is w_{q-1}, v, w_{q+1} . This q is taken to be the above i_j .

Proposition 3.3 *Let $w_{i_1}, w_{i_2}, \dots, w_{i_p}$ be the neighbors of v on the contour $C_{k-1} = w_1, w_2, \dots, w_m$, defined as above. Then $i_1 < i_2 < \dots < i_p$ (see Fig. 4).*

(The meaning of this proposition is that the circular ordering around v coincides with the order along the boundary list C_{k-1} . Bearing in mind that boundary lists are circular in nature, the proposition also states that at the particular starting point chosen for C_{k-1} (which is w_1), the list of neighbors of v on C_{k-1} does not ‘wrap around’ the circular list C_{k-1} .)

Proof. C_{k-1} was defined to be a counter-clockwise boundary list, and can be seen to be an output of the left hand walk performed on the graph G_{k-1} . In other words, if w_{i-1}, w_i, w_{i+1} is a fragment of C_{k-1} , then in G_{k-1} the vertex w_{i+1} immediately follows w_{i-1} on the clockwise circular ordering of neighbors around w_i .

Consider applying the algorithm to the graph $G_{k-1} \cup v$, and initializing a left hand walk with an edge $(w_{i_{j+1}}, v)$ satisfying $1 \leq j \leq p-1$. Mark by $b(f)$ the boundary list being constructed. The first step of the algorithm adds w_{i_j} to the list $b(f)$, since by our definitions w_{i_j} follows $w_{i_{j+1}}$ in the circular ordering around v . The next step adds $w_{i_{j+1}}$, since by our choice of the index i_j , the vertex following v in the clockwise circular ordering around w_{i_j} is $w_{i_{j+1}}$.

The algorithm now carries out the very same steps that were carried out when constructing C_{k-1} , thus creating the list $w_{i_j}, w_{i_{j+1}}, w_{i_{j+2}}, \dots$, until the initial edge $(w_{i_{j+1}}, v)$ is reached again. This will be the only point along the traversal that reaches v , since v appears at most once in every boundary list because it is not a cut-vertex of $G_{k-1} \cup v$.

Hence, we have established the fact that no other neighbor w_{i_q} of v appears between w_{i_j} and $w_{i_{j+1}}$, meaning that these two are consecutive on a circular list of neighbors of v ordered by their indices along C_{k-1} .

It remains to show that in the traversal along the fragment of C_{k-1} from w_{i_j} to $w_{i_{j+1}}$, no wrapping around occurs. This will imply $i_j < i_{j+1}$, as needed. Accordingly, if we assume that the walk does pass beyond w_m , the edge (v_2, v_1) would be included in the boundary list $b(f)$ being constructed. However, this edge resides on the boundary of the exterior face of the entire graph G , and therefore also on the boundary of the exterior face of $G_{k-1} \cup v$. This means that $b(f)$ is the boundary list of the exterior face of $G_{k-1} \cup v$, and as such it contains the edge (w_{i_1}, v) , by our definition of w_{i_1} . But we initialized $b(f)$ with the edge $(w_{i_{j+1}}, v)$, and we know that v appears exactly once in $b(f)$. Since $j+1$ cannot be equal to 1 in the range of j 's under discussion, this is a contradiction.

□

3.4 The algorithm

The algorithm employs the following one-dimensional arrays: A , indexed by the faces of the graph, and N and F , indexed by the vertices. During execution of the k th stage of the algorithm, $A(f)$ will contain the number of edges from $b(f)$ that are in G_{k-1} . Also, $N(v)$ will contain the number of neighbors of vertex v in G_{k-1} , and $F(v)$ will be the number of ‘ready’ faces that have v as their only vertex outside G_{k-1} . Here, a face f , that is not the exterior face of G , is said to be *ready* if $A(f) = |b(f)| - 2$, i.e., $b(f)$ has only two edges not in G_{k-1} . (This can only mean that there is a single vertex in $b(f) - G_{k-1}$ that is incident to these two edges.)

Before we get into the algorithm itself, we establish some facts about the contents of N and F , using the following notational conventions. Let v be a vertex not in G_{k-1} . Denote $N(v)$ by p , and let w_{i_1}, \dots, w_{i_p} be the neighbors of v on C_{k-1} . Also, let f_j be the left face of the edge (w_{i_j}, v) , for $1 \leq j \leq p$. Finally, let L_v be the circular ordering of all neighbors of v .

Proposition 3.4

- (i) $N(v) > F(v)$.
- (ii) $N(v) = F(v) + 1$ iff all the faces f_j , for $2 \leq j \leq p$, are ready.
- (iii) If $N(v) = F(v) + 1$, then the neighbors of v in G_{k-1} form a single interval in the list L_v .

Proof. (i) $F(v)$ contains the ready faces that have v as their sole vertex outside G_{k-1} . Thus, the boundary list of such a face must contain an edge of the form (w_{i_j}, v) , with w_{i_j} in G_{k-1} . This implies that each of these ready faces is the left face of one of the edges (w_{i_j}, v) , so that the set of ready faces accounted for in $F(v)$ is a subset of the p faces f_1, \dots, f_p . Hence, $N(v) = p \geq F(v)$. To prove $N(v) > F(v)$, we will show that f_1 cannot be a ready face.

Note that $b(f_1)$ is the boundary list of f_1 on G , so that $b(f_1)$ contains the edge (w_{i_1}, v) . If $b(f_1)$ has only v as a vertex not in G_{k-1} , it is entirely contained in $G_{k-1} \cup v$. By our choice of w_{i_1} , the left face of the edge (w_{i_1}, v) in the subgraph $G_{k-1} \cup v$ is the exterior face of this subgraph. Now, $G_{k-1} \cup v$ contains the edge (v_2, v_1) , which is on the boundary of the exterior face of the entire graph. Thus, (v_2, v_1) is on the boundary of the exterior face of the subgraph $G_{k-1} \cup v$ too, and it therefore belongs to $b(f_1)$. However, in the entire graph G , the face whose boundary list contains the edge (v_2, v_1) is the exterior face, which means that f_1 must be the exterior face of G . But recall that the exterior face of the entire graph G was excluded from the definition of a ready face. Hence, f_1 cannot be a ready face.

(ii) Assume $N(v) = F(v) + 1$. The above counting shows that each of the $p-1$ faces f_2, \dots, f_p must be ready. Conversely, since the set of faces accounted for in $F(v)$ consists of those faces from among f_2, \dots, f_p that are ready, then if they are *all* ready we must have $F(v) = p - 1$, which is $N(v) = F(v) + 1$.

(iii) Let $N(v) = F(v) + 1$, and assume that L_v contains a fragment of the form $w_{i_{j-1}}, \dots, u, w_{i_j}$ for some $2 \leq j \leq p$, meaning that there are vertices that separate a pair of adjacent neighbors of v in G_{k-1} . Recall that w_{i_1}, \dots, w_{i_p} is the list of neighbors of v in G_{k-1} , ordered counter-clockwise around v . Thus, u cannot be in G_{k-1} (otherwise it would be one of $w_{i_{j-1}}$ or w_{i_j}). Now, since u follows w_{i_j} in the clockwise circular ordering around v , the boundary list $b(f_j)$ must contain the edges (w_{i_j}, v) and (v, u) . This implies that f_j has two vertices outside G_{k-1} (which are v and u), and therefore it cannot be a ready face. This contradicts the assumption $N(v) = F(v) + 1$, thus completing the proof. □

Now to the algorithm. Assume we have just chosen v_k . Here is how the arrays are updated:

(1) update v_k 's neighbors:

For each neighbor v of v_k that is outside G_k , increment $N(v)$ by 1.

(2) update faces:

There are two faces to update: The left face of the edge (w_{i_1}, v_k) , which is f_1 , and the right face of (w_{i_p}, v_k) , which we shall call f_{p+1} . For these, increment $A(f_1)$ and $A(f_{p+1})$ by 1. (Recall that w_{i_1}, \dots, w_{i_p} is the ordered list of neighbors of v_k on C_{k-1} . Also, it might be the case that $f_1 = f_{p+1}$.)

(3) update ready faces:

If a face f becomes ready as a result of (2), find the only vertex v along $b(f)$ that is outside G_k , and increment $F(v)$ by 1.

Here now is the algorithm for building the new biconnected canonical ordering:

initialization:

- initialize all three arrays, A , N and F , to 0;
- take as (v_1, v_2) any edge on the boundary of the exterior face of G ;
- initialize a list of vertices with v_1 and v_2 , and update their neighbors
 - as in (1) above;
- set $A(f)$ to 1 for f , the left face of (v_1, v_2) ;
- If f is a triangle with vertices v_1, v_2, v_3 , set $F(v_3)$ to 1, since f is ready.

for k from 3 to n do

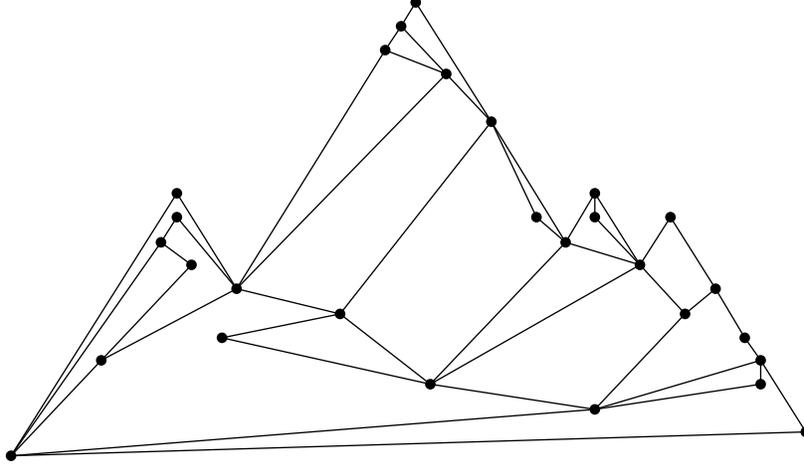


Figure 5: The graph from Fig. 1 as drawn by our algorithm.

if there is a vertex v not on the list, with $N(v) \geq 2$ and
 $N(v) = F(v) + 1$
 then add it to the list as v_k ;
 else find a vertex v not on the list, with legal support and
 $N(v) = 1$, and add it to the list as v_k ;
 update the data structures as in (1) – (3) above for v_k .
end-for

So much for computing the canonical ordering. The placement step of our algorithm is very similar to the original one of [FPP88], described in Section 2. The original placement step relies on the fact that at each stage the vertex v_k is in the exterior face of G_{k-1} and has neighbors on C_{k-1} . These conditions are guaranteed in the biconnected canonical ordering too. Our algorithm uses the same *leftson* and *rightson* arrays, indexed by the vertices.

One difference worth mentioning is the update for the case where the vertex has a support. Assume v_k has only w_i as a neighbor on G_{k-1} , and it should be drawn using a right support. In this case, we update the data structures as if there were a real edge (v_k, w_{i+1}) ; this means that the contour as it appears in the *rightson* chain contains the segment w_i, v_k, w_{i+1} . The case for a left support is analogous.

3.5 Remarks

- By Def. 3.2(c), the vertex v_k is drawn only after at least one of its neighbors has already been drawn in G_{k-1} . This prevents the situation

shown in Figure 1. Fig. 5 shows the same graph as drawn by our algorithm.

- The drawing obtained in the original algorithm of [FPP88] is of size bounded by $(2n - 4) \times (n - 2)$. This follows from the fact that the algorithm starts the placement step with the edge (v_1, v_2) drawn with length 2, and it increases this length by 2 at each step, ending with length $2n - 4$. The entire drawing can be enclosed in a triangle whose base is the edge (v_1, v_2) , and whose sides emanate from v_1 and v_2 with slopes $+1$ and -1 , respectively. Hence, the drawing's maximal height is $n - 2$. All this is true for our algorithm too, so that the same bounds apply.
- Our canonical ordering requires the graph to be biconnected. There are examples of planar non-biconnected graphs for which no biconnected canonical ordering exists, a fact that is also reflected in the proof of correctness below. However, every graph can be made biconnected by adding dummy edges, as follows. (i) given any two disconnected components, add a dummy edge to connect arbitrarily chosen vertices, one in each of them; (ii) given two components with a common cut-vertex v , add a dummy edge that connects arbitrary neighbors of v , one from each component. Deciding connectivity and biconnectivity, and identifying biconnected components can all be done in linear time. In our system, we use an algorithm described in [HT73] as a preliminary step for the drawing algorithm; after completing the placement step we remove the dummy edges.

4 Analysis of the algorithm

4.1 Correctness

The following theorem establishes correctness of the computation of a biconnected canonical ordering, the only part of our algorithm that is sufficiently new and subtle so as to require special analysis.

Theorem 4.1 *For each $2 \leq k \leq n$, let v_1, \dots, v_k be the sequence of vertices generated by the algorithm up to stage k . Then (i) Conditions (a)–(c) of Def. 3.2 are satisfied, and (ii) there exists a vertex v outside G_k , such that either $N(v) \geq 2$ and $N(v) = F(v) + 1$, or $N(v) = 1$ and v has a legal support.*

Proof. We proceed by induction on k . For $k = 2$, G_2 consists of the single edge (v_1, v_2) . Conditions (a) and (b) of Def. 3.2 hold immediately, while (c) is vacuous. For $k \geq 2$, we prove the existence claim (ii) first.

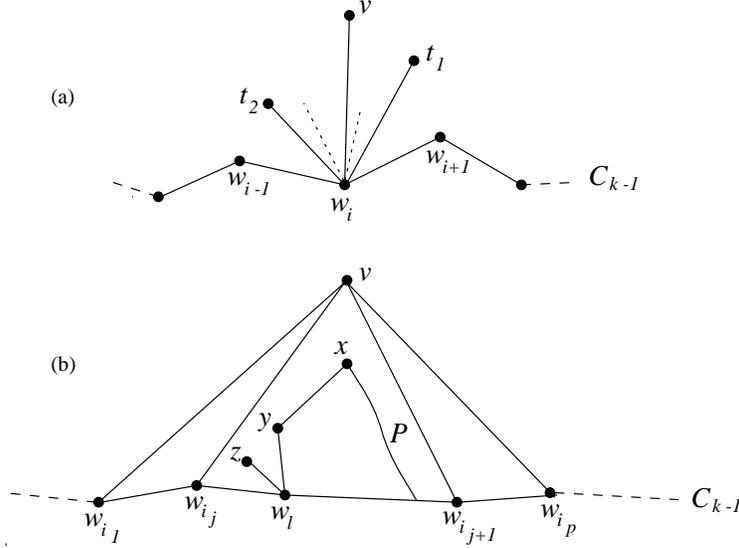


Figure 6: Illustrations for proof of theorem 4.1.

Assume we have built the sequence v_1, \dots, v_{k-1} such that conditions (a)–(c) of Def. 3.2 hold. We prove that there exists a vertex v as in (ii). The graph G is connected, so there are vertices outside G_{k-1} with $N(v) > 0$. First, assume that all these vertices have $N(v) = 1$. Let v be such a vertex, and let w_i be its sole neighbor on C_{k-1} . If v itself does not have a legal support, then the vertex t_1 that immediately follows w_{i+1} on the counter-clockwise circular ordering of neighbors around w_i satisfies $N(t_1) = 1$ and has a right support, and the vertex t_2 that immediately precedes w_{i-1} , satisfies $N(t_2) = 1$ and has a left support. See Fig. 6(a). In this case, one of t_1 or t_2 must have a legal support, and it can therefore be chosen to be v_k .

The more difficult case is when there are vertices with $N(v) \geq 2$, and in discussing it we shall need the notion of a *span*.

Let v be a vertex outside G_{k-1} , which has p neighbors in G_{k-1} . As before, we let $C_{k-1} = w_1, w_2, \dots, w_m$, where $w_1 = v_1$ and $w_m = v_2$. By Prop. 3.3 we can denote the p neighbors of v by $w_{i_1}, w_{i_2}, \dots, w_{i_p}$, as they constitute a sub-series of C_{k-1} . The fragment of C_{k-1} between w_{i_1} and w_{i_p} is called the *span* of v on C_{k-1} . The *length* of the span is defined to be $i_p - i_1$ (which is the number of vertices therein).

Now assume that there are vertices in the graph with $N(v) \geq 2$, and assume, for contradiction, that none of these vertices satisfy $N(v) = F(v) + 1$ as required in the theorem. We choose v to be a vertex with $N(v) \geq 2$ whose span on C_{k-1} is the shortest among the vertices with $N(v) \geq 2$. Let $w_{i_1}, w_{i_1+1}, \dots, w_{i_p}$ be the span of v on C_{k-1} . If there are two such vertices

with equal span length, but with different spans, pick one of them arbitrarily; If the two have the same span sequence, pick the one closer to w_{i_1+1} on the circular ordering around w_{i_1} , taken counter-clockwise (i.e., the one closer to the contour C_{k-1}).

By our assumption, v satisfies $N(v) > F(v) + 1$, implying that one or more of the faces f_2, \dots, f_p is not ready. Let f_{j+1} , for some $1 \leq j \leq p - 1$, be one of these, and let H be the subgraph of G induced by the closed polygonal line $w_{i_j}, w_{i_{j+1}}, \dots, w_{i_{j+1}}, v$. As mentioned earlier, this does not have to be a simple cycle, since some w_i might appear twice therein. Let e be the left face of the edge $(w_{i_{j+1}}, v)$ in the subgraph H . If e was a face in the original graph G too, it must have been ready, since it has only two edges outside G_{k-1} . However, by our assumptions, this is impossible. Consequently, the region e in G is not connected, and hence there is a path P in G that divides e in two.

We now claim that P cannot be a single edge. First, if P were an edge connecting v to some vertex w on C_{k-1} , then w must be between w_{i_j} and $w_{i_{j+1}}$, contradicting the order in the span of v . Second, if P were an edge connecting two w_i 's, then P would belong to the subgraph G_{k-1} , and as such, it could not be inside the exterior face of G_{k-1} . This is impossible, however, since by the (b) part of the induction hypothesis v is in the exterior face of G_{k-1} , so that the region e (which includes P) is in that exterior face too.

Therefore, P must have at least one internal vertex, call it x . Since we assume that G is biconnected, there is a path Q in G that connects x and w_{i_j} without passing through v (otherwise v is a cut-vertex of G). Let w_l be the first point on Q (when coming from x) that is in G_{k-1} , and let y be the point preceding w_l on Q (see Fig. 6(b)). Since w_l is on the boundary of e and x is in e , y must also be in e .

Now, if y satisfies $N(y) > 1$, we have found in y a contradiction to the minimality assumption on v . Here is why: First, we claim that y 's span is part of v 's. To see this, note that every edge emanating from y must reside entirely in e (by planarity), and it therefore can only lead to points in e or on its boundary. The intersection of the boundary of e with C_{k-1} is the sequence $S = w_{i_j}, \dots, w_{i_{j+1}}$, and hence y 's span can only be a subsequence of S , while S itself is part of v 's span. This implies that the length of y 's span cannot exceed the length of v 's; if it is strictly smaller, we have a contradiction to the choice of v . If v and y happen to have exactly the same span, the circular ordering around w_{i_1} (taken counter-clockwise) must be $w_{i_1+1}, \dots, y, \dots, v$, which contradicts the second part of the minimality assumption on v .

If $N(y) = 1$, so that y has only one neighbor on C_{k-1} , we find our contradiction a little further down the line, so to speak. Again, we have two cases to consider. If $w_l \neq w_{i_j}$, let z be the neighbor of w_l immediately following w_{l-1} along the clockwise circular ordering of neighbors around w_l . This z might be

y itself, or a neighbor of w_l that is closer to w_{l-1} . Now, by planarity, z is also in e , and if $N(z) > 1$, we have in z a contradiction to the minimality assumption on v , by the same arguments as above. If $N(z) = 1$, z has a legal left support on w_{l-1} , and therefore it can be chosen as v_k , which again contradicts our assumptions.

If $w_l = w_{i_j}$, we take z to be the neighbor of w_l immediately following w_{l+1} along the counter-clockwise circular ordering of neighbors around w_l . This z is in e , and contradicts our assumptions similarly (using a right support for the case $N(z) = 1$).

This completes the proof of existence. We now show that each of the three parts of Def. 3.2 holds.

- (a): G_k contains the edge (v_2, v_1) . In the original graph G , this edge is on the boundary of the exterior face. If in G_k the edge (v_2, v_1) was on the boundary of an interior face, it would have to remain interior in G too, since the added vertices when going from G_k to G can only split faces, and a bounded face will split into bounded faces only.
- (b): Consider the region added to the interior of G_{k-1} when the vertex v_k is added. If $N(v_k) = 1$, a single edge was added to G_{k-1} , going from the exterior boundary of G_{k-1} to a point in the exterior face. This added no region to the interior of G_{k-1} . By the induction hypotheses, all vertices in $G - G_{k-1}$ were in the exterior face of G_{k-1} , and therefore all vertices in $G - G_k$ are in the exterior face of G_k . If $N(v_k) > 1$, we added to the interior of G_{k-1} a region taken from its exterior face, and we must show that this region does not contain vertices from $G - G_k$. In going from G_{k-1} to G_k we added the vertex v_k and all the edges (v_k, w_{i_j}) with w_{i_j} on C_{k-1} . The region added to the interior of G_{k-1} contains the faces f_2, \dots, f_p , which are the left faces of the edges $(w_{i_2}, v_k), \dots, (w_{i_p}, v_k)$ in G_k . However, by our choice of v_k with $N(v_k) = F(v_k) + 1$, we ensured that these are all actual faces of G , and as such they have no vertices of G inside them.
- (c): That v_k has at least one neighbor in G_{k-1} is straightforward, since the v_k selected by the algorithm satisfies $N(v_k) \geq 1$. Also, the algorithm chooses v_k with one neighbor only if it has a legal support.

□

4.2 Implementation and running time

We would now like to show that if we are given the circular orderings of neighbors around the vertices and the boundary lists of the faces of G , the algorithm

described in Section 3 can be implemented to construct a biconnected canonical ordering in linear time, using a linear amount of memory (to include the representation of the inputs too).

Here is how the data structures for the inputs are set up. Every directed edge (u, v) has pointers to the next and previous edges on the circular ordering around u , and pointers to the next and previous edges on the boundary list of the left face f that contains the edge. It also has a pointer to the opposite edge (v, u) , and a pointer to f . Each vertex has a pointer to the list representing the circular ordering around it, and each face has a pointer to its boundary list.

A vertex that satisfies the conditions of Theorem 4.1(ii) is called *ready*. The existence claim in the theorem showed that the set of ready vertices never becomes empty during execution, and at each stage the algorithm chooses one of the ready vertices as the next v_k . To handle this set of vertices we use a doubly linked queue, termed the *ready queue*, which makes it possible to insert and remove a vertex in constant time.

At each stage of the algorithm, a vertex is removed from the ready queue and is taken to be v_k . After doing so, the following steps are carried out:

Updating N : For every u that is a neighbor of v_k , the entry $N(u)$ is updated by running through the entire list of neighbors of v_k , and incrementing $N(u)$ by 1 for each u not yet in G_k . If u is in the ready queue, then after incrementing $N(u)$ it is removed, since $N(u) = F(u) + 1$ no longer holds. If u satisfies $N(u) = 1$, we check if it has a legal support, as follows. We find the two faces incident with the dual edges (v_k, u) and (u, v_k) , inspect the two vertices on either side of v_k along these faces, and check if either of them is already in G_k . We also need to make sure the support is legal. If u has a legal support we add it to the ready queue.

Updating A : Only the two endmost faces need to be updated — f_1 , the left face of the edge (w_{i_1}, v_k) , and f_{p+1} , the right face of the edge (w_{i_p}, v_k) . For these, $A(f_1)$ and $A(f_{p+1})$ are incremented by 1. To find these edges, we use Prop. 3.4, to the effect that in L_{v_k} (the list of neighbors around v_k) the vertices that belong to G_{k-1} , i.e., $w_{i_1} \dots w_{i_p}$, form a single interval, whose ends are thus easy to find. Now, if one of these two faces reached a situation where $A(f) = b(f) - 2$, it is declared ready, and a traversal of its boundary list is carried out to find the vertex v not in G_k . We then increment $F(v)$ by one, and if now $N(v) = F(v) + 1$, the vertex v is inserted into the ready queue.

To analyze the running time, note that an entire list of neighbors is considered only when a vertex is chosen as the next v_k . In total, this means that

we traverse every edge twice, once in each direction. We also traverse edges when we search for the vertex v in a ready face. This is done once per face, when the face becomes ready. Since an edge appears in at most two faces, each edge is traversed twice more, giving a total of four traversals per edge. Now, by Euler’s formula, a planar graph with n vertices and e edges satisfies $e \leq 3n - 3$, which implies that the total running time is $O(n)$.

As for memory space requirements, we have arrays with entries for vertices, edges and faces. Since another consequence of Euler’s formula is that the number of faces is also $O(n)$, all these are linear in n too. The queue we use for ready vertices does not hold a vertex twice, and hence its size is also linear in n .

As mentioned at the end of Section 3.1, the boundary lists can be derived from the embedding using a right hand walk. Thus, it suffices to require that the input contains lists representing the embedding, and to include a preliminary step in the algorithm that constructs the boundary lists. Given an edge (v_{i-1}, v_i) , the beginning of the **repeat–until** loop in the right hand walk algorithm of Section 3.1 reads:

take as v_{i+1} the vertex immediately following v_{i-1} in the counter–clockwise circular ordering of neighbors around v_i .

To implement this we have to establish a mapping, which, given an edge (u, v) , returns the dual edge (v, u) . In [LEDA91] this mapping is constructed in linear time and space (using bucket–sort). With this in mind, the entire right hand walk algorithm can be implemented in linear time and space in a straightforward way, and can be integrated as the first step of our algorithm without affecting the asymptotic complexity.

Acknowledgements: We wish to thank M. Chrobak and the two referees for their helpful suggestions.

References

- [CNAO85] Chiba, N., T. Nishizeki, S. Abe, and T. Ozawa, “A Linear Algorithm for Embedding Planar Graphs Using PQ -trees”, *J. Comput. Sys. Sci.* **30** (1985), 54–76.
- [CP90] Chrobak, M. and T.H. Payne, “A Linear Time Algorithm for Drawing a Planar Graph on a Grid”, *Inf. Proc. Lett.* **54** (1995), 241–246.
- [DH89] Davidson R. and D. Harel, “Drawing Graphs Nicely Using Simulated Annealing”, *ACM Transactions on Graphics*, October 1996,

to appear. (Also, Technical report, The Weizmann Institute of Science, Rehovot, Israel, 1989; revised 1992, 1993.)

- [FPP88] Fraysseix, H. de, J. Pach, and R. Pollack, “Small Sets Supporting Fáry Embeddings of Planar Graphs”, *Proc. 20th ACM Symp. on Theory of Comput.*, pp. 426–433, 1988.
- [HS94] Harel, D. and M. Sardas, “Randomized Graph Drawing with Heavy-Duty Preprocessing”, *J. Visual Lang. and Comput.*, **6** (1995), 233–253. (Also, *Proc. Workshop on Advanced Visual Interfaces*, ACM Press, New York, 1994, pp. 19–33.)
- [HT73] Hopcroft, J.E. and R.E. Tarjan, “Efficient Algorithms for Graph Manipulation”, *Comm. Assoc. Comput. Mach.* **16** (1973), 372–378.
- [Kan92] Kant, G., “Drawing Planar Graphs Using the *lmc*-Ordering” *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, 1992, pp. 101–110,
- [Knu68] Knuth, D.E., *The Art of Computer Programming; vol. 1: Fundamental Algorithms*, Addison Wesley, Reading, MA, 1968; 2nd ed., 1973.
- [LEDA91] “Library of Efficient Datatypes and Algorithms (LEDA)”, software package, Max-Planck Intitut für Informatik, Saarbrücker, 1991.
- [Mel96] Mehlhorn, K., and P. Mutzel “On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm”, Technical Report no. 117/94, Max-Planck-Institut für Informatik, Saarbrücken, 1994; to appear in *Algorithmica*.