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

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

1994, revised March 98

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).

1 Implementation History

The first implementation was written by the authors in 94. Stefan Näher extended the implementation to multi-graphs in 95. In March 98, Kurt Mehlhorn extended the implementation to multi-graphs with self-loops.

2 Introduction

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

$$bool\ planar\ (graph\&\ G,\ bool\ embed\ =\ false)$$

and

$$bool\ planar\ (graph\&\ G,\ list\ <edge>\&\ P,\ bool\ embed\ =\ false).$$

*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).

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 HT_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 6 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 HT_PLANAR is based on the LEDA platform of combinatorial and geometric computing [MNU97, KN89]. 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 HT_PLANAR and a demo, and report on our experimental experience.

Procedure HT_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]. Our procedure HT_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 [MM95]. We want to remark that the description of the embedding phase given in [Meh84, section IV.10] is false. The essential part of [MM95] is reprinted in section 5.

This document defines the files *planar.h*, *planar.c*, and *demo.c*. *planar.c* contains the code for procedure HT_PLANAR, *demo.c* contains the code for a demo, and *planar.h* consists of the declarations of procedure HT_PLANAR. The third file is defined in section 7, the structure of the first two files is as follows:

⟨*planar.h*⟩≡

```
bool        planar (graph& G, bool embed);
bool        planar (graph& G, list <edge>& P, bool embed);
void         Make_biconnected_graph(graph& G);
```

⟨*planar.c*⟩≡

⟨*includes*⟩;
⟨*typedefs, global variables and class declarations*⟩;
⟨*auxiliary functions*⟩;
⟨*first version of planar*⟩;
⟨*second version of planar*⟩;

We include parts of LEDA (who would want to work without it) [MNU97, KN89]. We need stacks, graphs, and graph algorithms.

⟨*includes*⟩≡

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

The second version of HT_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*⟩≡

```
bool HT_PLANAR(graph& G, list <edge>& P, bool embed)
{
  if (HT_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.

⟨*second version of planar*⟩+≡

```
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.

⟨*second version of planar*⟩+≡

```
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.

⟨*second version of planar*⟩+≡

```
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 (HT_PLANAR(H,false))
  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 K_5 or $K_{3,3}$. We still need to translate back to G.

⟨*second version of planar*⟩+≡

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

The first version of HT_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 HT_PLANAR alters the input graph, it works on a copy of it.

The above assumes that G is a graph without self-loops and parallel edges. In the spring of 98 we added to ability to deal with self-loops and parallel edges. We use the function *copy_simplified_graph* to make a copy of *Gin* in G; the copy contains no self-loops and only one edge from every bundle of parallel edges.

The original algorithms requires a graph with at least three nodes. If we have less we just add them.

⟨*first version of planar*⟩≡

```
bool HT_PLANAR(graph& Gin, bool embed)
/* |Gin| is a directed graph. HT\_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 a map in that case. */
{
  float T = used_time();
  GRAPH <node,edge> G;
  edge_array <edge> companion_in_G(Gin,nil);
  node_array <node> link(Gin);
  bool Gin_is_map = Gin.make_map();
  if ( embed && !Gin_is_map )
    error_handler(1,"HT_PLANAR: sorry: can only embed maps");
  edge_array<int> offset(Gin,0);
  copy_simplified_graph(Gin,G,link,companion_in_G,offset,Gin_is_map);
  //if (embed) assert(G.is_map());
  int n = G.number_of_nodes();
  int m = G.number_of_edges();
  if (n >= 3 && m > 6*n - 12) return false;
  /* An undirected planar graph with three or more nodes
   * has at most $3n - 6$ edges; a directed graph may
   * have twice as many */
  // for some strange region the planarity test and the embedding
  // function want at least three nodes
  for (int i = 0; i < 3 - n; i++) G.new_node();
  list<edge> el = Make_Biconnected(G);
  edge e;
```

```
  forall(e,el)
  { edge r = G.new_edge(target(e),source(e));
    G.set_reversal(e,r);
  }
  if ( !Gin_is_map ) G.make_map(el);
  //else assert(G.is_map());
#ifdef BOOK
  cout << "time to copy and to make bidirected and connected  " << used_time(T);
  newline;
cout << "number of nodes and edges  " << n << "  " << Gin.number_of_edges();
  newline;
#endif

  GRAPH<node,edge> H;
  edge_array<edge> companion_in_H(Gin);
  { node v;
    edge e;
    forall_nodes(v, G) G[v] = H.new_node(v);
    forall_edges(e, G) G[e] = H.new_edge(G[source(e)],G[target(e)],e);
    forall_edges(e, Gin)
    { if ( Gin.source(e) != Gin.target(e) )
      companion_in_H[e] = G[companion_in_G[e]];
    }
    forall_edges(e, G)
      H.set_reversal(G[e],G[G.reversal(e)]);
  }
 // assert(H.is_map());

  ⟨test planarity⟩;
#ifdef BOOK
cout << "time to test planarity   " << used_time(T);
  newline;
#endif
  if (embed)
  { ⟨construct embedding⟩
#ifdef BOOK
cout << "time to construct embedding   " << used_time(T);
    newline;
#endif
  }
  return true;
}
```

3 Copy and Simplify

Let G be a graph. We make a simplified copy H of G; we do not copy self-loops and we copy only one edge in a bundle of parallel edges. For every node v and edge e of H we store the original in $H[v]$ and $H[e]$, respectively, and for each node v of G and each non-loop edge e of G we store the representative in H in *v_in_H*$[v]$ and *e_in_H*$[e]$, respectively.

If an embedding is to be constructed we do slighly more. This is to be described below.

⟨*auxiliary functions*⟩≡

```
static node_array<int> node_ord;
static int source_ord(const edge& e) { return node_ord[source(e)]; }
static int target_ord(const edge& e) { return node_ord[target(e)]; }

static void copy_simplified_graph(const graph& G, GRAPH<node,edge>& H,
                      node_array<node>& v_in_H, edge_array<edge>& e_in_H,
                      edge_array<int>& offset, bool Gin_is_map)
{
  node v;
  forall_nodes(v,G) v_in_H[v] = H.new_node(v);

  list<edge> el = G.all_edges();

  if ( el.empty() ) return;

  node_ord.init(G,0);
  int  nr = 0;
  int  n = G.number_of_nodes();

  forall_nodes(v,G) node_ord[v] = nr++;
  el.bucket_sort(0,n-1,&source_ord);
  el.bucket_sort(0,n-1,&target_ord);

  // now parallel edges are adjacent in el

  if ( !Gin_is_map )
  { // we delete self loops and parallel edges
    list_item it = el.first();
    while ( it )
    { edge e0 = el[it];
      node v = G.source(e0); node w = G.target(e0);
      if ( v == w )
        e_in_H[e0] = nil;
      else
        e_in_H[e0] = H.new_edge(v_in_H[v],v_in_H[w],e0);

      edge e;
      list_item it1 = el.succ(it);
      while (it1 &&
             source(e0) == source(e = el[it1]) &&
             target(e0) == target(e) )
      { e_in_H[e] = e_in_H[e0];
        it1 = el.succ(it1);
      }
      it = it1;
    }
    return ;
  }
  ⟨copy simplified graph when Gin_is_map is true⟩
}
```

When *Gin_is_map* is true, we need to work harder. We want to achieve the following:

- In each bundle of uedges one uedge is selected, i.e., the two edges comprising the uedge are selected. The two edges are copied to H (if they are not self-loops).
- The other uedges in the bundle are numbered. We store the number with both edges in a uedge.

We scan over the edges in *el*. Whenever we encounter an edge *e0* whose offset is not defined, we treat the bundle of uedges containing the edge *e0*. Let *r0* be the reversal of *e0*. If *e0* is a self-loop we do not insert a representative into H. Otherwise, we insert representatives for *e0* and *r0*. We then scan over all edges parallel to *e0*. We set their representative to the representative of *e0* and we number them. We use the negative number for the reversal.

Assume now that we sort the edges in a bundle by offset. Then the edges in a bundle will be oriented clockwise at one of the endpoints and counterclockwise at the other end. Hence they match up perfectly.

For a bundle of self-loops the self-loops will be nested.

In order to cope with self-loops we need to test again whether a node is already numbered.

⟨*copy simplified graph when Gin_is_map is true*⟩≡

```
// we need to work harder because we want to embed
list_item it = el.first();
while ( it )
{ edge e0 = el[it];
  if ( offset[e0] != 0 ) { it = el.succ(it); continue; }
  edge r0 = G.reversal(e0);
  node v = G.source(e0); node w = G.target(e0);
  if ( v == w )
    { e_in_H[e0] = nil; e_in_H[r0] = nil; }
  else
  { e_in_H[e0] = H.new_edge(v_in_H[v],v_in_H[w],e0);
    e_in_H[r0] = H.new_edge(v_in_H[w],v_in_H[v],r0);
    H.set_reversal(e_in_H[e0],e_in_H[r0]);
  }
  edge e;
  list_item it1 = it; int off = 1;
  while (it1 &&
        source(e0) == source(e = el[it1]) &&
        target(e0) == target(e))
  { e_in_H[e] = e_in_H[e0]; e_in_H[G.reversal(e)] = e_in_H[r0];
    if (offset[e] == 0)
    { offset[e] = off; offset[G.reversal(e)] = -off;
      off++;
    }
    it1 = el.succ(it1);
  }
  it = it1;
}
//assert(H.is_map());
edge e;
//forall_edges(e,G) assert(offset[e] == -offset[G.reversal(e)]);
//forall_edges(e,G) assert(G.source(e) == G.target(e) || e_in_H[e] != nil);
//forall_edges(e,G) assert(offset[e] != 0);
```

4 The Planarity Test

We are now ready for the planarity test proper. We follow [Meh84, page 95]. We first compute *dfsnumber*s and *parent*s, 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*⟩≡

```
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;
}
```

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

⟨*typedefs, global variables and class declarations*⟩≡

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

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*⟩+≡

```
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 \cite[page 101]{ Me:book } */
  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);
}
```

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

⟨*auxiliary functions*⟩+≡

```
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
```

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

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

```
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);
```

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.

⟨*auxiliary functions*⟩+≡

```
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)⟩;
  ⟨process all edges leaving the spine⟩;
  ⟨test strong planarity and compute Att⟩;
  return true;
}
```

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].

⟨*determine the cycle C(e0)*⟩≡

```
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);
```

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*⟩+≡

```
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)
{
  /* 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)
  {
    /* 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;
    }
  };
```

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*⟩≡

```
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⟩;
      ⟨update stack S of attachments⟩;
    } // end if
  } //end forall
  ⟨prepare for next iteration⟩;
  w = parent[w];
} //end while
```

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*⟩≡

```
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
```

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*⟩≡

```
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);
```

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*⟩≡

```
while (!S.empty() && (S.top())->clean(dfsnum[parent[w]],alpha)) delete(S.pop());
```

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*⟩≡

```
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);

  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]);
```

5 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, $1 \leq i \leq m$, 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, $1 \leq i \leq m$, 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 non - standard 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 $1 \leq i < r$ if $t = L$, and $AL(w_i) = \emptyset$ for $1 \leq i < r$, if $t = R$. The lists $AL(w_i)$, $AR(w_i)$, $0 \leq i \leq r$, 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, $0 \leq l \leq j-1$, 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 $t \neq \alpha(e')$, 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

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

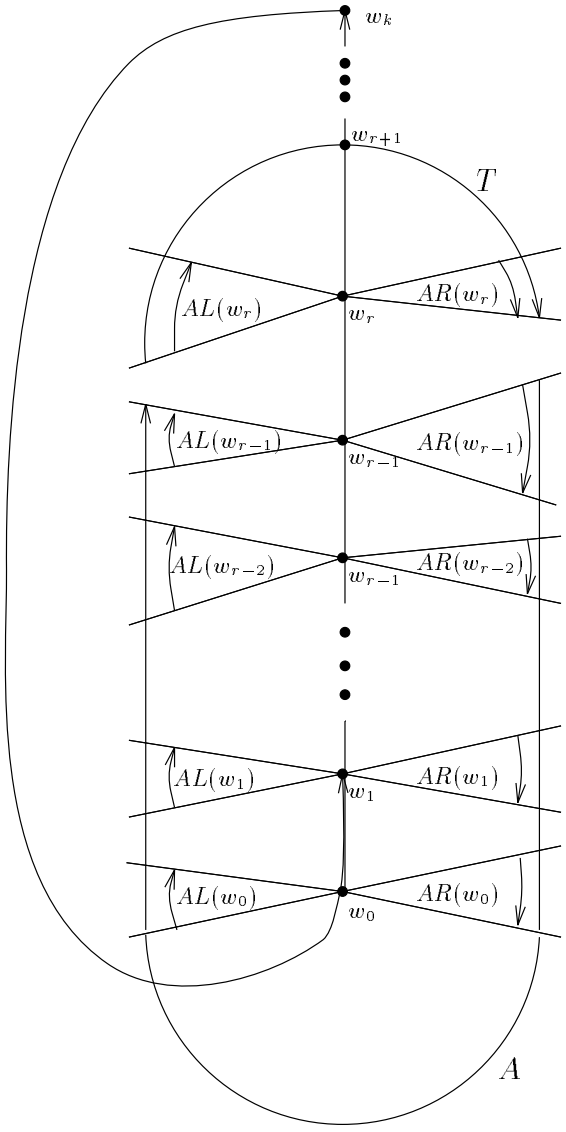

Figure 1: A call *embedding* (e_0, t) returns lists T and A.

```
(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

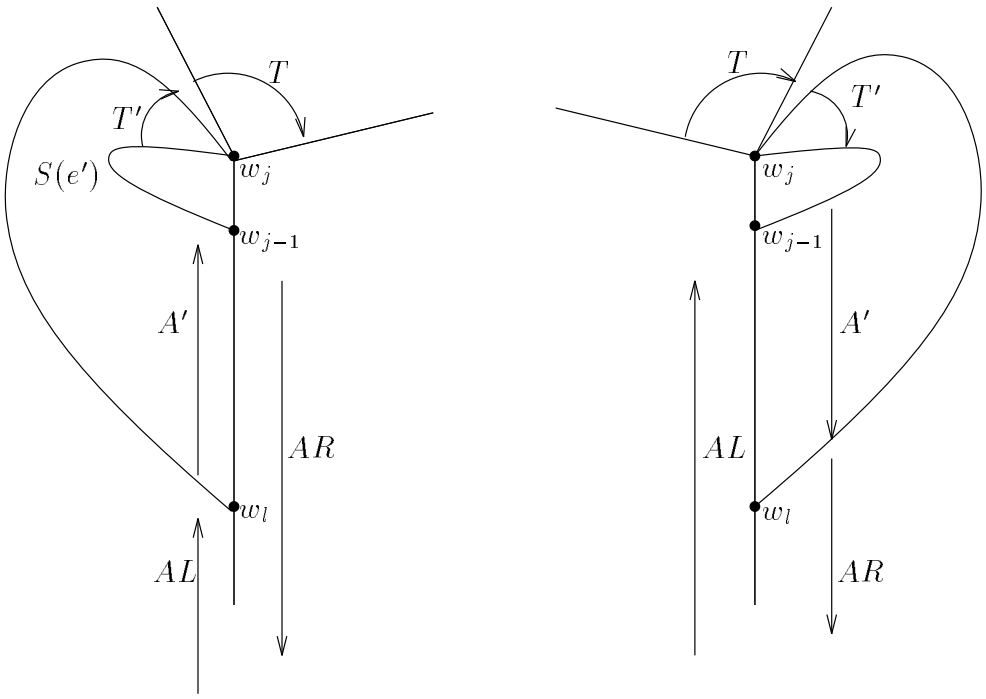

Figure 2: Glueing $S(e')$ to the left or right side of the path $w_0, \ldots, w_k$ respectively.

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.

We still need to explain how we deal with self-loops and with multiple edges. The embedding algorithm works on a simple graph without self-loops. Consider now a bundle of parallel uedges. When constructing the graph G from *Gin* we determined a numbering of the edges of G such that the two edges comprising a uedge have the same number and such that all parallel uedges are numbered consecutively.

This is simple. We give each edge the number of the representative in H and then add its offset.

⟨*construct embedding*⟩≡

```
{ 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,H); //,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<double> sort_Gin(Gin,0);

  int mh = H.number_of_edges();  // mh - 1 is maximal value for sort_num
  double two_m = 2*(1 + Gin.number_of_edges());

                               // maximal absolute value of offset is
                               // Gin.number_of_edges

  edge ein;
  forall_edges(ein,Gin)
  if ( Gin.source(ein) == Gin.target(ein) )
    sort_Gin[ein] = two_m * ( mh + node_ord[Gin.source(ein)]) + offset[ein];
  else
    sort_Gin[ein] = two_m * sort_num[companion_in_H[ein]] + offset[ein];

  Gin.sort_edges(sort_Gin);
}
```

It remains to describe procedure *embedding*.

⟨*auxiliary functions*⟩+≡

```
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,GRAPH<node,edge>& H) //, edge_array<edge>& reversal)
{ ⟨embed: determine the cycle C(e0)⟩;
  ⟨process the subsegments⟩;
  ⟨prepare the output⟩;
}
```

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)*⟩≡

```
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;
```

Lines (2) to (14).

⟨*process the subsegments*⟩≡

```
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⟩;
      ⟨update lists T, Al, and Ar⟩;
    } // end if
  } //end forall
  ⟨compute adjacency list of w and prepare for next iteration⟩;
  w = parent[w];
} //end while
```

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

⟨*embed recursively*⟩≡

```
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,H); //,reversal);
}
else         { /* back edge */
               Tprime.append(G[e]); //$e$
               Aprime.append(H.reversal(G[e])); //reversal of $e$
}
```

Lines (6) to (9).

⟨*update lists T, Al, and Ar*⟩≡

```
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$
}
```

Lines (11) to (13).

⟨*compute adjacency list of w and prepare for next iteration*⟩≡

```
T.append(H.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
}
```

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

⟨*prepare the output*⟩≡

```
A.clear();

A.conc(Ar);
A.append(H.reversal(G[back_edge_into_w0]));
A.conc(Al);
```

6 Efficiency

Under LEDA 3.0 the space requirement of the first version of HT_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 HT_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.

7 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_and_test.c*⟩≡

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

#include "planar.c"

⟨procedure to draw graphs⟩
main()
{ ⟨initiation and declarations⟩
  while (true)
  { ⟨select graph⟩
    ⟨test graph for planarity and show output⟩
    ⟨reset window⟩
  }
  return 0;
}
```

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

⟨*procedure to draw graphs*⟩≡

```
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);

}
```

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

⟨*initiation and declarations*⟩≡

```
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 interactively");
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 highlighted.");
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 of 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(" ");
```

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

⟨*select graph*⟩≡

```
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;
         }
  }
```

We test the planarity of our graph G using our procedure HT_PLANAR.

⟨*test graph for planarity and show output*⟩≡

```
if (HT_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 ...");
    Make_Biconnected(G);
    list<edge> n_edges;
    G.make_map(n_edges);
    HT_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;
  HT_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);
}
```

We reset the graphics window.

⟨*reset window*⟩≡

```
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);
```

Another small program for timing purposes

⟨*timing.c*⟩≡

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

graph G;

main()
{ for(int n = 1000; n<= 32000; n = 2*n)
  { G.clear();
    random_planar_graph(G,n);
    Make_Simple(G);
    list<edge> new_edges;
    G.make_map(new_edges);

    newline;
    newline;
    HT_PLANAR(G,true);
    newline;
  }
}
```

References

[KN89] K.Mehlhorn and Stefan Näher. LEDA: A library of efficient data types and algorithms. In *MFCS 89*, LNCS, 1989. CACM, to appear.

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

[MM95] K. Mehlhorn and P. Mutzel. On the Embedding Phase of the Hopcroft and Tarjan Planarity Testing Algorithm. *Algorithmica*, 16(2):233–242, 1995.

[MNU97] Kurt Mehlhorn, S. Näher, and Ch. Uhrig. The LEDA User Manual (Version R 3.5). Technical report, Max-Planck-Institut für Informatik, 1997. http://www.mpi-sb.mpg.de/LEDA/leda.html.

[Wil84] S.G. Williamson. Depth-First Search and Kuratowski Subgraphs. *JACM*, 31(4):681–693, 1984.