*

An Implementation of the Hopcroft and Tarjan Planarity Test and Embedding Algorithm

Kurt Mehlhorn
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany

Petra Mutzel
Institut für Informatik,
Universität zu Köln, 50969 Köln, Germany

Stefan Näher
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany

Abstract

We describe an implementation of the Hopcroft and Tarjan planarity test and embedding algorithm. The program tests the planarity of the input graph and either constructs a combinatorial embedding (if the graph is planar) or exhibits a Kuratowski subgraph (if the graph is non-planar).

Contents

*This work was supported in part by the German Ministry for Research and Technology (Bundesministerium für Forschung und Technologie) under grant ITS 9103 and by the ESPRIT Basic Research Actions Program under contract No. 7141 (project ALCOM II).

1. Introduction.

We descibe two procedures to test the planarity of a graph G:

$$\textbf{bool}\ \mathit{planar}(\textbf{graph}\ \&G, \textbf{bool}\ \mathit{embed} = \mathit{false})$$

and

$$\textbf{bool}\ \mathit{planar}(\textbf{graph}\ \&G, \textbf{list}\langle\textbf{edge}\rangle\ \&P, \textbf{bool}\ \mathit{embed} = \mathit{false}).$$

Both take a directed graph G and test it for planarity. If the graph is planar and bidirected, i.e., for every edge of G its reversal is also in G, and the argument *embed* is true, then they also compute a combinatorial embedding of G (by suitably reordering its adjacency lists). If the graph G is non-planar then the first version of *planar* only records that fact. The second version in addition returns a subgraph of G homeomorphic to $K_{3,3}$ or K_5 (as a list P of edges). For a planar graph G the running time of both versions is linear (cf. section 34 for more detailed information). For non-planar graphs G the first version runs in linear time but the second version runs in quadratic time. We are aware of the linear time algorithm of Williamson [Wil84] to find Kuratowski subgraphs but have not implemented it.

The implementation of *planar* is based on the LEDA platform of combinatorial and geometric computing [Nae93, MN89]. It is part of the LEDA-distribution (available through anonymous ftp at cs.uni-sb.de). In this document we describe the implementation of both versions of *planar* and a demo, and report on our experimental experience.

Procedure *planar* is based on the Hopcroft and Tarjan linear time planarity testing algorithm as described in [Meh84, section IV.10]. For the sequel we assume knowledge of section IV.10 of [Meh84]. A revised version of that section is included in this document (see section 42) for the convenience of the reader. Our procedure *planar* differs from [Meh84, section IV.10] in two respects: Firstly, it works for arbitrary directed graphs and not only for biconnected undirected graphs. To this end we augment the input graph by additional edges to make it biconnected and bidirected. The augmentation does not destroy planarity. Secondly, the embedding phase follows the presentation in [MM94]. We want to remark that the description of the embedding phase given in [Meh84, section IV.10] is false. The essential part of [MM94] is reprinted in section 26.

This document defines the files *planar.h*, *planar.c*, and *demo.c*. *planar.c* contains the code for procedure *planar*, *demo.c* contains the code for a demo, and *planar.h* consists of the declarations of procedure *planar*. The third file is defined in section 35, the structure of the first two files is as follows:

⟨**planar.h** 1⟩ ≡
  bool *planar*(**graph** &G, **bool** *embed* = *false*);
  bool *planar*(**graph** &G, **list**⟨**edge**⟩ &P, **bool** *embed* = *false*);
  void *Make_biconnected_graph*(**graph** &G);

2.⟨**planar.c** 2⟩ ≡
  ⟨includes 3⟩;
  ⟨typedefs, global variables and class declarations 11⟩;
  ⟨auxiliary functions 9⟩;
  ⟨first version of *planar* 5⟩;
  ⟨second version of *planar* 4⟩;

3. We include parts of LEDA (who would want to work without it) [Nae93, MN89]. We need stacks, graphs, and graph algorithms.

⟨ includes 3 ⟩ ≡

```
#include <LEDA/stack.h>
#include <LEDA/graph.h>
#include <LEDA/graph_alg.h>
#include "planar.h"
```

See also section 36.

This code is used in sections 2 and 35.

4. The second version of *planar* is easy to describe. We first test the planarity of G using the first version. If G is planar then we are done. If G is non-planar then we cycle through the edges of G. For every edge e of G we test the planarity of $G-e$. If $G-e$ is planar we add e back in. In this way we determine a minimal (with respect to set inclusion) non-planar subgraph of G, i.e., either a K_5 or a $K_{3,3}$.

⟨ second version of *planar* 4 ⟩ ≡

```
bool planar(graph &G, list<edge> &P, bool embed = false)
{
  if (planar(G, embed)) return true;
  /* We work on a copy H of G since the procedure alters G; we link every vertex and
  edge of H with its original. For the vertices we also have the reverse links. */
  GRAPH<node, edge> H;
  node_array<node> link(G);
  node v;
  forall_nodes (v, G) link[v] = H.new_node(v);
  /* This requires some explanation. H.new_node(v) adds a new node to H, returns the
  new node, and makes v the information associated with the new node. So the statement
  creates a copy of v and bidirectionally links it with v */
  edge e;
  forall_edges (e, G) H.new_edge(link[source(e)], link[target(e)], e);
  /* link[source(e)] and link[target(e)] are the copies of source(e) and target(e) in H.
  The operation H.new_edge creates a new edge with these endpoints, returns it, and
  makes e the information of that edge. So the effect of the loop is to make the edge set
  of H a copy of the edge set of G and to let every edge of H know its original. We can
  now determine a minimal non-planar subgraph of H */
  list<edge> L = H.all_edges();
  edge eh;
  forall (eh, L) {
    e = H[eh];      // the edge in G corresponding to eh
    node x = source(eh);
    node y = target(eh);
    H.del_edge(eh);
    if (planar(H)) H.new_edge(x, y, e);
          // put a new version of eh back in and establish the correspondence
  }
  /* H is now a homeomorph of either K5 or K3,3. We still need to translate back to G.
  */
```

```
    P.clear( );
    forall_edges (eh, H) P.append(H[eh]);
    return false;
  }
```

This code is used in section 2.

5. The first version of *planar* is also quite simple to describe. Graphs with at most three vertices are always planar. So assume that G has more than three vertices. We first add edges to G to make it bidirected and then add some more edges to make it biconnected (of course, without destroying planarity). Then we test the planarity of the extended graph and construct an embedding. Since *planar* alters the input graph, it works on a copy of it.

⟨ first version of *planar* 5 ⟩ ≡

```
  bool planar(graph &Gin, bool embed = false)
  /* Gin is a directed graph. planar decides whether Gin is planar. If it is and embed ≡
  true then it also computes a combinatorial embedding of Gin by suitably reordering its
  adjacency lists. Gin must be bidirected in that case. */
  {
    int n = Gin.number_of_nodes( );
    if (n ≤ 3) return true;
    if (Gin.number_of_edges( ) > 6 * n − 12) return false;
    /* An undirected planar graph has at most 3n − 6 edges; a directed graph may have
    twice as many */
    ⟨ make G a copy of Gin and add edges to make G bidirected 6 ⟩;
    ⟨ make G biconnected 8 ⟩;
    ⟨ test planarity 13 ⟩;
    if (embed) ⟨ construct embedding 26 ⟩;
    return true;
  }
```

This code is used in section 2.

6. We make G a copy of *Gin* and bidirectionally link all vertices and edges. Then we add edges to G to make it bidirected. In *Gin_is_bidirected* we record whether we needed to add edges.

⟨ make G a copy of *Gin* and add edges to make G bidirected 6 ⟩ ≡

```
  GRAPH⟨node, edge⟩ G;
  edge_array⟨edge⟩ companion_in_G(Gin);
  node_array⟨node⟩ link(Gin);
  bool Gin_is_bidirected = true;
  {
    node v;
    forall_nodes (v, Gin) link[v] = G.new_node(v);      // bidirectional links
    edge e;
    forall_edges (e, Gin)
      companion_in_G[e] = G.new_edge(link[source(e)], link[target(e)], e);
  }
  ⟨ bidirect G 7 ⟩;
```

This code is used in section 5.

7. We bidirect G. We first assign numbers to nodes and edges. We make sure that the two versions of the same undirected edge get the same number but versions of distinct undirected edges get different numbers. Then we sort the edges according to numbers. Finally we step through the sorted list of edges and add missing edges.

$\langle$ bidirect G 7 $\rangle \equiv$

```
{
  node_array⟨int⟩ nr(G);
  edge_array⟨int⟩ cost(G);
  int cur_nr = 0;
  int n = G.number_of_nodes();
  node v;
  edge e;
  forall_nodes (v, G) nr[v] = cur_nr++;
  forall_edges (e, G)
    cost[e] = ((nr[source(e)] < nr[target(e)]) ? n * nr[source(e)] + nr[target(e)] :
          n * nr[target(e)] + nr[source(e)]);
  G.sort_edges(cost);
  list⟨edge⟩ L = G.all_edges();
  while (¬L.empty()) {
    e = L.pop();
    /* check whether the first edge on L is equal to the reversal of e. If so, delete it from
    L, if not, add the reversal to G */
    if (¬L.empty() ∧ (source(e) ≡ target(L.head())) ∧ (source(L.head()) ≡ target(e)))
      L.pop();
    else {
      G.new_edge(target(e), source(e));
      Gin_is_bidirected = false;
    }
  }
}
```

This code is used in section 6.

8. Making the Graph Biconnected.

We make G biconnected. We first make it connected by linking all roots of a DFS-forest. Assume now that G is conected. Let a be any articulation point and let u and v be neighbors of a belonging to different biconnected components. Then there are embeddings of the two components with the edges $\{u, a\}$ and $\{v, a\}$ on the boundary of the unbounded face. Hence we may add the edge $\{u, v\}$ without destroying planarity. Proceeding in this way we make G biconnected.

In *Make_biconnected_graph* we change the graph while working on it. But we modify only those adjacency lists that will not be touched later.

We need the biconnected version of G (G will be further modified during the planarity test) in order to construct the planar embedding. So we store it as a graph H. For every edge of *Gin* and G we store a link to its copy in H. In addition every edge of H is made to know its reversal.

```
⟨ make G biconnected 8 ⟩ ≡
  Make_biconnected_graph(G);
  ⟨ make H a copy of G 12 ⟩;
```

This code is used in section 5.

9. We give the details of the procedure *Make_biconnected_graph*. We first make G connected by linking all roots of the DFS-forest. In a second step we make G biconnected.

```
⟨ auxiliary functions 9 ⟩ ≡
  void Make_biconnected_graph(graph &G)
  {
    node v;
    node_array⟨bool⟩ reached(G, false);
    node u = G.first_node( );
    forall_nodes (v, G) {
      if (¬reached[v]) {
        /* explore the connected component with root v */
        DFS(G, v, reached);
        if (u ≠ v) {
          /* link v's component to the first component */
          G.new_edge(u, v);
          G.new_edge(v, u);
        }    //end if
      }    // end not reached
    }    //end forall
    /* G is now connected. We next make it biconnected. */
    forall_nodes (v, G) reached[v] = false;
    node_array⟨int⟩ dfsnum(G);
    node_array⟨node⟩ parent(G, nil);
    int dfs_count = 0;
    node_array⟨int⟩ lowpt(G);
    dfs_in_make_biconnected_graph(G, G.first_node( ), dfs_count, reached, dfsnum, lowpt,
        parent);
  }    // end Make_biconnected_graph
```

See also sections 10, 15, 16, 18, and 27.

This code is used in section 2.

10. We still have to give the procedure *dfs_in_make_biconnected_graph*. It determines articulation points and adds appropriate edges whenever it discovers one. For a proof of correctness we refer the reader to [Meh84, section IV.6].

```
⟨ auxiliary functions 9 ⟩ +≡
  void dfs_in_make_biconnected_graph(graph &G, node v, int &dfs_count,
          node_array⟨bool⟩ &reached,
          node_array⟨int⟩ &dfsnum, node_array⟨int⟩ &lowpt, node_array⟨node⟩
          &parent)
  {
    node w;
    edge e;
    dfsnum[v] = dfs_count++;
    lowpt[v] = dfsnum[v];
    reached[v] = true;
    if (¬G.first_adj_edge(v)) return;      // no children
    node u = target(G.first_adj_edge(v));      // first child
    forall_adj_edges (e, v) {
      w = target(e);
      if (¬reached[w]) {
        /* e is a tree edge */
        parent[w] = v;
        dfs_in_make_biconnected_graph(G, w, dfs_count, reached, dfsnum, lowpt, parent);
        if (lowpt[w] ≡ dfsnum[v]) {
          /* v is an articulation point. We now add an edge. If w is the first child and v
            has a parent then we connect w and parent[v], if w is a first child and v has no
            parent then we do nothing. If w is not the first child then we connect w to the
            first child. The net effect of all of this is to link all children of an articulation
            point to the first child and the first child to the parent (if it exists) */
          if (w ≡ u ∧ parent[v]) {
            G.new_edge(w, parent[v]);
            G.new_edge(parent[v], w);
          }
          if (w ≠ u) {
            G.new_edge(u, w);
            G.new_edge(w, u);
          }
        }     // end if lowpt = dfsnum
        lowpt[v] = Min(lowpt[v], lowpt[w]);
      }     // end tree edge
      else lowpt[v] = Min(lowpt[v], dfsnum[w]);      // non tree edge
    }     // end forall
  }     // end dfs
```

11. Because we use the function *dfs_in_make_biconnected_graph* before its declaration, let's add it to the global declarations.

⟨ typedefs, global variables and class declarations 11 ⟩ ≡

```
void dfs_in_make_biconnected_graph(graph &G, node v, int &dfs_count,
     node_array⟨bool⟩ &reached, node_array⟨int⟩ &dfsnum, node_array⟨int⟩
     &lowpt, node_array⟨node⟩ &parent);
```

See also sections 14, 17, and 20.

This code is used in section 2.

12. We make H a copy of G and create bidirectional links between the vertices and edges of G and H. Also, each edge in *Gin* gets a link to its copy in H and every edge of H gets to know its reversal. More precisely, $H[G[v]] = v$ for every node v of G and $H[G[e]] = e$ for every edge e of G, and *companion_in_H*[*ein*] is the edge in H corresponding to the edge *ein* of *Gin* for every edge *ein* of *Gin*. Finally, if $e = (u, v)$ is an edge of H then $reversal[e] = (v, u)$.

⟨ make H a copy of G 12 ⟩ ≡

```
GRAPH⟨node, edge⟩ H;
edge_array⟨edge⟩ companion_in_H(Gin);
{
  node v;
  forall_nodes (v, G) G.assign(v, H.new_node(v));
  edge e;
  forall_edges (e, G) G.assign(e, H.new_edge(G[source(e)], G[target(e)], e));
  edge ein;
  forall_edges (ein, Gin) companion_in_H[ein] = G[companion_in_G[ein]];
}
edge_array⟨edge⟩ reversal(H);
compute_correspondence(H, reversal);
```

This code is used in section 8.

13. The Planarity Test.

We are now ready for the planarity test proper. We follow [Meh84, page 95]. We first compute *dfsnumbers* and *parents*, we delete all forward edges and all reversals of tree edges, and we reorder the adjaceny lists as described in [Meh84, page 101]. We then test the strong planarity. The array *alpha* is needed for the embedding process. It records the placement of the subsegments.

⟨ test planarity 13 ⟩ ≡

```
node_array<int> dfsnum(G);
node_array<node> parent(G, nil);
reorder(G, dfsnum, parent);
edge_array<int> alpha(G, 0);
{
  list<int> Att;
  alpha[G.first_adj_edge(G.first_node())] = left;
  if (¬strongly_planar(G.first_adj_edge(G.first_node()), G, Att, alpha, dfsnum, parent))
    return false;
}
```

This code is used in section 5.

14. We need two global constants *left* and *right*.

⟨ typedefs, global variables and class declarations 11 ⟩ +≡

```
const int left = 1;
const int right = 2;
```

15. We give the details of the procedure *reorder*. It first performs DFS to compute *dfsnum*, *parent*, *lowpt1* and *lowpt2*, and the list *Del* of all forward edges and all reversals of tree edges. It then deletes the edges in *Del* and finally it reorders the edges.

⟨ auxiliary functions 9 ⟩ +≡

```
void reorder(graph &G, node_array<int> &dfsnum, node_array<node> &parent)
{
  node v;
  node_array<bool> reached(G, false);
  int dfs_count = 0;
  list<edge> Del;
  node_array<int> lowpt1(G), lowpt2(G);
  dfs_in_reorder(Del, G.first_node(), dfs_count, reached, dfsnum, lowpt1, lowpt2, parent);
  /* remove forward and reversals of tree edges */
  edge e;
  forall (e, Del) G.del_edge(e);
  /* we now reorder adjacency lists as described in [Meh84, page 101] */
  node w;
  edge_array<int> cost(G);
```

```
forall_edges (e, G) {
   v = source(e);
   w = target(e);
   cost[e] = ((dfsnum[w] < dfsnum[v]) ? 2 * dfsnum[w] : ((lowpt2[w] ≥ dfsnum[v]) ?
         2 * lowpt1[w] : 2 * lowpt1[w] + 1));
}
G.sort_edges(cost);
}
```

16. We still have to give the procedure *dfs_in_reorder*. It's a bit long but standard.

⟨ auxiliary functions 9 ⟩ +≡

```
void dfs_in_reorder(list⟨edge⟩ &Del, node v, int &dfs_count, node_array⟨bool⟩
         &reached,
         node_array⟨int⟩ &dfsnum, node_array⟨int⟩ &lowpt1, node_array⟨int⟩
         &lowpt2,
         node_array⟨node⟩ &parent)
{
   node w;
   edge e;
   dfsnum[v] = dfs_count++;
   lowpt1[v] = lowpt2[v] = dfsnum[v];
   reached[v] = true;
   forall_adj_edges (e, v) {
      w = target(e);
      if (¬reached[w]) {
         /* e is a tree edge */
         parent[w] = v;
         dfs_in_reorder(Del, w, dfs_count, reached, dfsnum, lowpt1, lowpt2, parent);
         lowpt1[v] = Min(lowpt1[v], lowpt1[w]);
      }     //end tree edge
      else {
         lowpt1[v] = Min(lowpt1[v], dfsnum[w]);     // no effect for forward edges
         if ((dfsnum[w] ≥ dfsnum[v]) ∨ w ≡ parent[v])
            /* forward edge or reversal of tree edge */
            Del.append(e);
      }     //end non-tree edge
   }     // end forall
   /* we know lowpt1[v] at this point and now make a second pass over all adjacent edges
   of v to compute lowpt2 */
   forall_adj_edges (e, v) {
      w = target(e);
      if (parent[w] ≡ v) {
         /* tree edge */
         if (lowpt1[w] ≠ lowpt1[v]) lowpt2[v] = Min(lowpt2[v], lowpt1[w]);
         lowpt2[v] = Min(lowpt2[v], lowpt2[w]);
      }     //end tree edge
      else     // all other edges
      if (lowpt1[v] ≠ dfsnum[w]) lowpt2[v] = Min(lowpt2[v], dfsnum[w]);
   }     //end forall
}     //end dfs
```

17. Because we use the function *dfs_in_reorder* before its declaration, let's add it to the global declarations.

$\langle$ typedefs, global variables and class declarations 11 $\rangle$ +≡

void *dfs_in_reorder*(**list**⟨**edge**⟩ &*Del*, **node** *v*, **int** &*dfs_count*, **node_array**⟨**bool**⟩ &*reached*,
node_array⟨**int**⟩ &*dfsnum*, **node_array**⟨**int**⟩ &*lowpt1*, **node_array**⟨**int**⟩ &*lowpt2*,
node_array⟨**node**⟩ &*parent*);

18. We now come to the heart of the planarity test: procedure *strongly_planar*. It takes a tree edge $e0 = (x, y)$ and tests whether the segment $S(e0)$ is strongly planar. If successful it returns (in *Att*) the ordered list of attachments of $S(e0)$ (excluding x); high DFS-numbers are at the front of the list. In *alpha* it records the placement of the subsegments.

strongly_planar operates in three phases. It first constructs the cycle $C(e0)$ underlying the segment $S(e0)$. It then constructs the interlacing graph for the segments emanating from the spine of the cycle. If this graph is non-bipartite then the segment $S(e0)$ is non-planar. If it is bipartite then the segment is planar. In this case the third phase checks whether the segment is strongly planar and, if so, computes its list of attachments.

$\langle$ auxiliary functions 9 $\rangle$ +≡

```
bool strongly_planar(edge e0, graph &G, list<int> &Att, edge_array<int> &alpha,
        node_array<int> &dfsnum, node_array<node> &parent)
{
  ⟨ determine the cycle C(e0) 19 ⟩;
  ⟨ process all edges leaving the spine 21 ⟩;
  ⟨ test strong planarity and compute Att 25 ⟩;
  return true;
}
```

19. We determine the cycle $C(e0)$ by following first edges until a back edge is encountered. *wk* will be the last node on the tree path and *w0* is the destination of the back edge. This agrees with the notation of [Meh84].

$\langle$ determine the cycle $C(e0)$ 19 $\rangle$ ≡

```
node x = source(e0);
node y = target(e0);
edge e = G.first_adj_edge(y);
node wk = y;
while (dfsnum[target(e)] > dfsnum[wk])      // e is a tree edge
{
  wk = target(e);
  e = G.first_adj_edge(wk);
}
node w0 = target(e);
```

This code is used in section 18.

20. The second phase of *strongly_planar* constructs the connected components of the interlacing graph of the segments emananating from the the spine of the cycle $C(e0)$. We call a connected component a *block*. For each block we store the segments comprising its left and right side (lists *Lseg* and *Rseg* contain the edges defining these segments) and the ordered list of attachments of the segments in the block; lists *Latt* and *Ratt* contain the DFS-numbers of the attachments; high DFS-numbers are at the front of the list. Blocks are so important that we make them a class.

We need the following operations on blocks.

The constructor takes an edge and a list of attachments and creates a block having the edge as the only segment in its left side.

flip interchanges the two sides of a block.

head_of_Latt and *head_of_Ratt* return the first elements on *Latt* and *Ratt* respectively and *Latt_empty* and *Ratt_empty* check these lists for emptyness.

left_interlace checks whether the block interlaces with the left side of the topmost block of stack S. *right_interlace* does the same for the right side.

combine combines the block with another block *Bprime* by simply concatenating all lists.

clean removes the attachment w from the block B (it is guaranteed to be the first attachment of B). If the block becomes empty then it records the placement of all segments in the block in the array *alpha* and returns true. Otherwise it returns false.

add_to_Att first makes sure that the right side has no attachment above *w0* (by flipping); when *add_to_Att* is called at least one side has no attachment above *w0*. *add_to_Att* then adds the lists *Ratt* and *Latt* to the output list *Att* and records the placement of all segments in the block in *alpha*. We advise the reader to only skim the rest of the section at this point and to come back to it when the procedures are first used.

⟨ typedefs, global variables and class declarations 11 ⟩ +≡

```
class block {
private: list<int> Latt, Ratt;    // list of attachments
  list<edge> Lseg, Rseg;    // list of segments represented by their defining edges
public: block(edge e, list<int> &A)
  {
    Lseg.append(e);
    Latt.conc(A);    // the other two lists are empty
  }
  ~block() { }
  void flip()
  {
    list<int> ha;
    list<edge> he;
    /* we first interchange Latt and Ratt and then Lseg and Rseg */
    ha.conc(Ratt); Ratt.conc(Latt); Latt.conc(ha);
    he.conc(Rseg); Rseg.conc(Lseg); Lseg.conc(he);
  }
  int head_of_Latt() { return Latt.head(); }
  bool empty_Latt() { return Latt.empty(); }
  int head_of_Ratt() { return Ratt.head(); }
  bool empty_Ratt() { return Ratt.empty(); }
```

```
bool left_interlace(stack<block *> &S)
{ /* check for interlacing with the left side of the topmost block of S */
  if (Latt.empty()) error_handler(1, "Latt is never empty");
  if (¬S.empty() ∧ ¬((S.top())→empty_Latt()) ∧ Latt.tail() < (S.top())→head_of_Latt())
    return true;
  else return false;
}
bool right_interlace(stack<block *> &S)
{ /* check for interlacing with the right side of the topmost block of S */
  if (Latt.empty()) error_handler(1, "Latt is never empty");
  if (¬S.empty() ∧ ¬((S.top())→empty_Ratt()) ∧ Latt.tail() < (S.top())→head_of_Ratt())
    return true;
  else return false;
}
void combine(block *&Bprime)
{ /* add block Bprime to the rear of this block */
  Latt.conc(Bprime→Latt);
  Ratt.conc(Bprime→Ratt);
  Lseg.conc(Bprime→Lseg);
  Rseg.conc(Bprime→Rseg);
  delete (Bprime);
}
bool clean(int dfsnum_w, edge_array<int> &alpha, node_array<int> &dfsnum)
{ /* remove all attachments to w; there may be several */
  while (¬Latt.empty() ∧ Latt.head() ≡ dfsnum_w) Latt.pop();
  while (¬Ratt.empty() ∧ Ratt.head() ≡ dfsnum_w) Ratt.pop();
  if (¬Latt.empty() ∨ ¬Ratt.empty()) return false;
  /* Latt and Ratt are empty; we record the placement of the subsegments in alpha.
  */
  edge e;
  forall (e, Lseg) alpha[e] = left;
  forall (e, Rseg) alpha[e] = right;
  return true;
}
void add_to_Att(list<int> &Att, int dfsnum_w0, edge_array<int> &alpha,
        node_array<int> &dfsnum)
{ /* add the block to the rear of Att. Flip if necessary */
  if (¬Ratt.empty() ∧ head_of_Ratt() > dfsnum_w0) flip();
  Att.conc(Latt);
  Att.conc(Ratt);
  /* This needs some explanation. Note that Ratt is either empty or {w0}. Also if
  Ratt is non-empty then all subsequent sets are contained in {w0}. So we indeed
  compute an ordered set of attachments. */
  edge e;
  forall (e, Lseg) alpha[e] = left;
  forall (e, Rseg) alpha[e] = right;
}
};
```

21. We process the edges leaving the spine of $S(e0)$ starting at node wk and working backwards. The interlacing graph of the segments emanating from the cycle is represented as a stack S of blocks.

⟨ process all edges leaving the spine 21 ⟩ ≡

```
node w = wk;
stack⟨block *⟩ S;
while (w ≠ x) {
  int count = 0;
  forall_adj_edges (e, w) {
    count ++;
    if (count ≠ 1)      // no action for first edge
    {
      ⟨ test recursively 22 ⟩;
      ⟨ update stack S of attachments 23 ⟩;
    }     // end if
  }     // end forall
  ⟨ prepare for next iteration 24 ⟩;
  w = parent[w];
}     // end while
```

This code is used in section 18.

22. Let e be any edge leaving the spine. We need to test whether $S(e)$ is strongly planar and if so compute its list A of attachments. If e is a tree edge we call our procedure recursively and if e is a back edge then $S(e)$ is certainly strongly planar and $target(e)$ is the only attachment. If we detect non-planarity we return flase and free the storage allocated for the blocks of stack S.

⟨ test recursively 22 ⟩ ≡

```
list⟨int⟩ A;
if (dfsnum[w] < dfsnum[target(e)]) {
  /* tree edge */
  if (¬strongly_planar(e, G, A, alpha, dfsnum, parent)) {
    while (¬S.empty()) delete (S.pop());
    return false;
  }
}
else A.append(dfsnum[target(e)]);      // a back edge
```

This code is used in section 21.

23. The list A contains the ordered list of attachments of segment $S(e)$. We create an new block consisting only of segment $S(e)$ (in its L-part) and then combine this block with the topmost block of stack S as long as there is interlacing. We check for interlacing with the L-part. If there is interlacing then we flip the two sides of the topmost block. If there is still interlacing with the left side then the interlacing graph is non-bipartite and we declare the graph non-planar (and also free the storage allocated for the blocks). Otherwise we check for interlacing with the R-part. If there is interlacing then we combine B with the topmost block and repeat the process with the new topmost block. If there is no interlacing then we push block B onto S.

⟨ update stack S of attachments 23 ⟩ ≡

```
block *B = new block (e, A);
while (true) {
  if (B→left_interlace(S)) (S.top())→flip();
  if (B→left_interlace(S)) {
    delete (B);
    while (¬S.empty()) delete (S.pop());
    return false;
  }
  ;
  if (B→right_interlace(S)) B→combine(S.pop());
  else break;
}    // end while
S.push(B);
```

This code is used in section 21.

24. We have now processed all edges emanating from vertex w. Before starting to process edges emanating from vertex *parent*[w] we remove *parent*[w] from the list of attachments of the topmost block of stack S. If this block becomes empty then we pop it from the stack and record the placement for all segments in the block in array *alpha*.

⟨ prepare for next iteration 24 ⟩ ≡

```
while (¬S.empty() ∧ (S.top())→clean(dfsnum[parent[w]], alpha, dfsnum))
  delete (S.pop());
```

This code is used in section 21.

25. We test the strong planarity of the segment $S(e0)$.

We know at this point that the interlacing graph is bipartite. Also for each of its connected components the corresponding block on stack S contains the list of attachments below x. Let B be the topmost block of S. If both sides of B have an attachment above *w0* then $S(e0)$ is not strongly planar. We free the storage allocated for the blocks and return false. Otherwise (cf. procedure *add_to_Att*) we first make sure that the right side of B attaches only to *w0* (if at all) and then add the two sides of B to the output list *Att*. We also record the placements of the subsegments in *alpha*.

⟨ test strong planarity and compute *Att* 25 ⟩ ≡

```
Att.clear();
while (¬S.empty()) {
  block *B = S.pop();
  if (¬(B→empty_Latt()) ∧ ¬(B→empty_Ratt()) ∧ (B→head_of_Latt() >
        dfsnum[w0]) ∧ (B→head_of_Ratt() > dfsnum[w0])) {
    delete (B);
    while (¬S.empty()) delete (S.pop());
    return false;
  }
  B→add_to_Att(Att, dfsnum[w0], alpha, dfsnum);
  delete (B);
```

```
}      // end while
/* Let's not forget (as the book does) that w0 is an attachment of S(e0) except if w0 = x.
*/
if (w0 ≠ x) Att.append(dfsnum[w0]);
```

This code is used in section 18.

26. Constructing the Embedding.

We now discuss how the planarity testing algorithm can be extended so that it also computes a planar map. Consider a segment $S(e_0) = C + S(e_1) + \ldots + S(e_m)$ consisting of cycle C and emanating segments $S(e_1), \ldots, S(e_m)$ and recall that the proofs of Lemmas 8 and 9 describe how the embeddings of the $S(e_i)$'s have to be combined to yield a canonical embedding of $S(e_0)$. Our goal is to turn these proofs into an efficient algorithm.

The proofs of Lemmas 8 and 9 demonstrate two things:

- How to test whether $IG(C)$ is bipartite and how to construct a partition $\{L, R\}$ of its vertex set, and
- how to construct an embedding of $S(e_0)$ from the embeddings of the $S(e_i)$'s. This involves flipping of embeddings as we incrementally construct the embedding of $S(e_0)$.

Suppose now that some benign agent told us that $IG(C)$ were bipartite and gave us an appropriate partition $\{L, R\}$ of its vertex set, i.e., a partition $\{L, R\}$ such that no two segments in L and no two segments in R interlace and such that $A(e_i) \cap \{w_1, \ldots, w_{r-1}\} = \emptyset$ for any segment $S(e_i) \in R$. Here, as before, $w_0, \ldots, w_r$ denotes the stem of C. Then no flipping would ever be necessary; we can simply combine the embeddings of the $S(e_i)$'s as prescribed by the partition $\{L, R\}$. More precisely, to construct a canonical embedding of $S(e_0)$ draw the path $w_0, \ldots, w_k$ (consisting of stem $w_0, \ldots, w_r$, edge $e_0 = (w_r, w_{r+1})$ and spine $w_{r+1}, \ldots, w_k$) as a vertical upwards directed path, add edge (w_k, w_0), and then for i, $1 \le i \le m$, and $S(e_i) \in L$ extend the embedding of $C + S(e_1) + \ldots S(e_{i-1})$ by glueing a canonical embedding of $S(e_i)$ onto the left side of the vertical path, and for i, $1 \le i \le m$, and $S(e_i) \in R$ extend the embedding of $C + S(e_1) + \ldots + S(e_{i-1})$ by glueing a reversed canonical embedding of $S(e_i)$ onto the right side of the vertical path. Similarly, if the goal is to construct a reversed canonical embedding of $S(e_0)$ then, if $S(e_i) \in L$, a reversed canonical embedding of $S(e_i)$ is glued onto the right side of the vertical path, and if $S(e_i) \in R$, then a canonical embedding of $S(e_i)$ is glued onto the left side of the vertical path.

Who is the benign agent which tells us that $IG(C)$ is bipartite and gives us the appropriate partition $\{L, R\}$ of the segments emanating from $C = C(e_0)$? It's the call *stronglyplanar*(e_0). After all, it tests whether $IG(C)$ is bipartite and computes a bipartition of its vertex set. Let $S(e)$ be a segment emanating from C and let B be the connected component of $IG(C)$ containing $S(e)$. The call *stronglyplanar*(e_0) computes B iteratively. The construction of B is certainly completed when B is popped from stack S. Put $S(e)$ into R when $S(e) \in RB$ at that moment and put $S(e)$ into L otherwise. With this extension, algorithm *stronglyplanar* computes the partition $\{L, R\}$ of the segments emanating from C in linear time. We assume for notational convenience that the partition (more precisely, the union of all partitions for all cycles $C(e_0)$ encountered in the algorithm) is given as a function $\alpha : S \to \{L, R\}$ where S is the set of edges for which *stronglyplanar* is called.

We next give the algorithmic details of the embedding process. We first use procedure *stronglyplanar* to compute the mapping α. We then use a procedure *embedding* to actually compute an embedding. The procedure *embedding* takes two parameters: an edge e_0 and a flag $t \in \{L, R\}$. A call *embedding*(e_0, L) computes a canonical embedding of $S(e_0)$ and a call *embedding*(e_0, R) computes a reversed canonical embedding of $S(e_0)$. The call *embedding*$((1,2), L)$ embeds the entire graph.

The embedding of $S(e_0)$ computed by *embedding*(e_0, t) is represented in the following nonstandard way:

1. For the vertices $v \in V(e_0)$ we use the standard representation, i.e., the cyclic list of the incident edges corresponding to the clockwise ordering of the edges in the embedding.

2. For the vertices in the stem we use a non-standard representation. For each vertex $w_i \in \{w_0, \ldots, w_r\}$ let the lists $AL(w_i)$ and $AR(w_i)$ be such that the catenation of (w_i, w_{i+1}), $AR(w_i)$, (w_i, w_{i-1}), and $AL(w_i)$ corresponds to the clockwise ordering of the edges incident to w_i in the embedding. Here, $w_{-1} = w_k$. Note that $AR(w_i) = \emptyset$ for $1 \le i < r$ if $t = L$, and $AL(w_i) = \emptyset$ for $1 \le i < r$, if $t = R$. The lists $AL(w_i)$, $AR(w_i)$, $0 \le i \le r$, are returned in an implicit way: $AL(w_r)$ and $AR(w_r)$ are returned as the list $T = AL(w_r), (w_r, w_{r+1}), AR(w_r)$ and the other lists are returned as the list $A = AR(w_{r-1}), \ldots, AR(w_0), (w_0, w_k), AL(w_0), \ldots, AL(w_{r-1})$, cf. Figure 1.

The procedure *embedding* has the same structure as the procedure *stronglyplanar* and is given in Program 1 on page 18. It first constructs the stem and the spine (line (1)) of cycle $C(e_0)$, then walks down the spine (lines (3) to (14)), and finally computes the lists T and A it wants to return (lines (15) and (16)).

We first discuss the walk down the spine. Suppose that the walk has reached vertex w_j. We first recursively process the edges emanating from w_j (lines (4) to (10)), and then compute the cyclic adjacency list of vertex w_j and prepare for the next iteration (lines (11) to (13)).

We discuss lines (4) to (10) first. In general, some number of edges emanating from w_j and all edges incident to vertices w_l with $l > j$ will have been processed already. In agreement with our previous notation call the processed edges $e_1, \ldots, e_{i-1}$. We claim that the following statement is an invariant of the loop (4) to (10): T concatenated with (w_j, w_{j-1}) is the cyclic adjacency list of vertex w_j in the embedding of $C + S(e_1) + \ldots + S(e_{i-1})$, and AL and AR are the catenation of lists $AL(w_0), \ldots, AL(w_{j-1})$ and $AR(w_{j-1}), \ldots, AR(w_0)$ respectively where $(w_l, w_{l+1}), AR(w_l), (w_l, w_{l-1}), AL(w_l)$ is the cyclic adjacency list of vertex w_l, $0 \le l \le j-1$, in the embedding of $C + S(e_0) + \ldots + S(e_{i-1})$. The lists T, AL, and AR are certainly initialized correctly in line (2). Assume now that we process edge $e' = e_i$ emanating from w_j. The flag $\alpha(e')$ indicates what kind of embedding of $S(e_i)$ is needed to build a canonical embedding of $S(e_0)$; the opposite kind of embedding of $S(e_i)$ is needed to build a reversed canonical embedding of $S(e_0)$. So the required kind is given by $t \oplus \alpha(e')$, where $L \oplus L = R \oplus R = L$ and $L \oplus R = R \oplus L = R$. The call *embedding*$(e', t \oplus \alpha(e'))$ computes the cyclic adjacency lists of the vertices in $V(e')$ and returns lists T' and A' as defined above. If $S(e_i)$ has to be glued to the left side of the vertical path $w_0, \ldots, w_k$, i.e., if $t = \alpha(e')$ then we append T' to the front of T and A' to the end of AL, cf. Figure 2. Analogously, if $S(e_i)$ has to be glued to the right side of the path $w_0, \ldots, w_k$, i.e., if $t \ne \alpha(e')$, then we append T' to the end of T and A' to the front of AR. This clearly maintains the invariant.

Suppose now that we have processed all edges emanating from w_j. Then (w_j, w_{j-1}) concatenated with T is the cyclic adjacency list of vertex w_j (line (11)).

We next prepare for the treatment of vertex w_{j-1}. Let T' and T'' be the list of darts incident to w_{j-1} from the left and from the right respectively and having their other endpoint in an already embedded segment. List T' is a suffix of AL and list T'' is a prefix of AR. The catenation of $T', (w_{j-1}, w_j)$, T'', and (w_{j-1}, w_{j-2}) is the current clockwise adjacency list of vertex w_{j-1}. Thus lines (12) and (13) correctly initialize AL, AR, and T for the next iteration.

Suppose now that all edges emanating from the spine of $C(e_0)$ have been processed, i.e., control reaches line (15). At this point, list T is the ordered list of darts incident to w_r (except (w_r, w_{r-1})) and the two lists AL and AR are the ordered list of darts incident to the two sides of the stem of $C(e_0)$. Thus T and the catenation of $AR, (w_0, w_k)$, and AL are the two components of the output of *embedding*(e_0, t). We summarize in

```
(0)  procedure embedding(e_0: edge, t: {L, R})
     (* computes an embedding of S(e_0), e_0 = (x, y), as described in the text;
        it returns the lists T and A defined in the text *)
(1)  find the spine of segment S(e_0) by starting in node y and always
        take the first edge of every adjacency list until a back edge is
        encountered. This back edge leads to node w_0 = lowpt[y].
        Let w_0, ..., w_r be the tree path from w_0 to x = w_r and
        let w_{r+1} = y, ..., w_k be the spine constructed above.
(2)  AL ← AR ← empty list of darts;
        T ← (w_k, w_0);                                          (* a list of darts *)
(3)  for j from k downto r + 1
(4)  do for all edges e' (except the first) emanating from w_j
(5)     do (T', A') ← embedding(e', t ⊕ α(e'))
(6)        if t = α(e')
(7)        then T ← T' conc T; AL ← AL conc A'
(8)        else T ← T conc T'; AR ← A' conc AR
(9)        fi
(10)    od
(11)    output (w_j, w_{j-1}) conc T;          (* the cyclic adjacency list of vertex w_j *)
(12)    let AL = AL' conc T' and AR = T'' conc AR'
           where T' and T'' contain all darts incident to w_{j-1};
(13)    AL ← AL'; AR ← AR'; T ← T' conc (w_{j-1}, w_j) conc T''
(14) od
(15) A ← AR conc (w_0, w_k) conc AL;
(16) return T and A
(17) end
```

Program 1

Theorem 1 *Let $G = (V, E)$ be a planar graph. Then G can be turned into a planar map (G, σ) in linear time.*

In our implementation we follow the book except in three minor points. G has only one directed version of each edge but H has both. In the embedding phase we need both directions and therefore construct the embedding of H and later translate it back to *Gin*. Secondly, we do not construct the embedding of H vertex by vertex but in one shot. To that effect we compute a labelling *sort_num* of the edges of H and later sort the edges. Thirdly, the book makes reference to edges (w_{i-1}, w_i) and their reversals. To make these edges available we compute an array *tree_edge_into* that contains for each node the incoming tree edge.

We finally want to remark on our convention for drawing lists. In Figures 1 and 2 the arrows indicate the end (!!!) of the lists.

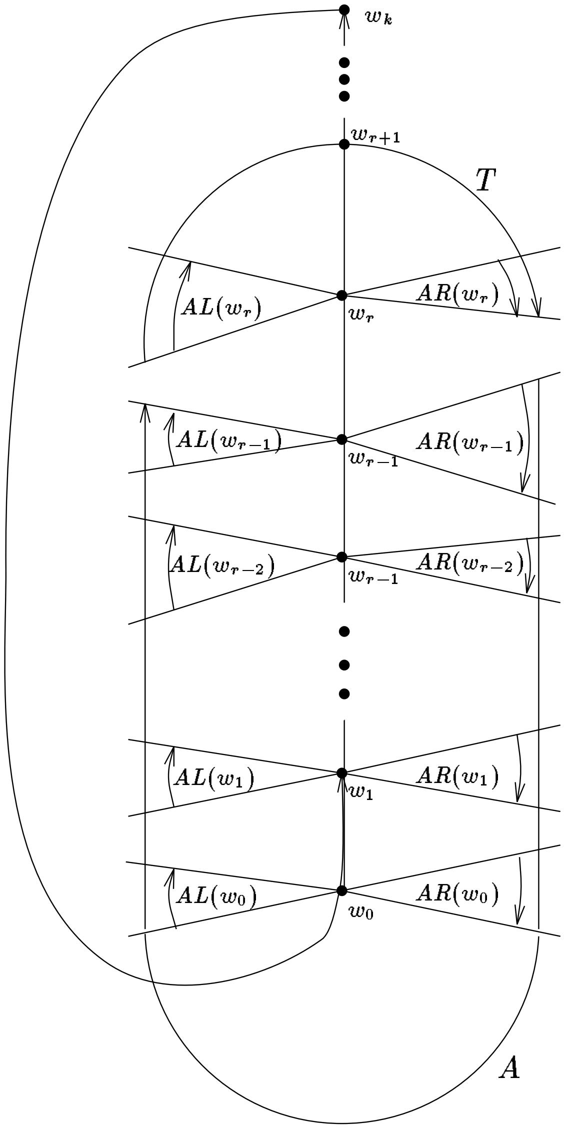

Figure 1: A call *embedding* (e_0, t) returns lists T and A.

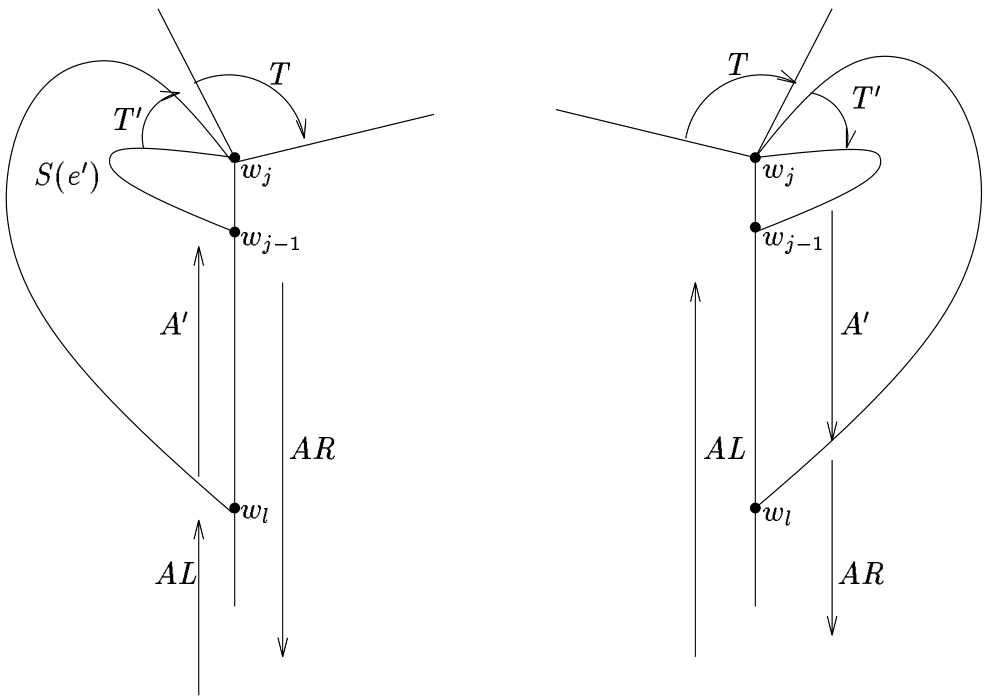

Figure 2: Glueing $S(e')$ to the left or right side of the path $w_0, \ldots, w_k$ respectively.

⟨ construct embedding 26 ⟩ ≡

```
{
  list⟨edge⟩ T, A;      // lists of edges of H
  int cur_nr = 0;
  edge_array⟨int⟩ sort_num(H);
  node_array⟨edge⟩ tree_edge_into(G);
  embedding(G.first_adj_edge(G.first_node()), left, G, alpha, dfsnum, T, A, cur_nr,
      sort_num, tree_edge_into, parent, reversal);
  /* T contains all edges incident to the first node except the cycle edge into it. That
  edge comprises A */
  T.conc(A);
  edge e;
  forall (e, T) sort_num[e] = cur_nr++;
  edge_array⟨int⟩ sort_Gin(Gin);
  {
    edge ein;
    forall_edges (ein, Gin) sort_Gin[ein] = sort_num[companion_in_H[ein]];
  }
  Gin.sort_edges(sort_Gin);
}
```

This code is used in section 5.

27. It remains to describe procedure *embedding*.

⟨ auxiliary functions 9 ⟩ +≡

```
void embedding(edge e0, int t, GRAPH⟨node, edge⟩ &G, edge_array⟨int⟩ &alpha,
        node_array⟨int⟩ &dfsnum, list⟨edge⟩ &T, list⟨edge⟩ &A, int &cur_nr,
        edge_array⟨int⟩ &sort_num, node_array⟨edge⟩ &tree_edge_into,
        node_array⟨node⟩ &parent, edge_array⟨edge⟩ &reversal)
{
  ⟨ embed: determine the cycle C(e0) 28 ⟩;
  ⟨ process the subsegments 29 ⟩;
  ⟨ prepare the output 33 ⟩;
}
```

28. We start by determining the spine cycle. This is precisley as in *strongly_planar*. We also record for the vertices $w_{r+1}, \ldots, w_k$, and w_0 the incoming cycle edge either in *tree_edge_into* or in the local variable *back_edge_into_w0*. This is line (1) of Program1.

⟨ embed: determine the cycle $C(e0)$ 28 ⟩ ≡

```
node x = source(e0);
node y = target(e0);
tree_edge_into[y] = e0;
edge e = G.first_adj_edge(y);
node wk = y;
```

```
while (dfsnum[target(e)] > dfsnum[wk])      // e is a tree edge
{
   wk = target(e);
   tree_edge_into[wk] = e;
   e = G.first_adj_edge(wk);
}
node w0 = target(e);
edge back_edge_into_w0 = e;
```

This code is used in section 27.

29. Lines (2) to (14).

⟨ process the subsegments 29 ⟩ ≡

```
node w = wk;
list<edge> Al, Ar;
list<edge> Tprime, Aprime;
T.clear();
T.append(G[e]);      // e = (wk, w0) at this point, line (2)
while (w ≠ x) {
   int count = 0;
   forall_adj_edges (e, w) {
      count++;
      if (count ≠ 1)      //no action for first edge
      {
         ⟨ embed recursively 30 ⟩;
         ⟨ update lists T, Al, and Ar 31 ⟩;
      }      // end if
   }      //end forall
   ⟨ compute w's adjacency list and prepare for next iteration 32 ⟩;
   w = parent[w];
}      //end while
```

This code is used in section 27.

30. Line (5). The book does not distinguish between tree and back edges but we do here.

⟨ embed recursively 30 ⟩ ≡

```
if (dfsnum[w] < dfsnum[target(e)]) {
   /* tree edge */
   int tprime = ((t ≡ alpha[e]) ? left : right);
   embedding(e, tprime, G, alpha, dfsnum, Tprime, Aprime, cur_nr, sort_num,
        tree_edge_into, parent, reversal);
}
else {
   /* back edge */
   Tprime.append(G[e]);      //e
   Aprime.append(reversal[G[e]]);      //reversal of e
}
```

This code is used in section 29.

31. Lines (6) to (9).

```
⟨ update lists T, Al, and Ar 31 ⟩ ≡
  if (t ≡ alpha[e]) {
    Tprime.conc(T);
    T.conc(Tprime);      // T = Tprime conc T
    Al.conc(Aprime);      // Al = Al conc Aprime
  }
  else {
    T.conc(Tprime);      // T  = T conc Tprime
    Aprime.conc(Ar);
    Ar.conc(Aprime);      // Ar  = Aprime conc Ar
  }
```

This code is used in section 29.

32. Lines (11) to (13).

```
⟨ compute w's adjacency list and prepare for next iteration 32 ⟩ ≡
  T.append(reversal[G[tree_edge_into[w]]]);      // (w_{j-1}, w_j)
  forall (e, T) sort_num[e] = cur_nr++;
  /* w's adjacency list is now computed; we clear T and prepare for the next iteration by
  moving all darts incident to parent[w] from Al and Ar to T. These darts are at the rear
  end of Al and at the front end of Ar */
  T.clear();
  while (¬Al.empty() ∧ source(Al.tail()) ≡ G[parent[w]])
      // parent[w] is in G, Al.tail in H
  {
    T.push(Al.Pop());      // Pop removes from the rear
  }
  T.append(G[tree_edge_into[w]]);      // push would be equivalent
  while (¬Ar.empty() ∧ source(Ar.head()) ≡ G[parent[w]])      //
  {
    T.append(Ar.pop());      // pop removes from the front
  }
```

This code is used in section 29.

33. Line (15). Concatenate Ar, (w_0, w_r), and Al.

```
⟨ prepare the output 33 ⟩ ≡
  A.clear();
  A.conc(Ar);
  A.append(reversal[G[back_edge_into_w0]]);
  A.conc(Al);
```

This code is used in section 27.

34. Efficiency.

Under LEDA 3.0 the space requirement of the first version of *planar* is approximately $160(n+m)+100\alpha m$ Bytes, where n and m denote the number of nodes and edges respectively and α is the fraction of edges in the input graph that do not have their reversal in the input graph. For the pseudo-random planar graphs generated in the demo we have $\alpha = 0$ and $m = 4n$ and hence the space requirement is about $800n$ Bytes. The second version needs an additional $54n + 66m$ Bytes.

The running time of *planar* is about 50 times the running time of `STRONG_COMPONENTS`. On a 50 MIPS SPARC10 workstation the planarity of a planar graph with 16000 nodes and 30000 edges ($\alpha = 0$) is tested in about 10 seconds. It takes 5.4 seconds to make the graph bidirected and biconnected, about 3.9 seconds to test its planarity, and another 6.1 seconds to construct an embedding. The space requirement is about 15 MByte.

35. A Demo.

The demo allows the user to either interactively construct a graph using LEDA's graph editor or to construct a random graph, or to construct a "pseudo-random" planar graph (the graph defined by an arrangement of random line segments). The graph is then tested for planarity. If the graph is planar a straight-line embedding is output. If the graph is non-planar a Kuratowski subgraph is highlighted.

The demo proceeds in cycles. In each cycle we first clear the graphics window W and the graph G and then give the user the choice of a new input graph.

```
⟨ demo.c 35 ⟩ ≡
  ⟨ includes 3 ⟩;
  ⟨ procedure to draw graphs 37 ⟩;
  main( )
  {
    ⟨ initiation and declarations 38 ⟩;
    while (true) {
      ⟨ select graph 39 ⟩;
      ⟨ test graph for planarity and show output 40 ⟩;
      ⟨ reset window 41 ⟩;
    }
    return 0;
  }
```

36. We need to include *planar.h* and various parts of LEDA.

```
⟨ includes 3 ⟩ +≡
#include "planar.h"
#include <LEDA/graph.h>
#include <LEDA/graph_alg.h>
#include <LEDA/window.h>
#include <LEDA/graph_edit.h>
```

37. We need a simple procedure to draw a graph in a graphics window. The numbering of the nodes is optional.

```
⟨ procedure to draw graphs 37 ⟩ ≡
  void draw_graph(const GRAPH⟨point, int⟩ &G, window &W, bool
          numbering = false)
  {
    node v;
    edge e;
    int i = 0;
    forall_edges (e, G) W.draw_edge(G[source(e)], G[target(e)], blue);
    if (numbering)
      forall_nodes (v, G) W.draw_int_node(G[v], i++, red);
    else
      forall_nodes (v, G) W.draw_filled_node(G[v], red);
  }
```

This code is used in section 35.

38. We give the user a short explanation of the demo and declare some variables.

⟨ initiation and declarations 38 ⟩ ≡

```
panel P;
P.text_item("This␣demo␣illustrates␣planarity␣testing␣and␣planar␣straight\
    -line");
P.text_item("embedding.␣You␣have␣two␣ways␣to␣construct␣a␣graph:␣either␣i\
    nteractively");
P.text_item("using␣the␣LEDA␣graph␣editor␣or␣by␣calling␣one␣of␣two␣graph␣\
    generators.");
P.text_item("The␣first␣generator␣constructs␣a␣random␣graph␣with␣a␣certain");
P.text_item("number␣of␣nodes␣and␣edges␣(you␣will␣be␣asked␣how␣many)␣and␣\
    the␣");
P.text_item("second␣generator␣constructs␣a␣planar␣graph␣␣by␣intersecting\
    ␣a␣certain");
P.text_item("number␣of␣random␣line␣segments␣in␣the␣unit␣square␣(you␣will\
    ␣be␣asked␣how␣many).");
P.text_item("␣");
P.text_item("The␣graph␣is␣displayed␣and␣then␣tested␣for␣planarity.");
P.text_item("If␣the␣graph␣is␣non-planar␣a␣Kuratowski␣subgraph␣is␣highlig\
    hted.");
P.text_item("If␣the␣graph␣is␣planar,␣a␣straight-line␣drawing␣is␣produced.");
P.button("continue");
P.open();
window W;
GRAPH⟨point, int⟩ G;
node v, w;
edge e;
int n = 250;
int m = 250;
const double pi = 3.14;
panel P1("PLANARITY␣TEST");
P1.int_item("|V|␣=␣", n, 0, 500);
P1.int_item("|E|␣=␣", m, 0, 500);
P1.button("edit");
P1.button("random");
P1.button("planar");
P1.button("quit");
P1.text_item("␣");
P1.text_item("The␣first␣slider␣asks␣for␣the␣number␣n␣of␣nodes␣and");
P1.text_item("the␣second␣slider␣asks␣for␣the␣number␣m␣of␣edges.");
P1.text_item("If␣you␣select␣the␣random␣input␣button␣then␣a␣graph␣with");
P1.text_item("that␣number␣of␣nodes␣and␣edges␣is␣constructed,␣if␣you");
P1.text_item("select␣the␣planar␣input␣button␣then␣2.5␣times␣square-root␣o\
    f␣n");
P1.text_item("random␣line␣segments␣are␣chosen␣and␣intersected␣to␣yield");
P1.text_item("a␣planar␣graph␣with␣about␣n␣nodes,␣and␣if␣you␣select␣the");
P1.text_item("edit␣button␣then␣the␣graph␣editor␣is␣called.");
P1.text_item("␣");
```

This code is used in section 35.

39. We display the panel P1 until the user makes his choice. Then we construct the appropriate graph.

⟨ select graph 39 ⟩ ≡

```
int inp = P1.open(W);    // P1 is displayed until a button is pressed
if (inp ≡ 3) break;    // quit button pressed
W.init(0, 1000, 0);
W.set_node_width(5);
switch (inp) {
case 0:
  {    // graph editor
    W.set_node_width(10);
    G.clear( );
    graph_edit(W, G, false);
    break;
  }
case 1:
  {    // random graph
    G.clear( );
    random_graph(G, n, m);
    /* eliminate parallel edges and self-loops */
    eliminate_parallel_edges(G);
    list⟨edge⟩ Del = G.all_edges( );
    forall (e, Del)
      if (G.source(e) ≡ G.target(e)) G.del_edge(e);
      /* draw the graph with its nodes on a circle */
    float ang = 0;
    forall_nodes (v, G) {
      G[v] = point(500 + 400 * sin(ang), 500 + 400 * cos(ang));
      ang += 2 * pi/n;
    }
    draw_graph(G, W);
    break;
  }
case 2:
  {    // pseudo-random planar graph
    node_array⟨double⟩ xcoord(G);
    node_array⟨double⟩ ycoord(G);
    G.clear( );
    random_planar_graph(G, xcoord, ycoord, n);
    forall_nodes (v, G) G[v] = point(1000 * xcoord[v], 900 * ycoord[v]);
    draw_graph(G, W);
    break;
  }
}
```

This code is used in section 35.

40. We test the planarity of our graph G using our procedure *planar*.

```
⟨ test graph for planarity and show output 40 ⟩ ≡
  if (PLANAR(G, false)) {
    if (G.number_of_nodes( ) < 4)
      W.message("That's an insult: Every graph with |V| <= 4 is planar");
    else {
      W.message("G is planar. I compute a straight-line embedding ...");
      /* we first make G bidirected. We remember the edges added in n_edges */
      node_array<int> nr(G);
      edge_array<int> cost(G);
      int cur_nr = 0;
      int n = G.number_of_nodes( );
      node v;
      edge e;
      forall_nodes (v, G) nr[v] = cur_nr++;
      forall_edges (e, G)
        cost[e] = ((nr[source(e)] < nr[target(e)]) ? n * nr[source(e)] + nr[target(e)] :
            n * nr[target(e)] + nr[source(e)]);
      G.sort_edges(cost);
      list<edge> L = G.all_edges( );
      list<edge> n_edges;
      while (¬L.empty( )) {
        e = L.pop( );
        if (¬L.empty( ) ∧ source(e) ≡ target(L.head( )) ∧ target(e) ≡ source(L.head( )))
          L.pop( );
        else {
          n_edges.append(G.new_edge(target(e), source(e)));
        }
      }
      Make_biconnected_graph(G);
      PLANAR(G, true);
      node_array<int> xcoord(G), ycoord(G);
      STRAIGHT_LINE_EMBEDDING(G, xcoord, ycoord);
      float f = 900.0/(2 * G.number_of_nodes( ));
      forall_nodes (v, G) G[v] = point(f * xcoord[v] + 30, 2 * f * ycoord[v] + 30);
      forall (e, n_edges) G.del_edge(e);
      W.clear( );
      if (inp ≡ 0) draw_graph(G, W, true);      // with node numbering
      else draw_graph(G, W);
    }
  }
  else {
    W.message("Graph is not planar. I compute the Kuratowski subgraph ...");
    list<edge> L;
    PLANAR(G, L, false);
    node_array<int> deg(G, 0);
    int lw = W.set_line_width(3);
    edge e;
    forall (e, L) {
```

```
    node v = source(e);
    node w = target(e);
    deg[v]++;
    deg[w]++;
    W.draw_edge(G[v], G[w]);
  }
  int i = 1;
  /* We highlight the Kuratowski subgraph. Nodes with degree are drawn black. The
  nodes with larger degree are shown green and numbered from 1 to 6 */
  forall_nodes(v, G) {
    if (deg[v] ≡ 2) W.draw_filled_node(G[v], black);
    if (deg[v] > 2) {
      int nw = W.set_node_width(10);
      W.draw_int_node(G[v], i++, green);
      W.set_node_width(nw);
    }
  }
  W.set_line_width(lw);
}
```

This code is used in section 35.

41. We reset the graphics window.

⟨ reset window 41 ⟩ ≡

```
W.set_show_coordinates(false);
W.set_frame_label("click any button to continue");
W.read_mouse();      // wait for a click
W.reset_frame_label();
W.set_show_coordinates(true);
```

This code is used in section 35.

42. Some Theory.

We give the theory underlying the planarity test as described in [Meh84, section IV.10].

Our next topic is a linear time planarity testing algorithm. Since a graph is planar iff its biconnected components are (cf. [Meh84, section IV.6] for a linear time algorithm to compute the biconnected components of a graph) we can restrict our attention to biconnected graphs. Also we can confine ourselves to graphs with $m \le 3n - 6$. The planarity testing algorithm is an extension of depth-first-search. In the sequel we will always identify nodes with their DFS-number. A DFS on the directed version of $G = (V, E)$ partitions the darts of G into the sets T, F and B. For the planarity testing algorithm we consider the directed graph $(V, T \cup F^{-1})$ and call the edges in T tree edges and the edges in F^{-1} back edges. Also, we write B instead of F^{-1}. Note that this notation differs slightly from the one used in [Meh84, section IV.5]. There, reversals of tree edges were also called back edges.

We will now describe the idea underlying the planarity Algorithm. Let C be any cycle starting in the root of the *dfs*-tree and consisting of tree edges followed by one back edge. Such a cycle exists since G is assumed to be biconnected. For every edge $e = (x, y)$ emanating from the cycle, i.e., x lies on C but e is not an edge of the cycle we consider the segment $S(e)$ defined as follows. If e is a back edge then $S(e)$ is the cycle formed by the tree path from y to x together with the edge e. If e is a tree edge then $S(e)$ consists of the subgraph spanned by the set $V(e) = \{w;\ y \xrightarrow[T]{*} w\}$ of nodes reachable from y by tree edges, all back edges starting in a node in $V(e)$ and ending in a node on cycle C (which is then an ancestor of x), and the tree path from the lowest attachment of $S(e)$ to cycle C to node y.

Example: In Figure 3 the cycle C consists of the tree path from node 1 to node 9 and the back edge $(9, 1)$. The four edges $(9, 10)$, $(7, 5)$, $(7, 13)$ and $(6, 4)$ emanate from the cycle. The segment $S((9, 10))$ consists of the subgraph spanned by $\{10, 11, 12\}$, the back edges $(11, 8)$, $(11, 7)$ and $(12, 5)$, and the tree path from 5 to 10. The segment $S((9, 10))$ is attached to the cycle in the nodes 9, 8, 7 and 5. ∎

We test the planarity of G in a two step process. In the first step we test whether $C + S(e)$, the graph consisting of cycle C and segment $S(e)$, is planar for every edge e emanating from cycle C. This is equivalent to testing whether the segment $S(e)$ has a strongly planar embedding, i.e., an embedding where all attachments of $S(e)$ to the cycle C lie on the boundary of the outer face. In order to test the strong planarity of $S(e)$ we will use the algorithm recursively. Suppose now that the segments $S(e)$ are all strongly planar. We then try in a second step to merge the embeddings found in step one. The merging process has to decide for each segment $S(e)$ whether it should be placed inside or outside the cycle C. For this purpose, it only needs to take into account the set of attachments of the different segments emanating from C and their interaction. In our example, the segments $S((7, 5))$ and $S((6, 4))$ have to be embedded on different sides of C because these segments "interlace".

We will next describe the theory behind both steps in detail. With an edge $e = (x, y)$ we associate a **cycle** $C(e)$ and a **segment** $S(e)$ as follows. If e is a back edge then $C(e)$ and $S(e)$ consist of the tree path from y to x and the edge e. If e is a tree edge then let $V(e) = \{w;\ y \xrightarrow[T]{*} w\}$ be the set of tree successors of y and let $lowpt[y] = \min\{z;\ (w, z)$ is a back edge and $w \in V(e)\}$ be the lowest endpoint of a back edge starting in $V(e)$. The cycle $C(e)$ consists of a tree path from $lowpt[y]$ to w, where $w \in V(e)$ and $(w, lowpt[y]) \in B$ is such a back edge. The segment $S(e)$ consists of $C(e)$, the subgraph spanned by $V(e)$ and all back edges starting in a node in $V(e)$. Note that the segment $S(e)$ is uniquely defined but that there may be several choices for the cycle $C(e)$. We divide the tree path underlying the

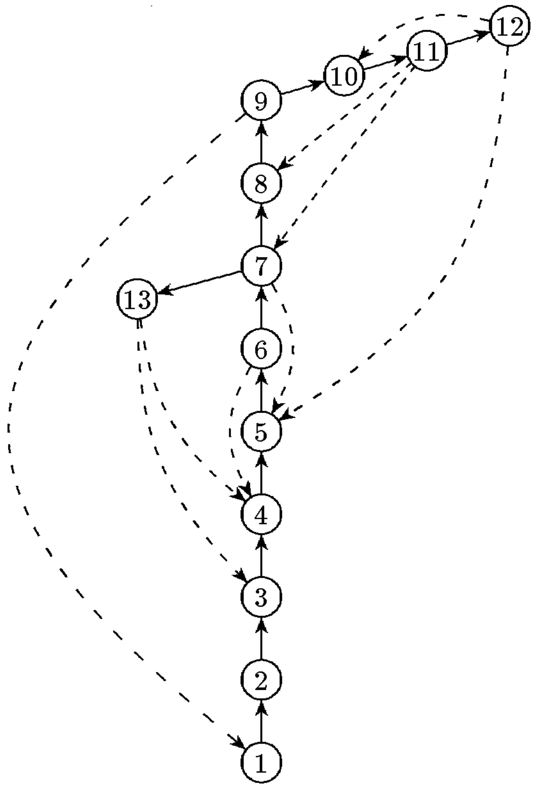

Figure 3: A *dfs*-tree of a planar graph

cycle $C(e)$ into two parts, its stem and its spine. The **stem** consists of the part ending in x. The **spine** is empty if e is a back edge and it is the part starting in y if e is a tree edge.

In our example, the cycle $C((9,10))$ consists of the tree path from 5 to 12 followed by the back edge $(12,5)$. The stem is the tree path from 5 to 9 and the spine is the tree path from 10 to 12. The cycle $C((1,2))$ consists of the tree path from 1 to 9 and the back edge $(9,1)$. Its stem is the node 1.

A segment $S(e)$ is called **strongly planar** if there is an embedding of $S(e)$ such that the stem of the cycle $C(e)$ borders the outer face. An embedding with this property is called a strongly planar embedding of $S(e)$. Let $w_0, w_1, \ldots, w_r$ with $e = (w_r, y)$ be the stem of $C(e)$. A strongly planar embedding of $S(e)$ is called **canonical (reversed canonical)** if for all i, $0 < i < r$, the edge $\{w_i, w_{i+1}\}$ immediately follows (precedes) the edge $\{w_i, w_{i-1}\}$ in the counterclockwise ordering of edges incident to w_i. Note that every strongly planar embedding is either canonical or reversed canonical.

In Figure 3 the embeddings of segments $S((9,10))$ and $S((7,13))$ are both strongly planar, the embedding of $S((7,13))$ is canonical and the embedding of $S((9,10))$ is reversed canonical.

Lemma 2 *Let G be a biconnected graph and let e be the unique tree edge starting in the root of the dfs-tree. Then $S(e) = G$ and G is planar iff $S(e)$ is strongly planar.*

Proof: Let $e = (1, 2)$ be the unique tree edge incident to node 1. Then $V(e) = \{2, \ldots, n\}$ and hence $S(e) = G$. Also, the stem of $C(e)$ consists only of vertex 1 and hence $S(e)$ is strongly planar iff it is planar. ∎

Lemma 2 shows that we can confine ourselves to a test of strong planarity. Now let e_0 be an edge and $C = C(e_0)$ be the cycle associated with e_0. An edge $e = (x, y)$ is said to **emanate** from C if x lies on the spine of C but e does not belong to C. Clearly, if e emanates from $C(e_0)$ then the stem of $C(e)$ is part of the tree path underlying $C(e_0)$ and $S(e)$ is a subgraph of $S(e_0)$. Also, $S(e_0)$ is the union of $C(e_0)$ and the segments $S(e)$, where e emanates from $C(e_0)$. The basis of step 1 of the planarity algorithm is the following

Lemma 3 *Let $C = C(e_0)$ be a cycle and let e emanate from C. Then $C + S(e)$ is planar iff $S(e)$ is strongly planar.*

Proof: "$\Rightarrow$": Consider any embedding of $C + S(e)$. The cycle C divides the plane into a bounded and an unbounded region. We may assume w.l.o.g. that the edge $e = (x, y)$ lies in the bounded region. Hence all nodes in $V(e)$ must lie in the bounded region since every node in $V(e)$ is reachable from y without passing through a node of C. If we remove the part of cycle C between x and $lowpt[y]$ then we have the desired strongly planar embedding of $S(e)$.

"$\Leftarrow$": Given a strongly planar embedding of $S(e)$ we can clearly add the missing part of C to obtain an embedding of $C + S(e)$. ∎

For step 2 of the algorithm we need the concepts of attachments and interlacing. Let $C = C(e_0)$ and let $e = (x, y)$ emanate from C. The set $A(e)$ of **attachments** of segment $S(e)$ to cycle C is defined to be the set $\{x, y\}$ if e is a back edge and the set $\{x\} \cup \{z;\ (w, z)$ is a back edge, $w \in V(e)$ and $z \notin V(e)\}$ if e is a tree edge. Two segments $S(e)$ and $S(e')$ where e and e' emanate from C are said to **interlace** if either there are nodes $x < y < z < u$ on cycle C such that $x, z \in A(e)$ and $y, u \in A(e')$ or $A(e)$ and $A(e')$ have three points in common (cf. Fig. 4; note that the segments shown may have further attachments). Clearly,

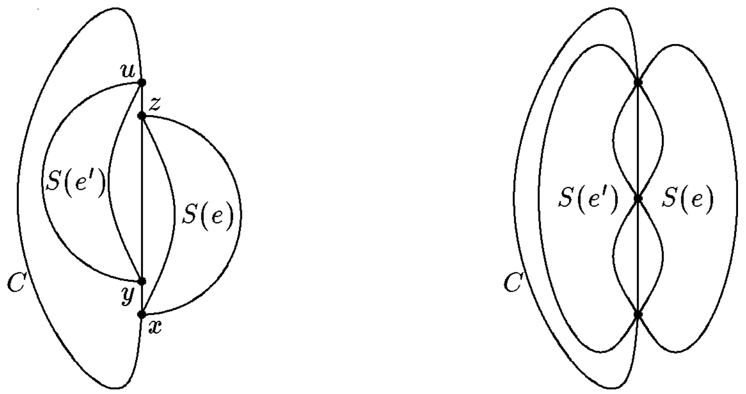

Figure 4: Interlacing segments

interlacing segments cannot be embedded on the same side of C. The **interlacing graph** $IG(C)$ with respect to cycle C is defined as follows: The nodes of $IG(C)$ are the segments

$S(e)$ where e emanates from C. Also, $S(e)$ and $S(e')$ are connected by an edge iff $S(e)$ and $S(e')$ interlace. The interlacing graph for the cycle $C((1,2))$ of Figure 3 is shown in Figure 5. This graph is bipartite with segments S_1 and S_3 forming one of the sides of the bipartite graph. Note also that the planar embedding of the graph of Figure 3 has S_1 and S_3 on one side of C and S_2 and S_4 on the other side of C.

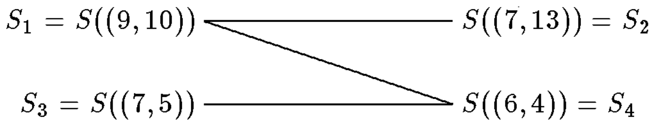

Figure 5: Interlacing graph

Lemma 4 *Let e_0 be a tree edge, let $C = C(e_0) = w_0 \underset{T}{\rightarrow} w_1 \underset{T}{\rightarrow} \cdots \underset{T}{\rightarrow} w_k \underset{B}{\rightarrow} w_0$ and let $e_0 = (w_r, w_{r+1})$. Let $e_1, \ldots, e_m$ be the edges leaving the spine of C, i.e., they leave the cycle in nodes w_j, $r < j \leq k$. Then $S(e_0)$ is planar iff $S(e_i)$ is strongly planar for every i, $1 \leq i \leq m$, and $IG(C)$ is bipartite, i.e., there is a partition L, R of $\{S(e_1), \ldots, S(e_m)\}$ such that no two segments in L resp. R interlace. Moreover, segment $S(e_0)$ is strongly planar iff in addition for every connected component B of $IG(C)$: either $\{w_1, \ldots, w_{r-1}\} \cap \bigcup_{S(e) \in B \cap L} A(e) = \emptyset$ or $\{w_1, \ldots, w_{r-1}\} \cap \bigcup_{S(e) \in B \cap R} A(e) = \emptyset$.*

Proof: "$\Rightarrow$": Note first that $S(e_0) = C + S(e_1) + \cdots + S(e_m)$. Hence, if $S(e_0)$ is planar then $C + S(e_i)$, $1 \leq i \leq m$, is planar and hence $S(e_i)$ is strongly planar by Lemma 3. Consider any planar embedding of $S(e_0)$. Let $L = \{S(e_i);\ S(e_i)$ is embedded inside cycle C, $1 \leq i \leq m\}$ and let R be the remaining segments. Then no two segments in L resp. R interlace because interlacing segments have to be embedded on different sides of C. Hence $IG(C)$ is bipartite. Finally, assume that $S(e_0)$ is strongly planar. Consider any strongly planar embedding of $S(e_0)$, i.e., tree path $w_0 \rightarrow w_1 \rightarrow w_2 \rightarrow \cdots \rightarrow w_r$ borders the outer face. Then no segment $S(e_i)$, $1 \leq i \leq m$, which is embedded outside C can have an attachment in $\{w_1, \ldots, w_{r-1}\}$ and hence $\{w_1, \ldots, w_{r-1}\} \cap \bigcup_{S(e) \in R} A(e) = \emptyset$.

"$\Leftarrow$": The proof of this direction is postponed. It will be given in Lemma 9. ▮

Lemma 4 suggests an algorithm for testing strong planarity. In order to test strong planarity of a segment $S(e_0)$, test strong planarity of the segments $S(e_i)$, $1 \leq i \leq m$, construct the interlacing graph and test for the conditions stated in Lemma 4. Unfortunately, the size of the interlacing graph might be quadratic and therefore we cannot afford to construct the interlacing graph explicitly. Rather, we compute the connected components (and their partition into left and right side) of $IG(C)$ and an embedding of $S(e_0) = C + S(e_1) + \cdots + S(e_m)$ by considering segment by segment. We start with cycle C and then try to add segment by segment. We will consider the segments $S(e_1), \ldots, S(e_m)$ in an order such that adding a canonical embedding of $S(e_{i+1})$ to an embedding of $C + S(e_1) + \cdots + S(e_i)$ can always be achieved (if at all) in a particularly simple way, namely by moving some of the $S(e_l)$, $l \leq i$, to the other side of C and then adding $S(e_{i+1})$ inside C and close to the tree path underlying C, cf. Figure 10. In that figure the segment $S(e_{i+1})$ emanates from w_j, $e_{i+1} = (w_j, y)$ and $z = \min A(e_{i+1})$ is the lowest attachment of $S(e_{i+1})$. Also, there is a face F inside C such that the tree path from z to w_j is on the boundary of F. Clearly, a canonical embedding of $S(e_{i+1})$ can be added inside F to the embedding of $C + S(e_1) + \cdots + S(e_i)$ in this case.

In order to follow this embedding strategy we should first consider all segments emanating from w_k, then all segments emanating from w_{k-1}, ... For any node w_j we consider the segments emanating from w_j in the order of lowest attachment, considering the segments with lower attachment first. Among the segments emanating from w_j and having the same lowest attachment, say w_i with $i < j$, we first consider the segments having only w_i and w_j as attachments and then all the others (there can be at most two segments of the latter kind because any two such segments interlace). We will now show how to compute this ordering on the edges emanating from C. We do so by showing how to reorder the adjacency list of each node such that the order of the adjacency list corresponds to the order defined above. For every node v let

$$\begin{aligned} lowpt[v] &= \min(\{v\} \cup \{z;\ v \overset{*}{\underset{T}{\rightarrow}} w \underset{B}{\rightarrow} z \text{ for some } w \in V\}), \quad \text{and} \\ lowpt2[v] &= \min(\{v\} \cup \{z;\ v \overset{*}{\underset{T}{\rightarrow}} w \underset{B}{\rightarrow} z \text{ for some } w \in V \text{ and } z \neq lowpt[v]\}). \end{aligned}$$

$lowpt[v]$ is the lowest node reachable from v by a sequence of tree edges followed by one back edge. Since G is assumed to be biconnected we have $lowpt[v] < v$ for all $v \neq 1$. $lowpt2[v]$ is the second lowest node reachable from v in this way, if there is one. The default value for both functions is v. The functions *lowpt* and *lowpt2* are easily computed during *dfs* since

$$lowpt[v] = \min(\{v\} \cup \{z;\ (v,z) \in B\} \cup \{lowpt[w];\ (v,w) \in T\})$$

and

$$\begin{aligned} lowpt2[v] = \min(\{v\} &\cup \{z;\ (v,z) \in B \text{ and } z \neq lowpt[v]\} \\ &\cup \{lowpt[w];\ (v,w) \in T, lowpt[w] \neq lowpt[v]\} \\ &\cup \{lowpt2[w];\ (v,w) \in T\}). \end{aligned}$$

These equations suggest to compute *lowpt* and *lowpt2* by two separate applications of *dfs*. In the first application of *dfs* one computes *lowpt* and in the second application one computes *lowpt2* using *lowpt*. We leave it to the reader to show that one *dfs* suffices to compute both functions. For an edge $e = (w_j, y)$ let

$$lowpt[e] = \textbf{if } e \in B \textbf{ then } y \textbf{ else } lowpt[y] \textbf{ fi}.$$

Then $lowpt[e] = \min A(e)$ and $|A(e)| \geq 3$ iff $e \in T$ and $lowpt2[y] < w_j$ for any edge $e = (w_j, y)$ emanating from the cycle C. We want to reorder the adjacency list of w_j such that an edge $e = (w_j, y)$ is before an edge $e' = (w_j, y')$ if either $lowpt[e] < lowpt[e']$ or $lowpt[e] = lowpt[e']$ and $|A(e)| = 2$ and $|A(e')| \geq 3$. Let $c : E \rightarrow \mathbb{N}$ be defined by

$$c((v,w)) = \begin{cases} 2 \cdot w & \text{if } (v,w) \in B; \\ 2 \cdot lowpt[w] & \text{if } (v,w) \in T \text{ and } lowpt2[w] \geq v; \\ 2 \cdot lowpt[w] + 1 & \text{if } (v,w) \in T \text{ and } lowpt2[w] < v. \end{cases}$$

Then reordering an adjacency list according to non-decreasing values of c yields the desired ordering of outgoing edges. We can do the reordering in linear time by bucket sort. Have $2n$ initially empty buckets. Step through the edges of G one by one and throw edge (v,w) into bucket $c((v,w))$. After having done so we go through the buckets in decreasing order. When edge (v,w) is encountered we add (v,w) to the front of v's adjacency list.

In our example, the edges out of node 7 are ordered $(7,8),(7,13),(7,5)$ and the edges out of node 11 are ordered $(11,12),(11,7),(11,8)$.

From now on, we assume that adjacency lists are reordered in the way described above. The reordering has the additional property that a cycle $C(e_0)$ for a tree edge $e_0 = (x,y)$ is very easy to find. We start at node y and construct a path by always taking the first edge out of each node until a back edge is encountered. This path is a spine of $C(e_0)$, as is easily verified.

We now resume the discussion of how to deal with the interlacing graph. As in Lemma 4, $C = C(e_0)$,

$$C = w_0 \underset{T}{\rightarrow} w_1 \underset{T}{\rightarrow} \cdots \underset{T}{\rightarrow} w_k \underset{B}{\rightarrow} w_0$$

and $e_0 = (w_r, w_{r+1})$ for some r. Let $e_1, \ldots, e_m$ be the edges leaving the spine of C *in order*, i.e., the edges leaving w_k are considered first and for each w_j the edges are ordered as described above. Let $IG_i(C)$ be the subgraph of $IG(C)$ spanned by $S(e_1), \ldots, S(e_i)$. If $IG_i(C)$ is non-bipartite then so is $IG(C)$ and hence $S(e_0)$ is not strongly planar. If $IG_i(C)$ is bipartite then every connected component (= block) of $IG_i(C)$ is. If B is a block of $IG_i(C)$ then we use LB, RB to denote the partition of B induced by the bipartite graph.

Our next goal is to describe how the blocks of $IG_{i+1}(C)$ can be obtained from the blocks of $IG_i(C)$. Let $e_{i+1} = (w_j, y)$. For every block B of $IG_i(C)$ let

$$ALB = \{w_h;\ 0 \le h < j \text{ and } w_h \in A(e) \text{ for some } S(e) \in LB\}$$

be the set of attachments (below w_j) of segments in LB. ARB is defined similarly.

Lemma 5 *If $IG_i(C)$ is bipartite, then:*

a) *There is some ordering of the blocks of $IG_i(C)$, say $B_1, B_2, \ldots, B_h, B_{h+1}, \ldots$ such that*

$$\max(ALB_l \cup ARB_l) \le \min(ALB_{l+1} \cup ARB_{l+1})$$

for $1 \le l < h$ and $ALB_l = ARB_l = \emptyset$ for $l > h$.

b) *$IG_{i+1}(C)$ is bipartite iff for all l, $1 \le l \le h$, either $\max ALB_l \le \min A(e_{i+1})$ or $\max ARB_l \le \min A(e_{i+1})$.*

c) **c)** *If $IG_{i+1}(C)$ is bipartite then the blocks of $IG_{i+1}(C)$ can be obtained as follows: Assume w.l.o.g. that $\max ALB_l \le \min A(e_{i+1})$ for all l. (This can always be achieved by interchanging LB and RB for some blocks B.) Let $d = \min(\{l;\ \max ARB_l > \min A(e_{i+1})\} \cup \{h+1\})$. Then the blocks of $IG_{i+1}(C)$ are $B_1, \ldots, B_{d-1}, B_d \cup \cdots \cup B_h \cup \{S(e_{i+1})\}, B_{h+1}, \ldots$.*

d) *If $IG_{i+1}(C)$ is bipartite and $S(e_l)$, $1 \le l \le i+1$, are strongly planar then there is a planar embedding of $C + S(e_1) + \cdots + S(e_{i+1})$ such that all segments in $\bigcup_l LB_l$ are embedded inside C and all segments in $\bigcup_l RB_l$ are embedded outside C.*

Proof: We use induction on i. For $i = 0$ little remains to be shown. $IG_0(C)$ is empty and $IG_1(C)$ consists of a single node. This shows a), b) and c). For part d) we only have to observe that $S(e_1)$ can be embedded inside as well as outside C, if $S(e_1)$ is strongly planar.

So let us turn to the case $i > 0$. We will show parts b), c), a) and d) in this order.

b) "$\Rightarrow$": Note first that it suffices to show the following

Claim 6 *If* $\max ALB_l > \min A(e_{i+1})$ *for some* l *then there is a segment* $S(e) \in LB_l$ *such that* $S(e)$ *and* $S(e_{i+1})$ *interlace.*

Suppose that we have shown Claim 6. If there were l, $1 \leq l \leq h$, such that $\max ALB_l > \min A(e_{i+1})$ and $\max ARB_l > \min A(e_{i+1})$ then $S(e_{i+1})$ interlaces with a segment $S(e) \in LB_l$ and a segment $S(e') \in RB_l$ by Claim 6. Since $S(e)$ and $S(e')$ belong to the same block there is a path from $S(e)$ to $S(e')$ in $IG_i(C)$. Since $IG_i(C)$ is bipartite this path necessarily has odd length. Together with edges $\{S(e), S(e_{i+1})\}$ and $\{S(e_{i+1}), S(e')\}$ we obtain an odd length cycle in $IG_{i+1}(C)$. Hence $IG_{i+1}(C)$ is non-bipartite, a contradiction. We still have to show Claim 6.

Proof of Claim 6: Let $z = \min A(e_{i+1})$. Since $\max ALB_l > z$ there must be a segment $S(e) \in LB_l$ such that $w \in A(e)$ for some w with $z \xrightarrow[T]{+} w \xrightarrow[T]{+} w_j$. Edge e emanates from node w_p for some $p \geq j$.

Case 1: $p > j$.

Then $z \xrightarrow[T]{+} w \xrightarrow[T]{+} w_j \xrightarrow[T]{+} w_p$, $z, w_j \in A(e_{i+1})$ and $w, w_p \in A(e)$. Hence segments $S(e)$ and $S(e_{i+1})$ interlace (cf. Figure 6).

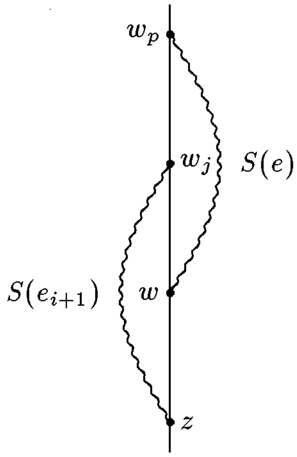

Figure 6: Case 1

Case 2: $p = j$.

Let $e = (w_j, u)$. Since e is considered before e_{i+1} and hence $\min A(e) \leq z$, edge e cannot be a back edge. (If it were a back edge then $\min A(e) = u = w > z$, a contradiction.) Hence e is a tree edge and $\min A(e) = lowpt[u]$.

Case 2.1: $lowpt[u] < z$.

Then $lowpt[u] \xrightarrow[T]{+} z \xrightarrow[T]{+} w \xrightarrow[T]{+} w_j$, $lowpt[u], w \in A(e)$ and $z, w_j \in A(e_{i+1})$. Hence segments $S(e)$ and $S(e_{i+1})$ interlace (cf. Fig. 7).

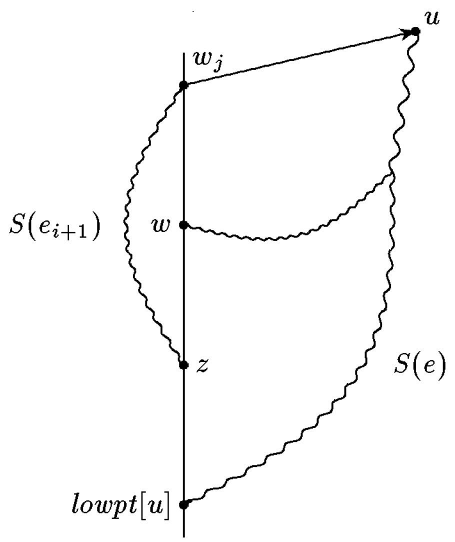

Figure 7: Case 2.1

Case 2.2: $lowpt[u] = z$.

Since $w \in A(e)$ we have $lowpt2[u] < w_j$. Since e is considered before e_{i+1} we must have $|A(e_{i+1})| \geq 3$, and hence edge e_{i+1} cannot be a back edge. Rather, it must be a tree edge and we must have $lowpt2[y] < w_j$. If $lowpt2[y] \neq lowpt2[u]$, say $lowpt2[y] \xrightarrow[T]{+} lowpt2[u]$, then we have $z \xrightarrow[T]{+} lowpt2[y] \xrightarrow[T]{+} lowpt2[u] \xrightarrow[T]{+} w_j$, $z, lowpt2[u] \in A(e)$, and $lowpt2[y], w_j \in A(e_{i+1})$. Hence $S(e_{i+1})$ and $S(e)$ interlace (cf. Fig. 8). If $lowpt2[y] = lowpt2[u]$ then $A(e)$ and $A(e_{i+1})$ have three points in common and hence $S(e_i)$ and $S(e_{i+1})$ interlace (cf. Figure 9). ∎

"$\Leftarrow$": Assume now that $\max ALB_l \leq \min A(e_{i+1})$ or $\max ARB_l \leq \min A(e_{i+1})$ for all l, $1 \leq l \leq h$. By interchanging LB_l and RB_l, if necessary, we can achieve that $\max ALB_l \leq \min A(e_{i+1})$ for all l, $1 \leq l \leq h$.

Claim 7 *Let $S(e) \in \bigcup_l LB_l$ be arbitrary. Then $S(e)$ and $S(e_{i+1})$ do not interlace.*

Proof: $A(e_{i+1}) \subseteq \{w;\ \min A(e_{i+1}) \xrightarrow[T]{*} w \xrightarrow[T]{*} w_j\}$ and $A(e) \subseteq \{w;\ w \xrightarrow[T]{*} \min A(e_{i+1})$ or $w_j \xrightarrow[T]{*} w\}$. Hence $S(e)$ and $S(e_{i+1})$ do not interlace. ∎

The bipartiteness of $IG_{i+1}(C)$ now follows from Claim 7 because it is safe to add $S(e_{i+1})$ to the "left side" of the interlacing graph.

c) Assume that $IG_{i+1}(C)$ is bipartite. Then for all l, $1 \leq l \leq h$, $\max ALB_l \leq \min A(e_{i+1})$ or $\max ARB_l \leq \min A(e_{i+1})$ by part b). By interchanging LB_l and RB_l, if necessary, we can achieve $\max ALB_l \leq \min A(e_{i+1})$ for all l, $1 \leq l \leq h$. Let $d = \min(\{l;\ \max ARB_l > \min A(e_{i+1})\} \cup \{h+1\})$.

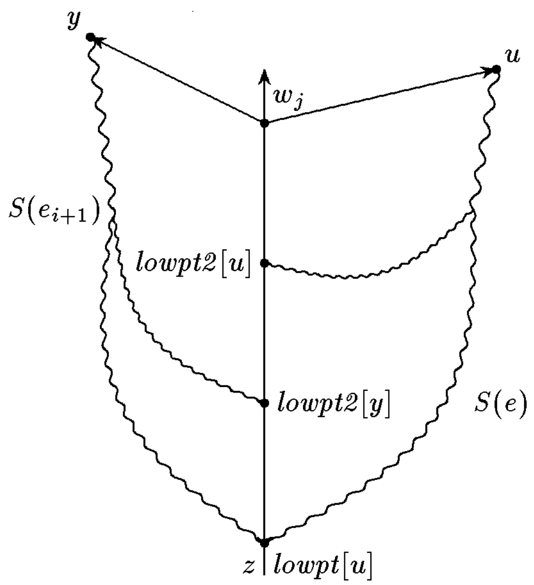

Figure 8: Case 2.2, $lowpt2[y] \neq lowpt2[u]$

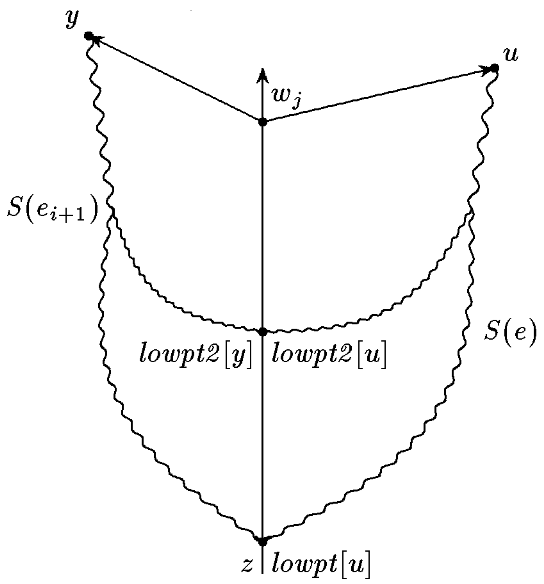

Figure 9: Case 2.2, $lowpt2[y] = lowpt2[u]$

Claim 8 *For all l: There is a segment $S(e) \in RB_l$ such that $S(e)$ and $S(e_{i+1})$ interlace iff $d \leq l \leq h$.*

Proof: "$\Leftarrow$": Let $d \leq l \leq h$. Then

$$
\begin{aligned}
\min A(e_{i+1}) &< \max ARB_d && \text{[by definition of } d\text{]}\\
&\le \max ARB_l && \text{[by induction hypothesis, part a) and } d \le l\text{]}\\
&< w_j && \text{[since } l \le h\text{]}
\end{aligned}
$$

and hence there is a segment $S(e) \in RB_l$ such that $S(e)$ and $S(e_{i+1})$ interlace by Claim 6.

"$\Rightarrow$": (Indirect.) Let $l < d$ or $l > h$ and let $S(e) \in RB_l$. Then $A(e) \subseteq \{w;\ w_j \xrightarrow[T]{*} w\}$ if $l > h$ and $A(e) \subseteq \{w;\ w_j \xrightarrow[T]{*} w$ or $w \xrightarrow[T]{*} \min A(e_{i+1})\}$ if $l < d$. The former inclusion follows from the definition of h, the latter inclusion follows from the definition of d, and part a) of the induction hypothesis. Also $A(e_{i+1}) \subseteq \{w;\ \min A(e_{i+1}) \xrightarrow[T]{*} w \xrightarrow[T]{*} w_j\}$ and hence $S(e)$ and $S(e_{i+1})$ do not interlace. ∎

We conclude from Claims 7 and 8 that $S(e_{i+1})$ is connected to segments in blocks $B_d, \ldots, B_h$. Hence the blocks of $IG_{i+1}(C)$ are $B_1, \ldots, B_{d-1}, B_d \cup \cdots \cup B_h \cup \{S(e_{i+1})\}, B_{h+1}, \ldots$. Let $B = B_d \cup \cdots \cup B_h \cup \{S(e_{i+1})\}$ be the new block. Then B can be partitioned into LB and RB where $LB = \bigcup_{d \le l \le h} LB_l \cup \{S(e_{i+1})\}$ and $RB = \bigcup_{d \le l \le h} RB_l$. Moreover, if $d \le h$, $\max ARB_d \le \min ARB_{d+1} \le \max ARB_{d+1} \le \cdots \le \min ARB_h \le \max ARB_h$ by part a) and $\max ALB_d \le \min ALB_{d+1} \le \max ALB_{d+1} \le \cdots \le \min ALB_h \le \max ALB_h \le \min A(e_{i+1})$ by part a) and the assumption that $\max ALB_l \le \min A(e_{i+1})$ for all l, $1 \le l \le h$.

a) Follows immediately from part c). The ordering of the blocks of $IG_{i+1}(C)$ given in part c) satisfies the conditions required in part a). This follows immediately from the discussion completing the proof of part c).

d) Assume that $IG_{i+1}(C)$ is bipartite and that $S(e_l)$, $1 \le l \le i+1$, are strongly planar. Let B'_1, B'_2, ... be the blocks of $IG_{i+1}(C)$. By part c) we have $B'_1 = B_1$, ..., $B'_{d-1} = B_{d-1}$, $B'_d = B_d \cup \cdots \cup B_h \cup \{S(e_{i+1})\}$, $B'_{d+1} = B_{h+1}$, ..., where B_1, B_2, ... are the blocks of $IG_i(C)$. Moreover, $LB'_l = LB_l$, $RB'_l = RB_l$ for $l < d$, $LB'_{d+l} = LB_{h+l}$, $RB'_{d+l} = RB_{h+l}$ for $l \ge 1$ and $LB'_d = \bigcup_{d \le l \le h} LB_l \cup \{S(e_{i+1})\}$ and $RB'_d = \bigcup_{d \le l \le h} RB_l$. By induction hypothesis there is a planar embedding of $C + S(e_1) + \cdots + S(e_i)$ such that all segments in $\bigcup_l LB_l$ are embedded inside C and all segments in $\bigcup_l RB_l$ are embedded outside C. By the proof of Claim 7 no segment $S(e) \in \bigcup_l LB_l$ has an attachment w which lies strictly between $\min A(e_{i+1})$ and w_j. Thus there is a face F inside C such that the tree path from $\min A(e_{i+1})$ to w_j is part of the boundary of F. All attachments of $S(e_{i+1})$ lie between $\min A(e_{i+1})$ and w_j inclusively. Moreover, $S(e_{i+1})$ is strongly planar and hence there is a planar embedding of $S(e_{i+1})$ where the tree path from $\min A(e_{i+1})$ to w_j borders the outer face. We can add this embedding to the embedding of $C + S(e_1) + \cdots + S(e_i)$ by putting it inside face F. In this way we obtain a planar embedding of $C + S(e_1) + \cdots + S(e_{i+1})$ (cf. Fig. 10). This completes the proof of Lemma 5. ∎

Lemma 9 *The if-part of Lemma 4 holds.*

Proof: If $IG(C)$ is bipartite and $S(e_i)$, $1 \le i \le m$, is strongly planar then by Lemma 5 d) there is an embedding of $C + S(e_1) + \cdots + S(e_m) = S(e_0)$ such that all segments in $\bigcup_i LB_i$ are embedded inside C and all segments in $\bigcup_i RB_i$ are embedded outside C. In particular, $S(e_0)$ is planar. Assume now that in addition $ALB_l \cap \{w_1, \ldots, w_{r-1}\} = \emptyset$ or $ARB_l \cap \{w_1, \ldots, w_{r-1}\} = \emptyset$ for all l where ALB_l and ARB_l are defined with $j = r+1$, i.e., when all edges $e_1, \ldots, e_m$ are embedded. We may assume w.l.o.g. (by interchanging L

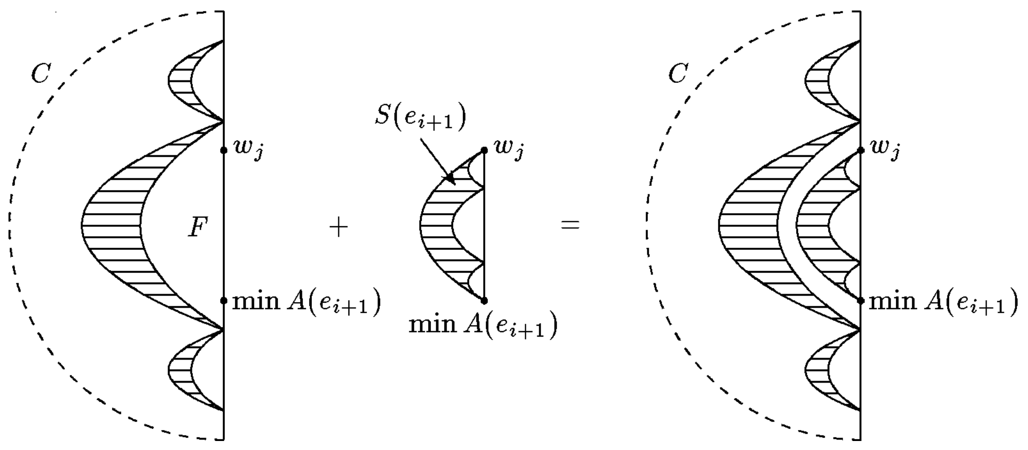

Figure 10: Addition of $S(e_{i+1})$ inside F

and R for some blocks) that $ARB_l \cap \{w_1, \ldots, w_{r-1}\} = \emptyset$ for all l. Thus outside C there are no attachments to nodes $w_1, \ldots, w_{r-1}$ and hence there is a face F outside C such that the stem $w_0, \ldots, w_r$ of $S(e_0)$ borders F. We can now turn F into the outer face and in this way obtain a canonical embedding of $S(e_0)$. ∎

We illustrate Lemma 5 on our example. Let C be the cycle which runs from node 1 to node 9 along tree edges and then back to node 1. There are four segments emanating from this cycle: $S_1 = S((9,10))$, $S_2 = S((7,13))$, $S_3 = S((7,5))$ and $S_4 = S((6,4))$. All four segments are strongly planar. When segment $S_2 = S((7,13))$ is considered, we have: $IG_1(C)$ has one block B_1 consisting of segment S_1. Say S_1 belongs to RB_1. Then $ALB_1 = \emptyset$ and $ARB_1 = \{5\}$. Lemma 5 b) is satisfied and hence $IG_2(C)$ is bipartite. We have $d = 1$ in Lemma 5 c) and hence $IG_2(C)$ has only block B_1, say $LB_1 = \{S_2\}$ and $RB_1 = \{S_1\}$. Then $ALB_1 = \{3,4\}$ and $ARB_1 = \{5\}$ when S_3 is considered. $IG_3(C)$ is bipartite and has two blocks B_1 and B_2, say $LB_1 = \{S_2\}$, $RB_1 = \{S_1\}$, $RB_2 = \{S_3\}$. Then $ALB_1 = \{3,4\}$, $ARB_1 = \{5\}$, $ARB_2 = \{5\}$, $ALB_2 = \emptyset$ when S_4 is considered. S_4 forces us to merge blocks B_1 and B_2, i.e., $d = 1$ in Lemma 5 c), and hence $IG_4(C)$ has only one block B_1. Moreover $LB_1 = \{S_2, S_4\}$ and $RB_1 = \{S_1, S_3\}$.

It is now easy to derive an efficient way of dealing with the interlacing graph from Lemma 5. Suppose that we processed edges $e_1, \ldots, e_i$ already and want to process edge e_{i+1} next. At this point we keep blocks $B_1, \ldots, B_h$ in a stack S where h is defined as in Lemma 5 a). Also for each l, $1 \le l \le h$, we maintain the multi-sets ALB_l and ARB_l in a doubly linked list. The lists ALB_l and ARB_l are *ordered* according to DFS-numbers. From the stack position corresponding to B_l we have pointers to the front and back end of lists ALB_l and ARB_l. The test for bipartiteness of $IG_{i+1}(C)$ given in Lemma 5 b) is now easily implemented by Program 2.

The running time of Program 2 is clearly $O(h-d+2)$. Also, it correctly computes d as defined in Lemma 5 c). The new blocks of $IG_{i+1}(C)$ are now easily formed by Program 3. The running time of Program 3 is also clearly $O(h-d+2)$ provided we are given $(A(e_{i+1})-\{w_j\})$. Also, it correctly computes lists ALB and ARB. Note that these lists are ordered according to the remark at the end of the proof of Lemma 5 c). We can now give the complete planarity testing algorithm, see Program 4.

```
l ← h + 1;
while max(ALB_{l-1} ∪ ARB_{l-1}) > lowpt[e_{i+1}]
do if ALB_{l-1} is non-empty and max ALB_{l-1} > lowpt[e_{i+1}]
    then interchange LB_{l-1} and RB_{l-1} fi;
    if ALB_{l-1} is non-empty and max ALB_{l-1} > lowpt[e_{i+1}]
    then IG_{i+1}(C) is not bipartite and hence
           the graph can be declared non-planar fi;
    l ← l - 1
od;
d ← l.
```

Program 2

```
ALB ← ARB ← ∅;
for l from d to h
do ALB ← ALB concatenated with ALB_l;
    ARB ← ARB concatenated with ARB_l
od;
ALB ← ALB concatenated with (A(e_{i+1}) - {w_j});
pop B_h, ..., B_d from stack S;
add B to stack S.
```

Program 3

Lemma 10 *Program 4 tests strong planarity in linear time and space.*

Proof: Observe first that line (1) determines the spine of cycle $C(e_0)$ in time proportional to the length of the spine. The stem $w_0, \ldots, w_r$ is not explicitly constructed; we only mention it in order to keep the same notation as in Lemmas 4 and 5. Next we argue that bipartiteness of $IG(C)$ is tested correctly. The correctness of loop (4)–(8) is obvious from the discussions above. Suppose now that we processed all edges emanating from w_j. In order to prepare for processing the edges emanating from w_{j-1} we only have to delete all occurrences of w_{j-1} on lists ALB_l and ARB_l. This is done in lines (9)–(14). Note that all occurrences of w_{j-1} must be in the top entries of stack S by Lemma 5 a). Hence lines (9)–(14) work correctly. When control reaches line (16) the interlacing graph $IG(C)$ is bipartite and hence $S(e_0)$ is planar. Moreover, for every block B in the stack S the lists ALB and ARB contain exactly the attachments below w_r of segments in the block. In line (18) we now test the condition for strong planarity given in Lemma 4. It states that for all blocks B of $IG(C)$ either $\{w_1, \ldots, w_{r-1}\} \cap \bigcup_{S(e) \in LB} A(e) = \emptyset$ or $\{w_1, \ldots, w_{r-1}\} \cap \bigcup_{S(e) \in RB} A(e) = \emptyset$. Of course, we can always interchange L and R such that the latter is the case. It remains to argue that lines (20) to (22) correctly compute the ordered set $A(e_0) - \{x\}$ of attachments. Let l_0 be minimal such that $\max(ALB_{l_0} \cup ARB_{l_0}) \geq w_1$. Then $ALB_l \cup ARB_l \subseteq \{w_0\}$ for $l < l_0$ and either $ALB_{l_0} \subseteq \{w_0\}$ or $ARB_{l_0} \subseteq \{w_0\}$ by line (18). Also $\min(ALB_l \cup ARB_l) \geq \max(ALB_{l-1} \cup ARB_{l-1}) \geq \max(ARB_{l_0} \cup ALB_{l_0})$ for $l > l_0$ by Lemma 5 a) and hence either $ALB_l = \emptyset$ or $ARB_l = \emptyset$ for $l > l_0$ by line (18). Thus lines (20) to (22) work correctly and the correctness proof is complete.

We still have to analyze the running time. Note first that *stronglyplanar* is called at most once for each edge. Also, each tree edge belongs to exactly one spine. Hence the total time spent in lines (1), (2), (3), (4), (5) (without counting the time spent within recursive calls),

```
(0)   procedure stronglyplanar(e_0 : edge);
      co tests whether segment S(e_0), e_0 = (x, y), is strongly planar.
         If so, it returns the ordered (according to dfsnum) list of
         attachments of S(e_0) excluding x oc
(1)   find the spine of cycle C(e_0) by starting in node y and always
      taking the first edge on every adjacency list until a back edge is
      encountered. This back edge leads to node w_0 = lowpt[y].
      Let w_0, ..., w_r be the tree path from w_0 to x = w_r and
      and let w_{r+1} = y, ..., w_k be the spine constructed above;
(2)   let S be an empty stack of blocks;
(3)   for j from k downto r + 1
(4)   do for all edges e' (except the first) emanating from w_j
(5)      do stronglyplanar(e');
(6)         let A(e') be the ordered list of attachments of S(e')
            as returned by the successful call stronglyplanar(e');
(7)         update stack S as described in Programs 2 and 3
(8)      od;
(9)      let B_h be the top entry in stack S;
(10)     while max(ALB_h ∪ ARB_h) = w_{j-1}
(11)     do remove node w_{j-1} from ALB_h and ARB_h;
(12)        if ALB_h and ARB_h become empty
(13)        then pop B_h from the stack; h ← h - 1 fi
(14)     od
(15)  od;
      co if control reaches this point then IG(C) is bipartite.
         We will now test for strong planarity and compute A(e_0) oc
(16)  L ← ∅;      co an empty list oc
(17)  for l from 1 to h
(18)  do if max ALB_l ≥ w_1 and max ARB_l ≥ w_1
(19)     then declare S(e_0) not strongly planar and stop fi;
(20)     if ALB_l ≠ ∅ and max ALB_l ≥ w_1
(21)     then L ← L conc ARB_l conc ALB_l
(22)     else  L ← L conc ALB_l conc ARB_l fi
(23)  od;
(24)  return L
(25)  end.
```

Program 4

(6), (8), (9) and (16) is $O(m)$. Let us look at line (7) next. Observe that line (7) is executed at most once for each edge. Also, at most one block is pushed on stack S in one execution of line (7), and execution time of line (7) is proportional to the number of entries removed from stack S. Since only m elements are added to stacks S altogether, only m elements can be removed and hence the total time spent in line (7) is $O(m)$. The same argument shows that the total time spent in lines (17)–(23) is $O(m)$, because the time spent in these lines is proportional to the number of elements removed from stacks S in these lines. Lines (10)–(14) still remain to be considered. Only endpoints of back edges are placed on lists ALB and ARB. No back edge is placed twice on a list and each back edge is removed at most once. Hence the total cost of lines (10)–(14) is $O(m)$. ∎

Theorem 11 *Let $G = (V, E)$ be a graph. Then planarity of G can be tested in time $O(n)$.*

Proof: If $m > 3n - 6$ then G is non-planar. If $m \leq 3n - 6$ then we can divide G into its biconnected components in time $O(m) = O(n)$. For each biconnected component we can test its planarity in linear time. Also, a graph is planar iff its biconnected components are planar. ∎

References

[Meh84] K. Mehlhorn. *Data Structures and Efficient Algorithms.* Springer Verlag, 1984.

[MM94] K. Mehlhorn 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.

[MN89] Kurt Mehlhorn and Stefan Näher. LEDA: A library of efficient data types and algorithms. In *MFCS 89*, LNCS, 1989. CACM, to appear.

[Nae93] St. Näher. LEDA Manual. Technical report, Max-Planck-Institut für Informatik, 1993.

[Wil84] S.G. Williamson. Depth-first search and Kuratowksi subgraphs. *Journal of the ACM*, 11:681–693, 1984.

Index

List of Refinements

⟨ auxiliary functions 9, 10, 15, 16, 18, 27 ⟩ Used in section 2.
⟨ bidirect G 7 ⟩ Used in section 6.
⟨ compute w's adjacency list and prepare for next iteration 32 ⟩ Used in section 29.
⟨ construct embedding 26 ⟩ Used in section 5.
⟨ `demo.c` 35 ⟩
⟨ determine the cycle $C(e0)$ 19 ⟩ Used in section 18.
⟨ embed recursively 30 ⟩ Used in section 29.
⟨ embed: determine the cycle $C(e0)$ 28 ⟩ Used in section 27.
⟨ first version of *planar* 5 ⟩ Used in section 2.
⟨ includes 3, 36 ⟩ Used in sections 2 and 35.
⟨ initiation and declarations 38 ⟩ Used in section 35.
⟨ make G a copy of Gin and add edges to make G bidirected 6 ⟩ Used in section 5.
⟨ make G biconnected 8 ⟩ Used in section 5.
⟨ make H a copy of G 12 ⟩ Used in section 8.
⟨ `planar.c` 2 ⟩
⟨ `planar.h` 1 ⟩
⟨ prepare for next iteration 24 ⟩ Used in section 21.
⟨ prepare the output 33 ⟩ Used in section 27.
⟨ procedure to draw graphs 37 ⟩ Used in section 35.
⟨ process all edges leaving the spine 21 ⟩ Used in section 18.
⟨ process the subsegments 29 ⟩ Used in section 27.
⟨ reset window 41 ⟩ Used in section 35.
⟨ second version of *planar* 4 ⟩ Used in section 2.
⟨ select graph 39 ⟩ Used in section 35.
⟨ test graph for planarity and show output 40 ⟩ Used in section 35.
⟨ test planarity 13 ⟩ Used in section 5.
⟨ test recursively 22 ⟩ Used in section 21.
⟨ test strong planarity and compute Att 25 ⟩ Used in section 18.
⟨ typedefs, global variables and class declarations 11, 14, 17, 20 ⟩ Used in section 2.
⟨ update lists T, Al, and Ar 31 ⟩ Used in section 29.
⟨ update stack S of attachments 23 ⟩ Used in section 21.