Contents

8 Embedded Graphs *page* 2
8.1 Drawings 3
8.2 Bidirected Graphs and Maps 5
8.3 Embeddings 10
8.4 Order-Preserving Embeddings of Maps and Plane Maps 15
8.5 The Face Cycles and the Genus of a Map 17
8.6 Faces, Face Cycles, and the Genus of Plane Maps 19
8.7 Planarity Testing, Planar Embeddings, and Kuratowski Subgraphs 23
8.8 Manipulating Maps and Constructing Triangulated Maps 69
8.9 Generating Plane Maps and Graphs 74
8.10 Faces as Objects 76
8.11 Embedded Graphs as Undirected Graphs 79
8.12 Order from Geometry 80
8.13 Miscellaneous Functions on Planar Graphs 82

Bibliography 86

8 Embedded Graphs

Drawings of graphs are ubiquitous. In this chapter we introduce important mathematical concepts related to embedded graphs and we discuss algorithms that draw and embed graphs and that deal with embedded graphs. We provide only a minimum of the required mathematics and refer the reader to [Whi73] for a detailed treatment.

We start with the definition of what it means to draw a graph and an example of a drawing algorithm. We discuss bidirected graphs and maps, our technical vehicle for dealing with embedded graphs, in Section 8.2 and the concepts of embedding and planar embedding in Section 8.3. In this section we also introduce functions that test the planarity of a graph, that construct a plane embedding of a planar graph, and that exhibit a Kuratowski subgraph in a non-planar graph. Their implementation is discussed in Section 8.7. Sections 8.4 and 8.5 introduce order-preserving embeddings, plane maps, face cycles, and the genus of maps. In Section 8.6 and 8.12 we relate combinatorics and geometry. In particular, we prove that a map is plane if and only if its genus is zero, we derive an upper bound on the number of edges of any planar graph and we show how to construct the map induced by geometric positions assigned to the nodes of a graph. In Section 8.8 we show how to modify maps, in Section 8.9 we discuss the generation of random plane maps, and in Section 8.13 we introduce functions that five-color a planar graph and choose a large independent set in a planar graph. Section 8.10 introduces face items as a means of dealing with faces in the same way as with nodes and edges. In Section 8.11 we discuss our design choice of representing maps by directed graphs instead of undirected graphs.

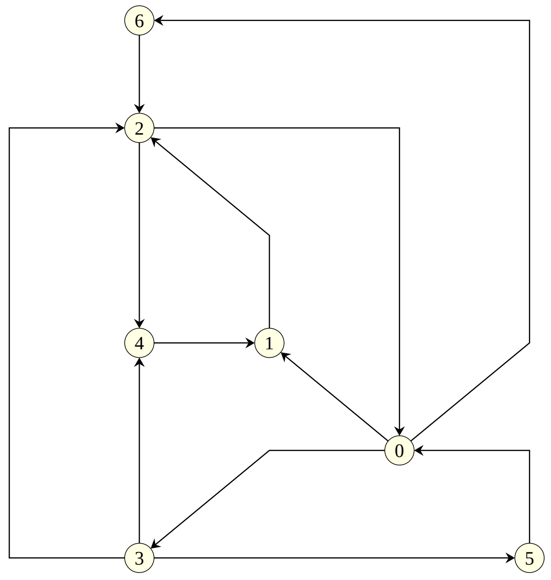

Figure 8.1 A drawing produced by one of the graph drawing algorithms in AGD [JMN].

8.1 Drawings

We have already seen many drawings of graphs in this book. We have never defined what we mean by a drawing, embedding, and planar embedding.

Let G be a graph and let S be a surface, e.g., the plane or the sphere or the torus. We will be almost exclusively concerned with the plane in this book. However, the concepts also apply to more complex surfaces.

A *drawing* I of G in S assigns a point $I(v) \in S$ to every node v of G and a Jordan curve[1] $I(e)$ to every edge $e = (v, w)$ such that:

(1) distinct points are assigned to distinct nodes, i.e., $I(v) \neq I(w)$ for $v \neq w$,
(2) the curve assigned to any edge connects the endpoints of the edge, i.e., if $e = (v, w)$ then $I(e)(0) = I(v)$ and $I(e)(1) = I(w)$.

A drawing in the plane is called a straight line drawing if every edge is drawn as a straight line segment. Figure 8.2 shows some drawings.

An algorithm, that takes a graph and produces a drawing for it, is called a *graph drawing algorithm*[2]. LEDA provides some graph drawing algorithms; see the section on graph drawing in the manual and *try the button layout in a GraphWin for a demonstration.* Many more graph drawing algorithms are available in the

[1] A Jordan curve c is a curve without self-intersections, i.e., a continuous mapping $c : [0,1] \longrightarrow S$ with $c(x) \neq c(y)$ for $0 \leq x < y < 1$.
[2] Graph drawing is an active area of research, see [BETT94, EM98, DETT98] for surveys.

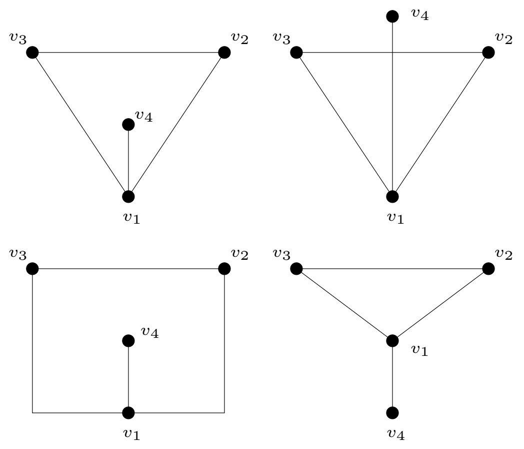

Figure 8.2 Some drawings of the same graph. All drawings except for the right upper drawing are embeddings.

systems AGD [JMN] and GDToolkit [Bat]. Both systems are based on LEDA. Figure 8.1 shows a drawing produced by an algorithm in AGD.

The functions

```
void SPRING_EMBEDDING(const graph& G,
            node_array<double>& xpos, node_array<double>& ypos,
            double xleft, double xright, double ybottom, double ytop,
            int iterations = 250);
void SPRING_EMBEDDING(const graph& G, const list<node>& fixed,
            node_array<double>& xpos, node_array<double>& ypos,
            double xleft, double xright, double ybottom, double ytop,
            int iterations = 250);
```

compute straight line drawings of a graph G using a so-called *spring embedder*[3]. A spring embedder works iteratively. It models the nodes of a graph as points in the plane that repulse each other, and it models each edge as a spring between the endpoints of the edge. In each iteration the force acting on any node is computed as the sum of repulsive forces (from all other nodes) and attractive forces (from incident edges), and the node is moved accordingly. The number of iterations is determined by the parameter *iterations*.

The x- and y-coordinates of the positions assigned to the nodes of G are returned in *xpos* and *ypos*, respectively, and the points are constrained to lie in the rectangle defined by *xleft*, *xright*, *ybottom*, and *ytop*. The second version of the function keeps the positions of the nodes in *fixed* fixed.

Drawings in which edges do not cross are particularly nice. We call such drawings embeddings. Out of the four drawings shown in Figure 8.1 three are embeddings. Embeddings are the topic of Section 8.3. The graphs in Figure 8.2 are undirected. For the purposes of this chapter it is convenient to distinguish between the two

[3] The name spring drawer would be more appropriate, as spring embedders do not produce embeddings, but drawings. However, the name spring embedder is in general use.

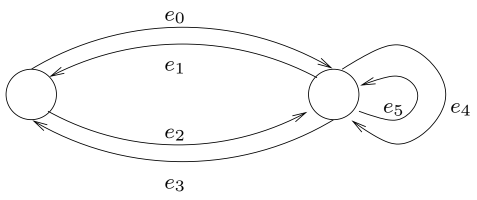

Figure 8.3 A bidirected graph: We have $reversal(e_{2i}) = e_{2i+1}$ and $reversal(e_{2i+1}) = e_{2i}$ for all i with $0 \le i \le 2$. Requirement (2) excludes the possibility that $reversal(e_0) = e_1$, and $reversal(e_3) = e_0$, and requirement (3) excludes the possibility that $reversal(e_4) = e_4$ and $reversal(e_5) = e_5$.

orientations of an edge. This leads to the concepts of bidirected graphs and maps, which we treat in the next section.

Exercise for 8.1

1 Implement a spring embedder.

8.2 Bidirected Graphs and Maps

A directed graph $G = (V, E)$ is called *bidirected* if there is a bijective function $reversal : E \to E$ such that for every edge $e = (v, w)$ with $e^R = reversal(e)$:

(1) $e^R = (w, v)$, i.e., $source(e) = target(e^R)$ and $target(e) = source(e^R)$,
(2) $reversal(e^R) = e$, and
(3) $e \neq e^R$.

Property (1) ensures that reversal deserves its name, and properties (2) and (3) ensure that reversal behaves properly in the presence of parallel edges and self-loops. Figure 8.3 shows an example of a bidirected graph and also illustrates properties (2) and (3). A bidirected graph has an even number of edges.

The function

```
bool G.is_bidirected();
```

returns *true* if G is bidirected and returns *false* otherwise. The function

```
void G.make_bidirected(<list<edge>& R);
```

adds a minimum number of edges to G so as to make G bidirected. The added edges are returned in R.

Every edge e of any graph G has a reversal information associated with it. It is accessed through

```
G.reversal(e)
```

and has type *edge*. The reversal information of an edge is either undefined (= *nil*) or is an edge e^R satisfying (1) to (3). The operation

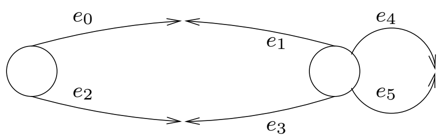

Figure 8.4 A map: Every pair of edges $\{e, e^R\}$ with *reversal*$(e) = e^R$ and *reversal*$(e^R) = e$ is drawn as two half-edges. For each half-edge the name of the half-edge is shown on the left side of the half-edge.

```
G.set_reversal(e,f)
```

sets the reversal information of e to f and the reversal information of f to e. The function checks whether the created reversal information is legal and aborts if it is not. If the reversal information of e was defined prior to the operation, the reversal information of e^R is set to *nil* by the operation. The same holds true for f.

A *map* is a graph in which the reversal information of every edge is defined. A map is always a bidirected graph and every bidirected graph can be turned into a map by setting the reversal information appropriately. The function

```
bool G.is_map()
```

returns *true* if G is a map and the functions

```
bool G.make_map()
void G.make_map(list<edge>& R)
```

turn G into a map by setting the reversal information of every edge. The first function requires that G is bidirected (if G is not bidirected, the function returns *false* and sets the reversal information of a maximal number of edges), the second function adds a minimum number of edges to G so as to make G bidirected and then turns G into a map. Both functions preserve reversal information, i.e., if *reversal*(e) is defined before the call, then *reversal*(e) is not changed by either call.

We call a pair of edges $\{e, e^R\}$ with *reversal*$(e) = e^R$ (and hence *reversal*$(e^R) = e$) a *uedge* (undirected edge) and say that e and e^R form the uedge. The uedge comprising e and e^R is denoted $\{e, e^R\}$ or $\{v, w\}$, where v and w are the two endpoints of e. The latter notation is ambiguous in the presence of parallel edges. We depict maps as shown in Figure 8.4. For every uedge $\{e, e^R\}$ we draw "two half-edges that meet" and label them e and e^R, respectively.

We have no iteration statement that iterates over the uedges of a graph. However, it is easy to obtain the effect of iterating over uedges.

```
forall_edges(e,G)
{ if ( index(e) > index(G.reversal(e)) ) continue;
  <body of loop>
}
```

Observe that the body of the loop is executed for exactly one edge in each uedge, namely the one with smaller index.

We describe the implementations of some of the functions introduced above. We also introduce a function that checks whether the reversal information of all edges is properly defined. This section may be skipped on first reading.

We start with a function *check_reversal_inf* that checks whether the reversal information of every edge is either nil or satisfies (1) to (3) and raises an error if this is not the case[4]. The function is non-trivial to write because it cannot assume that the reversal information of an edge has a meaningful value, i.e., the function has to cope with the possibility that $G.reversal(e)$ is non-nil and not an edge of G for some e.

We proceed as follows. We introduce a map *is_edge_of_G* from edges to bool that we initialize to *false*. We then set *is_edge_of_G*$[e]$ to *true* for all edges e of G. Next, we iterate again over all edges e of G and make sure that *reversal*(e) is either *nil* or an edge of G. In a third step we make sure that (1) to (3) holds for all edges e whose reversal information is not *nil*.

⟨*check_reversal_inf.c*⟩+≡

```
bool check_reversal_inf(const graph& G)
{ map<edge,bool> is_edge_of_G(false);
  edge e;
  forall_edges(e,G) is_edge_of_G[e] = true;
  forall_edges(e,G)
  { edge r = G.reversal(e);
    if ( r == nil || !is_edge_of_G[r]) return false;
  }
  forall_edges(e,G)
  { edge r = G.reversal(e);
    if (r == e || G.reversal(r) != e ||
       G.source(e) != G.target(r) || G.target(e) != G.source(r) )
    return false;
  }
  return true;
}
```

It is instructive to investigate what can go wrong when only the third *forall_edges* loop is executed. It would then be possible that r is different from *nil* but not an edge of G. The access to the reversal, target, or source of r could then result in a segmentation fault. The program above guards against this possibility by ensuring first that the reversal of any edge e of G is either *nil* or an edge of G.

We next show the implementation of the function *make_map*. Its implementation is derived from the function *Is_Bidirected* given in Section **??**.

A call of *G.make_map*( ) sets the reversal information of a maximal number of edges. We proceed as follows: let v_1, v_2, …, v_n be an arbitrary order on the nodes

[4] We use the function *check_reversal_inf* for testing purposes. Of course, all functions of the LEDA system are designed to preserve the invariant that the reversal of every function is either nil or an edge of G satisfying (1) to (3) and hence, if none of the implementers of LEDA had ever made a mistake, the function would have never raised an error.

of G, e.g., the ordering given by the internal numbering of the nodes[5]. We make two lists EST and ETS of all edges whose reversal information is undefined. EST starts with all edges out of v_1, followed by all edges out of v_2, ... For each i, the edges out of v_i are in increasing order of their target node. ETS starts with all edges into v_1, followed by all edges into v_2, ... For each i, the edges into v_i are in increasing order of the source node. We also want the self-loops incident to any v_i to appear in reverse order in the two lists.

The lists EST and ETS are easy to generate. We collect all edges whose reversal information is undefined in a list EST and use bucket sort to rearrange EST in increasing lexicographic order. We use the index of the source node of an edge as the primary key and the index of the target node as the secondary key. For ETS we interchange the roles of the primary and the secondary key, and we initialize ETS to the reversal of EST. The effect of initializing ETS with the reversal of EST instead of with ETS is that the self-loops incident to any v_i appear in reverse order in the two lists; this follows from the fact that bucket sort is stable.

Having rearranged both lists we establish the reversal information. EST starts with all edges out of v_1 sorted in order of increasing target and ETS starts with all edges into v_1 sorted in order of increasing source. Both lists start with all self-loops incident to v_1.

We scan over both lists and check whether the first edge on EST, call it e, can be paired with the first edge on ETS, call it r. We can pair e and r if none of them was paired previously and if $\mathit{source}(e) = \mathit{target}(r)$, $\mathit{target}(e) = \mathit{source}(r)$, and $e \neq r$. If e and r can be paired, we pair them by setting their reversal information appropriately. The function succeeds if all edges can be paired.

So assume that e and r cannot be paired. We show that at least one of e and r will never find a partner.

Assume first that $\mathit{source}(e) \neq \mathit{target}(r)$. If $\mathit{source}(e) < \mathit{target}(r)$ then ETS contains no further edge which ends in $\mathit{source}(e)$. Thus e cannot be paired. Similarly, if $\mathit{source}(e) > \mathit{target}(r)$ then EST contains no further edge that starts in $\mathit{target}(r)$. Thus r cannot be paired.

Assume next that $\mathit{source}(e) = \mathit{target}(r)$ and $\mathit{target}(e) \neq \mathit{source}(r)$. If $\mathit{target}(e)$ is less than $\mathit{source}(r)$ then ETS contains no further edge that starts in $\mathit{source}(e)$ and ends in $\mathit{target}(e)$ and hence e cannot be paired. If $\mathit{target}(e)$ is greater than $\mathit{source}(r)$ then EST contains no further edge that ends in $\mathit{target}(r)$ and starts in $\mathit{source}(r)$ and hence r cannot be paired.

Assume finally that $\mathit{source}(e) = \mathit{target}(r)$ and $\mathit{target}(e) = \mathit{source}(r)$ and $e = r$, i.e., e is a self-loop. Since EST and ETS contain the self-loops incident to any node in reverse order this can only happen if there is an odd number of self-loops incident to $\mathit{source}(e)$ and if e is the middle element of the block of self-loops incident

[5] The internal number of a node v is given by $\mathit{index}(v)$.

to *source*(e). In this situation it is OK if e stays unpaired and all other self-loops incident to *source*(e) are paired.

⟨*make_map.c*⟩≡

```
static int map_edge_ord1(const edge& e) { return index(source(e)); }
static int map_edge_ord2(const edge& e) { return index(target(e)); }
bool graph::make_map()
{
  int n = max_node_index();
  int count = 0;
  edge e,r;
  list<edge> EST;
  forall_edges(e,(*this)) if (e->rev == nil) EST.append(e);
  int number_of_undefined_reversals = EST.length();
  list<edge> ETS = EST; ETS.reverse();
  EST.bucket_sort(0,n,&map_edge_ord2); // secondary key
  EST.bucket_sort(0,n,&map_edge_ord1); // primary key
  ETS.bucket_sort(0,n,&map_edge_ord1); // secondary key
  ETS.bucket_sort(0,n,&map_edge_ord2); // primary key
  // merge EST and ETS to find corresponding edges
  while (! EST.empty() && ! ETS.empty())
  { e = EST.head();
    r = ETS.head();
    if ( e->rev != nil ) { EST.pop(); continue; }
    if ( r->rev != nil ) { ETS.pop(); continue; }
    if ( target(r) == source(e) )
    { if ( source(r) == target(e) )
      { ETS.pop(); EST.pop();
        if ( e != r )
        { e->rev = r; r->rev = e;
          count += 2;
        }
        continue;
      }
      else // target(r) == source(e) && source(r) != target(e)
      { if (index(source(r)) < index(target(e)))
          ETS.pop();  // r cannot be matched
        else
          EST.pop();  // e cannot be matched
      }
    }
    else // target(r) != source(e)
    { if (index(target(r)) < index(source(e)))
        ETS.pop();  // r cannot be matched
      else
        EST.pop();  // e cannot be matched
    }
   }
  return count == number_of_undefined_reversals;
}
```

Given the function above, it is trivial to extend a graph G to a map. A call *G.make_map*( ) determines the reversal information of a maximal number of edges. For any edge whose reversal information is still undefined, we add the reversed edge to G and set the reversal information accordingly.

$\langle$*make_map.c*$\rangle{+}{\equiv}$

```
  void graph::make_map(list<edge>& R)
  { if (make_map()) return;
    list<edge> el = all_edges();
    edge e;
    forall(e,el)
    { if (e->rev == nil)
      { edge r = new_edge(target(e),source(e));
        e->rev = r;
        r->rev = e;
        R.append(r);
      }
    }
  }
```

Exercises for 8.2

1 Does the function *check_reversal_inf* work if the map *is_edge_of_G* is replaced by an edge array?

2 Does the function *check_reversal_inf* work if the last two *forall_edges* loops are combined into one?

8.3 Embeddings

Embeddings are special drawings, namely drawings where no edge is drawn across a node, where the images of distinct edges do not cross, and where the two edges comprising a uedge are embedded the same. Formally, we define as follows:

A drawing I of a graph G into a surface S is called an *embedding* if the images of edges contain no images of points in their relative interiors[6], if the images of edges belonging to distinct uedges are disjoint except for endpoints[7], and if the curves assigned to edges belonging to the same uedge are reversals of each other[8].

Figure 8.1 shows three embeddings of a map M_0 into the plane; M_0 has nodes

[6] $I(e)(x) \neq I(v)$ for any edge e, node v, and real x with $0 < x < 1$

[7] $I(e)(x) \neq I(e')(y)$ for edges e and e' with $e \neq e'$ and $e' \neq \mathit{reversal}(e)$ and all x and y with $0 < x, y < 1$

[8] $I(e^R)(x) = I(e)(1-x)$ for all edges e, $e^R = \mathit{reversal}(e)$, and all x, $0 \leq x \leq 1$

v_1, v_2, v_3, and v_4 and uedges $\{v_1, v_2\}$, $\{v_1, v_3\}$, $\{v_1, v_4\}$, and $\{v_2, v_3\}$, and will be used as the running example in this chapter. An embedding into the plane is called a *planar embedding*, and a planar embedding in which every edge is mapped to a straight line segment is called a *straight line embedding.* A graph G is called *planar* if it has a planar embedding.

The function

```
bool Is_Planar(const graph& G)
```

tests whether the graph $G = (V, E)$ has a planar embedding. It returns *true* if G is planar and *false* otherwise. The running time is $O(n + m)$.

The functions

```
bool    PLANAR(graph& G, bool embed = false);
bool HT_PLANAR(graph& G, bool embed = false);
bool BL_PLANAR(graph& G, bool embed = false);
```

also test whether the graph G is planar. When *embed* is *true*, G is a map, and G is planar (the functions rise an error when *embed* is *true* and G is not a map), the functions in addition reorder the adjacency lists of G such that G becomes a plane map. The notion of plane map is explained in Section 8.4. All of this takes time $O(n + m)$.

There are two implementations of the planarity test and planar embedding algorithm: HT_PLANAR realizes the planarity testing algorithm of Hopcroft and Tarjan, see [HT74] or [Meh84, IV.10], and the embedding algorithm of Mehlhorn and Mutzel, see [MM95]. BL_PLANAR realizes the planarity testing algorithm of Lempel, Even, and Cederbaum, and Booth and Lueker, see [LEC67, Eve79, BL76], and the embedding algorithm of Nishizeki and Chiba, see [NC88]. The implementation of HT_PLANAR is documented in [MMN94] and the implementation of BL_PLANAR is discussed in Section 8.7. BL_PLANAR is the faster of our implementations and hence PLANAR is synonymous to BL_PLANAR.

The functions

```
bool    PLANAR(graph& G, list<edge>& el, bool embed = false);
bool HT_PLANAR(graph& G, list<edge>& el, bool embed = false);
bool BL_PLANAR(graph& G, list<edge>& el, bool embed = false);
```

behave like the functions above when G is planar. If G is non-planar, the functions also return a proof of non-planarity in the form of the edges *el* of a Kuratowski subgraph. The identification of Kuratowski subgraphs takes linear time $O(n + m)$ in BL_PLANAR and PLANAR, and takes quadratic time $O(n^2)$ in HT_PLANAR. We explain the notion of *Kuratowski subgraph.*

Figure 8.5 shows two non-planar graphs, the complete graph K_5 on five nodes and the complete bipartite graph $K_{3,3}$ with three nodes on each side. The non-planarity of both graphs will be shown in Lemma 3 in Section 8.6. It is a famous theorem of Kuratowski, see [Kur30, Whi73], that every non-planar graph G contains a subdivision[9] of either K_5 or $K_{3,3}$, i.e., there is a set *el* of edges in G forming a

[9] Let K be an arbitrary graph. A subdivision of K is obtained from K by subdividing edges. To subdivide an edge means to split the edge into two by placing a new vertex on the edge.

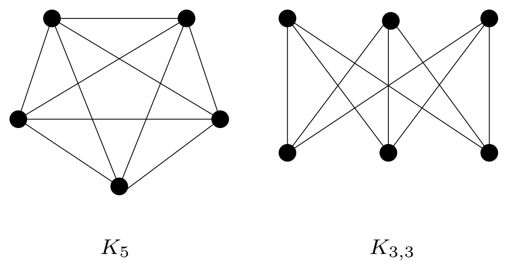

Figure 8.5 The Kuratowski graphs K_5 and $K_{3,3}$.

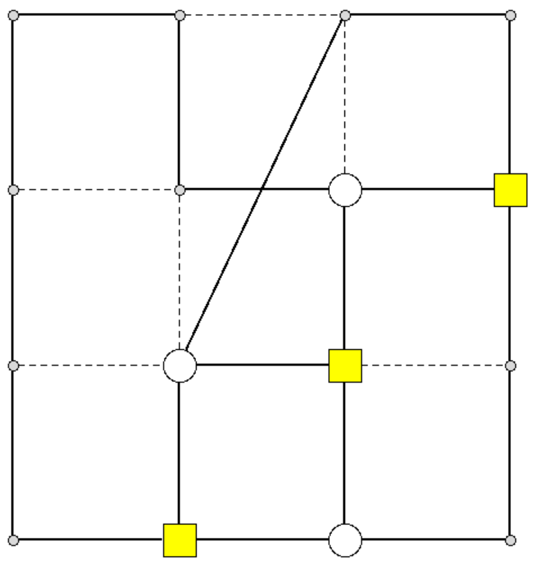

Figure 8.6 A non-planar graph and the Kuratowski subgraph proving non-planarity. The edges of the Kuratowski subgraph are shown in bold. This figure was generated with the xlman-demo gw_plan_demo.

subdivision of either K_5 or $K_{3,3}$. Figure 8.6 shows a Kuratowski subgraph of a non-planar graph.

There is also a function that gives more information about the Kuratowski subgraph than just the list of its edges.

```
int KURATOWSKI(graph& G, list<node>& V, list<edge>& E,
                     node_array<int>& deg);
```

returns zero if G is planar and returns one otherwise. If G is non-planar, it computes a Kuratowski subdivision K of G as follows: V is the list of all nodes and subdivision points of K. For all $v \in V$ which are subdivision points, the degree $deg[v]$ is equal to 2. If K is a K_5, then $deg[v]$ is equal to 4 for all nodes $v \in V$ that are not subdivision points. If K is a $K_{3,3}$, then $deg[v]$ is equal to -3 $(+3)$ for the nodes v on the left (right) side of the $K_{3,3}$.

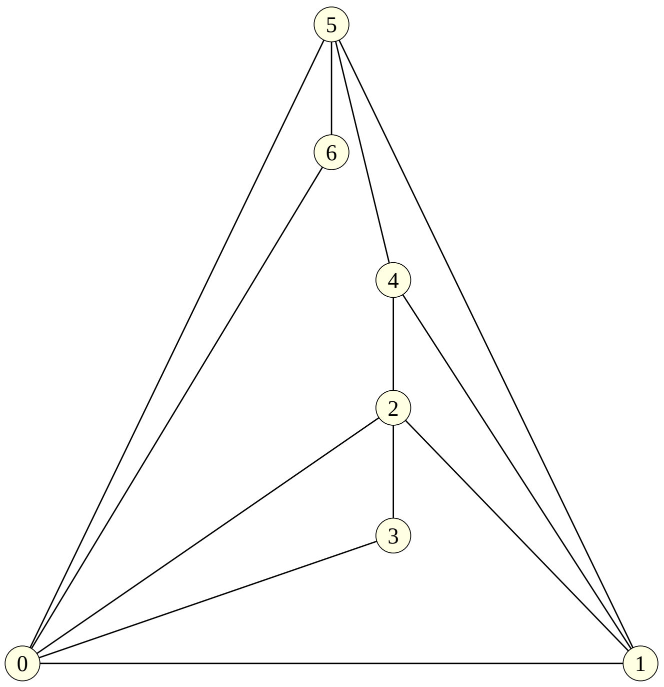

Figure 8.7 A straight line drawing produced by STRAIGHT_LINE_EMBEDDING. This figure was generated with the xlman-demo gw_plan_demo.

If G is a plane map, the function

```
int STRAIGHT_LINE_EMBEDDING(graph& G, node_array<int>& xcoord,
                                      node_array<int>& ycoord);
```

computes for each node v of G a point $(xcoord[v], ycoord[v])$ with integer coordinates in the range $[0\,..\,2(n-1)]$ such that the straight line embedding defined by these node positions is an order preserving embedding of G. The algorithm [Fár48, dFPP88] has running time $O(n^2)$. G must not have parallel edges and it must not have self-loops (since the existence of parallel edges or self-loops excludes the existence of a straight line embedding). Figure 8.7 shows a straight line drawing produced by this algorithm.

The function *Is_Planar* played an important role in the development of LEDA. We added the function to the system in 1991. The function had been implemented as part of a master's thesis and had been tested on a small number of examples (we did not have a large collection of planar graphs available to us). The master's thesis described the implementation; the actual program was not part of the thesis.

In 1993 we were sent a planar graph which, however, our program declared non-planar. When we started to revise the program we learned two things. First, we learned that writing a function

```
bool Is_Planar(const graph& G)
```

means asking for trouble. A function that answers a complex question like

Is G planar?

should not just return "YES" or "NO"; *it should justify its answer in a way that is easily checked by the caller of the function.*

Second, we learned that documentation and implementation had to be tied together more closely by the use of literate programming. Literate programming, first advocated by D.E. Knuth, suggests to embed an implementation into a document that describes the algorithm. All programs in this book are presented in a literate programming style. We first used CWEB [KL93] and later switched to noweb [Ram94].

In the case of planarity testing, the learning process led to reports [MMN94, MM95, HMN96] and to function

```
bool PLANAR(graph& G, list<edge>& el, bool embed)
```

which justifies its answers:

- When G is non-planar the function returns a proof of non-planarity in the form of the set *el* of edges of a Kuratowski subgraph. The caller can easily check that the edges in *el* form a Kuratowski subdivision of G.
- When G is planar, *embed* is set to *true*, and G is a map, the function reorders the adjacency lists of G such that G becomes a plane map. A caller of PLANAR has two ways to check whether the returned map is plane. He can either produce a planar drawing of G with the help of STRAIGHT_LINE_EMBEDDING and visually inspect the result, or he can compute the genus of G. The genus of maps will be discussed in Section 8.6 and it will be shown there that a map is plane iff its genus is zero. The genus of a map can be computed by a simple program.

The fact that PLANAR justifies its answers and that the answers are easily checked can be used to test the function on any input. Observe that testing is usually restricted to inputs where the answer is known by other means. The following test program exploits the fact that PLANAR can be tested on any input.

We choose integers n and m such that a random map with n nodes and m uedges has a fair chance of being planar and a fair chance of being non-planar, generate random maps with n nodes and about m edges, test them for planarity, and check the answer.

⟨*planar_test.c*⟩+≡

```
int main(){
int n = read_int("n = ");  int m = read_int("m = ");
graph G;
list<edge> el;
int P = 0; int K = 0;
while (P + K < 1000)
{ random_graph(G,n,m);
  list<edge> R;
  G.make_map(R);
  if ( PLANAR(G,el,true) )
    { assert(Genus(G) == 0); P++; }
  else
    { assert(CHECK_KURATOWSKI(G,el)); K++; }
}
cout << "\n\nnumber of plane graphs = " << P;
cout << "\n\nnumber of non-plane graphs = " << K; newline;
}
```

In a run with $n = 50$ and $m = 55$, the program above found 308 planar graphs and 692 non-planar graphs.

The function PLANAR was the first function in LEDA that justified its answers. By now, many functions do. We have seen many examples already in the preceding chapters and we will see more in the chapters to come. A general discussion of the role of program checking in LEDA can be found in Section **??**.

Exercises for 8.3

1 Let G be a non-planar graph. Show that the following strategy identifies the edges of a Kuratowski subgraph. Iterate over all edges e of G. If $G \setminus e$ is non-planar, remove e from G, and if $G \setminus e$ is planar leave G unchanged. The edges remaining in G form a Kuratowski subgraph.

2 Write a function

```
bool CHECK_KURATOWSKI(const graph& G, const list<edge>& el)
```

that returns *true* if the edges in *el* form a Kuratowski subdivision of G.

8.4 Order-Preserving Embeddings of Maps and Plane Maps

We define the notion of an order preserving embedding of a map.

For a vertex v, we use $A(v)$ to denote the set of edges with source v. The set $A(v)$ is stored as a cyclic list. For an edge e,

```
G.cyclic_adj_succ(e);
G.cyclic_adj_pred(e);
```

return the successor and predecessor of e, respectively, in the cyclic list $A(\mathit{source}(e))$.

We will, from now on, assume that the adjacency lists of the map M_0, our running example, are ordered as follows:

$$\begin{array}{ll} v_1: & e_1 = (v_1, v_2), e_2 = (v_1, v_4), e_3 = (v_1, v_3) \\ v_2: & e_4 = (v_2, v_3), e_1^R = (v_2, v_1) \\ v_3: & e_3^R = (v_3, v_1), e_4^R = (v_3, v_2) \\ v_4: & e_2^R = (v_4, v_1). \end{array}$$

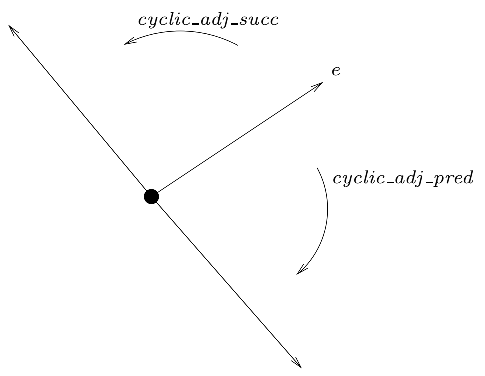

Figure 8.8 Order-preserving embeddings: The cyclic order of the edges in $A(v)$ agrees with the counter-clockwise ordering of the edges around v in the drawing.

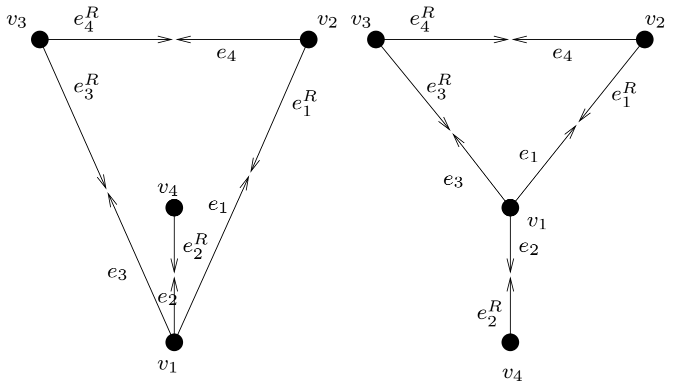

Figure 8.9 Two planar embeddings of the map M_0: In the embedding on the left the counter-clockwise ordering of the edges in $A(v_1)$ is e_1, e_2, e_3 and in the embedding on the right the ordering is e_1, e_3, e_2. The embedding on the left is order-preserving.

Consider a drawing of a map M into the plane (more generally, into any orientable surface) and let v be any node of M. The drawing defines a cyclic ordering on the edges $A(v)$ emanating from v, namely the counter-clockwise ordering[10] of the curves $I(e)$, $e \in A(v)$, around $I(v)$. A drawing is called *order-preserving* or *order-compatible* if for every node v the counter-clockwise ordering of the curves $I(e)$, $e \in A(v)$, around $I(v)$ agrees with the cyclic ordering of the edges in $A(v)$, see Figure 8.8. In Figure 8.9 one of the embeddings of M_0 is order-preserving and one is not. In all further drawings of maps in this chapter we will use order-preserving drawings.

A map is called *plane* if it has an order-preserving planar embedding. The function

```
bool Is_Plane_Map(const graph& G)
```

[10] A precise definition is as follows: for a positive real ϵ consider the first intersections of the curves $I(e)$, $e \in A(v)$, with the circle of radius ϵ around $I(v)$. For small enough ϵ this ordering does not depend on ϵ.

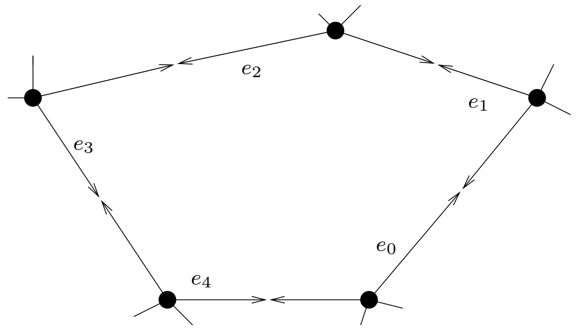

Figure 8.10 Face cycle successors and predecessors: We have $e_{i+1} = \mathit{face\_cycle\_succ}(e_i)$ for all i, $0 \le i < 5$. Indices are mod 5. The drawing convention for maps is used.

returns *true* if G is a plane map and returns *false* otherwise. We will see its implementation in Section 8.6.

8.5 The Face Cycles and the Genus of a Map

We define a partition of the edges of a map into cycles, the so-called *face cycles*. We introduce face cycles as purely combinatorial objects and will interpret them geometrically in the next section. Based on the concept of face cycles we will define the *genus* of a map.

For an edge e of a map M we define the face cycle successor and face cycle predecessor of e by:

```
  face_cycle_succ(e) = cyclic_adj_pred(reversal(e))
  face_cycle_pred(e) = reversal(cyclic_adj_succ(e)).
```

Figure 8.10 illustrates these definitions. The next lemma justifies the use of the names *succ* and *pred* and also shows that the function *face_cycle_succ* decomposes the edges of a map into cycles.

Lemma 1 *Let M be a map and let e be an edge of M. Then*

(a) $\mathit{face\_cycle\_pred}(\mathit{face\_cycle\_succ}(e)) = e$

(b) $\mathit{face\_cycle\_succ}(\mathit{face\_cycle\_pred}(e)) = e$

(c) *Let $e_0 = e$ and set $e_{i+1} = \mathit{face\_cycle\_succ}(e_i)$ for $i \ge 0$. Then there is a k such that $e_{k+1} = e_0$ and $e_i \ne e_j$ for all i and j with $0 \le i < j \le k$.*

Proof (a) and (b) We have

$$\begin{aligned}
&\mathit{face\_cycle\_pred}(\mathit{face\_cycle\_succ}(e))\\
&\quad= \mathit{reversal}(\mathit{cyclic\_adj\_succ}(\mathit{cyclic\_adj\_pred}(\mathit{reversal}(e))))\\
&\quad= \mathit{reversal}(\mathit{reversal}(e))
\end{aligned}$$

$$= \quad e$$

and

$$\begin{aligned} & \mathit{face\_cycle\_succ}(\mathit{face\_cycle\_pred}(e)) \\ = \quad & \mathit{cyclic\_adj\_pred}(\mathit{reversal}(\mathit{reversal}(\mathit{cyclic\_adj\_succ}(e)))) \\ = \quad & \mathit{cyclic\_adj\_pred}(\mathit{cyclic\_adj\_succ}(e)) \\ = \quad & e \end{aligned}$$

(c) Let k be minimal such that $e_{k+1} = e_i$ for some $i \le k$. Assume $i > 0$. From $e_{k+1} = \mathit{face\_cycle\_succ}(e_k)$ and $e_i = \mathit{face\_cycle\_succ}(e_{i-1})$ and part (a) we conclude $e_k = e_{i-1}$, a contradiction to the definition of k. Thus $i = 0$. □

For an edge e of a map M we define the *face cycle* containing e as the cycle $[e_0, e_1, \ldots, e_k]$ where $e_0 = e$, $e_{i+1} = \mathit{face\_cycle\_succ}(e_i)$ for $i \ge 0$, $e_{k+1} = e$, and $e_j \ne e_i$ for $0 \le i < j \le k$. Part (c) of the lemma above guarantees that this is a good definition. Every edge of M belongs to exactly one face cycle and the face cycles partition the edges of M.

We illustrate the concept of face cycle on our running example, the map M_0. The face cycle containing the edge $e_1 = (v_1, v_2)$ is

$$[e_1, e_4, e_3^R, e_2, e_2^R],$$

and the face cycle containing the edge $e_1^R = (v_2, v_1)$ is

$$[e_1^R, e_3, e_4^R].$$

Let us verify that this is indeed the case. We have

$$\mathit{face\_cycle\_succ}(e_1^R) = \mathit{cyclic\_adj\_pred}(\mathit{reversal}(e_1^R)) = \mathit{cyclic\_adj\_pred}(e_1) = e_3,$$

$$\mathit{face\_cycle\_succ}(e_3) = \mathit{cyclic\_adj\_pred}(\mathit{reversal}(e_3)) = \mathit{cyclic\_adj\_pred}(e_3^R) = e_4^R,$$

and

$$\mathit{face\_cycle\_succ}(e_4^R) = \mathit{cyclic\_adj\_pred}(\mathit{reversal}(e_4^R)) = \mathit{cyclic\_adj\_pred}(e_4) = e_1^R.$$

We want to stress that the concept of face cycles is purely combinatorial. It is made without any reference to a drawing of a map. A geometric interpretation is given in the next section.

We close this section with the definition of the *genus* of a map. Let M be a map with m edges, c connected components, n nodes, nz isolated nodes, and fc face cycles. Then

$$\mathit{genus}(M) = (m/2 + 2c - n - nz - fc)/2.$$

The genus of a map is always a non-negative integer, as we will show in the next section, and characterizes the surfaces into which a map can be embedded. For the map M_0 we have $m = 8$, $c = 1$, $n = 4$, $nz = 0$, and $f = 2$, and hence

$genus(M_0) = 0$. We will see in the next section that this implies that M_0 is a plane map.

The following program computes the genus of a map. We determine the number of nodes and edges and the number of isolated nodes in the obvious way, and we call *COMPONENTS* to determine the number of connected components. We determine the number of face cycles by tracing them one by one. We iterate over all edges e of G. If the face cycle of e has not been traced yet, we trace it and mark all edges on the cycle as considered.

$\langle genus.c \rangle \equiv$

```
int Genus(const graph& G)
{ if ( !Is_Map(G) ) error_handler(1,"Genus only applies to maps");
  int n = G.number_of_nodes();
  if ( n == 0 ) return 0;
  int nz = 0;
  node v;
  forall_nodes(v,G) if ( outdeg(v) == 0 )  nz++;
  int m = G.number_of_edges();
  node_array<int> cnum(G);
  int c = COMPONENTS(G,cnum);
  edge_array<bool> considered(G,false);
  int fc = 0;
  edge e;
  forall_edges(e,G)
  { if ( !considered[e] )
    { // trace the face to the left of e
      edge e1 = e;
      do { considered[e1] = true;
           e1 = G.face_cycle_succ(e1);
         }
      while (e1 != e);
      fc++;
    }
  }
  return (m/2 - n - nz - fc + 2*c)/2;
}
```

8.6 Faces, Face Cycles, and the Genus of Plane Maps

The purpose of this section is to relate combinatorics and geometry. We will define the faces of an embedding and relate it to the face cycles of a map. We will prove that a map is plane if and only if its genus is zero. We will also show that K_5 and $K_{3,3}$ are non-planar graphs.

Consider a map M and an embedding I of M into an orientable surface S. The removal of the embedding from S leaves us with a family of open connected subsets of S, called the *faces of the embedding*. In an embedding into the plane exactly one of the faces is unbounded and all other faces are bounded. The unbounded face is also called the *outer face*. We associate a set of edges with each face F, the boundary of F. An edge e belongs to the boundary of F if the "left side" of $I(e)$ is contained in F, formally, if for every point p in the relative interior of the embedding $I(e)$ of e and every sufficiently small disk centered at p, the part of the disk lying to the left of $I(e)$ is contained in F.

Consider the embeddings of M_0 shown in Figure 8.9. In the embedding on the left, the boundary of the unbounded face consists of the edges e_1^R, e_3, and e_4^R, and the boundary of the bounded face consists of the edges e_1, e_4, e_3^R, e_2, and e_2^R. In the embedding on the right, the boundary of the unbounded face consists of the edges e_1^R, e_2, e_2^R, e_3, and e_4^R, and the boundary of the bounded face consists of the edges e_1, e_4, and e_3^R. In the embedding on the left the face boundaries correspond to the face cycles of M_0.

The boundary of a face consists of one or more cycles[11], which we call *boundary cycles*. In the case of an order-preserving embedding boundary cycles and face cycles are the same.

Lemma 2 *Let I be an order-preserving embedding of a map M. The boundary cycles of the faces of I are in one-to-one correspondence to the face cycles of M.*

Proof Let $e = (v, w)$ be any edge of M and consider the boundary cycle C containing $I(e)$. Let $g = (w, z)$ be the edge such that $I(g)$ follows $I(e)$ in C. Then $I(g)$ follows $I(\mathit{reversal}(e))$ in the clockwise ordering of the embedded edges around $I(v)$. Since I is an order-preserving embedding we have $g = \mathit{face\_cycle\_pred}(e)$. Thus, boundary cycles and face cycles are the same. □

The next theorem shows that the genus of a map gives a combinatorial condition whether a map is plane. It is more generally true, see [Whi73], that the genus of a map M characterizes the oriented surfaces into which M can be embedded in an order-preserving way. The following theorem is due to Euler [Eul53] and Poincaré [Poi93].

Theorem 1 *Let M be any map. Then $\mathit{genus}(M) \geq 0$. Moreover, M is a plane map iff $\mathit{genus}(M) = 0$.*

Proof We observe first that it suffices to prove the claims for a connected map M. Let $M_1, \ldots, M_c$ be the connected components of M. Then[12] $m = \sum m_i$, $n = \sum n_i$, $nz = \sum nz_i$, $fc = \sum fc_i$, and $c = \sum c_i$ and hence

$$\mathit{genus}(M) = \sum \mathit{genus}(M_i).$$

[11] In a connected graph the boundary of each face consists of exactly one cycle.
[12] We use m_i to denote the number of edges in M_i and analogously for n_i, nz_i, fc_i, and c_i.

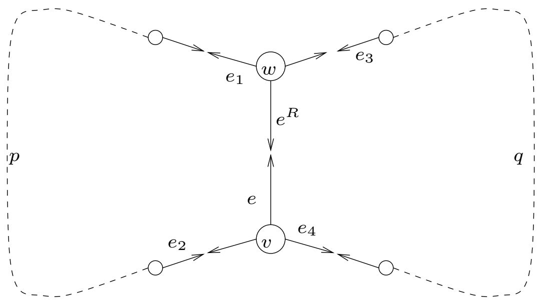

Figure 8.11 The edges e and e^R belong to distinct face cycles $e \circ p$ and $e^R \circ q$. Removal of e and e^R leaves us with a connected graph since p and q provide alternative connections between v and w. Let $e_1 = \mathit{face\_cycle\_succ}(e)$, $e_2 = \mathit{face\_cycle\_pred}(e)$, $e_3 = \mathit{face\_cycle\_pred}(e^R)$, and $e_4 = \mathit{face\_cycle\_succ}(e^R)$. Removal of e and e^R makes e_1 the face cycle successor of e_3, and e_4 the face cycle successor of e_2. No other successor relationship is affected. We conclude that the removal of e and e^R generates the face cycle $p \circ q$ and affects no other face cycles. Thus, $\mathit{fc}' = \mathit{fc} - 1$.

Let us assume for the moment that the claims hold for connected maps, i.e., we have $\mathit{genus}(M_i) \geq 0$ and M_i is plane iff $\mathit{genus}(M_i) = 0$ for all i. We conclude $\mathit{genus}(M) \geq 0$. If M is plane then all M_i's are plane. Thus, $\mathit{genus}(M_i) = 0$ for all i and hence $\mathit{genus}(M) = 0$. Conversely, $\mathit{genus}(M) = 0$ implies $\mathit{genus}(M_i) = 0$ for all i (since $\mathit{genus}(M_i) > 0$ for some i would imply $\mathit{genus}(M_j) < 0$ for some j). Thus, M_i is plane for all i and hence M is plane.

For connected maps we use induction on the number of edges. If $m = 0$ then $n = \mathit{nz} = 1$ and $\mathit{fc} = 0$. Thus, M is plane and $\mathit{genus}(M) = 0$. We turn to the induction step.

Assume first that M contains a uedge $\{e, e^R\}$ such that e and e^R belong to different face cycles. Removal of e and e^R generates a map M' with $m' = m - 2$, $n' = n$, $c' = c = 1$, $\mathit{nz}' = \mathit{nz} = 0$, and $\mathit{fc}' = \mathit{fc} - 1$, see Figure 8.11. Thus, $\mathit{genus}(M) = \mathit{genus}(M')$. By induction hypothesis, $\mathit{genus}(M') \geq 0$ and M' is plane iff $\mathit{genus}(M') = 0$. From $\mathit{genus}(M') \geq 0$ we conclude $\mathit{genus}(M) \geq 0$. We next show that M is plane iff $\mathit{genus}(M) = 0$. If M is plane then M' is plane (since an order-preserving embedding of M' is obtained from an order-preserving embedding of M by removing the images of e and e^R). Thus $\mathit{genus}(M') = 0$ by induction hypothesis and hence $\mathit{genus}(M) = 0$. Conversely, if $\mathit{genus}(M) = 0$ then $\mathit{genus}(M') = 0$ and hence there is an order-preserving embedding I' of M', by induction hypothesis. By Lemma 2 there is a face F in the embedding I' with boundary cycle $p \circ q$. We embed e and e^R into F and obtain an order-preserving embedding I of M.

Assume next that for every uedge $\{e, e^R\}$ of M, e and e^R belong to the same face cycle. Consider any node v and let $A(v) = (e_0, e_1, \ldots, e_{k-1})$ be the cyclic list of edges out of v. Then

$$e_i = \mathit{face\_cycle\_succ}(e_{i+1}^R)$$

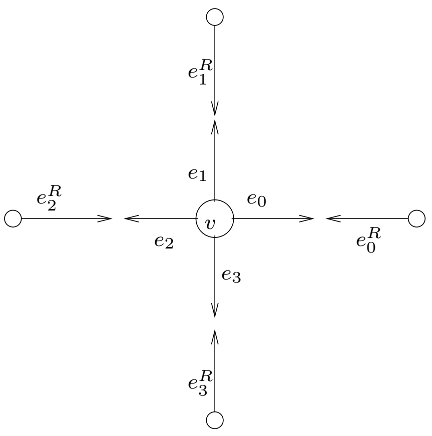

Figure 8.12 A node v with $A(v) = (e_0, e_1, e_2, e_3)$. There is a face cycle containing e_{i+1}^R and e_i for all i, $0 \le i < 4$. Indices are modulo 4.

for all i, $0 \le i < k$, by the definition of face cycles, see Figure 8.12. Since e_i and e_i^R belong to the same face cycle by assumption, all edges incident to v belong to the same face cycle and, since M is connected, all edges of M belong to the same face cycle. Thus, $fc = 1$. Since M is connected, the number of uedges is at least $n-1$. Thus, $m \ge 2(n-1)$, $c = 1$, $nz = 0$, and hence $genus(M) \ge 0$. We next show that M is plane iff $genus(M) = 0$. If M is plane consider an order-preserving embedding I of M. The face cycles of M are in one-to-one correspondence to the faces of the embedding. Since there is only one face cycle, there is only one face, and hence M cannot contain a cycle. Thus, $m = 2(n-1)$ and hence $genus(M) = 0$. Conversely, if $genus(M) = 0$ then $(m/2 + 2 - n - 1) = 0$ and hence $m = 2(n-1)$. The number of uedges is therefore equal to $n-1$ and hence the uedges form a tree. For a tree there is clearly an order-preserving embedding. □

The theorem above implies that the test of whether a graph G is a plane map is trivial to implement. We only have to test whether G is a map and whether the genus of G is zero.

```
bool Is_Plane_Map(const graph& G) { return Is_Map(G) && Genus(G) == 0; }
```

We draw some more consequences of Theorem 1. It implies an upper bound on the number of edges in a planar graph (without self-loops and parallel edges) and it implies that the Kuratowski graphs K_5 and $K_{3,3}$ are non-planar.

Lemma 3

(a) *Let M be a connected plane map in which every face cycle consists of at least d edges, where $d \ge 3$. Then*

$$m/2 \le \frac{d}{d-2}(n-2),$$

i.e., M has at most $(d/(d-2)) \cdot (n-2)$ uedges.

(b) *Let M be a connected planar map without self-loops and without parallel edges. Then M has at most $3n-6$ uedges, if $n > 3$, and a node of degree at most five.*

(c) *Let M be a connected bipartite planar map without self-loops and without parallel edges. Then M has at most $2n-4$ uedges, if $n \geq 4$.*

(d) *The complete graph K_5 on five nodes is not planar.*

(e) *The complete bipartite graph $K_{3,3}$ with three nodes on each side is not planar.*

Proof (a) If every face cycle consists of at least d edges then $m \geq d \cdot fc$. Thus,

$$0 = genus(M) = m/2 + 2 - n - fc \geq m/2 + 2 - n - m/d$$

and hence $(m/2) \cdot (1 - 2/d) \leq n - 2$ or $m/2 \leq (d/(d-2)) \cdot (n-2)$.

(b) and (c) Reorder the adjacency lists of M such that M becomes a plane map. If M has no self-loops and no parallel edges, every face cycle of M consists of at least three edges. If, in addition, M is bipartite, every face cycle of M consists of at least four edges. The bounds on the number of edges now follow from part (a). If every node would have degree six or more, the total number of edges would be at least $6n/2 = 3n$.

(d) A planar graph with five nodes and no self-loops and no parallel edges has at most nine uedges by part (b). The graph K_5 has $5 \cdot 4/2 = 10$ uedges.

e) A planar bipartite graph with six nodes and no self-loops and no parallel edges has at most eight uedges by part (c). The graph $K_{3,3}$ has $3 \cdot 3 = 9$ uedges. □

Exercise for 8.6

1 It is obvious from the definition of $genus(M)$ that $2 \cdot genus(M)$ is an integer. The purpose of this exercise is to show that $genus(M)$ is an integer. In the proof of Theorem 1 we have constructed for every connected map M a connected map M' such that $genus(M) = genus(M')$ and such that M' has a single face cycle. Let M'' be obtained from M' by removing an edge e and its reversal e^R. Determine the number of edges, nodes, face cycles, and connected components of M'' and conclude that $genus(M') - genus(M'')$ is an integer. Use this observation and induction to show that the genus of every map is an integer.

8.7 Planarity Testing, Planar Embeddings, and Kuratowski Subgraphs

This section is joint work with D. Ambras, R. Hesse, Christoph Hundack, and E. Kalliwoda.

We give the details of the planarity test, the planar embedding algorithm, and the algorithm for finding Kuratowski subgraphs. For each algorithm we will first derive the required theory and then give an implementation. All implementations run in linear time and are collected in the file

⟨*_bl_planar.c*⟩≡

```
#include <LEDA/graph_alg.h>
#include <LEDA/pq_tree.h>
#include <LEDA/array.h>
#include <assert.h>
⟨auxiliary functions⟩
⟨planarity test⟩
⟨planar embedding of biconnected maps⟩
⟨planar embedding of arbitrary maps⟩
⟨Kuratowski graphs in biconnected maps⟩
⟨Kuratowski graphs in arbitrary graphs⟩
```

8.7.1 *The Lempel–Even–Cederbaum Planarity Test*

We discuss the planarity testing algorithm invented by Lempel, Even, and Cederbaum [LEC67, Eve79]. We assume that $G = (V, E)$ is a biconnected graph[13], that $e_0 = (s, t)$ is an arbitrary edge of G, and that the nodes of G are st-numbered, i.e., s is numbered 1, t is numbered n, and every node distinct from s and t has a lower and a higher numbered neighbor.

We will first discuss the required theory and then describe an implementation based on PQ-trees.

The Theory: We identify nodes with their st-number, i.e., $V = \{1, \ldots, n\}$. Figure 8.13 shows an example of an st-numbered biconnected graph. We will use it as our running example.

Let $V_k = \{1, \ldots, k\}$ and let $G_k = (V_k, E_k)$ be the graph induced by V_k, i.e., E_k consists of all edges of G whose endpoints are both in V_k. We extend G_k to a graph B_k. For each edge (v, w) of G with $v \le k$ and $w \ge k + 1$ there is a node and an edge in B_k. They are called virtual nodes and virtual edges, respectively. We label every virtual node with its counterpart in G. Figure 8.14 shows the graph B_7 for our running example.

If G is planar, B_k has a plane embedding which resembles a bush: node v, $1 \le v \le k$, is drawn at height v, all virtual nodes are put on a horizontal line at height $k + 1$, and all edges are drawn as y-monotone curves[14]. We call such an embedding a *bush form* for B_k and we call the horizontal line at height $k + 1$ the horizon. The existence of bush forms will follow from the discussion to come. Figures 8.15 and 8.16 shows two bush forms for the graph of Figure 8.14.

The *leaf word* of a bush form is a sequence in $\{N, E\}^*$, where E represents a

[13] The rather trivial extension to arbitrary graphs will be given at the end of the section.
[14] A curve is y-monotone if any horizontal line intersects the curve at most once.

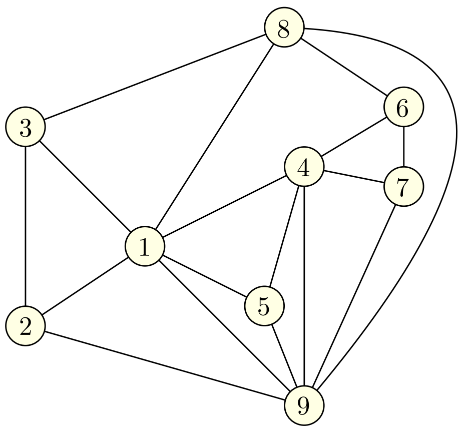

Figure 8.13 A biconnected *st*-numbered graph G. Node s is labeled 1 and node t is labeled 9.

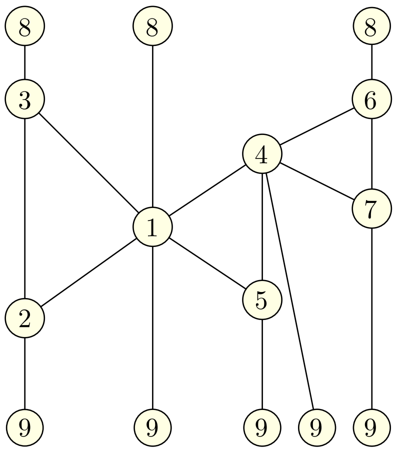

Figure 8.14 The graph B_7 for the graph G of Figure 8.13. There are three virtual nodes labeled 8, one for each edge connecting node 8 to a node labeled 7 or less in G, and there are five virtual nodes labeled 9, one for each edge connecting node 9 to a node labeled 7 or less in G. The nodes 4, 6, and 7 comprise a biconnected component which we denote H_0 for later reference.

virtual node labeled $k+1$, N represents a virtual node labeled $k+2$ or larger, and the virtual nodes are listed in their left-to-right order on the horizon. The bush form in Figure 8.15 has leaf word $ENNNENEN$ and the bush form in Figure 8.16 has leaf word $NEEENNNN$. A bush form for B_k is called *extendible* if all virtual nodes labeled $k+1$ are consecutive on the horizon, i.e., if its leaf word is in $N^*E^*N^*$. An extendible bush form $\hat{B}_k$ is readily extended to a bush form $\hat{B}_{k+1}$ for B_{k+1}. We move all nodes v, $v > k+1$, to height $k+2$, we merge all virtual nodes labeled $k+1$ into a single node (since they are consecutive on the horizon,

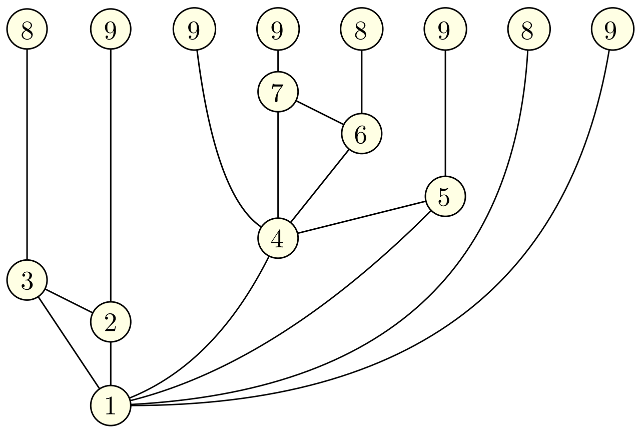

Figure 8.15 A bush form for the graph B_7 of Figure 8.14.

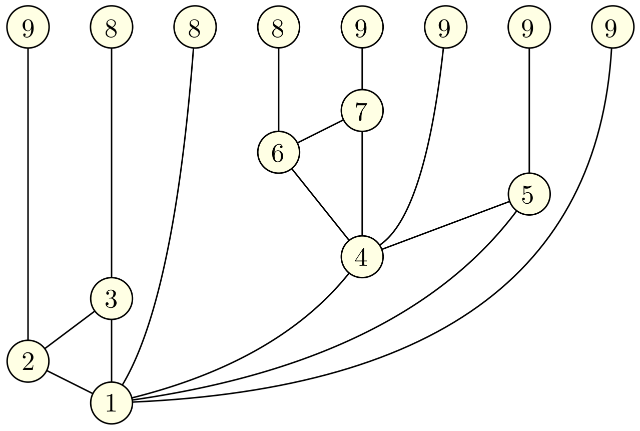

Figure 8.16 An extendible bush form for B_7.

merging does not destroy planarity), and add a new virtual edge and node for each edge $(k+1, w)$ with $w > k+1$.

The question is now how to decide whether B_k has an extendible bush form, and how to find an extendible bush form. We show:

Theorem 2 *B_{k+1} has a bush form iff B_k has a bush form and no obstructions. Moreover, if B_k has no obstructions then any bush form $\hat{B}_k$ of B_k can be transformed into an extendible bush form of B_k by a sequence of permutations and flippings.*

We still need to define several of the terms used in the theorem above. An obstruction is either an obstructing articulation point or an obstructing biconnected component. In the definition of either kind of obstruction we need the concepts of clean, mixed, or full subgraph. A subgraph of B_k is called *clean*, *mixed*, or *full* if none, some but not all, or all of its virtual nodes are labeled $k+1$.

An articulation point v of B_k is *obstructing* if there are three or more components of $B_k \setminus v$ that are mixed.

Consider the graph B_7 of Figure 8.14. Node 4 is an articulation point and $B_7 \setminus 4$ has three components: Two of them are mixed and one is full. Node 4 is non-obstructing. Please convince yourself that none of the articulation points is obstructing.

We come to biconnected components of B_k. A node y of a biconnected component H is called an *attachment* node of H if it is also the endpoint of an edge outside H. Attachment nodes are articulation points of B_k and hence are embedded on the boundary of the outside face in every bush form of B_k. In the graph B_7 the biconnected component H_0 has attachment nodes 4, 6, and 7.

Let $y_0, y_1, \ldots, y_{p-1}$ be the attachment nodes of a biconnected component H of B_k. We use y_0 for the lowest numbered attachment node; y_0 is also the lowest numbered node in H. Any bush form $\hat{B}_k$ of B_k induces an embedding of H (simply remove all nodes outside H and their incident edges). In this embedding of H the boundary of the outside face of H is a simple cycle, which we call the *boundary cycle*[15] of H in $\hat{B}_k$. A counter-clockwise traversal of the boundary cycle yields a cyclic order on the attachment nodes, which we call the cyclic order induced by the bush form. Consider Figures 8.15 and 8.16. The cyclic order of the attachment nodes 4, 6, and 7 is 4, 6, 7 in the first figure and is 4, 7, 6 in the second figure.

Lemma 4 *Let $y_0, y_1, \ldots, y_{p-1}$ be the attachment nodes of a biconnected component H of B_k in the cyclic order induced by some bush form $\hat{B}_k$ of B_k. Then any other bush form of B_k induces either the same cyclic order or its reversal.*

Proof Assume otherwise, i.e., there is a bush form $\hat{B}'_k$ such that the attachment nodes appear in a different cyclic order in $\hat{B}'_k$. Then there must be indices h, i, and j such that y_h and y_{h+1} (indices are $\bmod p$) are separated by y_i and y_j in the boundary cycle of H in $\hat{B}'_k$, see Figure 8.17. The embedding $\hat{B}'_k$ implies that any pair of paths connecting y_h to y_{h+1} and y_i to y_j, respectively, must cross. On the other hand, the embedding $\hat{B}_k$ implies the existence of non-crossing paths. □

Let $y_0, y_1, \ldots, y_{p-1}$ be the attachments of H in one of their cyclic orders[16]. The *component of B_k opposite to H at y_i* is the subgraph of B_k spanned by all nodes that are reachable from y_i without using an edge of H. We denote it by C_i. Each

[15] A node of H which is not an attachment node of H may lie on the boundary cycle of H in some bush forms and may not lie on the boundary cycle in others. Attachment nodes belong to the boundary cycle in every bush form.

[16] There are two by the preceding lemma. For the definition in this paragraph it does not matter which one is chosen.

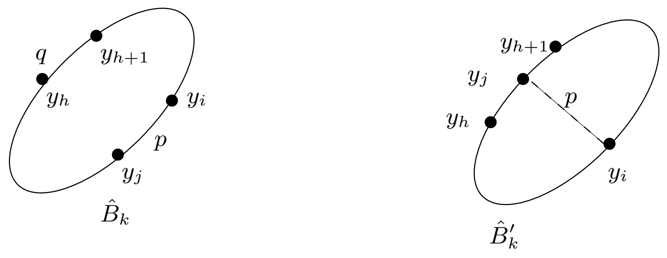

Figure 8.17 y_h and y_{h+1} are adjacent attachment nodes on the boundary cycle of H in $\hat{B}_k$, but are separated by y_i and y_j in the boundary cycle of H in $\hat{B}'_k$.

C_i is either clean, mixed, or full. We define the signature of H as the word

$$s_0 s_1 \dots s_{p-1} \in \{\text{clean,mixed,full}\}^*$$

where s_i describes the status of C_i. In the graph B_7, the component opposite to H_0 at 6 is full, the component opposite to H_0 at 7 is clean, and the component opposite to H_0 at 4 is mixed. The signature of H_0 is "mixed clean full" for the ordering 4, 7, 6 and "mixed full clean" for the ordering 4, 6, 7.

A biconnected component H is *non-obstructing* iff a cyclic shift of its signature is in

$$\text{clean}^* \ \text{mixed}_0^1 \ \text{full}^* \ \text{mixed}_0^1 \ \text{clean}^*,$$

where mixed_0^1 denotes zero or one occurrence of mixed, and is obstructing otherwise.

We come to permutations and flippings. Permutations apply to articulation points of B_k. Let v be an articulation point of B_k. Then, if $v > 1$, exactly one component of B_k with respect to v contains nodes lower than v, and if $v = 1$, no component does[17]. We call the component containing lower numbered nodes the *root component* of v and all other components *non-root components* of v.

In the graph B_7 of Figure 8.14 the root component of node 4 contains nodes 5, 1, 2, 3, two copies of 8, and three copies of 9.

Consider now any bush form $\hat{B}_k$ of B_k. A *sub-bush* of $\hat{B}_k$ with lowest numbered node v is the restriction of $\hat{B}_k$ to the union of some non-root components with respect to v. In particular, each non-root component of v corresponds to a sub-bush of $\hat{B}_k$. A *permutation operation* permutes the sub-bushes corresponding to the non-root components with respect to an articulation point v and a *flipping operation* flips over a sub-bush, see Figure 8.18.

We are now ready for the if-direction of Theorem 2.

Lemma 5 *If B_k has a bush form and no obstructions then any bush form $\hat{B}_k$ can be transformed into an extendible bush form by a sequence of permutations and flippings.*

[17] Observe that any node u with $u < v$ can reach 1 without passing through v by the virtue of st-numberings.

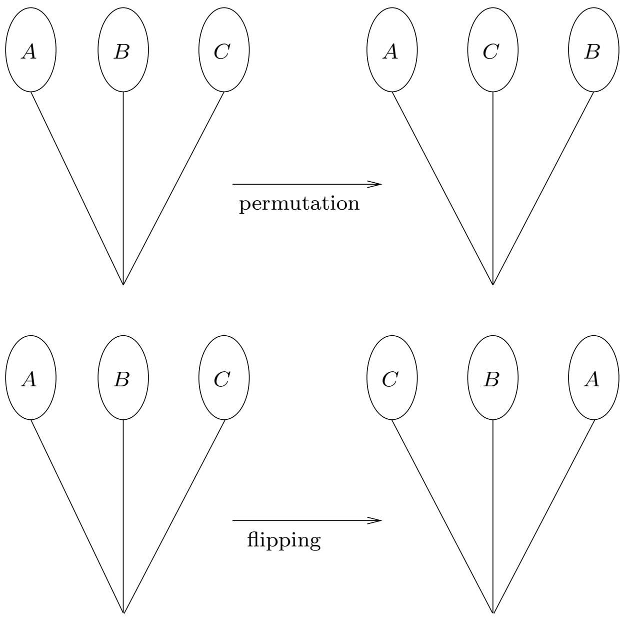

Figure 8.18 Permuting and flipping.

Proof We want to use induction over sub-bushes and therefore prove a slightly stronger claim. We call a sub-bush *incomplete* if there is a virtual node labeled $k+1$ outside the sub-bush and we call a sub-bush *strongly extendible* if its leaf word is in N^*E^* or E^*N^*. We show that every sub-bush can be transformed into an extendible sub-bush, i.e., a sub-bush whose leaf word is in $N^*E^*N^*$, and that every incomplete sub-bush can be transformed into a strongly extendible sub-bush.

Let $\hat{B}$ be any sub-bush. If $\hat{B}$ has only one virtual node, the claims are obvious. So, assume otherwise and let v be the lowest numbered node in $\hat{B}$. We distinguish cases according to whether v is an articulation point of $\hat{B}$ or not.

If v is an articulation point of $\hat{B}$ then at most two of the components of $\hat{B}$ with respect to v are mixed. We can therefore permute the components such that all full and all clean components are consecutive and such that the two mixed components bracket the full components, see Figure 8.19. We apply the induction hypothesis to the sub-bushes and therefore may assume that the sub-bushes are extendible or even strongly extendible (for incomplete sub-bushes). We complete the induction step with two observations. First, the mixed sub-bushes are incomplete except if there is at most one mixed sub-bush and this sub-bush contains all virtual nodes labeled $k+1$. Second, if $\hat{B}$ is incomplete then there is at most one mixed sub-bush since the root component of B_k with respect to v is mixed. Thus, $\hat{B}$ can be transformed into an extendible bush form and into a strongly extendible bush form if $\hat{B}$ is incomplete. The transformation consists of transformations of the

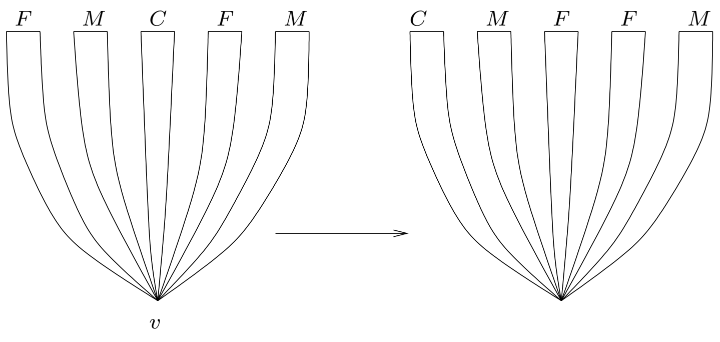

Figure 8.19 Permuting the sub-bushes of $\hat{B}$ with respect to v. C, M, and F stand for clean, mixed, and full sub-bushes, respectively.

sub-bushes, permuting the sub-bushes, and maybe flipping one of the mixed sub-bushes.

If v is not an articulation point of $\hat{B}$, let H be the biconnected component of $\hat{B}$ containing v. Let $y_0, y_1, \ldots, y_{p-1}$ with $v = y_0$ be the attachment points of H in B_k in one of their two cyclic orders. We have a sub-bush $\hat{B}_i$ of $\hat{B}$ for the component C_i of B_k opposite to y_i for all i, $1 \le i \le p-1$. Since H is non-obstructing and since C_0 is either clean or mixed (it cannot be full since it contains the edge (s,t)), we have

$$s_1 \ldots s_{p-1} \in \text{ clean}^* \text{ mixed}_0^1 \text{ full}^* \text{ mixed}_0^1 \text{ clean}^*$$

if C_0 is clean and we have

$$s_1 \ldots s_{p-1} \in \text{ clean}^* \text{ mixed}_0^1 \text{ full}^* \cup \text{ full}^* \text{ mixed}_0^1 \text{ clean}^*$$

if C_0 is mixed. In either case we conclude that $\hat{B}$ can be transformed into an extendible bush form and into a strongly extendible bush form if $\hat{B}$ is incomplete and hence C_0 is mixed. The transformation consists of transformations of sub-bushes followed (maybe) by a flipping of the two mixed sub-bushes. $\square$

Figure 8.20 illustrates Lemma 5. It shows a sequence of transformations that transform the bush form of Figure 8.15 into the extendible bush form of Figure 8.16.

We summarize. The Lempel–Even–Cederbaum planarity test constructs a sequence $\hat{B}_0$, $\hat{B}_1$, $\hat{B}_2$, ..., $\hat{B}_n$ of bush forms. In iteration $k+1$ the bush form $\hat{B}_k$ is first transformed into an extendible bush form $\hat{B}'_k$ and then extended to a bush form $\hat{B}_{k+1}$. The transformation to an extendible bush form uses permutations and flippings and is possible if $\hat{B}_k$ contains no obstructions.

The running time of the Lempel–Even–Cederbaum test is $O(n^2)$ in its original form. Booth and Lueker improved the running time to $O(n+m)$ by the introduction of the PQ-tree data structure, which we will discuss in the next section. In

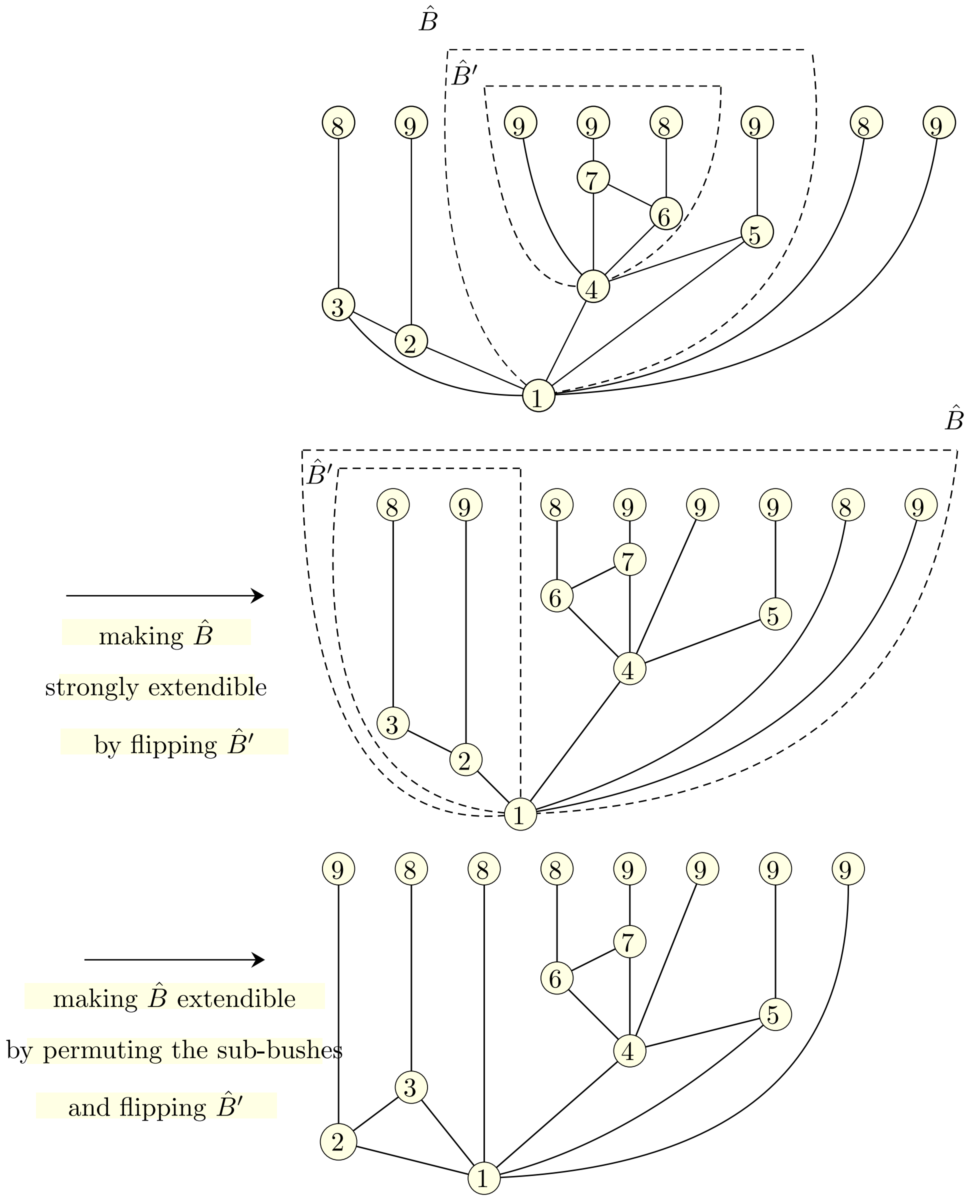
$\hat{B}$
$\hat{B}'$
$\hat{B}$
$\hat{B}'$
making $\hat{B}$ strongly extendible by flipping $\hat{B}'$
making $\hat{B}$ extendible by permuting the sub-bushes and flipping $\hat{B}'$
1
2
3
4
5
6
7
8
9

Figure 8.20 Transforming the bush form of Figure 8.15 into an extendible bush form.

Section 8.7.3 we will show that the existence of an obstruction implies the existence of a Kuratowski graph in G.

The PQ-Tree Data Structure: Booth and Lueker [BL76] introduced the PQ-data structure to keep track of the sequence of bush forms arising in the Lempel–

Even–Cederbaum planarity test. PQ-trees have wider applications than planarity testing but we will not discuss them here.

PQ-trees have the following interface.

```
pq_tree T(m);
```

declares a PQ-tree T which can represent bush forms in which every edge is labeled with an integer in $[1 \mathinner{..} m]$. After the declaration T represents the empty bush form with no nodes and no edges. We use S to denote the set of virtual edges in the current bush form. S is empty initially.

The operation

```
bool T.replace(list<int>& L, list<int>& U, list<int>& I)
```

adds a node to the current bush form. The node is incident to the virtual edges L in the current bush form and introduces new virtual edges U. We must have $L \subseteq S$, U is a set of integers (= edges) that have never been in S before, and $L = \emptyset$ iff $S = \emptyset$; the latter requirement corresponds to the fact that only node 1 is incident to no edge from below. The new set of virtual edges becomes $(S \setminus L) \cup U$.

The function returns *true* if the current bush form is extendible, i.e., can be transformed to a bush form in which all edges in L are contiguous on the horizon. The function returns *false* otherwise. Once a call of *replace* has returned *false*, the PQ-tree becomes non-functional and no further operations can be applied to it.

The last argument I is irrelevant for the planarity test and is only required for the construction of a planar embedding. We will discuss it in the next section.

The amortized running time of *replace* is proportional to the length of L plus the length of U and the running time of the declaration $T(m)$ is $O(m)$.

We are now ready for the planarity test. The function PLANTEST expects a biconnected graph G, an st-numbering *st_num* of its nodes, and a list *st_list* containing the nodes of G in increasing order of st-number, and returns *true* iff G is a planar graph.

If G has less than five nodes then G is planar. So assume that G has at least five nodes. We declare a PQ-tree $T(m)$, where m is one larger than the maximal index of any edge[18]. We use T to maintain the bush forms $\hat{B}_k$ for $k = 0, 1, 2, \ldots$.

We iterate over the nodes in increasing order of st-number. For each v, we collect the edges that connect v to lower numbered nodes in L, and we collect the edges that connect v to higher numbered nodes in U. Self-loops are ignored as they do not affect planarity. We update the bush form by

```
T.replace(L,U,I),
```

where I is a dummy argument. If the call is not successful, we break from the loop and return *false*, if the call is successful, we proceed to the next node. If all nodes can be added to the bush form we return *true*.

[18] The data type graph numbers edges with non-negative integers. The number of an edge is called its index. Since PQ-trees expect positive numbers, we identify any edge with its index plus one.

⟨*planarity test*⟩≡
```
static bool PLANTEST(graph& G, node_array<int>& st_num,
                               list<node>& st_list)
{
  int n = G.number_of_nodes();
  int m = G.max_edge_index() + 1;
  if (n < 5)  return true;
  pq_tree  T(m);
  int stv = 1;
  node v;
  forall(v,st_list)
  {
    list<int> L, U, I;
    edge e;
    forall_inout_edges(e,v)
    { node w = G.opposite(v,e);
      int stw = st_num[w];
      if (stw < stv) L.push(index(e)+1);
      if (stw > stv) U.push(index(e)+1);
     }
    if ( !T.replace(L,U,I) ) break;
    stv++;
   }
  return stv == n+1;
}
```

The program above performs the planarity test in time $O(n + m)$. This follows from the fact that the declaration of T requires time $O(m)$ and that the total cost of all *replace* operations is $O(n + m)$ and that an st-numbering can be computed in linear time (see Section **??**).

The program above is short and elegant. It performs a complex task, namely, to test whether a graph is planar, in linear time and a few lines of code. Of course, all the complexity is hidden in the implementation of PQ-trees.

Can you trust the program above? "*Yes, you can trust it*", but "*it would be unwise to do so*". We have not explained the inner workings of PQ-trees, their implementation is complex (almost 2000 lines), and most seriously there is no way to check the answer of the program above. It just says "yes" or "no". In the sections to come we will extend the program above to a program that can be checked. We show how to compute planar embeddings of planar graphs and Kuratowski subgraphs of non-planar graphs.

8.7.2 ***Planar Embeddings***

Chiba et al [CNAO85, NC88] have shown how to extend the planarity test of Lempel, Even, and Cederbaum to an embedding algorithm. We review their algorithm and give the implementation of functions

```
static bool PLAN_EMBED(graph& G, node_array<int>& st_num,
                                     list<node>& st_list);
bool BL_PLANAR(graph& G, bool embed);
```

The first function takes a biconnected map G, an st-numbering *st_num* of G, and the list of nodes of G in increasing order of st-number, and tests whether G is planar. If G is planar, it reorders the adjacency lists of G such that G becomes a plane map.

The second function applies to any map G. It returns *true* if G is planar and it returns *false* otherwise. If G is planar and *embed* is *true*, G is turned into a plane map. If *embed* is *true* and G is not a map, the function aborts. If *embed* is *false*, the function applies to any graph G.

Biconnected st-numbered Maps: We discuss the function PLAN_EMBED. The planarity testing algorithm constructs a sequence of bush forms $\hat{B}_0$, $\hat{B}_1$, $\hat{B}_2$, $\ldots$, $\hat{B}_n$. The construction is implicit in the sense that the bush forms are hidden in the internal structure of the PQ-tree. We want $\hat{B}_n$. The construction of $\hat{B}_{k+1}$ from $\hat{B}_k$ consists of two steps: first, $\hat{B}_k$ is transformed into an extendible bush form $\hat{B}'_k$ and then node $k+1$ is added to obtain $\hat{B}_{k+1}$.

For a node v let $L(v)$ be the set of edges (v,w) with $w < v$, and for any integer k with $k \geq v$ let $L_k(v)$ be the counter-clockwise order of the edges in $L(v)$ in the bush form $\hat{B}_k$. The embedding algorithm is based on the following observations:

- The cyclic order of the adjacency lists $A(v)$, $v \in V$, can be constructed from the lists $L_n(v)$, $v \in V$.
- The sequence $L_k(k)$ is readily extracted from the PQ-tree data structure.
- The sequence $L_{k+1}(v)$ is equal to $L_k(v)$ or $L_k^{rev}(v)$ for $k \geq v$.

We provide more details on the last item and postpone the discussion of the other two items.

Bush forms are transformed by permutations and flippings. Permutations have no effect on the order of the lists $L(v)$ for any v. They have a dramatic effect on the order of the lists $U(v)$, where $U(v)$ is the set of edges (v,w) with $v < w$. For this reason we do not keep track of the order of the $U(v)$'s during the construction process but determine their orders in a second phase (this is the subject of the first item). A flipping of a sub-bush with lowest numbered vertex w reverses the order of $L(v)$ for all v in the sub-bush with $v \neq w$ and does not affect the order of $L(v)$ for any other v. We conclude that $L_{k+1}(v)$ is equal to either $L_k(v)$ or $L_k^{rev}(v)$ for any v with $v \leq k$. We say that node v is flipped in iteration $k+1$ if $L_{k+1}(v) = L_k^{rev}(v)$. If v is not flipped in iteration $k+1$ then $L_{k+1}(v) = L_k(v)$.

We conclude that $L_n(v)$ is equal to $L_v(v)$ if v is flipped an even number of times and is equal to $L_v^{rev}(v)$ if v is flipped an odd number of times. We next show how to determine efficiently how often nodes are flipped. We could maintain a counter for

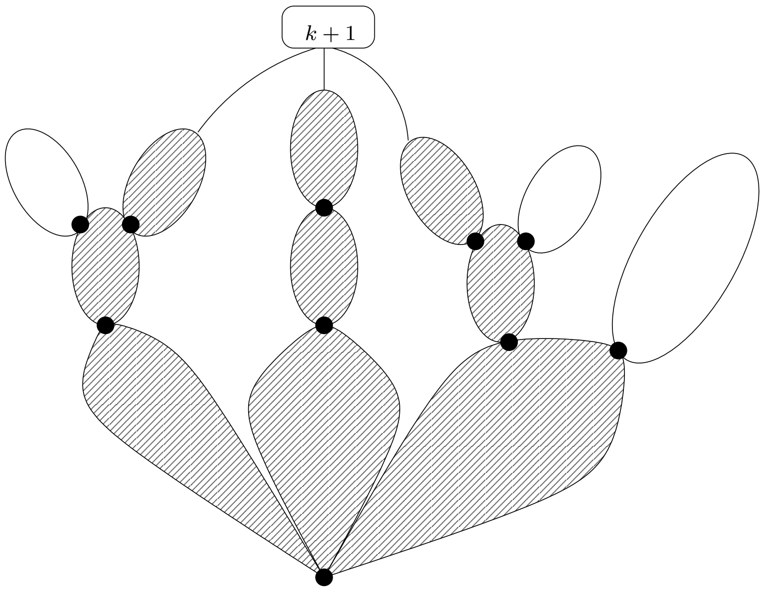

Figure 8.21 The biconnected components of B_k are indicated as ovals and articulation points are indicated as solid circles. The hatched biconnected components become part of H_{k+1}.

each node and increment it whenever the node is flipped. Since a linear number of nodes may be flipped in each iteration, this would result in a quadratic algorithm. We are aiming for linear running time and hence need a more compact way to maintain the counters.

In the graph B_{k+1} there is a unique biconnected component H_{k+1} having $k+1$ as its highest numbered node. We call H_{k+1} the biconnected component formed in iteration $k+1$.

Lemma 6 *All edges in $L(k+1)$ are contained in H_{k+1} and any biconnected component H of B_k is either contained in H_{k+1} or edge-disjoint from H_{k+1}, see Figure 8.21.*

Proof Consider any two lower neighbors u and v of $k+1$. They are connected by a path of length two through $k+1$ and they are connected by a path which avoids $k+1$, the second half-sentence being a consequence of st-numbering. Thus, all edges in $L(k+1)$ belong to H_{k+1} and the first part of the lemma is shown.

Any two edges belonging to the same biconnected component of B_k belong to the same biconnected component of B_{k+1}. This proves the second part of the lemma. □

For a biconnected component H of B_k let $V^+(H)$ denote the set of nodes of H except for the lowest numbered node of H. A flipping operation changes either the

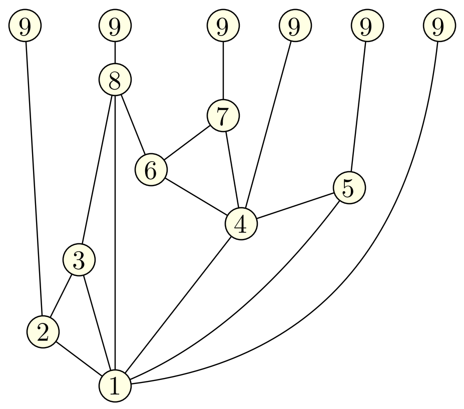

Figure 8.22 The bush form $\hat{B}_8$ obtained from adding node 8 to the bush form of Figure 8.16. The biconnected component H_8 consists of the biconnected components H_3, H_5, and H_7 and the edges in $L(8)$. The counter-clockwise order of the edges in $L(8)$ is $(8,3)$, $(8,1)$, $(8,6)$. The biconnected components H_3 and H_7 are flipped when going from the bush form $\hat{B}_7$ of Figure 8.15 to $\hat{B}_8$. Thus $I = 3, -3, 5, -7, (8,3), (8,1), (8,6)$, where the first 3 indicates that three components are merged into H_8, the sequence $-3, 5, -7$ indicates that the merged components are H_3, H_5, and H_7 and that H_3 and H_7 are flipped, and where $(8,3), (8,1), (8,6)$ form $L(8)$.

order of $L(v)$ for all nodes $v \in V^+(H)$ or for no node $v \in V^+(H)$. This follows from the fact that a biconnected component is either contained in a sub-bush or disjoint from it. We say that a biconnected component H is flipped in iteration $k+1$ if all nodes in $V^+(H)$ are flipped in iteration $k+1$.

Lemma 7 *There is a transformation of $\hat{B}_k$ to an extendible bush form in which only biconnected components H of B_k are flipped that become part of H_{k+1}.*

Proof Let $\hat{B}'_k$ be the extendible bush form produced by the strategy of Lemma 5 and assume that some biconnected component H that does not become part of H_{k+1} is flipped by the transformation from $\hat{B}_k$ to $\hat{B}'_k$. Let $y = y(\hat{B}'_k)$ be the lowest numbered node that is part of a biconnected component H that is flipped by the transformation to $\hat{B}'_k$ and does not become part of H_{k+1}. Consider the bush form $\hat{B}''_k$ obtained by flipping the smallest sub-bush $\hat{B}$ that contains H. $\hat{B}''_k$ is extendible since no leaf labeled $k+1$ is contained in $\hat{B}$. Moreover, either no biconnected component that does not become part of H_{k+1} is flipped in $\hat{B}''_k$ or $y(\hat{B}''_k) > y(\hat{B}'_k)$.

We conclude that $\hat{B}_k$ can be transformed into an extendible bush form in which only biconnected components are flipped that become part of H_{k+1}. □

We can now explain the third argument of function *replace* of class *pq_tree*. It consists of three parts, which in iteration $k+1$ are as follows (see Figure 8.22):

- An integer l specifying the number of components of B_k that are merged into H_{k+1}.

- A sequence j_0, j_1, …, j_{l-1} of integers, where $H_{|j_0|}$, …, $H_{|j_{l-1}|}$ are the biconnected components of B_k that are merged into H_{k+1}, and j_i is positive if H_{j_i} is not flipped in iteration $k+1$ and is negative otherwise.
- The edges[19] in $L(k+1)$ in their counter-clockwise order around $k+1$ in $\hat{B}_{k+1}$.

We denote the third argument of *replace* by I because it contains the instructions of how to obtain $\hat{B}_{k+1}$ from $\hat{B}_k$.

We are now ready for the implementation of PLAN_EMBED. It consists of three phases. In the first phase, we run the planarity test of the preceding section with three changes:

- We are now dealing with a map and therefore store only one direction of each edge in the PQ-tree. In phase one we are dealing with lists $L(v)$ and hence we store the direction from larger to smaller nodes. We construct the lists $L(v)$ and $U(v)$ by iterating over all edges out of v: edges to lower numbered nodes are put into $L(v)$ and the reversals of edges to higher numbered nodes are put into $U(v)$. We put edge reversals into $U(v)$ in order to guarantee that for each uedge the direction going from higher to smaller st-number is put into the PQ-tree. Self-loops are ignored in phase one.
- We define an array *EDGE* that stores for each integer in $[1..m]$ the edge corresponding to it.
- In iteration k we store the output I of PQ-tree operation *replace* in $I[k]$.

Here comes phase one.

⟨*PLAN_EMBED: phase 1*⟩≡
```
int n = G.number_of_nodes();
if ( G.number_of_edges() == 0 ) return true;
int m = G.max_edge_index() + 1;
// interface for pq_tree
pq_tree  T(m);
list<int>* I = new list<int>[n+1];
edge* EDGE  = new edge[m+1];  // EDGE[i+1] = edge with index i
edge  e;
forall_edges(e,G) EDGE[index(e)+1] = e;
// planarity test
int stv = 1;
node v;
forall(v,st_list)
{
  list<int> L, U;
  edge e;
```

[19] More precisely, the sequence of numbers identifying the edges.

```
    forall_adj_edges(e,v)
    { int stw = st_num[target(e)];
      if (stw < stv) L.push(index(e) + 1);
      if (stw > stv) U.push(index(G.reversal(e)) + 1);
    }
    if ( !T.replace(L,U,I[stv]) ) break;
    stv++;
  }
```

At the end of phase one, we either have $stv < n + 1$ and then G is non-planar, or $stv = n + 1$ and then G is planar and $I[k]$ contains the instruction list of the k-th iteration for all k, $1 \leq k \leq n$. Thus:

⟨*planar embedding of biconnected maps*⟩≡

```
  static int PLAN_EMBED_K(graph& G, node_array<int>& st_num,
                                    list<node>& st_list)
  { ⟨PLAN_EMBED: phase 1⟩
    if (stv == n+1) { ⟨PLAN_EMBED: phase 2⟩ }
    delete[] EDGE;
    delete[] I;
    return stv - 1;
  }
  static bool PLAN_EMBED(graph& G, node_array<int>& st_num,
                                    list<node>& st_list)
  { return PLAN_EMBED_K(G,st_num,st_list) == G.number_of_nodes(); }
```

The first version of the function is needed for the search for Kuratowski subgraphs in the next section. It returns the largest integer k such that B_k has a bush form.

We come to the second phase. The purpose of the second phase is to determine for each node the order of $L(v)$ in $\hat{B}_n$. This is either $L_v(v)$ or $L_v^{rev}(v)$ depending on whether v is flipped an even or an odd number of times.

Node n is not flipped at all. Consider now a node $j < n$ and assume that H_j is merged into H_k in iteration k. Then j is not flipped in iterations $j + 1$ to $k - 1$, is flipped in iteration k if $I[k]$ contains $-j$ in its second part and is not flipped in iteration k if $I[k]$ contains $+j$ in its second part, and is flipped in iterations later than k iff node k is flipped. Thus it is easy to compute the number of times any node v is flipped by iterating over all nodes in downward order of st-number.

It actually suffices to compute the parity of the number of times a node is flipped; the parity is $+1$ if the number is even and is -1 otherwise. Assume that we process node k and let j be such that H_j is merged into H_k in iteration k. Then the parity of j is equal to the sign of the occurrence of j in $I[k]$ times the parity of k. In the piece of code below, node k tells node j, if the parity of j is odd, by putting the indicator ODD as the first element of $I[j]$.

The order of $L_n(v)$ is equal to the third part of $I(v)$, if v is flipped an even number of times, and is equal to the reversal of the third part of $I(v)$ otherwise.

⟨*PLAN_EMBED: phase 2*⟩≡

```
node_array<list<edge> > L_n(G);
const int EVEN = +1; const int ODD = -1;
int stv = n;
forall_rev(v,st_list)
{
  if (stv == 1) break;   // for v = t down to s+1
  list<int>* I_v = &I[stv];
  int d  = 1;
  int l  = I_v->pop();
  if ( l == ODD )
  { d = -1;
    l = I_v->pop();
  }
  // l = number of components merged into H_v
  int i;
  for( i = 0; i < l; i++)
  { int j = d * I_v->pop();
    if (j < 0) I[-j].push(ODD);  // tell j that it is odd
  }
  if (d > 0)
    forall(i,*I_v) L_n[v].append(EDGE[i]);
  else
    forall(i,*I_v) L_n[v].push(EDGE[i]);
  stv--;
}
```

⟨*PLAN_EMBED: phase 3*⟩

We come to the third and last phase of PLAN_EMBED. We know $L_n(v)$ for every node v and want to compute the counter-clockwise order of the edges in $U(v)$, where $U(v)$ is the set of edges connecting v to higher numbered nodes. Self-loops will be treated as an add-on. We compute the ordering of the edges in $U(v)$ by so-called *leftmost depth-first search.*

Consider a depth-first search starting in t that uses only edges in $L(v)$ and that considers the edges in $L(v)$ in their counter-clockwise order. Such a depth-first search is called a leftmost depth-first search, as the edges in $L(v)$ are explored in left-to-right order (if drawn downwards from v) for any v and, more generally, the graph $\hat{B}_n$ is explored in a left-to-right fashion. This implies that for any node v, the edges in $U(v)$ are explored in left-to-right fashion, i.e., clockwise order, see Figure 8.23.

Lemma 8 *A leftmost depth-first search explores the edges in $U(u)$ in clockwise order for any node u.*

Proof Assume otherwise. Let u be the highest numbered node such that $U(v)$ is

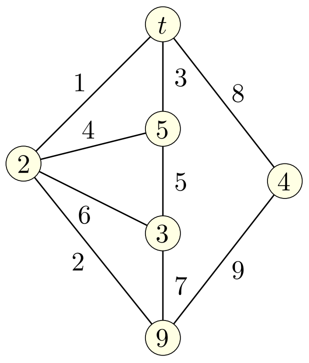

Figure 8.23 A leftmost depth-first search starting in t. For every node v the edges going to lower numbered neighbors are explored in left-to-right order. The edge labels indicate the order in which the edges are explored.

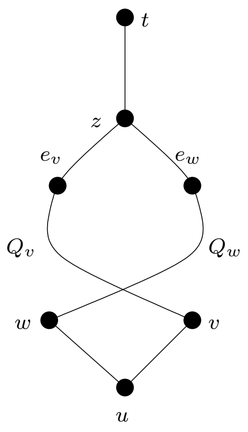

Figure 8.24 The edge (u, v) is after (u, w) in the clockwise order of edges in $U(u)$ but (v, u) is explored before (w, u).

ordered incorrectly, say edge (u, v) is after edge (u, w) in the clockwise order of edges in $U(u)$, but (v, u) is explored before (w, u). Consider the paths P_v and P_w from t to u, which follow the tree paths to v and w in the depth-first search tree, respectively, and then take the edge (v, u) or (w, u), respectively, see Figure 8.24. Let z be the node furthest from t and different from u that is common to both path. Let Q_v and Q_w be the induced paths from z to u passing through v and w, respectively, and let e_v and e_w be the first edges on these paths. Then e_v precedes e_w in the counter-clockwise order of the edges in $L(z)$.

The paths Q_v and Q_w are y-monotone, Q_v is left of Q_w "near" z, and Q_v is right of Q_w "near" u, and hence the two paths must cross. By definition of z they do not cross in a node and hence $\hat{B}_n$ is not a bush form of B_n. □

The following function LMDFS realizes leftmost depth-first search and builds a list *embed_list* containing all edges in $\cup_u U(u)$ in the order in which they are explored; the edge which is explored first comes last in the list, and the edge which is explored last comes first (since edges are pushed on the list and not appended). In other words, for each node u the edges in $U(u)$ occur in counter-clockwise order in *embed_list*. The edges do not necessarily occur consecutively.

LMDFS reuses the array *st_num* to record whether a node has been visited. leftmost depth-first search

⟨*auxiliary functions*⟩≡

```
static void LMDFS(graph& G, node v, const node_array<list<edge> >& L_n,
                  node_array<int>& st_num, list<edge>& embed_list)
{
  if (st_num[v] < 0) return;
  st_num[v] = -1;
  edge e;
  forall(e,L_n[v])
  { embed_list.push(G.reversal(e));
    LMDFS(G,target(e),L_n,st_num,embed_list);
  }
}
```

We use LMDFS in a function *embedding* that reorders the adjacency lists. We first build a list *embed_list* containing for each node v the set of edges in $A(v)$ in counter-clockwise order but not necessarily consecutively, and then use the sorting function *G.sort_edges*(*embed_list*) to rearrange the adjacency lists accordingly.

We build *embed_list* in three steps. In the first step we copy the lists $L\_n[v]$ to *embed_list*, in the second step we call LMDFS to add the edges in $\cup_v U(v)$ in their counter-clockwise order, and in the third step we deal with all self-loops. The self-loops can be added in any order, we only have to make sure that the two directions of a self-loop are placed next to each other. In this way there will be no crossings between self-loops.

⟨*auxiliary functions*⟩+≡

```
static void embedding(graph& G, node t, node_array<int>& st_num,
                      node_array<list<edge> >& L_n)
{
  list<edge> embed_list;
  node v; edge e;
  forall_nodes(v,G)
    forall(e,L_n[v]) embed_list.append(e);
  LMDFS(G,t,L_n,st_num,embed_list);
  // append self-loops at the end of the list
  edge_map<bool> treated(G,false);
  forall_nodes(v,G)
  { edge e;
    forall_adj_edges(e,v)
```

```
        if (target(e) == v && !treated[e])
        { embed_list.append(e); embed_list.append(G.reversal(e));
          treated[e] = treated[G.reversal(e)] = true;
        }
     }
    G.sort_edges(embed_list);
  }
```

After all this preparatory work phase three reduces to a call of embedding.

⟨*PLAN_EMBED: phase 3*⟩≡

```
  node t = st_list.tail();
  embedding(G,t,st_num,L_n);
```

The running time of PLAN_EMBED is $O(n+m)$. We have already argued that phase one takes linear time. Phase two touches every edge once and hence takes also linear time. Phase three consists of a depth-first search followed by extracting the adjacency lists from *embed_list* and hence takes linear time.

Arbitrary Maps: We give the implementation of *BL_PLANAR*(*G*, *embed*). Recall that *G* must be a map if *embed* is *true*. The implementation is fairly simple.

We extend *G* to a biconnected graph (if *embed* is *false*) and to a biconnected map (if *embed* is *true*), compute an st-numbering of *G*, call the planarity test for biconnected graphs and maps, respectively, and remove the added edges. The function *Make_Biconnected* is discussed in the exercises of Section **??**. It makes a graph biconnected by adding edges. It does so without destroying planarity.

⟨*planar embedding of arbitrary maps*⟩≡

```
  bool BL_PLANAR(graph& G, bool embed)
  { if (G.number_of_edges() <= 0)  return true;
    // prepare graph
    list<edge> el;
    if (embed)
    { if ( !G.make_map() )
        error_handler(1,"BL_PLANAR: can only embed maps.");
      Make_Biconnected(G,el);
      edge e;
      forall(e,el)
      { edge x = G.new_edge(target(e),source(e));
        el.push(x);
        G.set_reversal(e,x);
      }
    }
    else
      Make_Biconnected(G,el);
    node_array<int> st_num(G);
    list<node> st_list;
    ST_NUMBERING(G,st_num,st_list);
    bool plan;
    if (embed)
      plan = PLAN_EMBED(G,st_num,st_list);
    else
      plan = PLANTEST(G,st_num,st_list);
    // restore graph
    edge e; forall(e,el) G.del_edge(e);
    return plan;
  }
```

8.7.3 *Kuratowski Subgraphs*

We describe functions to extract Kuratowski subgraphs. We first give a simple algorithm with quadratic running time, then a linear time algorithm for biconnected graphs, and finally a linear time algorithm for arbitrary graphs.

We start with a simple algorithm that computes Kuratowski subgraphs in quadratic time $O((n+m)m)$. We iterate over all edges e of G. We hide e and check the planarity of $G \setminus e$. If $G \setminus e$ is non-planar, we leave e hidden, and if $G \setminus e$ is planar, we add e to the set of edges of the Kuratowski subgraph and restore it. At the end we restore all edges. The running time of this algorithm is m times the running time of the planarity test. The running time can be improved to $O(n^2)$ by observing that it suffices to consider $3n+7$ uedges of G, since a planar graph with n nodes can have at most $3n+6$ edges according to Lemma 3. We leave it to the exercises to implement this improvement.

$\langle$*auxiliary functions*$\rangle+\equiv$

```
  static void KURATOWSKI_SIMPLE(graph& G, list<edge>& K)
  { K.clear();
    if ( BL_PLANAR(G,false) )
      error_handler(1,"KURATOWSKI_SIMPLE: G is planar");
    list<edge> L = G.all_edges();
    edge e;
    forall(e,L)
    { G.hide_edge(e);
      if (BL_PLANAR(G,false))
      { G.restore_edge(e);
        K.append(e);
      }
    }
    G.restore_all_edges();
  }
```

We turn to the linear time algorithm of Karabeg and Hundack, Mehlhorn, and Näher [Kar90, HMN96] to find Kuratowski subgraphs. We assume that G a biconnected non-planar map without self-loops and parallel edges.

When the planarity test algorithm is run on G there will be a minimal k such that B_k has a bush form but B_{k+1} does not, because B_k contains an obstruction. Then $k+1 < n$ since $\hat{B}_{n-1}$ can always be extended. We show

Lemma 9 *If B_k has a bush form and contains an obstruction then G contains a Kuratowski subgraph.*

An obstruction is either an obstructing articulation point or an obstructing biconnected component. We deal with obstructing articulation points first and then with obstructing biconnected components. For both cases we need some simple facts about trees. For a tree T and a subset S of its nodes we use $T(S)$ to denote the smallest subtree of T connecting all nodes in S. If $|S| \leq 3$ then $T(S)$ contains a node r, called the *join* of S in T, such that the paths from r to the nodes in S are pairwise edge-disjoint ($r \in S$ is allowed). If $|S| = 3$, the join is unique.

Lemma 10 *Let v be an articulation point of B_k and let T be a depth-first search tree of B_k rooted at v. If w and z are distinct virtual nodes in some connected component C of B_k with respect to v then the join of $\{v, w, z\}$ in T is distinct from v, w, and z.*

Proof Let u be the first node reached in a depth-first search of C starting in v. Since C is a component with respect to v, $C \backslash v$ is connected. This implies that all nodes in $C \backslash v$ are descendants of u in T. □

In the sequel we use T_t to denote a tree on nodes $\{k+1, \ldots, n\}$ rooted at $t(=n)$ and where each node v, $v < n$, has an incoming edge from a higher numbered node. Such a tree exists since G is st-numbered.

We also use T_s to denote a depth-first search tree of B_k. T_s is rooted at s except if explicitly specified otherwise.

An Obstructing Articulation Point: Let v be an obstructing articulation point, i.e., at least three of the components with respect to v are mixed. Let C_i, $0 \leq i \leq 2$, be a mixed component with respect to v, let w_i be a leaf[20] labeled $k+1$ in C_i and let z_i be a large[21] leaf in C_i. Let T_s be a depth-first search tree of B_k rooted at v.

Let T_i be the subgraph of T_s spanned by v, w_i, and z_i, and let x_i be the join of T_i. Consider the subgraph K of G consisting of:

- T_0, T_1, T_2, and the tree $T_t(z_0, z_1, z_2)$.

Let r be the join of z_0, z_1, and z_2 in T_t. Then $r \neq k+1$ and hence K is a subdivision of $K_{3,3}$ with sides $\{x_0, x_1, x_2\}$ and $\{k+1, v, r\}$, see Figure 8.25.

[20] We will use leaf and virtual node as synonyms in this section.
[21] A large leaf is a leaf that is labeled $k+2$ or larger.

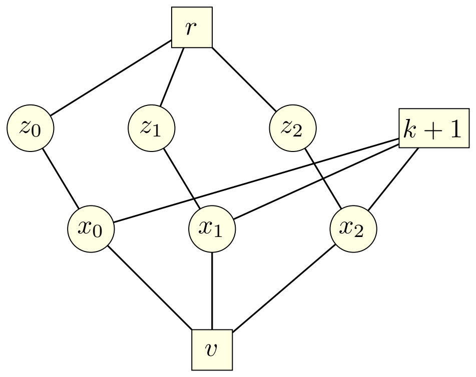

Figure 8.25 A $K_{3,3}$ with sides $\{x_0, x_1, x_2\}$ and $\{v, k+1, r\}$.

An Obstructing Biconnected Component: Let H be a biconnected component with attachment nodes y_0, y_1, ..., y_{p-1}. We assume that y_0 is the lowest numbered attachment node and that y_0, y_1, ..., y_{p-1} appear in this order on the boundary cycle of H in $\hat{B}_k$, where $\hat{B}_k$ is a bush form of B_k. Let C_i be the part of B_k opposite to H at y_i and let $s(C_i) \in \{\text{clean}, \text{mixed}, \text{full}\}$ be the status of C_i. We have

$$s(C_0)s(C_1)\dots s(C_{p-1}) \notin \text{ clean}^* \text{ mixed}_0^1 \text{ full}^* \text{ mixed}_0^1 \text{ clean}^*,$$

since H is obstructing.

Lemma 11 *One of the cases below arises:*

(1) *There are indices a, b, c, and d such that y_a, y_b, y_c, and y_d occur in this order on the boundary cycle of H, and C_a and C_c are non-clean and C_b and C_d are non-full.*

(2) *There are indices a, b, and c such that y_a, y_b, and y_c occur in this order on the boundary cycle of H, and C_a, C_b, and C_c are mixed.*

In either case, 0 is among the selected indices.

Proof Observe first, that C_0 is either clean or mixed, but never full (since there is a leaf labeled n in C_0 and $k+1 < n$). If

$$s(C_1)\dots s(C_{p-1}) \notin \text{ clean}^* \text{ mixed}_0^1 \text{ full}^* \text{ mixed}_0^1 \text{ clean}^*,$$

then there are a, b, c with $1 \le a < b < c \le p-1$ and C_a and C_c are non-clean and C_b is non-full. Since C_0 is non-full we are in case (1) with $d = 0$. So assume that

$$s(C_1)\dots s(C_{p-1}) \in \text{ clean}^* \text{ mixed}_0^1 \text{ full}^* \text{ mixed}_0^1 \text{ clean}^*.$$

Then C_0 is non-clean (and hence mixed) and $p - 1 \ge 2$ since H is non-obstructing otherwise.

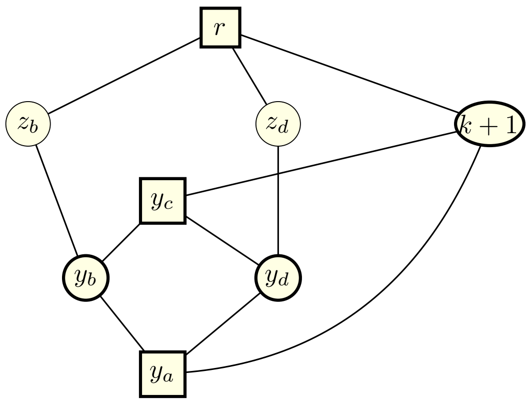

Figure 8.26 An obstructing cycle with four alternating attachments gives rise to a $K_{3,3}$ with sides $\{y_a, y_c, r\}$ and $\{y_b, y_d, k+1\}$.

If case (1) does not arise with $a = 0$ then there are no b, c, and d with $1 \le b < c < d \le p-1$ with C_b and C_d non-full and C_c non-clean, i.e., any C_c between two non-full C_b and C_d is clean. Thus, either $p - 1 = 2$ or

$$s(C_1)\dots s(C_{p-1}) \in \text{ clean}^* \text{ mixed}_0^1 \text{ full}^* \cup \text{ full}^* \text{ mixed}_0^1 \text{ clean}^*.$$

In the latter situation H is non-obstructing, and hence this case is excluded. In the former situation C_1 and C_2 must be mixed since H is non-obstructing otherwise. Thus, (2) arises. □

We next exhibit Kuratowski subgraphs for cases (1) and (2).

Assume first that there are indices a, b, c, and d such that y_a, y_b, y_c, and y_d occur in this order on the boundary cycle of H, C_a and C_c are non-clean and C_b and C_d are non-full. We call this an *obstructing cycle with four alternating attachments.* Consider the subgraph K of G consisting of:

- the boundary cycle of H,
- a path from y_a to a copy of $k+1$ in C_a,
- a path from y_c to a copy of $k+1$ in C_c,
- a path from y_b to a large leaf z_b in C_b,
- a path from y_d to a large leaf z_d in C_d,
- the tree $T_t(\{k+1, z_b, z_d\})$.

Let r be a join of $k+1$, z_b, and z_d in T_t; we may assume that $r \ne k+1$ (observe that $z_b \ne k+1$ and $z_d \ne k+1$). K is a subdivision of $K_{3,3}$ with sides $\{y_b, y_d, k+1\}$ and $\{y_a, y_d, r\}$, see Figure 8.26.

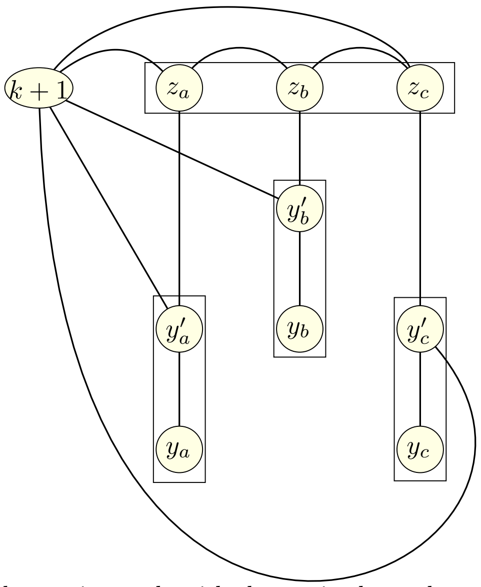

Figure 8.27 An obstructing cycle with three mixed attachments yields a K_5 after contraction of the paths from y_i to y'_i for $i \in \{a,b,c\}$ and contraction of the edges in tree $T_t(\{z_a, z_b, z_c\})$.

Assume next that there are indices a, b, and c such that y_a, y_b, and y_c occur in this order on the boundary cycle of H and C_a, C_b, and C_c are mixed. We call this a *cycle with three mixed attachments.* Consider the subgraph K of G consisting of:

- the boundary cycle of H,
- trees $T_s(\{y_i, w_i, z_i\})$ where $i \in \{a,b,c\}$, w_i is a leaf labeled $k+1$ in C_i, and z_i is a large leaf in C_i,
- tree $T_t(\{k+1, z_1, z_2, z_3\})$.

Let y'_i be the join of y_i, z_i, and w_i. Then y'_i is distinct from z_i and w_i but may be equal to y_i. Figure 8.27 illustrates the situation.

We can obtain a K_5 from K by contracting the paths connecting y_i with y'_i for $i \in \{a,b,c\}$ and by contracting the edges in $T_t(\{z_a, z_b, z_c\})$. We can now appeal to the fact that if a graph K can be contracted to a subdivision of $K_{3,3}$ or K_5 then it contains a subdivision of $K_{3,3}$ or K_5 before the contraction, see [NC88, Lemma 1.2] and the exercises. We will exploit this fact in our implementation.

For completeness we also exhibit the Kuratowski subgraphs directly. We distinguish three cases.

If $y_i = y'_i$ for all $i \in \{a,b,c\}$ and $T_t(\{k+1, z_a, z_b, z_c\})$ contains a node of degree four then K is a subdivision of K_5.

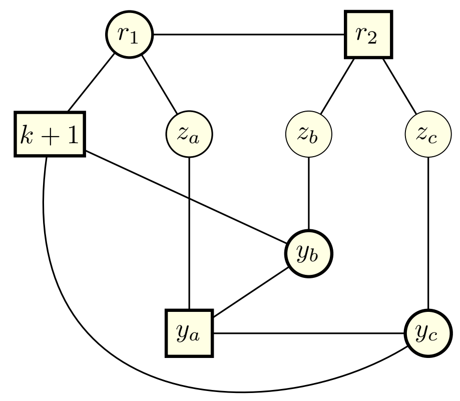

Figure 8.28 An obstructing cycle with three mixed attachments yields a $K_{3,3}$ if $y_i = y'_i$ for $i \in \{a,b,c\}$ and $T_t(\{k+1, z_a, z_b, z_c\})$ contains no node of degree four. In the figure, $k+1$ is paired with z_a.

If $y_i = y'_i$ for all $i \in \{a,b,c\}$ and $T_t(\{k+1, z_a, z_b, z_c\})$ contains no node of degree four then $T_t(\{k+1, z_a, z_b, z_c\})$ contains two nodes of degree three, say r_1 and r_2. The removal of the path joining r_1 and r_2 pairs $k+1$ with some z_i. We remove from K the path from y_i to the copy of $k+1$ in C_i and the part of the boundary cycle of H joining the other two y's and obtain a subdivision of $K_{3,3}$, see Figure 8.28, with sides $\{y_a, k+1, r_2\}$ and $\{y_b, y_c, r_1\}$.

If $y_i \neq y'_i$ for some $i \in \{a,b,c\}$, say $y_a \neq y'_a$, let r be the join of z_a, z_b, z_c in $T_t(\{z_a, z_b, z_c\})$. We obtain a subdivision of $K_{3,3}$ with sides $\{y_a, k+1, r\}$ and $\{y'_b, y'_c, y'_a\}$ from K by deleting the part of the boundary cycle of H that connects y_b and y_c, and by replacing $T_t(\{k+1, z_a, z_b, z_c\})$ by $T_t(\{z_a, z_b, z_c\})$, see Figure 8.29. This completes the proof of Lemma 9.

We turn to a linear time implementation. The following function assumes that G is a biconnected non-planar map without self-loops and parallel edges. It computes the set of edges of a Kuratowski subgraph of G in K.

⟨*Kuratowski graphs in biconnected maps*⟩≡
```
static void Kuratowski(graph& G, list<edge>& K)
{ node v; edge e;
  string current_case;  // for debugging purposes
  ⟨compute st-numbering⟩
  int k = PLAN_EMBED_K(G,st_num,st_list);
  if ( k == G.number_of_nodes() )
    error_handler(1,"Kuratowski: G must be non-planar");
  ⟨compute bush form B for B_k⟩
  ⟨obstructing articulation point⟩
  ⟨obstructing biconnected component⟩
}
```

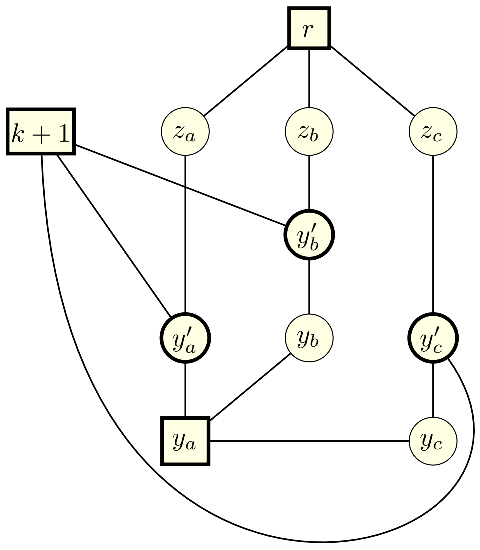

Figure 8.29 An obstructing cycle with three mixed attachments yields a $K_{3,3}$ if $y_a \neq y'_a$.

We start by computing an st-numbering of G. Next we call PLAN_EMBED_K to find k such that B_k has a bush form but B_{k+1} has not. We compute a bush form B for B_k and then search for an obstruction in B. This will be the most difficult part of the implementation. Having found an obstruction we extract a Kuratowski subgraph as shown in the proof of Lemma 9.

Compute *st*-Numbering: We compute an st-numbering and the nodes s and t.

$\langle$*compute st-numbering*$\rangle\equiv$

```
node_array<int> st_num(G);
list<node> st_list;
ST_NUMBERING(G,st_num,st_list);
node s = st_list.head();
node t = st_list.tail();
```

The Bush Form B for B_k: We construct a bush form B for B_k. We declare B of type *GRAPH*<*node*, *edge*> and let every node and edge of B know its original in G. We add a node *top_B* to B and connect it to every virtual node (by a uedge). In this way B becomes a biconnected map.

We st-number the nodes of B by first numbering the non-virtual nodes, then the virtual nodes, and finally the node *top_B*. We store the st-numbering in *st_numB*, the ordered list of nodes in *st_listB*. Finally, *sB* is the node in B that corresponds to s and *tB* is a virtual node in B that is connected to *sB* by an edge. *tB* is a large leaf in the root component of every articulation point and in the part of B opposite to y_0 for any biconnected component H with lowest attachment node y_0.

Having constructed the st-numbering we call PLAN_EMBED to compute a planar embedding of B. We restore the st-numbers as they are destroyed by the planar embedding program, and we delete the auxiliary node *top_B* from B and *st_listB*.

⟨*compute bush form B for B_k*⟩≡

```
GRAPH<node,edge> B;
list<node>      st_listB;
node_array<node> v_in_B(G,nil);
forall(v,st_list)
{ if ( st_num[v] > k ) break;
  node vB = v_in_B[v] = B.new_node(v);
  st_listB.append(vB);
}
node top_B = B.new_node();
forall_nodes(v,G)
{ if (st_num[v] > k) continue;
  forall_adj_edges(e,v)
  { node w = G.target(e);
    if ( st_num[w] < st_num[v] )  continue;
    edge r = G.reversal(e);
    node wB;
    if ( st_num[w] > k )
    { wB = B.new_node(w);
      st_listB.append(wB);
      B.set_reversal(B.new_edge(wB,top_B),B.new_edge(top_B,wB));
    }
    else
      wB =  v_in_B[w];
    edge e1 = B.new_edge(v_in_B[v],wB,e);
    edge r1 = B.new_edge(wB,v_in_B[v],r);
    B.set_reversal(e1,r1);
  }
}
node sB = v_in_B[s];  node tB;
forall_adj_edges(e,sB)
  if ( B[B.target(e)] == t) tB = B.target(e);
B.set_reversal(B.new_edge(sB,top_B),B.new_edge(top_B,sB));
st_listB.append(top_B);
node_array<int> st_numB(B);
int stn = 1;
forall(v,st_listB) st_numB[v] = stn++;
PLAN_EMBED(B,st_numB,st_listB); // destroys st-numbers
stn = 1;
forall(v,st_listB) st_numB[v] = stn++;
B.del_node(top_B); st_listB.Pop();  // remove top_B
```

Obstructing Articulation Points: We search for an obstructing articulation point and, if successful, extract a Kuratowski subgraph.

⟨*obstructing articulation point*⟩≡

```
array<node> z(3);
array<node> spec(3);
```

A successful search for an obstructing articulation point will store the obstructing articulation point in v, and for i, $0 \leq i < 3$, will store a large leaf in the i-th mixed component with respect to v in $z[i]$ and a leaf labeled $k+1$ in $spec[i]$.

The search (successful or not) will also compute some auxiliary information for internal use and for later use in the search for obstructing biconnected components.

We define an enum that we use to distinguish between leafs labeled $k+1$ and large leafs, and we define two functions so that node arrays can be used as type parameters.

⟨*auxiliary functions*⟩+≡

```
enum { K_PLUS_1 = 0, OTHERS = 1};
ostream& operator<<(ostream& o, const node_array<node>&) { return o; }
istream& operator>>(istream& i, node_array<node>&)       { return i; }
```

We give the declarations of the auxiliary informations and explain them below.

⟨*obstructing articulation point*⟩+≡

```
list<node> dfs_list;
node_array<edge> tree_edge(B,nil);
node_array<int> dfs_num(B,-1);
int dfs_count = 0;
DFS(B,sB,dfs_list,dfs_num,dfs_count,tree_edge);
edge_array<int> comp_num(B);
int num_comps = BICONNECTED_COMPONENTS(B,comp_num);
node_array<edge> up_tree_edge(G,nil);
array<node_array<node> > leaf(2);
leaf[K_PLUS_1] = leaf[OTHERS] = node_array<node>(B,nil);
array<node_array<node> > leaf_in_upper_part(2);
leaf_in_upper_part[K_PLUS_1] =
       leaf_in_upper_part[OTHERS] = node_array<node>(B,nil);
node_array<int>  num_mixed_non_root_comps(B,0);
node_array<node> spec_leaf_in_root_comp(B,nil);
array<node_array<node> > child(1,2);  // want indices one and two
child[1] = child[2] = node_array<node>(B,nil);
```

The auxiliary information is as follows: let T_s be a depth-first search tree of B rooted at s.

tree_edge$[v]$ is the tree edge into v in T_s for $v \neq s$ and is *nil* for $v = s$, *dfs_num*$[v]$

is the dfs-number of v, and *dfs_list* is the list of nodes of B in increasing order of dfs-number. All quantities just mentioned are computed by a call of the auxiliary function DFS, see below.

num_comps is the number of biconnected components, and *comp_num*$[e]$ is the number of the biconnected component containing e for any edge e of B. Both values are computed by calling the biconnected components function. We call *comp_num*$[e]$ the component number of e.

up_tree_edge$[v]$ is for any node v of G with *st_num*$[v] > k$ and $v \neq t$ an edge from a higher numbered node. It is *nil* for all other nodes of G. The up-tree edges define a tree T_t rooted at t on the nodes labeled $k+1$ and larger.

leaf$[$*K_PLUS_1*$][v]$ is a leaf labeled $k+1$ in the subtree of T_s rooted at v (*nil* if no such leaf exists).

leaf$[$*OTHERS*$][v]$ is a large leaf in the subtree of T_s rooted at v (*nil* if no such leaf exists).

The next four pieces of information are only defined for articulation points. The *upper part with respect to an articulation point* is the union of the non-root components with respect to the articulation point.

leaf_in_upper_part$[$*K_PLUS_1*$][v]$ is a leaf labeled $k+1$ in the upper part of v (*nil* if there is no such leaf).

leaf_in_upper_part$[$*OTHERS*$][v]$ is a large leaf in the upper part of v (*nil* if there is no such leaf).

child$[1][v]$ is a child of v in T_s that lies in a mixed non-root component with respect to v (*nil* if there is no such child).

child$[2][v]$ is a child of v in T_s that lies in a second mixed non-root component with respect to v (*nil* if there is no such child).

We next discuss how the auxiliary information is computed. The quantities related to depth-first search are computed by a variant of depth-first search.

⟨*auxiliary functions*⟩+≡

```
void DFS(const graph& G, node v,
         list<node>& dfs_list, node_array<int>& dfs_num,
         int& dfs_count, node_array<edge>& tree_edge)
{ dfs_list.append(v);
  dfs_num[v] = dfs_count++;
  edge e;
  forall_adj_edges(e,v)
  { node w = G.target(e);
    if ( dfs_num[w] == -1 )
    { tree_edge[w] = e;
      DFS(G,w,dfs_list,dfs_num,dfs_count,tree_edge);
    }
  }
}
```

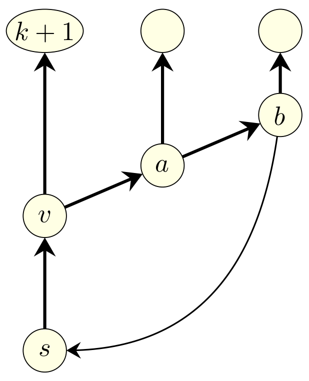

Figure 8.30 The root component of v consists of the nodes s, v, a, and b. Tree edges are drawn in bold. The tree edge (v,a) belongs to the same biconnected component as the tree edge into v, but the tree edge $(v,k+1)$ does not. The tree edge $(v,k+1)$ belongs to a non-root component with respect to v.

The up-tree is easily computed. We simply select for each node labeled larger than k an edge going to a node with higher st-number and then put the reversal of the edge into the tree.

⟨*obstructing articulation point*⟩+≡

```
forall_nodes(v,G)
{ if (st_num[v] <= k ) continue;
  edge e;
  forall_adj_edges(e,v)
  { node w = G.target(e);
    if ( st_num[w] > st_num[v] )
    { up_tree_edge[v] = G.reversal(e); break; }
  }
}
```

All other auxiliary information is computed by scans over T_s. We start with some simple observations, see Figure 8.30. We have, for any node v, the following:

- The tree edge into v belongs to the root component with respect to v.
- A tree edge out of v belongs to the root component with respect to v iff it belongs to the same biconnected component as the tree edge into v iff it has the same component number as the tree edge into v.
- A tree edge out of v belongs to a non-root component with respect to v iff its component number is different from the component number of the tree edge into v or if v is equal to (the copy of) s in B.
- The non-root components with respect to v are in one-to-one correspondence to the tree edges out of v.

The node labels *leaf*[*K_PLUS_1*] and *leaf*[*OTHERS*] are computed by a leaf to root scan of T_s.

⟨*obstructing articulation point*⟩+≡

```
forall_nodes(v,B)
{ if (st_numB[v] <= k) continue;
  if ( st_num[B[v]] == k + 1 )
    leaf[K_PLUS_1][v] = v;
  else
    leaf[OTHERS][v] = v;
}
forall_rev(v,dfs_list)  // down the tree
{ if (v == sB) continue;
  node pv = B.source(tree_edge[v]);
  assign(leaf[K_PLUS_1][pv],leaf[K_PLUS_1][v]);
  assign(leaf[OTHERS][pv],  leaf[OTHERS][v]);
}
```

where we used the following conditional assignment function *assign* to propagate information.

⟨*auxiliary functions*⟩+≡

```
void assign(node& x, const node& y) { if ( x == nil) x = y; }
```

We next compute for each articulation point v the number of mixed non-root components with respect to v and *leaf_in_upper_part*[][v].

A node v identifies a non-root component of its parent *pv* if either *pv* is equal to *sB* and *sB* has more than one child or if the tree edges into v and *pv* belong to different biconnected components. Actually, *sB* always has at least two children, one is a copy of t and the other contains a copy of $k+1$ in its subtree. Note that $k+1 \neq t$ since the planarity test cannot fail when node t is to be added.

The non-root component of *pv* identified by v is mixed if it contains a leaf labeled $k+1$ as well as a large leaf.

We are propagating information from the leaves to the root and hence know the number of mixed non-root components of v when v is reached. If a node v has three mixed non-root components we extract a Kuratowski subgraph.

⟨*obstructing articulation point*⟩+≡

```
forall_rev(v,dfs_list)   // down the tree
{ if (num_mixed_non_root_comps[v] >= 3)
  { ⟨v has three mixed non-root components⟩ }
  if ( v == sB) continue;
  node pv = B.source(tree_edge[v]);
  if ( pv == sB || comp_num[tree_edge[v]] != comp_num[tree_edge[pv]] )
  { if ( leaf[K_PLUS_1][v] && leaf[OTHERS][v] )
       num_mixed_non_root_comps[pv]++;
    assign(leaf_in_upper_part[K_PLUS_1][pv],leaf[K_PLUS_1][v]);
    assign(leaf_in_upper_part[OTHERS][pv],leaf[OTHERS][v]);
  }
}
```

Assume that v has three mixed non-root components. We iterate over all children of v and search for three children that define mixed non-root components. Whenever such a child is found we copy its two leaves to $y[i]$ and *spec*$[i]$ for $i = 0$, 1, and 2.

⟨*v has three mixed non-root components*⟩≡

```
current_case = "three mixed non-root components";
int i = 0;
forall_adj_edges(e,v)
{ node w = B.target(e);
  if ( w == sB || v != B.source(tree_edge[w]) ) continue;
  if ( leaf[K_PLUS_1][w] && leaf[OTHERS][w] )
  { z[i] = leaf[OTHERS][w]; spec[i] = leaf[K_PLUS_1][w];
    i++;
    if ( i == 3) break;
  }
}
⟨obstructing articulation point: extract Kuratowski graph⟩
```

The actual extraction of the Kuratowski subgraph will be discussed below.

If no articulation point has three mixed non-root components, we need to check whether there is an articulation point with two mixed non-root components and a mixed root component. It is slightly tricky to determine whether root components are mixed. We observe first that node s and hence node t is contained in any root component. Thus there is always a large leaf in the root component. In fact, it is the node tB.

We want to compute for each node v a leaf labeled $k+1$ in its root component (if any). Consider any path p in T_s from v to a leaf labeled $k+1$. The leaf belongs to the root component of v iff the target of the first edge of p belongs to the root component of v. This is the case if the first edge of p is the tree edge into v or is a tree edge out of v which belongs to the same biconnected component as the tree edge into v. We compute *spec_leaf_in_root_comp* by considering the two kinds of paths separately.

For the second kind of path we propagate information down the tree. We pass information about a leaf along a tree edge (v, w) if this edge belongs to the root component of v, i.e., if it has the same component number as the tree edge into v.

⟨*obstructing articulation point*⟩+≡

```
forall_rev(v,dfs_list)  // down the tree
{ if (v == sB) continue;
  node pv = B.source(tree_edge[v]);
```

```
    if ( pv != sB && comp_num[tree_edge[v]] == comp_num[tree_edge[pv]] )
      assign(spec_leaf_in_root_comp[pv],leaf[K_PLUS_1][v]);
  }
```

For the first kind of path we compute for every node v, *spec_leaf_via_tree_edge*$[v]$, a leaf labeled $k+1$ in the root component of v that is reachable through the tree edge into v (*nil* if there is no such leaf). A leaf labeled $k+1$ in the root component is then either a leaf that was already computed above or the leaf that can be reached via the tree edge into v.

spec_leaf_via_tree_edge is computed from the root towards the leaves of T_s. Let v be any node and consider the time when we process v. Let c be any child of v. A leaf in the root component of c that is reachable through the tree edge into c is either reachable through the tree edge into v or through a sibling of c.

If v has a leaf labeled $k+1$ that is reachable through the tree edge into v we simply pass this leaf to all children of v.

So assume that v has no leaf labeled $k+1$ that is reachable through the tree edge into v. We try to determine two children c_1 and c_2 of v that have a leaf labeled $k+1$ in their subtree. If there is none, then no child of v can reach a leaf labeled $k+1$ through one of its siblings, if there is exactly one child, then all siblings of this child can reach a leaf labeled $k+1$ through it, and if there are two children, then all children of v can reach a leaf labeled $k+1$ through a sibling.

When a node v is encountered that has two mixed non-root components and a mixed root component we have found an obstructing articulation point and proceed to extract a Kuratowski subgraph.

⟨*obstructing articulation point*⟩+≡

```
  node_array<node>  spec_leaf_via_tree_edge(B,nil);
  forall(v,dfs_list)  // up the tree
  { assign(spec_leaf_in_root_comp[v],spec_leaf_via_tree_edge[v]);
    if ( num_mixed_non_root_comps[v] == 2 && spec_leaf_in_root_comp[v] )
    { ⟨v has two mixed non-root and a mixed root component⟩ }
    if ( spec_leaf_via_tree_edge[v] != nil )
    { forall_adj_edges(e,v)
      { node c = B.target(e);
        if ( c == sB || v != B.source(tree_edge[c]) ) continue;
        spec_leaf_via_tree_edge[c] = spec_leaf_via_tree_edge[v];
      }
    }
    else
    { forall_adj_edges(e,v)
      { node c = B.target(e);
        if ( c == sB || v != B.source(tree_edge[c]) ) continue;
        if ( leaf[K_PLUS_1][c] )
        { if ( child[1][v] == nil )
            child[1][v] = c;
          else
```

```
            child[2][v] = c;
          }
        }
        if ( child[1][v] )
        { forall_adj_edges(e,v)
          { node c = B.target(e);
            if ( c == sB || v != B.source(tree_edge[c]) ) continue;
            if ( c != child[1][v] )
              spec_leaf_via_tree_edge[c] = leaf[K_PLUS_1][child[1][v]];
            else
              if ( child[2][v] )
                spec_leaf_via_tree_edge[c] = leaf[K_PLUS_1][child[2][v]];
          }
        }
      }
    }
```

Assume that v has two mixed non-root and a mixed root component. A leaf labeled $k+1$ in the root component of v is given by *spec_leaf_in_root_comp*$[v]$ and a large leaf is given by tB. For the other components we find the leaf labeled $k+1$ and the large leaf as in the case of three mixed non-root components.

⟨*v has two mixed non-root and a mixed root component*⟩≡

```
  current_case = "two mixed non-root and a mixed root component";
  z[0] = tB;
  spec[0] = spec_leaf_in_root_comp[v];
  int i = 1;
  forall_adj_edges(e,v)
  { node w = B.target(e);
    if ( w == sB || v != B.source(tree_edge[w]) ) continue;
    if ( v != sB && comp_num[e] == comp_num[tree_edge[v]] ) continue;
    if ( leaf[K_PLUS_1][w] && leaf[OTHERS][w] )
    { z[i] = leaf[OTHERS][w]; spec[i] = leaf[K_PLUS_1][w];
      i++;
      if ( i == 3) break;
    }
  }
```

⟨*obstructing articulation point: extract Kuratowski graph*⟩

Obstructing Articulation Point: Extraction of Kuratowski Graph: The node v is an obstructing articulation point. For every i, $0 \leq i < 3$, we have a large leaf in the i-th component in $z[i]$ and a leaf labeled $k+1$ in *spec*$[i]$.

We reroot the depth-first search tree at v and then extract the Kuratowski subgraph as described in the proof of Lemma 9.

⟨*obstructing articulation point: extract Kuratowski graph*⟩≡
```
// reroot the DFS-tree at v
dfs_list.clear();
dfs_num.init(B,-1);
tree_edge.init(B,nil);
int dfs_count = 0;
DFS(B,v,dfs_list,dfs_num,dfs_count,tree_edge);
list<edge> join_edges;
for (i = 0; i < 3; i++)
{ join(z[i],spec[i],v,tree_edge,B,join_edges);
  translate_to_G(join_edges,B); K.conc(join_edges);
}
join(B[z[0]],B[z[1]],B[z[2]],up_tree_edge,G,join_edges);
K.conc(join_edges);
check_before_return(G,K,st_num,leaf,tree_edge,dfs_num,k,
                    B,st_numB,sB,current_case);
return;
```

The function *check_before_return* calls *CHECK_KURATOWSKI*(G, K) to check whether K is a Kuratowski subgraph. If not, it opens two *GraphWins* and displays the edges in K in one of them and the bush form B in the other. We do not give details here. This visual debugging aid proved very valuable during the development phase of the algorithm.

The Join Function: Let T be a tree and let a, b, and c be the three nodes to be joined in T. For each node v the tree edge into v is stored in *tree_edge*$[v]$.

We trace the paths to the root from all three nodes and count, for each node of T, the number of paths containing it. Let r be the highest node which is reachable from all three nodes. The subtree joining the three nodes is the union of the paths from the three nodes to r. This union is not necessarily a disjoint union. We want to output each edge in the subtree only once and therefore mark nodes as we trace the paths. When a node is marked, its tree edge is added to the set L of edges comprising the subtree. The function returns r.

⟨*auxiliary functions*⟩+≡
```
node join(node a, node b, node c, const node_array<edge>& tree_edge,
          graph& B, list<edge>& L)
{ L.clear();
  node_array<int> num_desc(B,0);
  array<node> A(3); A[0] = a; A[1] = b; A[2] = c;
  int i;
  for (i = 0; i < 3; i++)
  { node v = A[i];
    num_desc[v]++;
    while ( tree_edge[v] != nil )
    { v = B.source(tree_edge[v]);
      num_desc[v]++;
    }
  }
  node r;
  for (i = 0; i < 3; i++)
  { node v = A[i];
    while (num_desc[v] < 3)
    { L.append(tree_edge[v]);
      num_desc[v] = 3;
      v = B.source(tree_edge[v]);
    }
    if ( i == 0 ) r = v;
  }
  return r;
}
void translate_to_G(list<edge>& L, const GRAPH<node,edge>& B)
{ list_item it;
  forall_items(it,L) L[it] = B[L[it]];
}
```

The function *translate* takes a list L of edges of B and replaces each edge by its counterpart in G.

Obstructing Biconnected Component: We come to obstructing biconnected components. We describe the search for an obstructing biconnected component and the extraction of a Kuratowski subgraph once an obstructing component has been found.

We exploit the fact that B is a plane map in our search for obstructing biconnected components. Consider any node v and the cyclic list $A(v)$ of edges out of v. If v is not an articulation point then all edges in $A(v)$ belong to the same biconnected component. If v is an articulation point then $A(v)$ decomposes into blocks, one for each biconnected component containing v. This follows from the fact that the boundary cycles of all biconnected component are part of the boundary of the outer face in every bush form.

Blocks that consist of at least two edges indicate the boundary cycle of a biconnected component. We find such blocks as follows. We iterate over all edges f out of v. If the cyclic predecessor of f in $A(v)$ belongs to a different biconnected component and the cyclic successor belongs to the same biconnected component, then f belongs to the boundary cycle of a non-trivial biconnected component, i.e., a biconnected component which is not just a single uedge. We maintain an edge array *treated_component* to record which biconnected components have already been treated.

If the component having f in its boundary cycle has not been treated yet, we determine its boundary cycle in *cycle_edges* and then determine whether one of the cases (1) or (2) of Lemma 11 applies.

In our search for biconnected components we iterate over the nodes of T_s from the root to the leaf. This has the advantage that we hit every biconnected component at its lowest node.

Let H be a biconnected component with attachment cycle $[y_0, y_1, \ldots, y_k]$, where y_0 is the lowest numbered node in the biconnected component. We need to know whether the component of B opposite to H at y_0 is mixed, i.e., contains a leaf labeled $k + 1$. We compute such a leaf in *spec_leaf_in_opposite_part*. For all i different from zero, the part of B opposite to H at y_i is simply the upper part of B with respect to y_i. We have collected information about upper parts already.

If the search for an obstructing biconnected component is unsuccessful, we give debugging information. After all, there must be either an obstructing articulation point or an obstructing biconnected component.

⟨*obstructing biconnected component*⟩≡

```
array<bool> treated_component(num_comps);
edge f;
forall(v,dfs_list)       // upwards
{ forall_adj_edges(f,v)
  { edge e1 = B.cyclic_adj_succ(f);
    edge e_pred = B.cyclic_adj_pred(f);
    if ( comp_num[e1] != comp_num[f] ||
         comp_num[f] == comp_num[e_pred] ) continue;
    if ( treated_component[comp_num[f]] ) continue;
    list<edge> cycle_edges;
    treated_component[comp_num[f]] = true;
    ⟨determine boundary cycle of component with lowest node y_0 = v⟩
    node spec_leaf_in_opposite_part = nil;
    ⟨compute leaf labeled k+1 in part opposite to y_0⟩
    ⟨obstructing cycle with four alternating attachments⟩
    if ( spec_leaf_in_opposite_part )
    { ⟨obstructing cycle with three mixed attachments⟩ }
  }
}
⟨unreachable point: give debugging information⟩
```

The boundary cycle of a biconnected component H is easily traced. We start with an edge f that emanates from v, the lowest node in the component, and that lies on the boundary cycle of the component. The unbounded face is to the right of f, see Figure 8.31. We will trace the boundary cycle in clockwise direction, i.e., keeping the unbounded face to our left, and store it in *cycle_edges*.

Assume that e is an edge such that its reversal belongs to the boundary cycle. Initially, e is equal to f. We show how to find the successor edge of e^{rev} in the boundary cycle. Let e_1 be the cyclic adjacency successor of e. We advance e_1 until the successor of e_1 belongs to a different biconnected component or the successor of e_1 is equal to e. The former case happens for nodes v that are attachment nodes of H and the latter case happens for nodes that lie on the boundary cycle of H

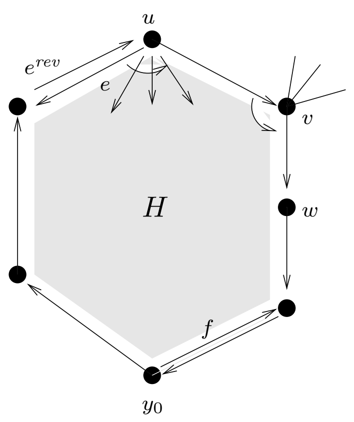

Figure 8.31 Scanning the boundary of a biconnected component H. We scan the boundary in clockwise direction. At each node, the reversal of a boundary edge is turned clockwise (i.e., through H) until the next boundary edge is reached. Two stopping criteria apply to the turning process: we stop if the cyclic adjacency successor does not belong to H or if all edges incident to the boundary node have been considered. The edge e^{rev} is a boundary edge into u. We turn its reversal e clockwise until the next boundary edge is reached. At node v the first stopping criterion applies and at a node which has no incident edge outside H the second stopping criterion applies.

but are not attachment nodes of H. Edge e_1 is the successor of e^{rev} on the cycle. We proceed in this way until the cycle is completely traced.

⟨*determine boundary cycle of component with lowest node y_0 = v*⟩≡

```
edge e0 = f;
node y0 = v;
edge e = f; // e1 was set to B.cyclic_adj_succ(f) above
do { while ( comp_num[B.cyclic_adj_succ(e1)] == comp_num[e] &&
             B.cyclic_adj_succ(e1) != e )
     { e1 = B.cyclic_adj_succ(e1); }
     cycle_edges.append(e1);
     e = B.reversal(e1);
     e1 = B.cyclic_adj_succ(e);
} while ( e != e0 );
```

We next show how to compute a leaf labeled $k+1$ in the part of B opposite to H at y_0 in constant time. Constant time is needed since y_0 can be the lowest numbered node of many biconnected components.

The part of B opposite to H at y_0 consists of the root component of y_0 and all non-root components with respect to y_0 that do not contain H. We have computed above two children of y_0 (if they exist) that define mixed non-root components. A leaf labeled $k+1$ can be found in either the root component or in one of the mixed children that does not contain H. A non-root component does not contain H if the tree edge into the child does not belong to H.

⟨*compute leaf labeled k+1 in part opposite to y_0*⟩≡

```
spec_leaf_in_opposite_part = spec_leaf_in_root_comp[v];
for (int i = 1; i <= 2; i++)
{ node c = child[i][v];
  if ( spec_leaf_in_opposite_part == nil
       && c &&  comp_num[tree_edge[c]] != comp_num[e0] )
    spec_leaf_in_opposite_part = leaf[K_PLUS_1][c];
}
```

Obstructing Cycle with Four Alternating Attachments: We search for a cycle with four alternating attachments. By Lemma 11 there are two ways such a cycle may occur: The component opposite to y_0 contributes either a large leaf or a leaf labeled $k+1$. We therefore perform two searches. In the first search we set *y0_type* to OTHERS and let C_0 contribute a large leaf and in the second search we set *y0_type* to K_PLUS_1 and let C_0 contribute a leaf labeled $k+1$. The second search is only performed when *spec_leaf_in_opposite_part* is defined.

For an attachment y_i different from y_0 the part opposite to H at y_i is equal to the upper part of B with respect to y_i.

We store the four attachments in $y[0]$ to $y[3]$ and we store the selected leaf in the i-th component in $z[i]$.

⟨*obstructing cycle with four alternating attachments*⟩≡

```
list<int> kinds;
kinds.append(OTHERS); kinds.append(K_PLUS_1);
int y0_type;
forall(y0_type, kinds)
{ array<node> y(4);
  y[0] = y0; y[1] = y[2] = y[3] = nil;
  array<node> z(4);
  if (y0_type == OTHERS)
  { z[0] = tB;
    current_case = "cycle with 4 attachments; y_0 connects to t";
  }
  else
  { z[0] = spec_leaf_in_opposite_part;
    current_case = "cycle with 4 attachments; y_0 connects to k + 1";
    if ( !spec_leaf_in_opposite_part ) break;
  }
  list_item it0 = cycle_edges.first();
  list_item it = cycle_edges.cyclic_succ(it0);
  int i = 1;
  while (it != it0)
  { node v = B.source(cycle_edges[it]);
    int kind = (i == 2 ? y0_type : 1 - y0_type);
    if ( leaf_in_upper_part[kind][v] )
    { y[i] = v;
```

```
        z[i] = leaf_in_upper_part[kind][v];
        i++;
      }
      if ( i == 4 )
      { ⟨build the Kuratowski graph⟩
        return;
      }
      it = cycle_edges.cyclic_succ(it);
    }
  }
```

Assume that we have found an obstructing cycle with four alternating attachments. We have the four attachments in $y[0]$ to $y[3]$ and the selected leaf in the i-th component in $z[i]$. Also *y0_type* tells us the type of the component C_0.

In the upper tree we need to take the subtree spanned by the two large leaves and node $k+1$.

⟨*build the Kuratowski graph*⟩≡

```
  translate_to_G(cycle_edges,B); K.conc(cycle_edges);
  list<edge> join_edges;
  int i;
  for (i = 0; i < 4; i++)
  { join(y[i],z[i],z[i],tree_edge,B,join_edges);
    translate_to_G(join_edges,B); K.conc(join_edges);
  }
  // subtree of T_t spanned by k+1 and two large leaves.
  if (y0_type == OTHERS) i = 0; else i = 3;
  join(B[z[i]],B[z[1]],B[z[2]],up_tree_edge,G,join_edges);
  K.conc(join_edges);
  check_before_return(G,K,st_num,leaf,tree_edge,dfs_num,k,
                      B,st_numB,sB,current_case);
```

Obstructing Biconnected Component with Three Mixed Opposing Parts: For case (2) we need that the component opposite to y_0 is mixed and that there are y_a, y_b such that C_a and C_b are mixed.

⟨*obstructing cycle with three mixed attachments*⟩≡

```
  array<node> y(3);
  array<node> spec_leaf_opposing(3);
  array<node> other_leaf_opposing(3);
  y[0] = y0;
  spec_leaf_opposing[0] = spec_leaf_in_opposite_part;
  other_leaf_opposing[0] = tB;
  int i = 1;
  list_item it0 = cycle_edges.first();
  list_item it = cycle_edges.cyclic_succ(it0);
```

```
while (it != it0)
{ node v = B.source(cycle_edges[it]);
  if ( leaf_in_upper_part[OTHERS][v] && leaf_in_upper_part[K_PLUS_1][v])
  { y[i] = v;
    spec_leaf_opposing[i] = leaf_in_upper_part[K_PLUS_1][v];
    other_leaf_opposing[i] = leaf_in_upper_part[OTHERS][v];
    i++;
  }
  if ( i == 3 )
  { ⟨obstructing cycle with three mixed attachments: extract Kuratowski⟩
    return;
  }
  it = cycle_edges.cyclic_succ(it);
}
```

It remains to extract the Kuratowski subgraph. We proceed as described in the proof of Lemma 9. We collect all edges shown in Figure 8.27 in K. K is not a Kuratowski graph yet, but is guaranteed to contain one.

⟨*obstructing cycle with three mixed attachments: extract Kuratowski*⟩≡

```
current_case = "obstructing cycle with three mixed attachments";
translate_to_G(cycle_edges,B); K.conc(cycle_edges);
list<edge> join_edges;
for(int j = 0; j <= 2; j++)
{ join(spec_leaf_opposing[j], other_leaf_opposing[j], y[j],
       tree_edge,B,join_edges);
  translate_to_G(join_edges,B); K.conc(join_edges);
}
node r = join(B[other_leaf_opposing[1]], B[other_leaf_opposing[2]],
              B[spec_leaf_opposing[0]], up_tree_edge,G,join_edges);
K.conc(join_edges);
join(r,r,t,up_tree_edge,G,join_edges);
K.conc(join_edges);
{ ⟨thin out K⟩ }
check_before_return(G,K,st_num,leaf,tree_edge,dfs_num,k,
                    B,st_numB,sB,current_case);
```

Thinning Out: K is now an appropriate set of edges in G. It might still be too big. We want to thin it out so that only a $K_{3,3}$ or a K_5 remains. This is easy to do. We construct an auxiliary graph AG, which has a node for each node of G that has degree three or more in K and which has an edge for each path in K connecting two such nodes and having only intermediate nodes of degree two. We associate with every edge of AG the path in G represented by it.

AG is a small graph; in fact, it has at most twelve nodes. We call the quadratic version of the Kuratowski algorithm to find a Kuratowski subgraph of AG and then translate is back to G.

⟨*thin out K*⟩≡

```
node v; edge e;

edge_array<bool> in_K(G,false);
node_array<int> deg_in_K(G,0);
forall(e,K)
{ in_K[e] = true;
  deg_in_K[G.source(e)]++; deg_in_K[G.target(e)]++;
}

GRAPH<node,list<edge> > AG;
node_array<node> link(G,nil);

forall_nodes(v,G)
  if ( deg_in_K[v] > 2 ) link[v] = AG.new_node(v);

forall_nodes(v,G)
{ if ( !link[v] ) continue;
  edge e;
  forall_inout_edges(e,v)
  { if ( in_K[e] )
    { // trace path starting with e
      list<edge> path;
      edge f = e; node w = v;
      while (true)
      { in_K[f] = false; path.append(f);
        w = G.opposite(w,f);
        if ( link[w] ) break;
        // observe that w has degree two and hence ...
        forall_inout_edges(f,w)
          if ( in_K[f] ) break;
      }
      edge e_new = AG.new_edge(link[v],link[w]);
      AG[e_new].conc(path);  // O(1) assignment
    }
  }
}

list<edge> el;
KURATOWSKI_SIMPLE(AG,el);

K.clear();
forall(e,el) K.conc(AG[e]);
```

There is a small optimization in the program above which we want to mention. Instead of

```
edge e_new = AG.new_edge(link[v],link[w]);
AG[e_new].conc(path);  // O(1) assignment
```

we could have written more elegantly

```
AG.new_edge(link[v],link[w],path);
```

The second version calls the copy constructor to construct a copy of *path* as the edge information of the new edge of AG, the first version concatenates *path* to the edge information of the new edge (which is initialized to the default value of lists, i.e., the empty list, by the new edge operation). Concatenation is a constant time operation. Concatenation empties *path* and this is all right. We have now completed the implementation of the linear time Kuratowski graph finder for biconnected graphs.

Arbitrary Graphs: We extend the algorithm to arbitrary graphs G. We first call the embedding algorithm to find out if G is planar. If it is, we are done.

So assume that G is non-planar. Then one of the biconnected components of G is non-planar. The idea is to search for a non-planar biconnected component of G and to call the algorithm of the preceding section for the biconnected component.

We give more details. A call *BICONNECTED_COMPONENTS*$(G, \mathit{comp\_num})$ returns the number *num_c* of biconnected components of G and computes for each edge of G the index of the biconnected component containing e.

We iterate over all edges of G and construct for every c, $0 \le c < \mathit{num\_c}$, the set $E[c]$ of edges in the component and the set $V[c]$ of nodes of the component. We determine the set $V[c]$ as the set of endpoints of edges in $E[c]$ and hence this set may contain duplicates.

When the edge and node sets of all biconnected components are determined, we iterate over all components. For each c, $0 \le c < \mathit{num\_c}$, we construct a copy of the component in H. The nodes and edges of H know their counterparts in G. Since $V[c]$ may contain duplicates, we maintain a node array *link*, in which we store for each node v in G, whether a copy of v has already been constructed in H. We reset *link* when the construction of H is completed. In this way the extraction of a biconnected component has cost proportional to the size of the component.

When the extraction of a component is completed, we test it for planarity. We break from the loop once a non-planar biconnected component is found.

If G is biconnected we take a short cut and make H a copy of G.

The identification of Kuratowski graphs is simplified if H is a map without self-loops and parallel edges. We therefore remove self-loops (or do not put them into H in the first place) and parallel edges, and we turn H into a map by adding edges. Every added edge is made to point to the same edge in G as its reversal. We then call *Kuratowski* to find a Kuratowski subgraph K of H. We turn K into a Kuratowski subgraph of G by replacing every edge by its counterpart in G.

⟨*Kuratowski graphs in arbitrary graphs*⟩≡

```
bool BL_PLANAR(graph& G, list<edge>& K, bool embed)
{
  if (BL_PLANAR(G, embed)) return true;
  edge_array<int> comp_num(G);
  int num_c = BICONNECTED_COMPONENTS(G,comp_num);
  GRAPH<node,edge> H;
  edge e;
```

```
    if ( num_c == 1 )
    { CopyGraph(H,G);
      Delete_Loops(H);
    }
    else
    { node_array<node> link(G,nil);
      array<list<edge> > E(num_c);
      array<list<node> > V(num_c);
      forall_edges(e,G)
      { node v = source(e);  node w = target(e);
        if (v == w) continue;
        int c = comp_num[e];  E[c].append(e);
        V[c].append(v); V[c].append(w);
      }
      int c; node v;
      for(c = 0; c < num_c; c++)
      { H.clear();
        forall(v,V[c]) if ( link[v] == nil ) link[v] = H.new_node(v);
        forall(e,E[c])
        { node v = source(e); node w = target(e);
          H.new_edge(link[v],link[w],e);
        }
        forall(v,V[c]) link[v] = nil;
        if (!BL_PLANAR(H,false)) break;
      }
    }
    K.clear();
    // H is a biconnected non-planar graph; we turn it into map
    Make_Simple(H);
    list<edge> R;
    H.make_map(R);
    forall(e,R) H[e] = H[H.reverse(e)];
    // auxiliary edges inherit original edge from their reversal
    Kuratowski(H,K);
    list_item it;
    forall_items(it,K) K[it] = H[K[it]];
    return false;
  }
```

8.7.4 ***Running Times***

Table 8.1 shows the running times of the functions discussed in this section. We used five kinds of graphs:

- Random planar maps with n nodes and $m = 2n$ uedges (P).
- Random planar maps with n nodes and $m = 2n$ uedges plus a $K_{3,3}$ on six randomly chosen nodes (P + $K_{3,3}$).

Graph	Gen	BL_PLANAR		Check	HT_PLANAR	
		T	T + J		T	T + J
P	0.76	1.59	1.82	0.23	2.6	4.18
	1.72	3.27	3.71	0.47	5.41	8.87
	3.47	6.67	7.43	0.95	11.38	19.22
P + $K_{3,3}$	0.97	1.1	5.66	0.17	2.54	–
	1.74	2.4	12.65	0.34	5.16	–
	3.56	5.47	20.01	0.69	11.02	–
P + K_5	1	0.98	5.72	0.16	2.61	–
	1.75	1.81	12.91	0.34	5.35	–
	3.58	3.26	22.06	0.67	10.86	–
MP	0.87	2.28	2.41	0.33	3.88	6.24
	1.5	4.59	4.84	0.66	7.81	12.98
	3.05	9.23	9.66	1.34	16.06	26.84
MP + e	0.87	1.26	5.47	0.23	1.05	–
	1.49	2.19	9.61	0.49	2.1	–
	3.06	5.87	23.81	0.96	4.28	–

Table 8.1 The running times of functions related to planarity: The column labeled Gen contains the time needed to generate the input graph. All other columns are as described in the text. We used $n = 2^i \cdot 5000$ for $i = 0$, 1, and 2. This table was generated with the program planarity_time in the demo directory.

- Random planar maps with n nodes and $m = 2n$ uedges plus a K_5 on five randomly chosen nodes (P + K_5).
- Maximal planar maps with n nodes (MP).
- Maximal planar maps on n nodes plus one additional edge between two random nodes that are not connected in G (MP + e).

We constructed the graphs using the generators discussed in Section 8.9 and then permuted the adjacency lists, so as to hide the graph structure.

We ran the following algorithms:

- *BL_PLANAR*(G), the Booth–Lueker planarity test (T) that gives a yes-no answer, but does not justify its answer.

- *BL_PLANAR*$(G, K, true)$, the Booth–Lueker planarity test that justifies its answers (T + J). If G is planar, it turns G into a planar map, and if G is non-planar, it exhibits a Kuratowski subgraph of G.
- The check whether the algorithm in the previous item worked correctly, i.e., the check *Genus*$(G) == 0$, if G is planar, and *CHECK_KURATOWSKI*(G, K), if G is non-planar.
- *HT_PLANAR*(G), the Hopcroft–Tarjan planarity test (T) that gives a yes-no answer, but does not justify its answer.
- *HT_PLANAR*$(G, K, true)$, the Hopcroft–Tarjan planarity test that justifies its answers (T + J). This algorithm was only run when the previous item declared G planar. The extraction of the Kuratowski subgraph would have taken hours, since there is no efficient Kuratowski finder implemented for the Hopcroft–Tarjan planarity test.

Exercises for 8.7

1 Show that the number of distinct permutations in which the virtual leaves of B_k can appear on the horizon is

$$2^C \cdot P,$$

where C is the number of biconnected components of B_k with three or more attachments and $P = \prod p_v!$ where the product is over all articulation points of B_k and p_v is the number of non-root components of B_k with respect to v.

2 Improve the running time of the simple search for Kuratowski subgraphs to $O(n^2)$. Make sure that your algorithm works in the presence of parallel edges and self-loops.

3 Let G be a graph, let $e = (a, b)$ be an edge of G, and let G' be obtained from G by contraction of e. Show that if G' contains a Kuratowski subgraph then G does.

4 We have shown in Lemma 9 that the existence of an obstruction in B_k guarantees the existence of the Kuratowski subgraph of G. Show that it guarantees that B_{k+1} has no bush form.

8.8 Manipulating Maps and Constructing Triangulated Maps

In the chapter on graphs we saw functions that allow us to add new nodes and edges to a graph G. In particular,

```
edge G.new_edge(node v, node w)
```

adds a new edge (v, w) to G and returns it. The edge is appended to *out_edges*(v) and to either *in_edges*(w) (if G is directed) or *out_edges*(w) (if G is undirected).

In this chapter the cyclic ordering of the adjacency lists plays a crucial role and hence we need much finer control over the positions where edges are inserted into adjacency lists. The following function gives full control:

```
edge G.new_edge(edge e1, edge e2,
                int d1 = LEDA::after, int d2 = LEDA::after)
```

adds a new edge $x = (v, w)$ to G, where $v = \mathit{source}(e1)$ and $w = \mathit{target}(e2)$, and returns the new edge. The new edge is inserted before or after edge *e1* into *out_edges*(v) as directed by *d1*. If G is directed, it is also inserted before or after edge *e2* into *in_edges*(w) as directed by *d2*. If G is undirected, it is also inserted before or after edge *e2* into *out_edges*(w) as directed by *d2*. The constants *LEDA*:: *after* and *LEDA*:: *before* are predefined constants.

If control about the position of insertion is needed at only one endpoint of the edge (or if the new edge is the first edge incident to a node) the functions

```
edge G.new_edge(edge e, node w, int dir = LEDA::after)
edge G.new_edge(node v, edge e, int dir = LEDA::after)
```

should be used. The former function adds a new edge $x = (\mathit{source}(e), w)$ to G. x is inserted before or after edge e into *out_edges*$(\mathit{source}(e))$ as directed by *dir* and appended to *in_edges*(w) (if G is directed) or *out_edges*(w) (if G is undirected). The operation returns the new edge x. If G is undirected we must have $\mathit{source}(e) \neq w$. The latter function is symmetric to the former.

Related to the *new_edge* function is the *move_edge* function. The call

```
G.move_edge(edge e, node v, node w)
```

requires that e is an edge of G. It makes v the source of e and w the target of e. For all versions of the *new_edge* function mentioned above, there is a corresponding version of the *move_edge* function, which takes the edge to be moved as an additional argument. The effect of *move_edge*(e, v, w) is similar, but distinct to the combined effect of *del_edge*(e) followed by *new_edge*(v, w). The effect is similar as e ceases to make the connection between its old source and target and as there is now an edge from v to w. The effect is distinct, as *move_edge* moves an already existing edge (which may for example have associated entries in edge arrays) and *new_edge* creates a new edge.

For maps it is frequently convenient to add an edge and its reversal in a single operation.

```
edge M.new_map_edge(edge e1, edge e2)
```

inserts a new edge $e = (\mathit{source}(e1), \mathit{source}(e2))$ after *e1* into the adjacency list of *source*$(e1)$ and the reversal to e after *e2* into the adjacency list of *source*$(e2)$. The following function splits a uedge in a map M.

```
edge M.split_map_edge(edge e)
```

splits edge $e = (v, w)$ and its reversal $r = (w, v)$ into edges (v, u), (u, w), (w, u), and (u, v), where u is a new node. It returns the edge (u, w).

We give an application of the functions above. We show how to *triangulate a map*. Let M be a map. The task is to add edges to M such that:

- the genus is not increased, in particular, a plane map stays plane, and

- every face cycle of the resulting map consists of at most three edges.

Both items are easy to achieve. As long as M is not connected we take any two nodes v and w in distinct components and join them by a uedge. This increases the number of edges by two, decreases the number of components by one, and either decreases the number of isolated nodes by two and increases the number of face cycles by one, or decreases the number of isolated nodes by one and leaves the number of face cycles unchanged, or leaves the number of isolated nodes unchanged and decreases the number of face cycles by one. In either case the genus is unchanged.

So assume that M is connected. As long as there is a face cycle consisting of four or more edges, we consider any such face cycle C and two nodes v and w on C that are not neighbors on C, say

$$C = [\dots, e_2, v, e_4, \dots, e_3, w, e_1, \dots].$$

We split C by adding edges (v,w) and (w,v). The edge (v,w) is added after e_4 to the list of out-edges of v and the edge (w,v) is added after e_1 to the list of out-edges of w; this is the reverse of the operation illustrated in Figure 8.11. Adding the two edges increases the number of face cycles by one; thus the genus is not changed.

We use the triangulation routine as a subroutine in our straight line drawing routine for planar graphs. The straight line drawing routine assumes that its input is a triangulated graph without parallel edges. We therefore have to make sure that the triangulation routine does not introduce parallel edges. Unfortunately, when face cycles are split independently, parallel edges may be introduced. We want to avoid this.

- If the genus of M is zero then no new edge is parallel to another edge of the graph (new or old).

Christian Uhrig and Torben Hagerup suggested a triangulation algorithm that achieves all three items above. Their algorithm runs in linear time $O(n+m)$. The algorithm steps through the nodes of M. For each node v, it triangulates all faces incident on v. For each node v, this consists of the following:

First, the neighbors of v are marked. During the processing of v, a node will be marked exactly if it is a neighbor of v.

Then the faces incident on v are processed in any order. A face with boundary $[v = x_1, x_2, \dots, x_n]$ is triangulated as follows: if $n \le 3$, nothing is done. Otherwise,

(1) if x_3 is not marked, a uedge $\{x_1, x_3\}$ is added, x_3 is marked, and the same strategy is applied to the face with boundary $[x_1, x_3, x_4, ..., x_n]$.

(2) if x_3 is marked, a uedge $\{x_2, x_4\}$ is added, and the same strategy is applied to the face with boundary $[x_1, x_2, x_4, x_5, ..., x_n]$.

When all faces incident to v are triangulated, all neighbors of v are unmarked.

The algorithm just described clearly triangulates all face cycles. We need to show that it does not introduce parallel edges.

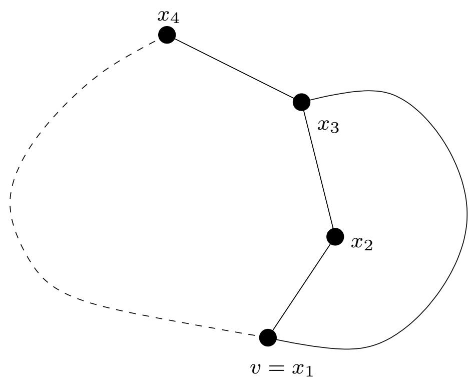

Figure 8.32 x_1, x_2, x_3, and x_4 are consecutive nodes on a face and the uedge $\{x_1, x_3\}$ exists. Then $\{x_2, x_4\}$ cannot exist.

During the processing of a node v, the marks on neighbors of v clearly prevent the addition of a parallel edge with endpoint v. After the processing of v, such an edge is not added because all faces incident on v have been triangulated. This takes care of the edges added in (1).

Whenever a uedge $\{x_2, x_4\}$ is added in step (2), the presence of a uedge $\{x_1, x_3\}$ implies that x_2 and x_4 are incident on exactly one common face, namely the face currently being processed, see Figure 8.32. Hence another edge $\{x_2, x_4\}$ will never be added.

The linear running time can be seen as follows. The time to process a node v is proportional to the degree of v plus the number of edges added during the processing of v. The total running time is therefore proportional to $O(n + m')$ where m' is the number of edges in the final graph. The number of uedges in the final graph is at most $3n$ by Lemma 3.

The following program implements the algorithm. We first add edges to make the graph connected, then make sure that all reversal informations are properly set, and finally add edges to triangulate the graph.

⟨*triangulate.c*⟩≡

```
list<edge> graph::triangulate_map()
{ node v;
  edge x, e, e1, e2, e3;
  list<edge> L;
  ⟨add edges to make the graph connected⟩
  if ( !make_map() )
  error_handler(1,"TRIANGULATE_PLANAR_MAP: graph is not a map.");
  node_array<int>  marked(*this,0);
  forall_nodes(v,*this)
  { list<edge> El = adj_edges(v);
    // mark all neighbors of v
```

```
      forall(e1,El) marked[target(e1)] = 1;
      ⟨process faces incident to v⟩
      //unmark all neighbors of v
      node w;
      forall_adj_nodes(w,v) marked[w] = 0;
    } // end of stepping through nodes
   return L;
  }
```

The two sub-steps are both fairly easy to implement. For the first sub-step we call COMPONENTs to determine the number of connected components and to label each node with its component number. If there is more than one component, we create an array *still_disconnected* with index set $[0 \mathinner{..} c-1]$, where c is the number of connected components. For each component except the one which contains s, the first node of G, we state that the component still needs to be connected with the component containing s. We then iterate over all nodes. Whenever we encounter a node v whose component still needs to be connected with s, we add the uedge $\{v, s\}$, and record that the component of v is now connected with the component of s.

⟨*add edges to make the graph connected*⟩≡

```
  node_array<int>  comp(*this);
  int c = COMPONENTS(*this, comp);
  if ( c > 1 )
  { node s = first_node();
    array<bool> still_disconnected(c);
    for (int i = 0; i < c; i++)
      still_disconnected[i] = ( i == comp[s] ? false : true);
    forall_nodes(v,(*this))
    { if ( still_disconnected[comp[v]] )
      { set_reversal(e1 = new_edge(s,v), e2 = new_edge(v,s));
        L.append(e1); L.append(e2);
        still_disconnected[comp[v]] = false;
      }
    }
  }
```

The faces incident to a node v are processed as described above. We store three consecutive edges of the face in *e1*, *e2*, and *e3*, respectively. If either of the three edges ends in v, the face cycle has length at most three and we are done.

So assume otherwise and let w be the endpoint of *e2*.

If w is not marked, we mark w and add the uedge $\{v, w\}$ inside the current face, i.e., we add the edge (w, v) after *e3* to $A(w)$ and we add the edge (v, w) after *e1* to $A(v)$. Also (v, w) becomes the new *e1*, *e2* becomes *e3*, and *e3* becomes the face cycle successor of *e2*.

If w is marked, we add the uedge $\{\mathit{source}(e2), \mathit{target}(e3)\}$ inside the current face, i.e., after edge *e2* at *source*(*e2*) and after the face cycle successor of *e3* at *target*(*e3*).

⟨*process faces incident to v*⟩≡
```
forall(e,El)
{
  e1 = e;
  e2 = face_cycle_succ(e1);
  e3 = face_cycle_succ(e2);
  if (target(e1) == v || target(e2) == v || target(e3) == v) continue;
  while (target(e3) != v)
  { node w = target(e2);
    if ( !marked[w] )
    { // we mark w and add the uedge {v,w}
      marked[w] = 1;
      L.append(x  = new_edge(e3,v));
      L.append(e1 = new_edge(e1,w));
      set_reversal(x,e1);
      e2 = e3;
      e3 = face_cycle_succ(e2);
    }
    else
    { //add the uedge {source(e2),target(e3)}
      e3 = face_cycle_succ(e3);
      L.append(x  = new_edge(e3,source(e2)));
      L.append(e2 = new_edge(e2,source(e3)));
      set_reversal(x,e2);
    }
  }//end of while
} //end of stepping through incident faces
```

8.9 Generating Plane Maps and Graphs

We discuss the generation of random plane maps and random plane graphs. We describe two methods to generate plane maps, a combinatorial method and a geometric method. We warn the reader that neither method generates plane maps according to the uniform distribution.

Combinatorial Constructions: The function
```
void maximal_planar_map(graph& G, int n);
```
generates a plane map with n nodes and $3n-6$ uedges, no self-loops and no parallel edges. The number of edges is the maximal possible, see Lemma 3, and, if $n \geq 3$, every face cycle is a triangle.

We give the implementation. If $n = 0$ we return the empty graph, if $n = 1$ we return the graph consisting of a single isolated node, and if $n = 2$ we return the graph consisting of two nodes and a single uedge. So let $n > 2$ and assume, that we have already constructed a maximal planar map with $n-1$ nodes. We select one of the existing edges, say e, at random and put a new node v into the face to the left of e.

Let $[e_1, e_2, e_3]$ be the face cycle containing e (when the third node is inserted the face cycle has length 2 instead of 3). For each i we add the edge $(\mathit{source}(e_i), v)$ to $A(\mathit{source}(e_i))$ after e_i and we append the edge $(v, \mathit{source}(e_i))$ to $A(v)$.

⟨*generate_planar_map.c*⟩≡

```
void maximal_planar_map(graph& G, int n)
{
  G.clear();
  if (n <= 0 ) return;
  node a = G.new_node();
  n--;
  if (n == 0) return;
  node b = G.new_node();
  n--;
  edge* E = new edge[n == 0? 2 : 6*n];
  E[0] = G.new_edge(a,b); E[1] = G.new_edge(b,a);
  G.set_reversal(E[0],E[1]);
  int m = 2;
  while (n--)
  { edge e = E[rand_int(0,m-1)];
    node v = G.new_node();
    while (target(e) != v)
    { edge x = G.new_edge(v,source(e));
      edge y = G.new_edge(e,v,LEDA::after);
      E[m++] = x; E[m++] = y;
      G.set_reversal(x,y);
      e = G.face_cycle_succ(e);
     }
   }
  delete[] E;
}
```

The function

```
void random_planar_map(graph& G, int n, int m);
```

generates a plane map with n nodes and $\min(m, 3n-6)$ uedges. It first generates a maximal plane map and then deletes a random set of uedges until the desired number of edges is obtained.

The functions

```
void maximal_planar_graph(graph& G, int n);
void random_planar_graph( graph& G, int n, int m);
```

first construct a plane map with the same parameters and then keep only one of the edges comprising each uedge.

Geometric Constructions: Geometry is a rich source of planar graphs. A simple way to generate a planar map is to choose n random points in the plane and to triangulate the resulting point set. We will see how to triangulate a point set in Section **??**. Alternatives are to compute the Delaunay triangulation of a set of random points, see Section **??**, or to choose a random set of segments and to compute the arrangement of the segments, see Section **??**.

The functions

```
void triangulation_map(graph& G, int n);
void triangulation_map(graph& G, node_array<double>& xcoord,
                       node_array<double>& ycoord, int n);
void triangulation_map(graph& G, list<node>& outer_face,
                       node_array<double>& xcoord,
                       node_array<double>& ycoord,
                       int n);
```

choose n random points in the unit square and set G to some triangulation. G will be a plane map. The first function only returns the triangulation, the second function also returns the point coordinates, and the third function also returns the list of vertices lying on the convex hull (in clockwise order).

The function

```
void random_planar_map(graph& G, node_array<double>& xcoord,
node_array<double>& ycoord, int n, int m);
```

first constructs a triangulated planar map and then deletes all but m edges.

All functions above are also available with *map* replaced by *graph* in the function name. The modified functions keep only one edge of each uedge.

8.10 Faces as Objects

The face cycles of maps played an important role in the preceding sections. It is therefore only natural to introduce them as a type of their own. For succinctness, we use the type name *face*.

8.10.1 *Concepts*

The operation

```
M.compute_faces()
```

computes the set of face cycles of the map M; the function aborts if M is not a map. After this operation and till the next modification of M by a *new_node*, *new_edge*, *del_node*, or *del_edge* operation, the face cycles of M are available in much the same way as the edges and nodes of M are available.

For example,

```
int        M.number_of_faces();
list<face> M.all_faces();
```

return the number of faces and the list of all faces of M, respectively. If f is a face, the predecessor and successor face of f in the list of all faces is returned by

M.succ_face(f) and *M.pred_face*(f), respectively, and the first and last face in the list of all faces is returned by *M.first_face*( ) and *M.last_face*( ), respectively. The four functions just mentioned return *nil* if the requested object does not exist. The iteration statement

```
forall_faces(f,M)
```

iterates over all face cycles of M.

The function

```
face M.face_of(edge e)
```

returns the face cycle of M which contains the edge e and the functions

```
list<edge> M.adj_edges(face f)
edge       M.first_face_edge(face f)
int        M.size(face f)
```

return the list of all edges in the face cycle f, the first edge in this cycle, and the number of edges in the face cycle, respectively. The iteration statement

```
forall_face_edges(e,f)
```

iterates over all edges e in the face cycle f.

For a node v, the function

```
list<face> M.adj_faces(node v)
```

returns the list of faces incident to v. More precisely, if $A(v) = [e_0, e_1, \ldots, e_{k-1}]$ is the list of edges out of v then the list $[\mathit{face\_of}(e_0), \ldots, \mathit{face\_of}(e_{k-1})]$ is returned.

Similarly, for a face f, the function

```
list<node> M.adj_nodes(face f)
```

returns the list of all nodes of M incident to f. More precisely, if $f = [e_0, e_1, \ldots, e_{k-1}]$, the list $[\mathit{source}(e_0), \ldots, \mathit{source}(e_{k-1})]$ is returned.

There is a small number of update operations which do not destroy the list of faces of a map. The operation

```
edge M.split_face(edge e1, edge e2)
```

inserts the edge $e = (\mathit{source}(e_1), \mathit{source}(e_2))$ and its reversal into M and returns e. The edges e_1 and e_2 must belong to the same face. This face cycle is split into two by the operation by inserting e after e_1 into the list of edges out of $\mathit{source}(e_1)$ and by inserting e^R after e_2 into the list of edges out of $\mathit{source}(e_2)$. The operation

```
face M.join_faces(edge e)
```

deletes the edge e and its reversal from M and updates the list of faces accordingly. Let f and g be the face cycles containing e and e^R, respectively. Assume first that $f \neq g$. If both f and g consist of a single edge[22] then the number of face cycles goes down by two and *nil* is returned. If at least one of f or g consists of more than one edge, then f and g are joined into a single face and this face is returned. When we coined the name for the operations we assumed that the latter case would be the "normal" use of the operation. Assume next that $f = g$. If f consists of exactly two edges, namely e and e^R then the number of face cycles goes down by one and *nil* is returned. If f consists of at least three edges and either e or e^R is the face cycle successor of the other then the number of face cycles is unchanged and f is

[22] This case occurs, for example, in a graph with one node and one uedge.

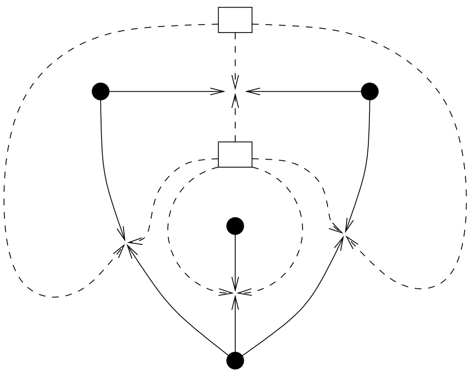

Figure 8.33 The dual of our map M_0. The dual has two nodes (shown as squares) and four uedges (drawn dashed).

returned. Finally, if neither e nor e^R is the face cycle successor of the other, then the number of faces goes up by one and one of the new faces is returned.

8.10.2 ***The Dual of a Map***

The (combinatorial) dual of a map M is another map D, see Figure 8.33:

- D has one node for each face cycle of M. More precisely, the nodes of D and the face cycles of M are in one-to-one-correspondence. We use $d(f)$ to denote the node of D corresponding to the face cycle f of M.
- D has one edge for each edge of M. Let e be any edge of M, let f be the face cycle containing e, and let g be the face cycle containing e^R. Then D contains the edge $d(e) = (d(f), d(g))$.
- Let $f = [e_0, e_1, \ldots, e_{k-1}]$ be a face cycle of M. Then the cyclic adjacency list of the node $d(f)$ of D is equal to $[d(e_0), d(e_1), \ldots, d(e_{k-1})]$.

The following program computes the dual D of a map M. We first compute the face cycles of M. We then put a node into D for each face cycle of M and record the correspondence in a *face_array*<*node*>. We then iterate over all face cycles of M and for each face cycle over the edges comprising the face cycle. For each edge we constructs its dual and record the correspondence. Observe that the edges incident to any dual node are constructed in the order in which they are supposed to appear in the adjacency list of the dual node. Finally, we establish the reversal information of all dual edges.

⟨*dual.c*⟩≡
```
void graph::dual_map(graph& D) const
{ D.clear();
  graph& M = *((graph*)this); // cast away the const
  M.compute_faces();
```

```
    face f; edge e;
    face_array<node> dual(M);
    forall_faces(f,M) dual[f] = D.new_node();
    edge_array<edge> dual_edge(M);
    forall_faces(f,M)
    { node df = dual[f];
      forall_face_edges(e,f)
      { face g = M.face_of(M.reversal(e));
        dual_edge[e] = D.new_edge(df,dual[g]);
      }
    }
    forall_edges(e,M)
      D.set_reversal(dual_edge[e],dual_edge[M.reversal(e)]);
  }
```

8.10.3 ***Faces of Planar Maps***

There are two functions that deal with faces of planar maps. The function

```
void M.make_planar_map()
```

assumes that M is a bidirected graph. It first calls $M.make\_map(\ )$ to turn M into a map. It then calls $PLANAR(M, true)$ to turn M into a plane map. It finally calls $M.compute\_faces(\ )$ to compute the face cycles of M.

The function

```
list<edge> M.triangulate_planar_map()
```

calls $M.triangulate\_map(\ )$ followed by $M.compute\_faces(\ )$ and returns the list of edges added to M by the former call.

Exercise for 8.10

1 Is the dual of the dual of a map M isomorphic to M? Give a counterexample. Under which conditions does the claim hold? State and prove a lemma.

8.11 Embedded Graphs as Undirected Graphs

The reader may wonder about the use of directed graphs in this chapter. After all, in maps we always combine a pair of directed edges into a uedge. We chose bidirected graphs to represent maps mainly for two reasons.

Although maps are basically undirected graphs, the two orientations of an undirected edge play a major role in the functions operating on maps. In particular, the face cycle successor of an edge and the reversal of an edge are “directed concepts” and hence would require additional arguments if maps were realized by undirected graphs. For example, one could distinguish the two orientations of an undirected edge by specifying a node to indicate the source node of the oriented edge. This would, however, not work for self-loops.

The second reason is that maps are frequently constructed incrementally and that the two orientations of an edge are constructed at different moments of time. We saw one example already in the program *dual_map* that constructs the dual of a map. Such constructions are difficult to implement with a representation that can only represent maps. The problem is that we arrive at a map at the end of the construction process but have no map during the construction process.

Our choice of directed graphs to represent maps wastes space, since the two edges comprising a uedge are stored in two lists at each endpoint of the uedge. One list for each endpoint would suffice for most functions presented in this chapter.

8.12 Order from Geometry

The following problem arises frequently. A graph is constructed by drawing it in a *GraphWin* and the combinatorial structure of the graph is supposed to reflect the drawing, i.e., for every node v the cyclic order of $A(v)$ is supposed to agree with the counter-clockwise order of the edges out of v in the drawing.

Let us be more precise. For every edge e let $d(e)$ be a vector (not necessarily, non-zero) in the plane. We define an order on two-dimensional vectors. For a non-zero vector d let $\alpha(d)$ be the angle between the positive x-axis and d, i.e., the angle by which the positive x-axis has to be turned in counter-clockwise direction until it aligns with d. A vector d_1 *precedes* a vector d_2 if $\alpha(d_1) < \alpha(d_2)$ and a vector d_1 is *equivalent* to a vector d_2 if $\alpha(d_1) = \alpha(d_2)$. The zero vector precedes all other vectors. The implementation of this order on vectors is discussed in Chapter **??** on geometry kernels.

The functions
```
bool SORT_EDGES(graph &G,
                const edge_array<NT>& dx, const edge_array<NT>& dy)
bool SORT_EDGES(graph &G,
                const node_array<NT>& x, const node_array<NT>& y)
```
reorder all adjacency lists in non-decreasing order of the vectors $d(e)$, $e \in E$. For the first function, the vector associated with an edge e is $(dx[e], dy[e])$, and for the second function, the vector associated with an edge $e = (v, w)$ is $(x[w]-x[v], y[w]-y[v])$.

The functions return *true* if G is a plane map after the reordering. When will this be the case? Assume that G is a map and that the vectors $d(e)$ come from a planar drawing of G, i.e., $d(e)$ is a vector tangent to the image of e as it leaves its source. If G has no self-loops and no parallel edges[23] then G will be a plane map after the call of *SORT_EDGES*. In fact, it will be a plane map for which the given drawing is an order-preserving embedding.

[23] Observe that sorting edges by angle leaves the relative order of self-loops and the relative order of parallel edges undefined.

We next give an application of the function SORT_EDGES to the task described in the introductory paragraph. The goal is to deduce a plane map from a straight line drawing of the map. Assume that *gw* is a GraphWin with an associated graph *G*, i.e., defined by

⟨*gw_sort_edges_demo*⟩≡
```
graph G;
⟨gw_sort_edges_demo: auxiliary functions⟩
int main()
{ GraphWin gw(G,"Plane Map from Geometry");
  gw.set_init_graph_handler(init_handler);
  gw.set_new_edge_handler(new_edge_handler);
  gw.set_del_edge_handler(del_edge_handler);
  gw.set_new_node_handler(new_node_handler);
  gw.set_del_node_handler(del_node_handler);
  gw.set_move_node_handler(move_node_handler);
  gw.set_directed(true);
  gw.display();
  gw.add_help_text("gw_sort_edges_demo");
  gw.display_help_text("gw_sort_edges_demo");
  gw.edit();
  return 0;
}
```

We define an auxiliary function *sort* that queries for each node *v* of *G* its position in *gw* and then calls SORT_EDGES. We call *sort* whenever an edge is added to the graph (and hence the new edge handler is called) or if a new graph is read in by *gw* (and hence the init handler is called). When an edge is added, we also add the reversal to make sure that we deal with a map.

The effect of the call of *sort* is to rearrange the adjacency lists according to the counter-clockwise order in which the edges incident to any node appear in the drawing. We print the graph at the end of sort in order to allow a visual comparison between the drawing and the representation of the graph. The graph will be a plane map as long as the drawing is a planar embedding.

⟨*gw_sort_edges_demo: auxiliary functions*⟩≡
```
void sort(GraphWin& gw)
{
  node_array<double> x(G), y(G);
  node v;
  forall_nodes(v,G)
  { point p = gw.get_position(v);
    x[v] = p.xcoord(); y[v] = p.ycoord();
  }
  SORT_EDGES(G,x,y);
  cout << "\n\nThe adjacency lists are:\n";
  G.print();
}
void init_handler(GraphWin& gw)
{ list<edge> L;
  G.make_map(L);
  sort(gw);
}
void new_edge_handler(GraphWin& gw, edge e)
{ G.set_reversal(e,gw.new_edge(G.target(e),G.source(e)));
  sort(gw);
}
bool del_edge_handler(GraphWin& gw, edge e)
{ gw.del_edge(G.reversal(e)); return true; }
void new_node_handler(GraphWin& gw,node)   {}
void del_node_handler(GraphWin& gw)        {}
void move_node_handler(GraphWin& gw,node v) { sort(gw); }
```

We will see more functions that relate geometry and graphs in Chapter **??** on geometric algorithms.

Exercises for 8.12

1 Extend the gw_drawing_demo.c such that it can also cope with edges that contain bends.

2 Write a function that checks whether the geometric positions assigned to the nodes of a map define a straight line embedding of the map. Hint: Read Section **??** on line segment intersection before working on this exercise.

8.13 Miscellaneous Functions on Planar Graphs

There are many problems that are simpler for planar graphs than for arbitrary graphs. We collect two in this section.

8.13.1 *Five Coloring*

Every planar graph can be four-colored, i.e., the nodes of the graph can be labeled with the integers 1 to 4 such that any edge connects two nodes of distinct color. We have not implemented a four coloring algorithm but only a five coloring algorithm.

The function

```
void FIVE_COLOR(graph& G, node_array<int>& C);
```

attempts to color the nodes of G using five colors, more precisely, it computes for every node v a color $C[v] \in \{1, \ldots, 5\}$, such that $C[\mathit{source}(e)] \neq C[\mathit{target}(e)]$ for every edge e. The function runs in linear time and is guaranteed to succeed if G is planar and contains no self-loops and no parallel edges[24].

We sketch how the algorithms works. In a planar graph there is always a node with at most five neighbors (Lemma 3). Let v be a node with at most five neighbors. If v has less than five neighbors, we recursively five-color the graph $G \setminus v$ and then use a color for v which is not used by any of its neighbors. If v has degree 5, we have to work slightly harder. We observe that there must be two neighbors of G which are not connected by an edge (otherwise the neighbors of v would form a complete graph on five nodes; this is, however, impossible in a planar graph by Lemma 3). Let w and z be two neighbors of v that are not connected by an edge. We remove v and merge w and z into a single node. This can be done without destroying planarity as Figure 8.34 shows. When merging w and z we also delete any parallel edges which may result from the merging process. We five-color the resulting graph G' recursively. In order to obtain a coloring of G we unmerge w and z, give w and z the color of the node that represented them both in G', and give v a color which is not used on its neighbors.

To obtain linear running time is slightly tricky and we leave it for the exercises.

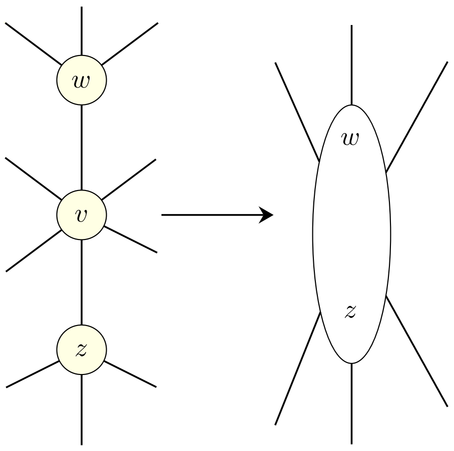

Figure 8.34 Merging the neighbors w and z of v.

8.13.2 *Independent Sets of Small Degree*

An independent set in a graph G is a set I of nodes no two of which are connected by an edge. A five coloring of a graph yields an independent set of size at least $n/5$, since at least one of the colors is used on at least $n/5$ of the nodes and since all edges have their endpoints in different color classes. Sometimes, it is desirable to have an independent set all of whose nodes have small degree.

[24] Self-loops are clearly an obstruction to colorability. Parallel edges are no “real” problem; it is just that our algorithm is not able to handle them.

The function
`void INDEPENDENT_SET(const graph& G, list<node>& I)`
computes an independent set I all of whose nodes have degree at most 9. If G is planar and has no parallel edges, it is guaranteed that $|I| \geq n/6$. The algorithm is due to David Kirkpatrick and Jack Snoeyink [KS97] and is extremely simple and elegant.

The algorithm starts by removing all nodes that have degree 10 or more. It then repeatedly chooses a node v of smallest degree, adds v to I, and removes v and its neighbors from G.

We describe an implementation. We start by making an isomorphic copy H of G; H is of type *GRAPH<node, edge>*, and each node v of H stores in $H[v]$ the node of G to which it corresponds. We saw the implementation of *CopyGraph* in Section **??**. We will work on H.

We delete all self-loops from H and turn H into a map. Recall that turning a graph into a map pairs a maximum number of edges and adds reversals for the unpaired edges. After turning H into a map, each edge is part of a uedge.

We then determine all nodes of degree at least 10 and delete all such nodes.

Next we collect all nodes of H of degree i, $0 \leq i \leq 9$ in a linear list $LD[i]$. In the course of the algorithm the lists $LD[i]$ may contain nodes that were already deleted from H. We need to be able to identify those nodes and therefore maintain an array *node_of_H*.

The construction of the independent set can now begin. As long as H is not empty, we select a node v from the lowest indexed non-empty list. We continue the selection process until we select a node that belongs to the current H. We add $H[v]$ to I (recall that $H[v]$ is the node in G that corresponds to v), and we delete v and its neighbors from H; we do not remove them from the lists LD though (this could be done by maintaining an array *pos_in_LD* that stores for each node v the item in LD that contains v). We collect all neighbors of v in a list *affected_nodes* and add them to the lists LD according to their new degrees.

⟨*_independent_set*⟩≡
```
void INDEPENDENT_SET(const graph& G, list<node>& I)
{ I.clear();
  GRAPH<node,edge> H;
  CopyGraph(H,G);
  node v; edge e;
  list<edge> E = H.all_edges();
  forall(e,E) { if (H.source(e) == H.target(e) ) H.del_edge(e); }
  H.make_map(E); // E is a dummy argument
  list<node> HD; // high degree nodes
  forall_nodes(v,H) if (H.degree(v) >= 10) HD.append(v);
  forall(v,HD) H.del_node(v);
  array<list<node> > LD(10);
  forall_nodes(v,H) LD[H.degree(v)].append(v);
  node_array<bool> node_of_H(H,true);
  while (H.number_of_nodes() > 0)
  { int i = 0;
    while (i < 10)
    { if ( LD[i].empty() ) { i++; continue; }
      v = LD[i].pop();
      if ( node_of_H[v] ) break;
    }
    I.append(H[v]);
    list<node> affected_nodes;
    forall_inout_edges(e,v)
    { node w = H.opposite(v,e);
      edge f;
      forall_inout_edges(f,w)
        affected_nodes.append(H.opposite(w,f));
      H.del_node(w); node_of_H[w] = false;
    }
    H.del_node(v); node_of_H[v] = false;
    forall(v,affected_nodes)
      if ( node_of_H[v] ) LD[H.degree(v)].append(v);
  }
}
```

Exercises for 8.13

1 Extend the function *FIVE_COLORING* so that it can handle parallel edges.

2 Implement the function FIVE_COLORING. Try to achieve linear running time.

3 Modify the implementation of INDEPENDENT_SET such that the lists *LD* contain only nodes of *H* and every node at most once.

4 A separator in a graph G is a set S of nodes of G such that removal of S decomposes G into two or more subgraphs none of which has more than $2n/3$ nodes. Planar graphs have separators of size $O(\sqrt{n})$ and there are linear time algorithms to compute them, see [Tar77] or [Meh84, IV.10]. Implement the planar separator theorem and provide it as a LEP.

Bibliography

[Bat] G. Di Battista. GD-Toolkit. Check the item “friends” of the LEDA-web page for a pointer.

[BETT94] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, 1994.

[BL76] K.S. Booth and G.S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using *PQ*-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.

[CNAO85] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *Journal of Computer and System Sciences*, 30(1):54–76, 1985.

[DETT98] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.

[dFPP88] H. de Fraysseix, J. Pach, and R. Pollack. Small sets supporting Fáry embeddings of planar graphs. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC’88)*, pages 426–433, 1988.

[EM98] P. Eades and P. Mutzel. Graph drawing algorithms. In M.J. Athallah, editor, *Algorithms and Theory of Computation Handbook*. CRC Press, 1998.

[Eul53] L. Euler. Demonstratio nonulaarum insignium proprietatum, quibus solida hedris planis inclusa sunt praedita. *Novi Comm. Acad. Sci. Petropol.*, 4:140–160, 1752/53.

[Eve79] S. Even. *Graph Algorithms*. Pitman, 1979.

[Fár48] I. Fáry. On straight line representations of planar graphs. *Acta. Sci. Math. (Szeged)*, 11:229–233, 1948.

[HMN96] C. Hundack, K. Mehlhorn, and S. Näher. A simple linear time algorithm for identifying Kuratowski subgraphs of non-planar graphs. Unpublished, 1996.

[HT74] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21:549–568, 1974.

[JMN] M. Jünger, P. Mutzel, and S. Näher. The AGD graph drawing library. http://www.mpi-sb.mpg.de/AGD/.

[Kar90] A. Karabeg. Classification and detection of obstructions to planarity. *Linear and Multilinear Algebra*, 26:15–38, 1990.

[KL93] D. Knuth and S. Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison-Wesley, 1993.

[KS97] D. Kirkpatrick and J. Snoeyink, 1997. personal communication.

[Kur30] C. Kuratowski. Sur le problème the courbes guaches en topologie. *Fundamenta Mathematicae*, 15:271–283, 1930.

[LEC67] A. Lempel, S. Even, and

I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Theory of Graphs, International Symposium, Rome*, pages 215–232, 1967.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer, 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.

[MMN94] K. Mehlhorn, P. Mutzel, and S. Näher. An implementation of the Hopcroft and Tarjan planarity test and embedding algorithm, 1994. available at the first author's WEB-page.

[NC88] T. Nishizeki and N. Chiba. *Planar Graphs: Theory and Algorithms*. Annals of Discrete Mathematics (32). North-Holland Mathematics Studies, 1988.

[Poi93] H. Poincaré. Sur la généralisation d'un theorem d'Euler relativ aux polyédres. *Comptes Rend. Acad. Sci. Paris*, 117:144–145, 1893.

[Ram94] N. Ramsey. Literate programming simplified. *IEEE Software*, 11:97–105, 1994.

[Tar77] R. Liptonand R. E. Tarjan. A separator theorem for planar graphs. In *Conference on Theoretical Computer Science, Waterloo*, pages 1–10, 1977.

[Whi73] A.T. White. *Graphs, Groups, and Surfaces*. North Holland, 1973.