October, 1993

The Hopcroft-Tarjan Planarity Algorithm

William Kocay*
Computer Science Department
University of Manitoba
Winnipeg, Manitoba, CANADA, R3T 2N2
e-mail: bkocay@cs.umanitoba.ca

Abstract

This is an expository article on the Hopcroft-Tarjan planarity algorithm. A graph-theoretic analysis of a version of the algorithm is presented. An explicit formula is given for determining which vertex to place first in the adjacency list of any vertex. The intention is to make the Hopcroft-Tarjan algorithm more accessible and intuitive than it currently is.

1. Introduction

Let G be a simple 2-connected graph with vertex set $V(G)$ and edge set $E(G)$. The number of vertices of G is denoted by n. If $u, v \in V(G)$, then $A(u)$ denotes the set of all vertices adjacent to u. $v \to u$ means that v is adjacent to u, so that $v \in A(u)$ (and since G is undirected, also $u \to v$). Refer to the book by Bondy and Murty [3] for other graph-theoretic terminology. Hopcroft and Tarjan [13] gave a linear-time algorithm to determine whether G is planar, using a depth-first search. The depth-first search computes two low-point arrays for the vertices, $L_1(u)$ and $L_2(u)$, and assigns a weight to the edges of G, where the weight is computed from the low-points. The edges incident on each $u \in V(G)$ are then ordered according to their weights. The algorithm then uses the revised graph in which incident edges have been ordered, and performs another depth-first search, the so-called *PathFinder* to embed the graph in the plane.

The purpose of this article is to explain exactly why the particular weighting scheme used for the edges is necessary, to elucidate why it works, and to suggest another, more intuitive way of embedding the graph in the plane, once the low-point depth-first search has been executed. The Hopcroft-Tarjan algorithm is complicated and subtle. For example, see the description of it in the paper of Williamson [20]. In [2], Di Battista, Eades, Tamassia, and Tollis state that "The known planarity algorithms

* This work was supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada.

that achieve linear time complexity are all difficult to understand and implement. This is a serious limitation for their use in practical systems. A simple and efficient algorithm for testing the planarity of a graph and constructing planar representations would be a significant contribution". It may not be possible to construct a simple planarity algorithm, but the graph-theoretic analysis of the algorithm presented here is intended to make the algorithm easier to understand and implement. A number of textbooks on graph theory and algorithms have been published since the paper of Hopcroft and Tarjan, many of which discuss planarity algorithms. The books by Even [9], Foulds [10], and Gould [12] all discuss the Hopcroft-Tarjan algorithm, but none of them attempt to prove that the algorithm works, because of the difficulty. The books by Gibbons [11], McHugh [16], Bondy and Murty [3], and Chartrand and Oellermann [5] all choose to describe the planarity algorithm of Demoucron, Malgrange, and Pertuiset [7] because it is conceptually much easier to understand than the Hopcroft-Tarjan algorithm. The book by Nishizeki and Chiba [17] describes the algorithm developed by Lempel, Even, Cederbaum [15], and Booth and Lueker [4], because it also is more intuitive. In the book by Williamson [19], the chapter on planarity ends with exercise 7.31. The reader is asked to "try and convince at least two friends that your algorithm works in linear time. A major hurdle will be the soporific effect on the friends". It is very easy to prove that a depth-first search based algorithm is linear. It is hard to prove that the Hopcroft-Tarjan algorithm works. This paper intends to make the Hopcroft-Tarjan algorithm more accessible and intuitive.

Consider a depth-first search of G. It constructs a rooted spanning tree T of G, called a DF-tree of G. The *root* node is the vertex from which the search was initiated. Each vertex $v \in V(G)$, except the root, has a unique ancestor vertex, being that vertex from which DFS(v) was called. Denote it by $a(v)$. Fig. 1 shows a DF-tree in a graph G. $a(v)$ is the vertex immediately above v in the tree. The depth-first search assigns a numbering to the vertices, the *depth-first* numbering, written $D(u)$, where $u \in V(G)$. It is the order in which the vertices are visited by the search. The vertices in Fig. 1 are numbered according to their DF-numbering. The edges of the tree are given by $E(T) = \{uv \mid u = a(v)\}$.

DFS(u: vertex)
{ extend a DF-search in a graph G from vertex u }
begin
  { *DFCount* is a global counter initially set to 0 }
  $DFCount := DFCount + 1$
  $D(u) := DFCount$

for each $v \to u$ do begin
{ initially, all $D(v)$ were set to 0 }
if $D(v) = 0$ then begin
$a(v) := u$
DFS(v)
end
end
end { DFS }

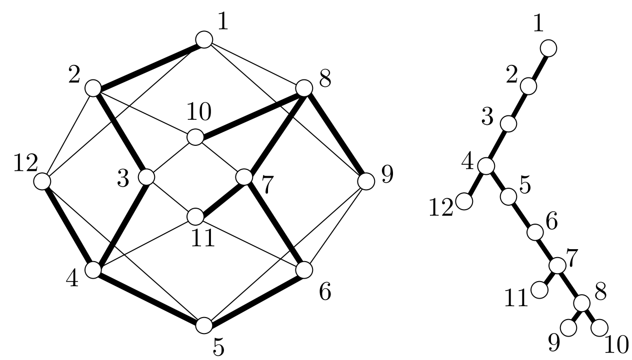

Fig. 1, A DFS in a graph

The tree T is drawn in Fig. 1 with its root at the top, the tree descends from the root. The *leaves* of T are the vertices with no descendants. Starting at any vertex v in T, the sequence $v_0 = v$, $v_1 = a(v_0)$, $v_2 = a(v_1)$, $\ldots$, is a sequence of vertices with decreasing DF-numbers, eventually reaching the root vertex of T. The depth-first search divides the edges of G into two categories. If uv is an edge with either $a(v) = u$ or $a(u) = v$, then uv is a *tree edge*. Otherwise uv is a *frond*.

Given any vertex u, write $S(u)$ for the set of all descendants of u, that is, all vertices v such that u occurs on the path from v to the root. Let $S^*(u) = \{u\} \cup S(u)$. Let $T(u) = \{w \in A(v) \mid v \in S^*(u)\}$. In words, $T(u)$ consists of all vertices adjacent to u or any descendant of u. If $a(v) = u$, the *branch* at u containing v is $B_u(v) = \{wx \mid w \in S^*(v)\}$, namely the set of all edges incident on v or any descendant of v. The edge $uv \in B_u(v)$ is called the *stem* of the branch. At this point it is convenient to follow [13] or [9] and renumber the vertices so that $D(u) = u$, that is, to refer to vertices directly by their DF-number. We then define

$L_1(u) = \min T(u)$
$L_2(u) = \min T(u) - \{L_1(u)\}$

Since G is 2-connected, every vertex has degree at least 2, so that $|T(u)| \geq 2$. It follows that the low-points are well-defined. The algorithm cannot refer to vertices directly by their DF-number, but must work with the graph as it is given. So the algorithm must compute the low-points as

$L_1(u) = x \in T(u)$, such that $D(x) = \min\{D(v) \mid v \in T(u)\}$
$L_2(u) = x \in T(u)$, such that $D(x) = \min\{D(v) \mid v \in T(u) - \{L_1(u)\}\}$

However, it is much easier to decsribe the algorithm if vertices can be referred to via their DF-number. This means that all comparisons of vertices such as "if $v < w \ldots$" must be evaluated via the DF-number, namely "if $D(v) < D(w) \ldots$", etc.

It is easy to modify the DFS to compute $L_1(u)$, and only slightly more difficult to modify it to compute $L_2(u)$ as well. See [13] or [9] for details. Suppose that $u = a(v)$. Then $u \in T(v)$. Therefore $L_1(v) \leq u$. It is a classic result of Hopcroft and Tarjan [1] that u is a cut-vertex of G if $L_1(v) = u$, when u is not the root node. Since G is assumed to be a 2-connected graph, it has no cut-vertices, so we can conclude that $L_1(v) < u$ whenever $u = a(v)$ and u is not the root node. It follows that $L_2(v) \leq u$.

2. Ordering the Edges

The graph G is stored by adjacency lists, namely $Adj[u]$ is a linked list containing all vertices adjacent to u. So $Adj[u]$ is an ordered list of adjacent vertices. If $v \in A(u)$, then the node in $Adj[u]$ corresponding to v represents the edge $uv \in E(G)$.

Hopcroft and Tarjan assign a weight to each edge. The weight is stored as a field in each node of the linked list. $wt_u[v]$ denotes the weight of edge uv in $Adj[u]$. It can be defined as follows. Suppose that $v \in A(u)$. $wt_u[v]$ is computed by DFS(u), by adding several statements.

$$wt_u[v] = \begin{cases} 2v, & \text{if } uv \text{ is a frond with } v < u \\ 2L_1(v), & \text{if } a(v) = u \text{ and } L_2(v) = u \\ 2L_1(v) + 1, & \text{if } a(v) = u \text{ and } L_2(v) < u \\ 2n + 1 & \text{otherwise} \end{cases}$$

The low-point depth-first search computes $D(u)$, $a(u)$, $L_1(u)$, $L_2(u)$, and $wt_u[v]$, for all $u \in V(G)$ and all $v \in A(u)$. Call this first DFS the *LowPtDFS*. Following it the edge-weights are in the range $1..2n+1$. The edges are then sorted by a bucket sort, and the adjacency lists $Adj[u]$ are reconstructed in order of increasing $wt_u[v]$. In the Hopcroft-Tarjan algorithm a second depth-first search is then called, the *PathFinder*. It uses the re-ordered adjacency lists to recursively generate paths in the graph and adds the paths one by one to build a planar embedding of G. The algorithm is based on paths.

A different method is here proposed of embedding G, based on embedding the DF-tree T branch by branch rather than by embedding paths. The reason for this is that the structure of trees is recursive in terms of their branches. This recursion can be used to give a graph-theoretic proof by induction that the algorithm works, using the branches of T. The algorithm is very similar to the path-embedding algorithm, but the use of branches seems simpler and is mathematically advantageous. The first thing is to understand the purpose and function of the weights.

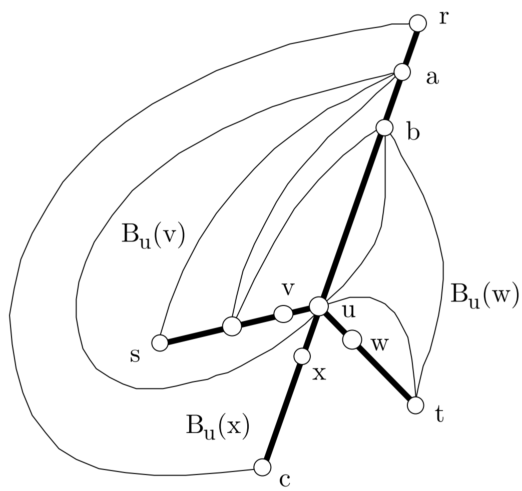

Fig. 2, Branches and fronds at u in a DF-tree T

Consider a DF-tree T constructed by the *LowPtDFS*, drawn schematically in the plane, as in Fig. 2, where only some of the vertices and edges are shown. The tree edges are shown in bold. At u there are two fronds, ua and ub, and there are three branches $B_u(v)$, $B_u(w)$, $B_u(x)$ with stems uv, uw, and ux. The fronds ua and ub have weights $wt_u[a] = 2a$ and $wt_u[b] = 2b$, since frond-weights are determined by their endpoints. To compute the weight of the stems, notice that $L_1(x) = r$ and $L_2(x) = u$, so that $wt_u[x] = 2r$, where r is the root vertex. $L_1(v) = a$ and $L_2(v) = b < u$, so $wt_u[v] = 2a+1$. $L_1(w) = b$ and $L_2(w) = u$, so $wt_u[w] = 2b$. The graph is to be embedded by another DFS which traverses T recursively, embedding the branches and fronds. The embedding takes place from the outside in. At vertex u they are embedded in the order $B_u(x)$, ua, $B_u(v)$, ub, and $B_u(w)$, since $wt_u[x] = 2r < wt_u[a] = 2a < wt_u[v] = 2a+1 < wt_u[b] = 2b = wt_u[w]$.

Embedding a branch like $B_u(v)$ with $L_1(v) = a$ is very much like embedding a frond ua. In fact if we insert a vertex of degree two on the frond ua, it behaves exactly like a branch whose L_1-value equals a. This is why the weight of a frond is determined by its endpoint, and that of a branch by the low-point of its stem vertex. Because $L_2(v) < u$ there will be other fronds inside the branch between u and a. Thus we can not know while visiting u whether it would be possible to embed the frond ua after $B_u[v]$ has been embedded. Therefore the frond *must* be embedded before the branch. This is why the frond is given weight $2a$ and the stem weight $2a + 1$.

A branch like $B_u(w)$ has $L_2(w) = u$. Therefore after the branch has been embedded it will still be possible to embed the frond ub, where $b = L_1(w)$ inside the branch. The order in which the frond and branch should be embedded is still not determined. Therefore they are given equal weight $2b$. So branches are ordered according to the L_1-value of the stem vertex. The purpose of the L_2-value is to distinguish the two types of branches. The factor of 2 in the weight computation is to distinguish between branches whose L_2-value is less than u, or equal to u, by making their weight odd or even, respectively.

2.1 Definition. A branch $B_u(v)$, where $u = a(v)$, with $L_2(v) = u$ is called a *type I* branch. If $L_2(v) < u$ it is called a *type II* branch.

So at vertex u, the ordering of $Adj[u]$ defined by the weights places fronds and type I branches before type II branches with the same L_1-value. A frond behaves like a type I branch. The definition of wt assigns $wt_u[v] = 2n + 1$ for some edges uv. These are either downward fronds, that is, fronds where $v > u$, or else tree edges with $a(u) = v$. These are embedded from their other endpoint, and so can be ignored while visiting u. Thus they are given a weight which places them after all other fronds and branches.

3. The Branch Points of T

Let T be a DF-tree constructed by a DFS. The DFS starts at the root of T. It descends the tree by recursively calling itself. While visiting node u, DFS(v) may be called recursively for several $v \in Adj[u]$.

3.1 Definition. A vertex u is a *branch point* of T, if T has a branch $B_u(v)$ where $v \in Adj[u]$ is not the first vertex of $Adj[u]$. The root of T is also a branch point. Given any v in T we define $b(v)$, the branch point of v as follows.

1) $b(v) = v$, if v is the root of T;
2) if v is not the root, then let $u = a(v)$.

$$b(v) = \begin{cases} b(u), & \text{if } v \text{ is the first node of } Adj[u]; \\ u & \text{otherwise.} \end{cases}$$

So the branch points of a given DF-tree T depend on which node comes first in the adjacency lists. Since the sequence $u_1 = a(v)$, $u_2 = a(u_1)$, $\ldots$ eventually reaches the root, $b(v)$ is uniquely defined, and is always a branch point of T. It is easily computed by a DFS initiated from the root of T.

```
DFS(u: vertex)
{ DFS to compute the branch points of an existing DF-tree T }
{ b(u) is already known }
begin
  FirstTime := true    { indicates first v ∈ Adj[u] }
  { a(u), D(u) have been computed by a previous DFS }
  for each v ∈ Adj[u] do begin
    if a(v) = u then begin
      { descend the tree }
      if FirstTime then b(v) := b(u)
      else b(v) := u
      DFS(v)
    end
    FirstTime := false
  end
end   { DFS }
```

The *LowPtDFS* cannot compute the branch points. This is because the branch points depend on the ordering of $Adj[u]$, and this is not determined until after the *LowPtDFS* has been executed. Given a DF-tree T and any ordering of the adjacency lists, a set of branch points of T is then determined. A subsequent DFS using the reordered $Adj[u]$ will descend the same tree T, although the branches will be searched in an order determined by $Adj[u]$.

Let $L(T) = \{v \mid \text{the first } w \in Adj[v] \text{ corresponds to a frond } vw\}$. Clearly $L(T)$ contains all the leaves of T. It may also contain other vertices which are not leaves. If $v \in L(T)$ is not a leaf, then v is always a branch point of T. For each $v \in L(T)$, let P_v denote the path in T from v to $b(v)$. The length of the path is $\ell(P_v)$, the number of edges in the path.

Referring to Fig. 2, the branches at u were ordered by the weights as $B_u(x)$, $B_u(v)$, and $B_u(w)$. The branch points are then as follows: $b(x) = r$, $b(v) = u$, $b(w) = u$, $b(u) = r$, $b(c) = r$, etc. The branch points divide the tree into paths. In Fig. 2, $L(T)$ consists of the leaves $\{c, s, t\}$, and the paths P_c, P_s, and P_t together contain all of T.

3.2 Lemma. *Let T be a DF-tree of G, with branch points defined by the ordered adjacency lists of G. Then $\{P_v \mid v \in L(T)\}$ is a partition of T into paths.*

Proof. By induction on $|L(T)|$. Let r denote the root of T. If $L(T) = \{v\}$, then v is a leaf and $T = P_v$, a path from v to r. Suppose that $|L(T)| = k$, where $k > 1$. Let v be the last leaf visited, and let $u = b(v)$. Pick $w \in P_v$ such that $a(w) = u$. Then $T' = T - P_v$ is a DF-tree for the graph $G - S^*(w)$ with $|L(T')| = k - 1$. Clearly $L(T') = L(T) - v$. Hence

$\{P_x \mid x \in L(T')\} \cup \{P_v\}$ is a partition of T into paths. The lemma follows.

At vertex u in T, the first node $v \in Adj[u]$ is crucial to the success of the algorithm. This is why the branch points are important. When uv is a tree edge, the DFS continues to descend the tree. When uv is a frond the descent stops. In this case the path P_u in T from u to $b(u)$, plus the frond uv is the path found by the *PathFinder* DFS of Hopcroft and Tarjan [13]. They also consider each frond as a path of length one. Hopcroft and Tarjan embed G in the plane by embedding these paths one by one.

We view the algorithm from a slightly different perspective. We use a DFS to embed G branch by branch of T. A frond uv, where $v < u$, is embedded either on the left side of T or on the right side. So we have an embedding of T in the plane, with the fronds arranged around T giving an embedding of G. We first determine an ordering of $Adj[u]$ so that the branch points are guaranteed to permit an embedding when G is planar. Then, following Hopcroft and Tarjan, we keep two linked lists of fronds, LF and RF containing the fronds and branches embedded on the left of T and on the right of T, respectively. LF and RF can be viewed as stacks, whose tops contain the number of the vertex currently marking the upper limit in the tree to which fronds may be embedded. As a first approximation we have

```
EmbedBranch(u: vertex)
begin
  { NonPlanar is a global flag }
  for each v ∈ Adj[u] do begin
    NonPlanar := true    { assume G will be found non-planar }
    if a(v) = u then begin   { uv is a tree edge }
      if b(v) = u then begin
        { uv begins a new branch at u }
        if L1(v) is too small to permit an embedding then Exit
        place Bu(v) either on LF or RF
      end
      EmbedBranch(v)
      if NonPlanar then Exit
    end
    else if v < u then begin   { uv is a frond }
      EmbedFrond(u, v)
      if EmbedFrond is unsuccessful then Exit
    end
  end
  NonPlanar := false    { no conflicting fronds were found }
end   { EmbedBranch }
```

EmbedFrond is a procedure which attempts to embed the frond uv by placing it either on LF or RF. It returns a boolean value indicating whether uv was successfully embedded.

4. Ordering the Adjacency Lists

The ordering of $Adj[u]$ by weights is not sufficient to guarantee that an embedding is possible without further refinement. We present several examples.

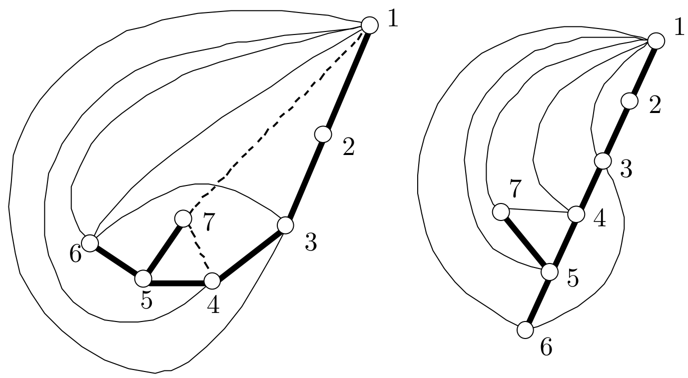

Fig. 3, Example 1

Example 1 shows two attempts at embedding the same graph. The tree T is the same in both attempts, but the ordering of the branches differs. In the example on the left, the adjacency lists are ordered strictly according to the weights computed. On the right the ordering has been adjusted slightly. The branches and fronds are the same in each case. At vertex 3, $B_3(4)$ is a type I branch with $wt_3[4] = 2$. There is a frond (3,1) with $wt_3[1] = 2$. In the left example, the frond was embedded first, then the branch, thereby forcing the branch inside the frond on the left side of T. In the right example, the branch $B_3(4)$ was embedded before the frond. At vertex 4, there is a type II branch $B_4(5)$ of weight 3. There is also a frond (4,1) of weight 2. On the left, the frond is embedded first; on the right the branch is taken first. At vertex 5, there are two type II branches, $B_5(6)$ and $B_5(7)$. Both have weight 3. There is also a frond (5,1) of weight 2. On the left, the frond is taken first, then $B_5(6)$, then $B_5(7)$; on the right $B_5(6)$ is taken first, then the frond, then $B_5(7)$. The thing to notice is that in the example on the left, it is impossible to embed $B_5(7)$ without forcing a crossing of the edge (6,3). So the ordering of $Adj[u]$ by weights must be adjusted if this graph is to be successfully embedded.

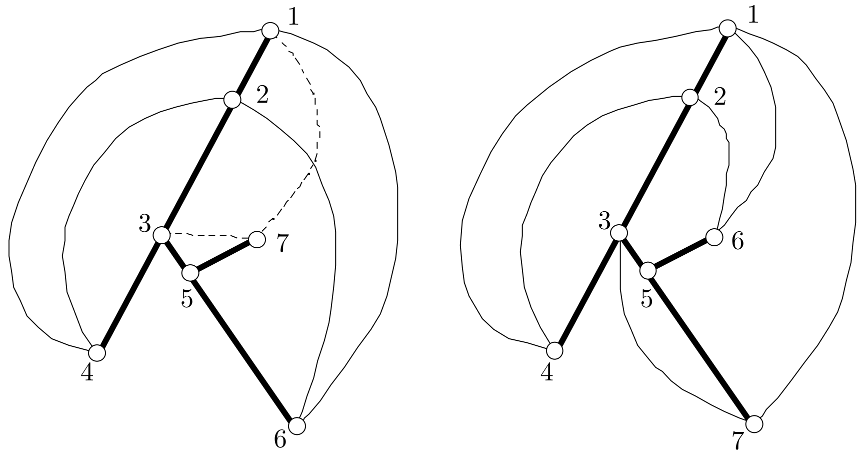

Fig. 4, Example 2

Example 2 also shows two attempts at embedding a graph. At vertex 3 there are two type II branches, $B_3(4)$ and $B_3(5)$, of weight 3. In both cases $B_3(4)$ is taken first. At vertex 5 there are also two type II branches, $B_5(6)$ and $B_5(7)$, both of weight 3. On the left $B_5(6)$ is taken first. We then find that $B_5(7)$ cannot be embedded without crossing the edge (6,2). On the right $B_5(7)$ is taken first. Notice that in the diagram on the left, moving $B_5(7)$ to the other side of the path from 3 to 6 is not permissible, since that would change the branch points of the tree. This is equivalent to reordering the adjacency lists, which could then affect the portion of the branch $B_3(5)$ already traversed at that point.

Example 3 shows a graph successfully embedded in the plane. At vertex 3 there is a type I branch $B_3(4)$ with $wt_3[4] = 2$ and a type II branch $B_3(5)$ with $wt_3[5] = 3$. The type I branch is taken first and the type II branch is then embedded inside the face created by the frond (4,1). At vertex 9 there is a type I branch $B_9(10)$ and a type II branch $B_9(11)$. Again the type I branch is taken first. At vertex 15 there is a frond (15,1) with $wt_{15}[1] = 2$ and a type II branch $B_{15}(16)$. The frond is taken first. Returning up the tree a number of minor branches are embedded, all successfully. The thing to notice is that at vertex 15, if the branch $B_{15}(16)$ had been embedded before the frond, then it would have been impossible to embed the minor branches when returning up the tree without introducing a crossing. So the frond must come first.

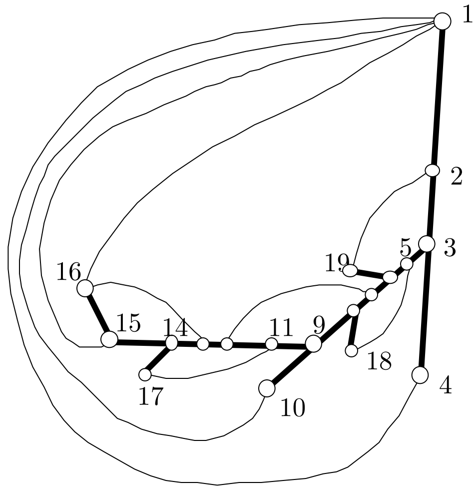

Fig. 5, Example 3

Example 4 shows a very similar graph successfully embedded in the plane. As in example 3 there are two branches at vertex 3, of which the type I branch is taken first. At vertex 9 there are again two branches and the type I branch is taken first. At vertex 15 there is a frond (15,1) and a type II branch $B_{15}(16)$. This time the branch is taken first, and the minor branches are then succesfully embedded when returning up the tree. The branch *must* be embedded before the frond. If the frond (15,1) were taken first it would not be possible to embed the minor branches without introducing a crossing.

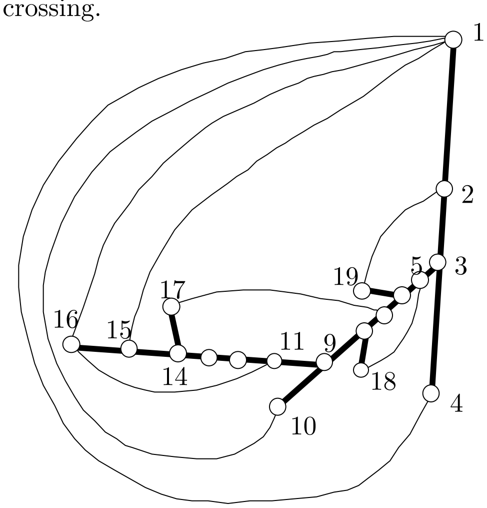

Fig. 6, Example 4

4.1 Definition. Let T be a DF-tree in a graph G. We define two integer functions $N(v)$ and $N'(v)$ for every $v \in V(G)$ as follows.

1) If v is the root of T then $N(v) = N'(v) = 0$.
   Otherwise let $u = a(v)$.
2) If $B_u(v)$ is a type I branch then $N(v) = N'(v) = 0$.
3) If $B_u(v)$ is a type II branch, choose the unique vertex x such that $a(x) = L_2(v)$. Then $L_2(v) < x \leq u < v$.

   3.1) If $b(v) = 1$ then $N(v) = 0$.
   If $L_2(v) < b(v)$ then $N(v) = 1$.
   Otherwise $L_2(v) \geq b(v) \neq 1$. Define

$$N(v) = \begin{cases} N(x) + 1, & \text{if } N(x) \neq 0 \\ 2, & \text{if } N(x) = 0 \text{ and } u \to L_1(v) \\ N(u), & \text{if } N(x) = 0 \text{ and } u \not\to L_1(v) \end{cases}$$

   3.2) If $b(u) = 1$ then $N'(v) = 0$.
   If $L_2(v) < b(u)$ then $N'(v) = 1$.
   Otherwise $L_2(v) \geq b(u) \neq 1$. Define

$$N'(v) = \begin{cases} N(x) + 1, & \text{if } N(x) \neq 0 \\ 2, & \text{if } N(x) = 0 \text{ and } u \to L_1(v) \\ N(u), & \text{if } N(x) = 0 \text{ and } u \not\to L_1(v) \end{cases}$$

The definition of $N(v)$ and $N'(v)$ appears complicated, but is only a mathematical formulation of a simple idea which is necessary in order to guarantee that the algorithm works. It can be computed quite easily, just by following the definition. $N(v)$ counts the number of steps needed to reach a vertex smaller than $b(v)$, where the steps are based on fronds to $L_2(v)$ or $L_1(v)$, as described below. If this is not possible then $N(v) = 0$. If $b(v) = 1$ then there is no vertex smaller than $b(v)$, so $N(v) = 0$. $N'(v)$ counts the number of steps needed to reach a vertex smaller than $b(u)$. If v is the first node in $Adj[u]$, then $b(v) = b(u)$, so $N(v) = N'(v)$. Consider the example of Fig. 6. $N(1) = 0$. $L_2(2) = 1$, so $N(2) = 0$. $L_2(3) = 2 \geq b(3)$, so $N(3) = 0$. Similarly $N(4) = 0$. $L_2(5) = 2 < b(5) = 3$, so $N(5) = 1$. $L_2(6) = 2 < b(6) = 3$, so $N(6) = 1$. $L_2(7) = 5 \geq b(7) = 3$, so $N(5) = 1 + N(6) = 2$, since $a(6) = 5$. Continuing in this way, we have $N(8) = 2$, $N(9) = 3$, $N(10) = 0$, $N(11) = N(12) = N(13) = N(14) = 1$, $N(15) = N(16) = 2$. We see that $N(v)$ counts the number of steps needed to reach a vertex smaller than $b(v)$. If $L_2(v) < b(v)$, then one step is sufficient, that is, the L_2-frond in $B_u(v)$ joins to a vertex $< b(v)$; hence $N(v) = 1$. If $B_u(v)$ is a type I branch then it is not possible to reach a vertex smaller than $b(v)$ via the L_2 function; hence $N(v) = 0$ in this case. Otherwise $B_u(v)$ is a type II branch with $L_2(v) \geq b(v)$; for example in Fig. 6, $L_2(7) = 5 > b(7) = 3$. In this case, pick x where $a(x) = L_2(v)$; in this example we get $x = 6$. Continuing with the example, starting at vertex 7, we can use two L_2-fronds, (18,5) and (19,2), giving a path (7,8,18,5,6,19,2) to reach vertex $2 < b(7) = 3$. Starting at vertex 9, we can use three L_2-fronds, (12,7), (18,5), and (19,2) to reach vertex $2 < b(9) = 3$. So $N(7) = 2$ and $N(9) = 3$.

$N'(v)$ differs from $N(v)$ only in the use of $b(u)$ in the comparison instead of $b(v)$. So $N(v) = N'(v)$ when v is the first node in the adjacency list. When v is not the first node, $N'(v)$ is the value that $N(v)$ *would have, if v were the first node.* $N'(v)$ is used to determine the correct ordering of $Adj[u]$. It is really $N'(v)$ that we want, but since $N'(v)$ is defined in terms of $N(v)$, we must compute $N(v)$ as well.

The functions $N(v)$ and $N'(v)$ can easily be computed by adding several statements modelled on definition 4.1 to the DFS of section 3 that computes the branch points. $N(v)$ and $N'(v)$ depend on the branch points of those vertices of T which are ancestors of v. The branch points in turn depend on the ordering of the adjacency lists. $N'(v)$ is used to re-order the adjacency lists. Initially, $Adj[u]$ is ordered according to the weight function $wt_u[v]$. Furthermore we assume that if $v, w \in Adj[u]$ have equal weight, where uv is a frond, and uw is the stem of a branch, that v precedes w in the adjacency list. That is, we assume that fronds precede branches of equal weight. This can easily be accomplished in the bucket sort by placing fronds at the head of a bucket and branch-stems at the tail of a bucket.

Let v^* be the first vertex of $Adj[u]$, and let W be the smallest odd number $\geq wt_u[v^*]$. Let $A^*(u) = \{v \in Adj[u] \mid wt_u[v] \leq W\}$. $A^*(u)$ represents all type I and type II branches at u with L_1-value $= \lfloor W/2 \rfloor$ (and possibly a frond at u to vertex $\lfloor W/2 \rfloor$). We now compute $N'(v)$ for each $v \in A^*(u)$ and adjust the ordering of $Adj[u]$ according to the following rule.

4.2 Rule.

1) If there is a $v \in A^*(u)$ such that $B_u(v)$ is a type II branch for which $N'(v)$ is even, then v is moved to the head of $Adj[u]$. (If there is more than one such v, only one of them is moved.)
2) Otherwise, if there is a $v \in A^*(u)$ such that $B_u(v)$ is a type I branch, then v is moved to the head of $Adj[u]$. (If there is more than one such v, only one of them is moved.)
3) Otherwise $Adj[u]$ is unchanged.

The computation of $N'(v)$ and the adjustment of the ordering of $Adj[u]$ is accomplished by the DFS which computes the branch points of T, since the ordering of the adjacency lists affects the branch points. If x is the unique vertex such that $a(x) = w$, for some vertex w, then we need to be able to find x, given w. Edge wx is the stem of the branch $B_w(x)$ at w. The easiest way to find x is for the algorithm to maintain an array $Stem[w]$, for vertices w.

```
BranchPtDFS(u: vertex)
{ DFS to compute b(v) and N(v), for all v ∈ Adj[u], and to }
{ reorder Adj[u]. Upon entry b(w) and N(w) are known, }
{ where w is u or any ancestor of u }
begin
  v := first vertex of Adj[u]
  W := wt_u[v]
  if W is even then W := W + 1
  v_I := 0;   v_II := 0   { markers for type I and II branches }
  for each v ∈ Adj[u] such that wt_u[v] ≤ W do begin
    if a(v) = u then begin   { uv is a tree edge }
      N(v) := 0   { assume N(v) will be 0 }
      if wt_u[v] is even then v_I := v   { B_u(v) is a type I branch  }
      else begin   { B_u(v) is a type II branch }
        if L_2(v) < b(u) then N(v) := 1
        else if b(u) ≠ 1 then begin
          x := Stem[L_2(v)]
          if N(x) ≠ 0 then N(v) = N(x) + 1
          else if u → L_1(v) then N(v) := 2
            else N(v) = N(u)
        end
        if N(v) is even then begin v_II := v; go to 1; end
      end
    end   { if a(v) = u }
  end   { for }
1: if v_II ≠ 0 then put v_II at head of Adj[u]
  else if v_I ≠ 0 then put v_I at head of Adj[u]
  { Adj[u] is now reordered, descend T }
  FirstTime := true    { indicates first v ∈ Adj[u] }
  for each v ∈ Adj[u] do begin
    if a(v) = u then begin
      b(v) := u   { assume the branch point will be u }
      if FirstTime then b(v) := b(u)
      else if wt_u[v] is even then N(v) := 0   { type I branch }
            else N(v) := 1   { type II branch }
      Stem[u] := v   { assign stem for branch about to be searched }
      BranchPtDFS(v)
    end
    FirstTime := false
  end
end   { BranchPtDFS }
```

4.3 Lemma. *The BranchPtDFS computes $b(v)$ and $N(v)$ correctly, for all $v \in V(G)$.*
Proof. By induction on the number of levels of recursion. Since we have identified vertices with their DF-number, the root is vertex 1. We must assign b(1) := 1 and $N(1) := 0$ before calling *BranchPtDFS*(1). So it is true with 0 levels of recursion. Suppose it is true up to k levels of recursion, and let u be a vertex visited at the $(k-1)^{\text{st}}$ level of recursion, where $k \geq 1$. The first for-loop of the algorithm takes each $v \in A^*(u)$ in turn, if uv is a tree edge. It initiallizes $N(v) := 0$. If $B_u(v)$ is a type I branch, it leaves $N(v)$ with the value of 0, which is correct. It records the vertex v in the value v_{I}. If $B_u(v)$ is a type II branch, it compares $L_2(v)$ with $b(u)$. If $L_2(v)$ is smaller, it sets $N(v) = 1$. This is the value of $N'(v)$. If $b(u) = 1$, the value of $N(v)$ is left as 0. Otherwise, if $L_2(v)$ is larger than $b(u)$, it finds x as in definition 4.1. The value given to $N(v)$ is then one of $N(x) + 1$, 2, or $N(u)$, as in 4.1. Again this is the value of $N'(v)$. If $N'(v)$ is even, it records the vertex v in the value v_{II}. So after the for-loop has been executed, $N(v)$ contains the value of $N'(v)$.

At this point one of v_{I} or v_{II} may be moved to the head of *Adj*$[u]$. Since $N(v) = N'(v)$ when v is the first node in *Adj*$[u]$, $N(v)$ has the correct value for the first vertex in the list. It also sets $b(v) = b(u)$ for the first node in the list, if uv is a tree edge. All remaining tree edges uv have $b(v) = u$, and $N(v) = 0$ or 1, according as they belong to type I or II branches, respectively. These are the values assigned by the algorithm. It follows that $b(v)$ and $N(v)$ are correctly computed for each $v \in$ *Adj*$[u]$ where uv is a tree edge. It then calls *BranchPtDFS* recursively. Therefore the lemma is true for every node visited at the k^{th} level of recursion. By induction it is true for all levels.

5. Graph-Theoretic Analysis of the Algorithm, $|L(T)| = 1$.

The re-ordering of *Adj*$[u]$ according to rule 4.2 is exactly what is needed in order to guarantee that the algorithm *EmbedBranch* will be successful whenever G is planar. The proof is by induction on n (the number of vertices of G), and on $|L(T)|$. We assume that G is a 2-connected graph and that $n \geq 3$.

Suppose first that $|L(T)| = 1$. Let $L(T) = \{v\}$, and let r denote the root of T. Since G is 2-connected, the first vertex in *Adj*$[v]$ is r, representing a frond to the root of T. Without loss of generality vr will be embedded on the left side of T. The cycle C formed by vr and P_v divides the plane into two regions, the inside and outside of C. The algorithm maintains two linked lists of fronds, LF and RF (note that a linked list is a sequence). Without loss of generality, we can assume that the left side of T refers to the inside region and the right side to the outside region of C. Every frond is to be placed either in LF or RF. Hopcroft and Tarjan use these linked lists as stacks, and discard fronds that can no longer have an effect on the remaining part of the algorithm. But it is advantageous to keep the entire linked list. Let $LF = (f_1, f_2, f_3, \ldots)$ and $RF = (g_1, g_2, g_3, \ldots)$. The fronds are ordered according to their smaller endpoint, namely, if $f_i = a_i b_i$ where $a_i < b_i$, then $a_1 \leq a_2 \leq a_3 \leq \ldots$, and similarly for RF. As we shall see, the sequences LF and RF constructed by the algorithm completely determine the combinatorial embedding of G in the plane. Therefore when we say that the algorithm embeds G in the plane, we mean that is constructs the lists LF and RF.

5.1 Definition. If xy and wz are two fronds such that $x < w < y < z$ then they cannot both be on the same side of C. They are called *conflicting* fronds. The *frond graph* of G and T is denoted FG. Its vertices are the fronds of G. Two fronds are adjacent if they conflict.

5.2 Lemma. *If G is planar and $|L(T)| = 1$, then FG is a bipartite graph.*

Proof. $V(FG)$ can be partitioned into those fronds inside C and those outside C. Since G is planar, there are no conflicting fronds both inside C or both outside C. Hence FG is bipartite.

It follows that FG contains no odd cycles. In general the frond graph will consist of a number of connected components $F_1, F_2, \ldots, F_m$. (Hopcroft and Tarjan call each F_i a *block* of fronds.) For each i, let (L_i, R_i) be the bipartition of F_i, where L_i consists of those fronds of F_i embedded on the left side of T, and R_i those fronds embedded on the right side of T. The bipartition $(\cup L_i, \cup R_i)$ of FG is not unique if $m > 1$, for we may exchange L_i and R_i in any F_i to obtain another bipartition of FG corresponding to another embedding of G. This is what Hopcroft and Tarjan call *switching sides*.

The diagram shows a DF-tree T for which FG has 4 components, indicated by the shading of the lines. A nesting of the components is evident.

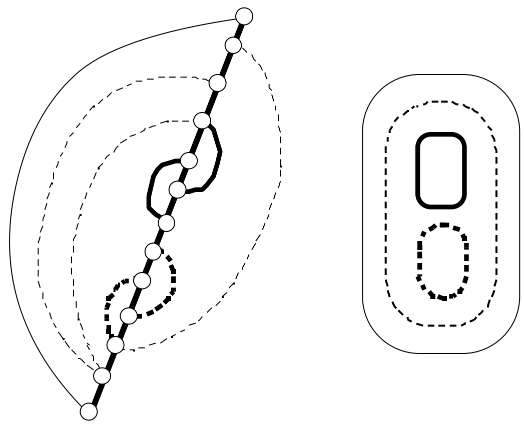

Fig. 7, Components of the frond graph, $|L(T)| = 1$.

Initially $FG = \emptyset$. In order to avoid the situation when $FG = LF = RF = \emptyset$ it is convenient to create two "dummy" fronds $f_0 = (0, n)$ and $g_0 = (0, n)$, and a "dummy" component F_0 for which $L_0 = \{f_0\}$ and $R_0 = \{g_0\}$. The algorithm descends T to the leaf v. All fronds are embedded while *EmbedBranch* is re-ascending T. As fronds are assigned either to LF or RF, FG is gradually constructed. The first frond embedded is vr. Without loss of generality, the algorithm will first try to place a frond on the left, if possible. So the algorithm will set $f_1 = vr$. Let u_i denote the minimum vertex of any frond in L_i, namely $u_i = \min\{a \mid \exists ab \in L_i\}$. Similarly $v_i = \max\{a \mid \exists ab \in L_i\}$, $x_i = \min\{a \mid \exists ab \in R_i\}$, and $y_i = \max\{a \mid \exists ab \in R_i\}$. If F_i consists of a single frond, then R_i will be empty, so that x_i and y_i are not always defined. In this case we take $x_i = n$ and $y_i = 0$. In this way most of the inequalities will be satisfied even when $R_i = \emptyset$. Every F_i will have $L_i \neq \emptyset$. In terms of the DF-numbering the first frond is $f_1 = (1, n)$.

Let u be a vertex of T being visited by *EmbedBranch* when a frond uw is encountered, where $w < u$. Any component F_i such that $u_i, x_i \geq u$ cannot possibly affect the remaining execution of the algorithm, since no remaining frond can conflict with any frond of F_i. Therefore the algorithm does not need to know all components of FG, only those for which $u > u_i$ or $u > x_i$. Let F_0, F_1, …, F_m denote all such components, in the order in which they were constructed, where $m \geq 1$. We can assume that $u \leq v_i$ for every i, and $u \leq y_i$ for every i for which $R_i \neq \emptyset$, since fronds are embedded while ascending T.

5.3 Lemma. *Let $1 \leq j \leq m$. Each component F_j "fits inside" F_{j-1}, namely F_{j-1} contains a frond ab such that $a \leq u_j < v_j \leq b$, and if $R_j \neq \emptyset$*

then $a \le x_j < y_j \le b$.

Proof. The frond graph is a dynamic structure, changing as fronds are embedded. By assumption $F_1, \ldots, F_m$ consists of those components for which $u_i < u \le v_i$ or $x_i < u \le y_i$, where u is the vertex currently being visited by *EmbedBranch*. Wlog, let $u_j < u$.

Suppose first that $u_{j-1} \le u_j$. Then $u_j < u \le v_{j-1}$. F_{j-1} is connected, so it contains a sequence of fronds connecting u_{j-1} to v_{j-1} and alternating between L_{j-1} and R_{j-1}. Therefore $a < u_j < b$ for some frond ab of this sequence in F_{j-1}. It follows that $v_j \le b$ and that $a \le x_j < y_j \le b$ if $R_j \ne \emptyset$ since F_j does not conflict with F_{j-1}.

On the other hand, suppose that $u_j < u_{j-1}$. Then since F_j does not conflict with F_{j-1}, $u_j < u \le v_j \le u_{j-1} < v_{j-1}$. However at least one of $u_i < u \le v_i$ or $x_i < u \le y_i$ holds for every i, so that $x_{j-1} < u \le y_{j-1}$ must hold. We can then apply the previous argument using x_{j-1} and y_{j-1} in place of u_{j-1} and v_{j-1}.

Let ℓ denote the largest integer such that the smaller endpoint of f_ℓ is $< u$ and let ℓ_w denote the smaller endpoint. Similarly, let r denote the largest integer such that the smaller endpoint of g_r is $< u$ and let r_w denote this endpoint. Because of F_0 we can be sure that ℓ and r are always defined. There are three possible situations that can arise for vertex u, as illustrated below.

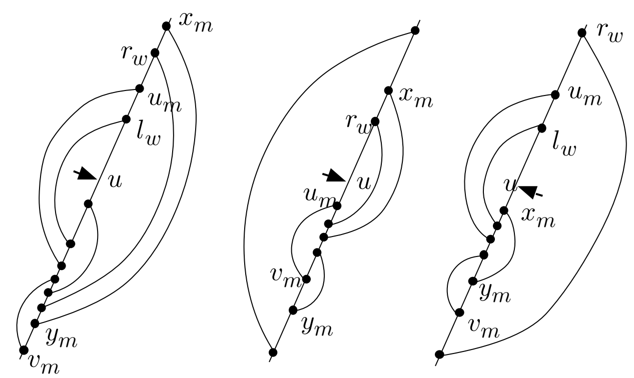

Fig. 8, Situation of vertex u wrt F_m, cases 1, 2, and 3

1. $u > u_m$ and $u > x_m$.
   In this case F_m contains a frond to u_m and x_m, so $u > \ell_w \ge u_m$ and $u > r_w \ge x_m$. ℓ_w and r_w both belong to fronds of F_m.
2. $u \le u_m$ and $u > x_m$.
   In this case $u_m \ge u > \ell_w$ and $u > r_w \ge x_m$. ℓ_w belongs to a frond of F_i where $i < m$. r_w belongs to a frond of F_m. By Lemma 5.3, $\ell_w \le r_w$.

3. $u > u_m$ and $u \leq x_m$.
   In this case $u > \ell_w \geq u_m$ and $x_m \geq u > r_w$. ℓ_w belongs to a frond of F_m. r_w belongs to a frond of F_i, where $i < m$. By Lemma 5.3, $r_w \leq \ell_w$.

There are several possible situations that can arise for vertex w.

4. $w \geq \ell_w$ and $w \geq r_w$.
   This can occur with cases 1, 2, or 3 above. uw does not conflict with any frond of LF or RF. uw is embedded on the left. It is inserted in LR immediately after f_ℓ. A new component F_{m+1} of FG is created with $u_{m+1} = w$ and $v_{m+1} = u$. $R_{m+1} = \emptyset$. The values of m and ℓ are increased by 1.
5. $w \geq \ell_w$ and $w < r_w$.
   This can only occur with cases 1 and 2 above, since $\ell_w < r_w$. uw conflicts with a frond of R_m, but not L_m. uw is embedded on the left. It is added to L_m, and inserted in LF immediately after f_ℓ. ℓ is increased by 1. If case 2 applies then uw may conflict with a frond of F_{m-1}. Set u_m equal to w and execute the following steps:

   while $w < x_{m-1}$ do Merge(F_{m-1}, F_m)

   Merge(F_{m-1}, F_m) is a procedure which updates the values u_m, v_m, x_m and y_m stored in the data structures, as follows.

   if $u_m < u_{m-1}$ then $u_{m-1} := u_m$
   if $v_m > v_{m-1}$ then $v_{m-1} := v_m$
   if $x_m < x_{m-1}$ then $x_{m-1} := x_m$
   if $y_m > y_{m-1}$ then $y_{m-1} := y_m$
   discard F_m
   $m := m - 1$

   It also conceptually merges L_{m-1} with L_m, and R_{m-1} with R_m, although it is not actually necessary to store these sets explicitly.

6. $w < \ell_w$ and $w \geq r_w$.
   This can only occur with cases 1 and 3 above, since $r_w < \ell_w$. uw conflicts with a frond of L_m, but not R_m. uw is embedded on the right. It is added to R_m, and inserted in RF immediately after g_r. r is increased by 1. If case 3 applies then uw may conflict with a frond of F_{m-1}. Set x_m equal to w and execute the following steps:

   while $w < u_{m-1}$ do Merge(F_{m-1}, F_m)
7. $w < \ell_w$ and $w < r_w$.
   This can occur with cases 1, 2, or 3 above. uw conflicts with a frond of LF and RF. If case 1 applies then G is non-planar, since uw conflicts with fronds f_ℓ and g_r of F_m, thereby creating an odd cycle in F_m. Execute the following steps:

while $w < \ell_w$ and $w < r_w$ do begin
  if $u > u_m$ and $u > x_m$ (case 1) then Exit (G is non-planar)
  SwitchSides(L_m, R_m)
end

SwitchSides is a procedure that exchanges L_m and R_m and updates the data structures as follows.

SwitchSides(L_m, R_m)
begin
  if $u \le u_m$ then begin { case 2 }
    while $u_{m-1} > \ell_w$ do Merge(F_{m-1}, F_m)
    $\ell_w := r_w$; $\ell := r$
    adjust r so that g_r is first frond preceding x_m in RF
    set r_w to smaller endpoint of g_r
  end
  else begin { $u > u_m$, case 3 }
    while $x_{m-1} > r_w$ do Merge(F_{m-1}, F_m)
    $r_w := \ell_w$; $r := \ell$
    adjust ℓ so that f_ℓ is first frond preceding u_m in LF
    set ℓ_w to smaller endpoint of f_ℓ
  end
  exchange the portion of the linked list LF between u_m and v_m
    with the portion of RF between x_m and y_m.
  exchange u_m and x_m, v_m and y_m, L_m and R_m
  Merge(F_{m-1}, F_m)
end { SwitchSides }

The while-loop will then stop with either $w \ge \ell_w$ (case 5) or $w \ge r_w$ (case 6). In either case uw can then be embedded.

F_1, F_2, …, F_m are connected components of the frond graph FG. If uw conflicts with both L_i and R_i, for some i, then uw cannot be embedded without creating an odd cycle in F_i. Hence G is non-planar in this case. Otherwise uw conflicts with at most one of L_i and R_i, for each i. It is then always possible to embed uw, for example by exchanging L_i and R_i for each i for which uw conflicts with R_i.

5.4 Lemma. *Let $|L(T)| = 1$. If the algorithm runs to completion then G is planar.*

Proof. The embedding is built by adding one frond at a time to an existing embedding, which is initially a planar embedding of T. Each frond is embedded without crossings, inside a face of the existing embedding. When

SwitchSides is executed, L_m and R_m are exchanged. This does not introduce any crossings into the embedding. So the embedding is always planar, right up to completion.

It follows that if G is non-planar, the algorithm will not run to completion. In proving that the algorithm works, we can therefore assume that G is planar.

5.5 Theorem. *If G is planar and $|L(T)| = 1$, the algorithm will embed G in the plane.*

Proof. Suppose that the algorithm does not run to completion. So it stops while visiting a vertex u with a frond uw, $w < u$, where $w < \ell_w$ and $w < r_w$ (case 7). If case 1 applies ($u > u_m$ and $u > x_m$) then uw conflicts with a frond of L_m and R_m so that uw cannot be embedded without creating an odd cycle in F_m. So G is non-planar in this case. If case 2 or 3 applies, then *SwitchSides* is executed. uw conflicts with the fronds to ℓ_w and r_w. Suppose first that case 2 applies. Then ℓ_w belongs to a component F_i where $i < m$ and r_w belongs to F_m. The components $F_{i+1}, \ldots, F_m$ contain no fronds in LF which conflict with uw. Therefore they all conflict with uw due to fronds in RF. Two components F_m and F_{m-1} are merged when uw conflicts with both of them. Therefore the components of FG are always connected. They are merged by *SwitchSides* into one connected component (while $u_{m-1} > \ell_w$ do) since they all conflict with uw, and then the resulting L_m and R_m are exchanged. This is done by exchanging the portions of the linked lists LF and RF that belong to L_m and R_m, using u_m, v_m, x_m, and y_m as pointers into the linked lists. At this point in the algorithm $m = i + 1$. It may be that F_i is a component which conflicts with uw in both L_i and R_i. The algorithm is now in a position to determine this by adjusting the values of ℓ_w and r_w, and comparing them with w. The new value of ℓ_w is the previous value of r_w, since the frond associated with r_w has been moved to the left. The new value of r_w is the smaller endpoint of the frond of RF immediately preceding F_m. Finally F_m and F_{m-1} are merged.

If case 3 applies the proof is very similar, except that right and left are interchanged. Thus we can conclude that if G is planar, eventually all components F_i which conflict with uw in one of L_i or R_i will be aligned so that uw can be embedded. The theorem follows.

Notice that the proof that the algorithm works when $|L(T)| = 1$ does not require rule 4.2. When $|L(T)| = 1$, the ordering of the adjacency lists is sufficient if just the L_1-value is used to order the fronds.

6. Graph-Theoretic Analysis of the Algorithm, $|L(T)| > 1$.

The algorithm *EmbedBranch* in section 3 uses a procedure *EmbedFrond* when a frond uv is encountered while visiting vertex u. The action of *EmbedFrond*(u, v) is described in section 5. Now when a branch $B_u(v)$ is encountered the entire branch must be embedded either on the left or right of T. The branch behaves very much like a frond uw where $w = L_1(v)$. If it is not possible to embed a frond uw then it will not be possible to embed $B_u(v)$ either. This can be tested by *EmbedFrond*(u, w). If uw can be embedded, say on LF, then the fronds xy of $B_u(v)$ for which $y < u$ must also be embedded in LF. Those for which $x, y \geq u$ can be embedded either in LF or RF. There is a very simple device which can be used to accomplish this. Suppose that $B_u(v)$ is to be embedded in LF. Embed a frond uw in LF, where $w = L_1(v)$. Mark uw as a "false frond". Embed a "branch marker" uu in RF. Mark it also as "false". Place uw and uu in the same component F_m of FG. The purpose of the false frond uw is to determine whether there is room to embed $B_u(v)$. The purpose of the branch marker uu is to force all fronds xy of $B_u(v)$ for which $y < u$ to the same side of T. The algorithm *EmbedBranch* can be modified by replacing the group of statements beginning "if $b(v) = u$ then begin" with the following.

if $b(v) = u$ then begin
  { uv begins a new branch at u }
  $w := L_1(v)$
  EmbedFrond(u, w)
  if EmbedFrond is unsuccessful then Exit
  mark uw as a false frond
  embed a branch marker uu on the side opposite to uw,
    in the same F_m
end

The definition 5.1 of conflicting fronds is not sufficient when $|L(T)| > 1$. We refer to fronds, false fronds, and branch markers collectively as fronds.

6.1 Definition. Let $B_u(v)$ be a branch with branch marker uu and false frond ut, where $t = L_1(v)$. Let xy and wz be any fronds where $x > y$, $w > z$, and $x \geq w$. Then:
1) The branch marker uu conflicts with the corresponding false frond ut;
2) xy and wz conflict if $b(x) = b(w)$ and $x > w > y > z$;
3) xy and uu conflict if $b(x) = u$, $y < u$, and $xy \in B_u(v)$.

The frond graph is denoted FG. Its vertices are the fronds of G. Two fronds are adjacent if they conflict.

If $B_u(v)$ is as above, with false frond ut embedded in LF, then any frond xy of $B_u(v)$ for which $y < u$ will conflict with the branch marker uu

in RF. It will therefore be embedded in LF, as part of F_m. A frond xy for which $x, y \geq u$ will not conflict with uu. It will be embedded in either LF or RF. No frond xy of $B_u(v)$ will conflict with the false frond ut since $b(x) \neq b(u)$. If it becomes necessary to exchange the bipartition of F_m, the entire branch will automatically be switched.

The easiest way to mark a frond false is to make one of its endpoints negative, use $(-u, t)$ and $(-u, u)$ instead of ut and uu. The program knows that a negative value indicates a false frond. They become *sentinels* in the linked lists LF and RF. In a similar fashion Hopcroft and Tarjan use "end of stack markers" to delimit the branches of T. The false fronds are handy because they contribute to the proof that the algorithm works. They form part of the frond graph, which is still bipartite, and can be used to find odd cycles when G is non-planar.

6.2 Lemma. *If G is planar, then FG is bipartite.*
Proof. If $|L(T)| = 1$, definition 6.1 reduces to 5.1 and the result follows by 5.2. We claim that (LF, RF) is a bipartition of FG. If it is not, then LF, say, will contain conflicting fronds uv and xy, where $u \geq v$ and $x \geq y$.
Case 1. uv and xy are true fronds.
Since $b(u) = b(x)$, uv and xy are in the same branch, say $B_z(w)$, where $w = b(u)$. The first frond st of the branch together with the path in T connecting s to t creates a cycle C such that uv and xy are crossing fronds both inside C. But G is planar, a contradiction.
Case 2. uv is a false frond, xy is a true frond.
As in case 1, uv and xy are in a branch $B_z(w)$, where $b(u) = b(x) = w$, and a cycle C is given. u is the branch point of a branch $B_u(q)$ which contains a frond to $v = L_1(q)$. This creates two crossing paths inside C, a contradiction.
Case 3. uv is a branch marker, xy is a true frond.
Since $b(x) = u$, xy is contained in a branch B at u. The branch marker uu corresponding to B must be in RF, since $xy \in LF$. So uv must be the branch marker for a different branch, a contradiction.
Case 4. uv and xy are false fronds.
This case is almost identical to case 2.
Case 5. uv is a branch marker, xy is a false frond.
This case is almost identical to case 3.
Case 6. uv and xy are branch markers.
Since $b(u) = x$ and $b(x) = u$, we must have $u = x = 1$, a contradiction.

Knowing that FG must be bipartite, an odd cycle in FG indicates that G is non-planar. This is the key to proving that the algorithm works. An odd cycle may involve false fronds or branch markers, but still proves that G is non-planar, by lemma 6.2.

When the frond graph is defined in this way, it is not necessary to introduce *segments*, as in [13] or [19]. Rather the embedding consists of the tree T, together with its fronds. FG is always bipartite. Similarly the branch points $b(v)$ together with the paths P_v preclude the need to define a *tree of paths* as in [19].

If $|L(T)| > 1$, pick $v \in L(T)$ where v is the last leaf of T visited by *EmbedBranch*. Let $u = b(v)$ and let r denote the root of T. Then P_v is a vu-path, and $u \neq r$. Let uw be the stem of the branch at u containing v, and set $x = a(v)$. Let $s = L_1(v)$. This is illustrated below.

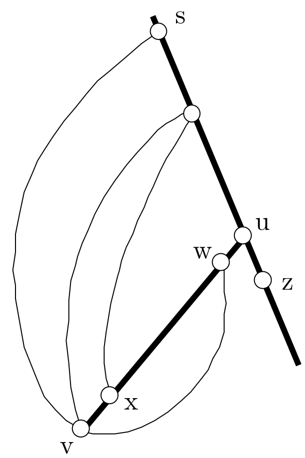

Fig. 9, The branch $B_u(w)$ containing $v \in L(T)$, $\ell(P_v) > 1$.

The proof that the algorithm will embed G when $|L(T)| > 1$ is based on a decomposition of G into $B_u(w)$ and $G - B_u(w)$, that is, T is decomposed into P_v and $T - P_v$, as in lemma 3.2. Construct a new graph G' from G as follows. Contract all the edges between s and $a(u)$ into a single vertex t, and delete all vertices and edges except t and those of $B_u(w)$. This is illustrated in Fig. 10. If G is planar, then so is G'. A DF-tree T' of G' consists of a single path from v to t. Any frond at in G' corresponds to at least one frond ay in G, where $s \leq y < u$.

6.3 Lemma. *If G is planar, then G' can be embedded so that all fronds to t are on the same side of T'.*
Proof. Let FG' denote the frond graph for G', with components $F_1, F_2, \ldots$, F_m, where $m \geq 1$. If it is not possible to embed G' with all fronds to t on the left side of T', then exchange the bipartition of each F_i as necessary, so that all fronds yt are on the left, where $y > a$ and a is as small as possible. So there is a frond $at \in F_i$ on the right side, for some i, which cannot be

moved to the left by exchanging the bipartition of F_i without bringing a frond bt to the right, where $b > a$. Choose b as small as possible. Then there are then no fronds to t between a and b. Let P be a shortest path in F_i connecting $at \in R_i$ to $bt \in L_i$. Then $\ell(P)$ is odd, since at and bt are on opposite sides of T'. P consists of a sequence of fronds bt, u_1v_1, u_2v_2, $\ldots$, u_jv_j, at, where $j = \ell(P) - 1$ is even and each u_kv_k conflicts with $u_{k+1}v_{k+1}$. This is illustrated with $j = 2$. Without loss of generality we can choose u_1v_1 so that v_1 is as small as possible. Similarly we can then choose u_2v_2 so that v_2 is as small as possible, and so forth, without changing $\ell(P)$.

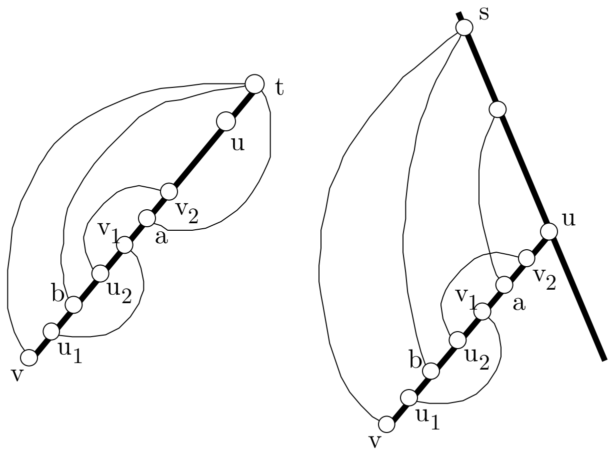

Fig. 10, G' and G, $b \to s$

<u>Case 1</u>. bt in G' has a corresponding frond sb in G.
Let x_1 be the vertex with $a(x_1) = b$ and let z be the vertex with $a(z) = a$. In G, $B_b(x_1)$ is a type II branch which precedes the frond sb in $Adj[b]$. Therefore $N(x_1)$ is even, by rule 4.2. There are no fronds to t between a and b, so in G, $v_1 = L_2(x_1)$. Similarly $v_2 = L_2(x_2)$, where $a(x_2) = v_1$, and so forth, until we reach v_j with $a(x_{j+1}) = v_j$. We have $L_1(x_k) = s$ and $L_2(x_k) = v_k = a(x_{k+1})$, for $k = 1, 2, \ldots, j$. By definition 4.1, $N(x_k)$ depends on $N(x_{k+1})$.

Suppose first that $a \to y$, where $s < y < u$. Then $N(x_{j+1}) = 1$, by 4.1. It follows that $N(x_j) = N(x_{j+1}) + 1$, and so on, until $N(x_1) = j + 1$, which is odd, a contradiction.

We conclude that s is the only vertex $< u$ adjacent to a. So as is a frond. The branch $B_a(z)$ precedes as in $Adj[a]$, so $N(z)$ is even, by rule 4.2. We claim that $N(z) \neq 0$. Now $L_2(z) = v_j$ so $N(z)$ depends on $N(x_{j+1})$. The only way that $N(z) = 0$ would be possible is if the last option in item 3.1 of definition 4.1 applied. But this requires that $a \not\to s$,

a contradiction. So $N(z)$ is even and $\neq 0$. Vertex x_j lies between u_j and a, so $L_2(x_j) = L_2(z) = v_j$. The last option in item 3.1 of definition 4.1 applies, so that $N(x_j) = N(z)$, which is even. $N(x_{j-1})$ depends on $N(x_j)$. Either $N(x_{j-1}) = 1$ or $N(x_{j-1}) = N(x_j)+1$, in both cases an odd number. There are no fronds to t between a and b, so we eventually must have $N(x_1) = N(z) + j - 1$, an odd number, which is impossible.
Case 2. bt in G' corresponds to by in G, where $y > s$, $s \not\to b$.
In this case, the path P in FG' corresponds to a path of odd length in FG connecting by to the frond aw corresponding to at. Since by and aw both conflict with the branch marker uu, an odd cycle exists in FG. By lemma 6.2, G is non-planar. The example in Fig. 11 shows a $K_{3,3}$ homeomorph in G when $j = 2$. The dashed portions of T are not part of the $K_{3,3}$. The dark and light shading of some of the nodes indicates the bipartition of $K_{3,3}$.

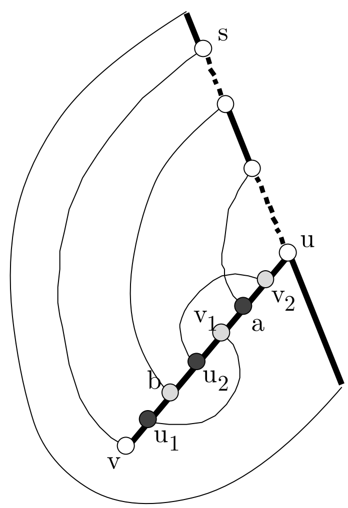

Fig. 11, G, $b \not\to s$

6.4 Lemma. *Let $|L(T)| > 1$. If the algorithm runs to completion then G is planar.*
Proof. The embedding is built up by adding one frond at a time to either LF or RF. If a frond uv conflicts with fronds of both LF and RF in F_m, then FG contains an odd cycle, so that G is non-planar, and the algorithm stops. If no odd cycle is created, uv will be embedded inside a face of the existing embedding.

6.5 Theorem. *Let G be a 2-connected planar graph. Then the algorithm will embed G in the plane.*

Proof. By induction on n and $|L(T)|$. By theorem 5.3, we know that the theorem is true when $|L(T)| = 1$, for all n. Let $n > 3$ and suppose that the theorem holds for all graphs with fewer than n vertices. Let $k > 1$ and suppose that the theorem is also true for all graphs on n vertices whose DF-tree T satisfies $|L(T)| < k$. Let G have n vertices with $|L(T)| = k$.

As in Fig. 9 above, r denotes the root of T, $v \in L(T)$ is the last leaf of T visited by *EmbedBranch*, $u = b(v)$, $a(w) = u$, $x = a(v)$, and $s = L_1(v)$. $B_u(w)$ is the last branch visited at vertex u. Let z be the first vertex of $Adj[u]$. uz is either a frond or the stem of a branch. In either case $wt_u[z] \leq wt_u[w]$ and there is a cycle C containing s, u, and z, such that the entire branch $B_u(w)$ is embedded inside C.

<u>Case 1</u>. $\ell(P_v) > 1$.

Contract edge vx in G and in T to get a graph G'' with tree T''. If G is planar then so is G''. T'' is a DF-tree of G'', and the weights of edges in G'' are the same as in G. Therefore the ordering of the adjacency lists is the same in G'' as in G, except possibly at vertex x, which is now a leaf. G'' has $n-1$ vertices, so the algorithm will embed G''. Each frond will be placed in LF or RF. When the algorithm is given G as input, the execution of *EmbedBranch*(w) called from u is exactly the same as in the graph G' of lemma 6.1, except that fronds to vertices $< u$ must be embedded on the left side. By lemma 6.1 this can be accomplished by exchanging the bipartition of several F_i, if necessary. This is what the algorithm will do. Because of the branch markers all fronds to vertices $< u$ will always be on the same side of T. Therefore $B_u(w)$ will be successfully embedded. The remaining execution of the algorithm is exactly the same in G as in G''. The algorithm cannot know whether the branch $B_u(w)$ just embedded was contracted as in G'' or not, the fronds to vertices $< u$ are identical in both cases. So if $\ell(P_v) > 1$, G will be successfully embedded in the plane.

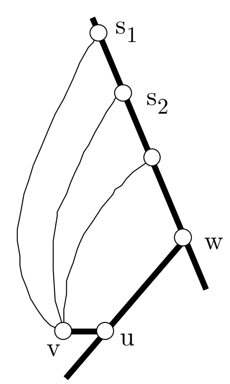

Fig. 12, The branch $B_u(v)$ when $\ell(P_v) = 1$

Case 2. $\ell(P_v) = 1$.
In this case $a(v) = b(v) = u$. Let $w = b(u)$, $s_1 = L_1(v)$, and $s_2 = L_2(v)$. Use induction on $m = \deg(v)$. If $m = 2$, then we can replace the branch $B_u(v)$ with a single frond us_1. The resulting graph has $n-1$ vertices, so will be successfully embedded. Given G, the algorithm will call *EmbedBranch*(v) at vertex u before embedding the frond vs_1. The rest of the execution will be identical, except that LF and RF will contain an additional false frond us_1 and branch marker uu, which will not affect the execution. So G will be embedded when $m = 2$. If $m > 2$, construct a new graph H from G by replacing $B_u(v)$ with a frond us_1 and a new branch B containing v together with the remaining $m-1$ fronds at v. So in H, $\deg(v) = m-1$, and the frond us_1 will precede the branch B in $Adj[u]$. By the induction hypothesis, H will be successfully embedded. We compare the execution of the algorithm given H and G. In H there are fronds ww, uu, us_1, us_2, and vs_2. In G there are ww, uu, us_1, vs_1 and vs_2. If uu and ww are embedded on the same side in H, then the execution of the algorithm in G will be the same as in H except that all the fronds at v will be embedded by a recursive call *EmbedBranch*(v). So if uu and ww are on the same side in H, G will be successfully embedded.

If $s_1, s_2 < w$, the frond us_1 and the false frond us_2 in H both conflict with ww. Since us_2 conflicts with uu, we must have uu and ww on the same side, so G will be successfully embedded.

If $s_1 < w$ but $s_2 \geq w$, then us_1 conflicts with ww in H, but us_2 does not. If uu and ww are in different components of the frond graph for H, then we can switch the component containing uu to put uu and ww on the same side. If they are in the same component on opposite sides, then FH contains a path P of odd length connecting uu to ww. This path will also occur in G. But in G, us_1 conflicts with both uu and ww, creating an odd cycle in FG, a contradiction. It follows that the result holds when $m > 1$. By induction, the theorem holds for all values of $|L(T)| \geq 1$ and all n.

Once it has been determined that G is planar, a planar embedding must be constructed. This consists of two parts, 1) constructing a combinatorial embedding, and 2) constructing a geometric embedding of G. A geometric embedding is the actual layout, representing the vertices as points in the plane, and the edges as non-crossing curves in the plane. The combinatorial embedding is given by the cyclic order of edges at each vertex [18]. Read's algorithm [18] is one method that can be used to construct a geometric embedding once a combinatorial embedding is given. A method of finding a combinatorial embedding is given by Nishizeki and Chiba [6,17], using PQ-trees. The Hopcroft-Tarjan algorithm as presented here actually provides the geometric embedding, almost for free.

6.6 Theorem. *If G is planar, the linked lists LF and RF completely determine a combinatorial embedding of G.*

Sketch of proof. It is clear that a "walk-around" traversal of T together with its fronds determines the cyclic order of the edges at each vertex in G. If $|L(T)| = 1$, T is a single path, and the walk-around is determined by the order of the fronds on LF and RF. If $|L(T)| > 1$, let $v \in L(T)$ be the last leaf visited by the algorithm, let $u = b(v)$, $w = L_1(v)$, and use induction. The false frond uw and branch marker uu will appear on opposite sides of T. A traversal of LF and RF will encounter the branch marker uu and corresponding false frond uw, and "walk around" the branch. It is fairly straightforward to develop an algorithm which will traverse the lists LF and RF and directly assign the cyclic ordering of the edges. The details will be given elsewhere.

If G is non-planar it contains a homeomorph of a Kuratowski subgraph $K_{3,3}$ or K_5. K_5 occurs only in very exceptional circumstances (Kelmans[14]), so that if G is non-planar, we usually want to find a $K_{3,3}$ homeomorph in G. This problem is addressed by Williamson [19,21]. It is not too hard to see that the DF-tree T together with its branch points $b(v)$ are sufficient to find the paths in G which make up the $K_{3,3}$. The algorithm for this is not presented here. It is based on finding an odd cycle in FG.

In programming the Hopcroft-Tarjan algorithm one first begins with the *LowPtDFS* in order to determine $L_1(v)$ and $L_2(v)$. The adjacency lists are then sorted by weight. Next, the *BranchPtDFS* can be executed in order to establish the first node in $Adj[u]$. Then *EmbedBranch* is executed in order to construct the lists LF and RF which determine the combinatorial embedding of G. An alternative is to combine *BranchPtDFS* and *EmbedBranch* into a single DFS, which is easy to do. In either case it is obvious that the algorithm has linear running time, since a DFS considers each edge of G exactly twice.

Acknowledgement. I would like to thank Almira Karabeg for bringing the book by Williamson [19] to my attention.

References

1. A.O. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachussetts, 1974.
2. Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis, "Algorithms for Drawing Graphs: an Annotated Bibliography", preprint, 1993.
3. J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, American Elsevier Publishing, New York, 1976.

4. K.S. Booth and G.S. Lueker, “Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms”, JCSS 13 (1976), 335-379.
5. Gary Chartrand and Ortrud Oellermann, *Applied and Algorithmic Graph Theory*, McGraw-Hill, Inc., New York, 1993.
6. Norishige Chiba and Takao Nishizeki, “A linear algorithm for embedding planar graphs using PQ-trees”, Journal of Computer and System Sciences 30 (1985), 54-76.
7. G. Demoucron, Y. Malgrange, and R. Pertuiset, “Graphes planaires: reconnaissance et construction des représentations planaires topologiques”, Rev. Francaise Recherche Opérationelle 8 (1965) 34-47.
8. Narsingh Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall, Inc., New Jersey, 1974.
9. Shimon Even, *Graph Algorithms*, Computer Science Press, Maryland, 1979.
10. L.R. Foulds, *Graph Theory Applications*, Springer-Verlag, New York, 1992.
11. Alan Gibbons, *Algorithmic Graph Theory*,Cambridge University Press, Cambridge, U.K., 1985.
12. Ronald Gould, *Graph Theory*, Benjamin/Cummings Publishing, Inc., Menlo Park, California, 1988
13. John Hopcroft and Robert Tarjan, “Efficient planarity testing”, JACM 21 (1974), 449-568.
14. A.K. Kelmans, “Graph planarity and related topics”, Rutcor Research Report #28-92, Rutgers University, New Brunswick, New Jersey, 1992.
15. A. Lempel, S. Even, and I. Cederbaum, “An algorithm for planarity testing of graphs”, in *Theory of Graphs, International Symposium, Rome*, 1966, Ed. P. Rosenstiehl, Gordon and Breach, New York, 1967.
16. James A. McHugh, *Algorithmic Graph Theory*, Prentice Hall, Inc., New Jersey, 1990.
17. T. Nishizeki and N. Chiba, *Planar Graphs: Theory and Algorithms*, Annals of Discrete Mathematics, vol. 32, North Holland, Amsterdam, 1988.
18. R.C. Read, “A new method for drawing a planar graph given the cyclic order of the edges at each vertex”, Congressus Numerantium 56 (1987), 31-44.
19. S. Gill Williamson, *Combinatorics for Computer Science*, Computer Science Press, Maryland, 1985.
20. S.G. Williamson, “Embedding graphs in the plane — algorithmic aspects”, Annals of Discrete Mathematics 6 (1980), 349-384.
21. S.G.Williamson,“Depth-first search and Kuratowski subgraphs”,JACM 31 (1984), 681-693.