The Left-Right Planarity Test

Ulrik Brandes ⋆

Department of Computer & Information Science, University of Konstanz

Abstract

A graph is planar if and only if it can be embedded in the plane without crossings. I give a detailed exposition of simple and efficient, yet poorly known algorithms for planarity testing, embedding, and Kuratowski subgraph extraction based on the left-right characterization of planarity.

Key words: Graph algorithms, planarity, algorithm review

1 Introduction

Two things appear to constitute the folklore about graph planarity testing:

(1) There are two main strands of linear-time algorithms, the *vertex-addition approach* pioneered by Lempel, Even, and Cederbaum (1967), and the *path-addition approach* pioneered by Hopcroft and Tarjan (1974).
(2) Both are a real challenge to understand, implement, and teach.

This is not a review of the exciting history of planarity testing at large, however, but of the lesser known *left-right approach*, which is seemingly different and usually associated with de Fraysseix and Rosenstiehl (1982). Even though the developments from its origins in Wu (1955) to its latest version in de Fraysseix, Ossona de Mendez, and Rosenstiehl (2006) and de Fraysseix (2008) is interesting in itself, my main goal here is to meet the apparent demand for an accessible exposition.

The left-right approach is remarkably elementary and does not require tricky data structures (e.g., Booth and Lueker 1976), a complicated embedding phase

⋆ I would like to thank Sabine Cornelsen, Giuseppe Di Battista, Daniel Kaiser, Martin Mader, and Maurizio Patrignani for helpful comments, suggestions, and corrections.

(e.g., Mehlhorn and Mutzel 1996), or even special treatment of biconnected components. Moreover, it was found to be extremely fast (Boyer, Cortese, Patrignani, and Di Battista, 2004) and can be augmented easily to return a Kuratowski subgraph if the input is not planar.

This work is motivated by the stark contrast between the elegance and simplicity of the left-right approach and its minimal adoption. It yields, I am convinced, the simplest linear-time planarity algorithms known to date, but to the best of my knowledge, there is not a single exposition or implementation independent from the original group of authors.

The absence of an easily readable, yet fully detailed description may be the main cause for its lack of popularity. In an attempt to remedy this situation, the original description of de Fraysseix, Ossona de Mendez, and Rosenstiehl (2006) is simplified with minor corrections, and it is extended by a new motivation, implementation-level pseudo-code, and more straightforward Kuratowski subgraph extraction. While the planarity test given here differs from the original paper, similar improvements have been introduced independently into the only previous implementation, available in PIGALE (de Fraysseix and Ossona de Mendez, 2002).

From the present description it should be possible to teach the algorithm in no more than two sessions of an advanced algorithms course. With a planar graph data structure art hand, transforming the pseudo-code into an implementation should be a matter of hours.

The remainder is organized such that readers solely interested in understanding the left-right approach can stop reading after Section 5. Therefore, only minimal background on graph planarity and the associated algorithmic problems is provided in Section 2. A new motivation for the left-right approach is given in Section 3, and the planarity characterization on which it is based in Section 4. The left-right algorithm for planarity testing and planar embedding is given in Section 5, including detailed pseudo-code. Kuratowski subgraph extraction for non-planar graphs is treated separately in Section 6, and the relation to other planarity criteria and algorithms as well as some notes on the history of the left-right approach are postponed until Section 7.

2 Planarity

We consider simple undirected graphs $G = (V, E)$, since directions, loops and multiple edges have no effect on planarity, and denote $n = n(G) = |V|$ and $m = m(G) = |E|$ throughout.

A *drawing* of a graph is a mapping of its vertices onto points in the plane, and of its edges onto curves connecting their endpoints. Where possible without confusion, we neglect the distinction between vertices, edges, etc., and their drawings. A drawing of a graph is *planar*, if edges do not intersect except at common endpoints. A graph is planar, if it admits a planar drawing.

A planar drawing divides the plane into connected regions, called *faces*. Each bounded face is an *inner* face, and the single unbounded one is called the *outer* face.

A *(combinatorial) embedding* consists of cyclic orderings of the incident edges for every vertex. An embedding is *realized* by a drawing, if the clockwise ordering of the edges around each vertex in the drawing agrees with the embedding. Note that an embedding represents an equivalence class of drawings that realize it. An embedding is planar, if it can be realized in a planar drawing.

Given a graph G, there are four major algorithmic problems related to planarity:

(1) Decide whether G is planar.
(2) If G is planar, find a planar embedding.
(3) If G is not planar, find a Kuratowski subgraph.
(4) Given a planar embedding of G, realize it in a planar drawing.

A Kuratowski subgraph is an inclusion-minimal subgraph certifying non-planarity. Since it is of less general interest, the topic is deferred to Section 6.

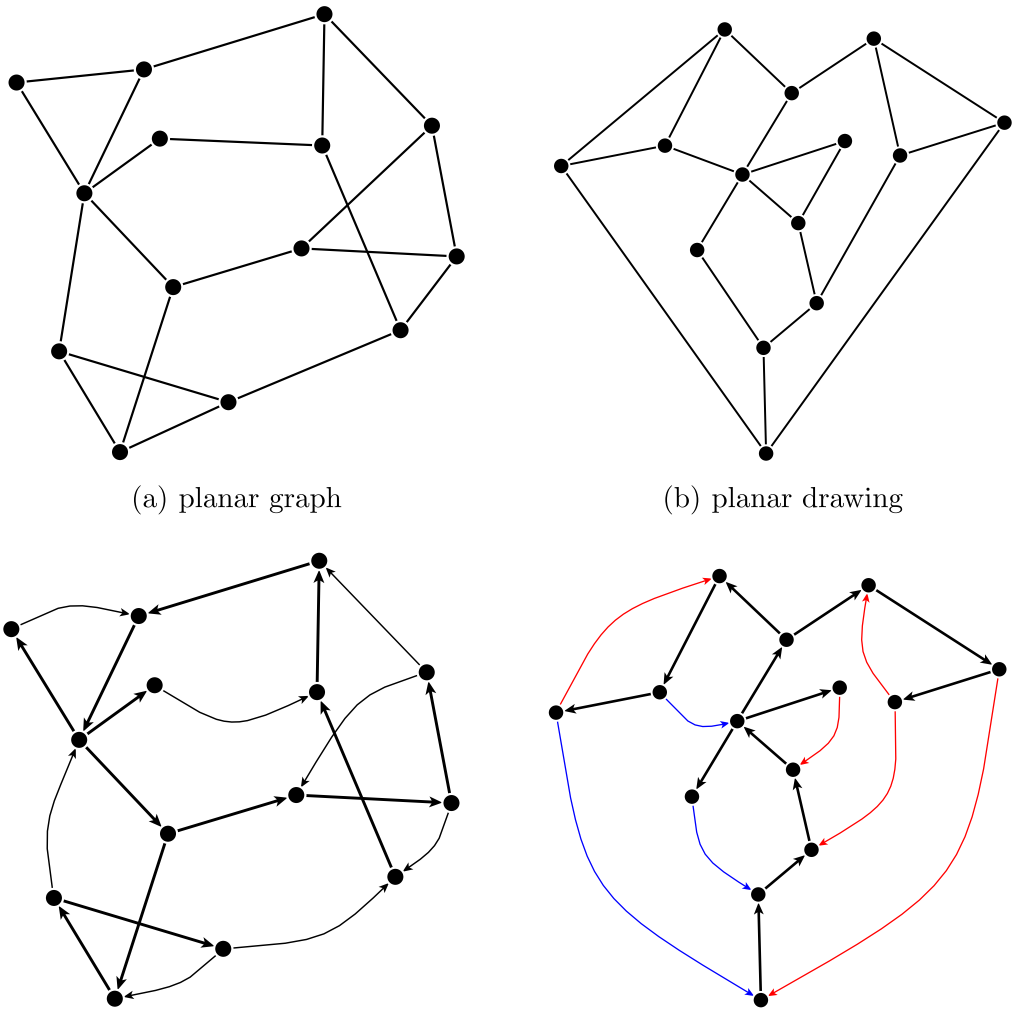
Our focus will be on the first two problems and we only note that, given a planar embedding, realizations may be subject to various criteria such as integer coordinates, straight-line edges, small area, polygonal edges with few bends and/or slopes, etc., and there are many algorithms for drawing planar graphs according to such criteria (see, e.g., Nishizeki and Rahman 2004).

The linear-time testing and embedding algorithm described in Section 5 is based on a rather intuitive criterion that is motivated and established in the next two sections.

3 Motivation

Since planarity is about the absence of crossings, cycles are the root cause of difficulties: cycles yield closed curves that disconnect regions of the plane, and one has to be careful about where to place which part of the graph.

There are only two significantly different ways to draw a simple cycle planarly,



(a) planar graph (b) planar drawing

(c) DFS orientation (d) LR partition

Fig. 1. Example of a planar graph (from Cai, Han, and Tarjan 1993). In both the planar and non-planar drawing, the same depth-first search (DFS) orientation is shown with thick tree edges and curved back edges. In any planar drawing the back edges can be partitioned into left and right, depending on whether their fundamental cycle is counterclockwise or clockwise. Note that the non-planar drawing contains self-intersecting fundamental cycles for both back edges entering the DFS root.

namely clockwise or counterclockwise. It turns out that fixing the orientation of some cycles may impose constraints on the choices for overlapping others via the ordering of edges around vertices. We will show that testing planarity amounts to deciding whether there is a consistent orientation of all cycles. Despite a potentially exponential number of cycles, this can be done efficiently, because constraints need not be resolved for all cycles, but only for a small set of cycles that represent the entire cycle structure.

Representative cycles are determined from a depth-first search as described next. This is followed by some apparent orientation constraints that also relate cycle orientations to embeddings. In Section 4, more precise versions of these constraints are proven to characterize planarity. The proof is constructive and immediately yields a planar combinatorial embedding, if one exists.

3.1 Depth-first search

The left-right planarity criterion is inherently related to depth-first search (DFS). Important aspects of this relation are hinted at in this section, and DFS terminology is introduced along the way.

Recall that a depth-first search on a connected undirected graph $\bar{G} = (V, \bar{E})$ yields a *DFS orientation* of $\bar{G}$, i.e. a directed graph $\vec{G} = (V, \vec{E})$ in which each undirected edge is oriented according to its traversal direction. Once the graph is oriented, we will only work with its directed version and hence neglect the distinction between $\bar{E}$ and $\vec{E}$. In the oriented graph, we denote by $E^+(v) = \{(v, w) \in E\}$ the set of all outgoing edges of $v \in V$, so that $E = \bigcup_{v \in V} E^+(v)$.

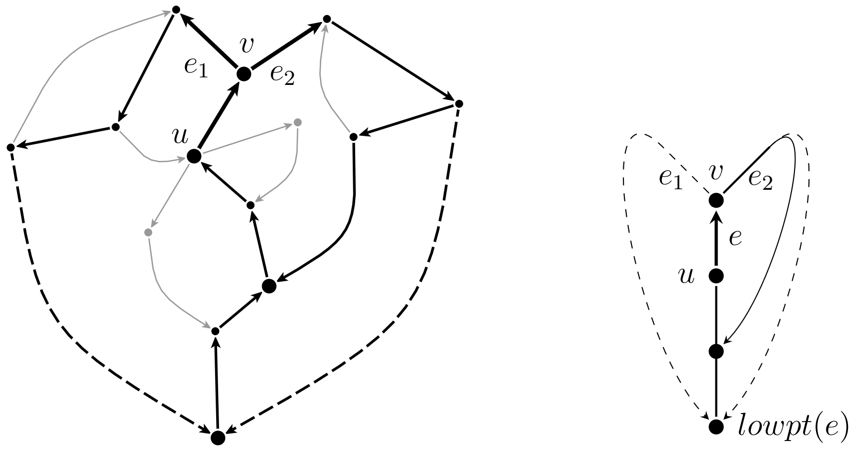
In addition to an orientation, a DFS traversal yields a bipartition of the edges into $E = T \uplus B$, where those in T are called *tree edges* and induce a rooted spanning tree (the *DFS tree*), and the non-tree edges in B are called *back edges*. See Fig. 3. We write $u \to v$ and $v \hookrightarrow w$ for $(u, v) \in T$ and $(v, w) \in B$. Also, we use $\xrightarrow{+}$ for the transitive, and $\xrightarrow{*}$ for the reflexive and transitive closure of $\to$ and call the unique sequence inducing $u \xrightarrow{*} v$ a *tree path*.

If $v \xrightarrow{*} w$ ($v \xrightarrow{+} w$), v is said to be (strictly) *lower* than w, and w (strictly) *higher* than v. A vertex is *lowest* (*highest*) in a set of vertices, if no other member of that set is lower (higher). The *height* of a vertex v is its distance from the root.

The characterizing property of DFS orientations is that the *target* w of every back edge $v \hookrightarrow w$ is a tree ancestor of (i.e., strictly below) its *source* v. Thus, each back edge $v \hookrightarrow w$ induces a *fundamental cycle* $C(v \hookrightarrow w) = w \xrightarrow{+} v \hookrightarrow w$, and these will be our primary objects of interest. Two cycles are called *overlapping*, if they share an edge, and it is the overlap of cycles that makes planarity testing challenging.

Lemma 1 *Let $G = (V, T \uplus B)$ be a DFS-oriented graph.*

(1) The fundamental cycles are exactly the simple directed cycles of G.
(2) Two distinct fundamental cycles are either disjoint, or their intersection forms a tree path.



(a) original graph (b) sketch

Fig. 2. A fork with branching point v in the graph of Figure 1, and a sketched representation showing only those back edges that are return edges of $e = u \to v$. Note that edges to the lowpoint of e are dashed, and that e_2 is chordal but e_1 is not.

PROOF.

(1) All fundamental cycles are simple and, because of DFS, directed. Now consider any simple directed cycle and let $v \in V$ be lowest on that cycle. Since every cycle contains at least one back edge, let $x \hookrightarrow u$ be the first back edge after v. Vertex v is lowest, so that u must be in $v \xrightarrow{*} x$. Since the cycle is simple, $u = v$ and there are no more edges.
(2) Let $w \xrightarrow{*} v \hookrightarrow w$ and $u \xrightarrow{*} x \hookrightarrow u$ be two fundamental cycles. Since they are distinct, $v \hookrightarrow w \neq x \hookrightarrow u$. Since there is exactly one path between any pair of vertices in a tree, tree paths can join and fork at most once. A non-empty intersection of $w \xrightarrow{*} v$ and $u \xrightarrow{*} x$ must, therefore, be a treepath itself. $\square$

For two overlapping cycles, the last edge $u \to v$ on the shared tree path together with the succeeding edge $e_1 = (v, w_1), e_2 = (v, w_2)$ on each cycle is called their *fork*, and v its *branching point*. We will see that finding a planar combinatorial embedding reduces to finding an appropriate ordering of all triplets of edges that form a fork.

It will be convenient to fix a linearization of the cyclic ordering of outgoing edges around a vertex. Since every vertex v has at most one incoming tree edge, the clockwise order of outgoing edges is split at the incoming tree edge, or between any two consecutive outgoing edges if v is the root of a DFS tree.

In the next section, two simple observations help understand how cycle orientations impose fork orderings.

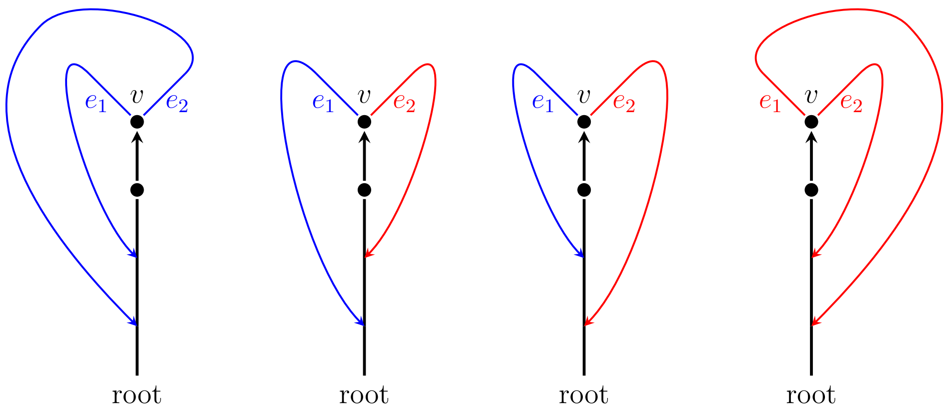


Fig. 3. In a planar drawing, overlapping fundamental cycles are nested, if and only if they are oriented alike. If the root is on the outer face, the lowest vertex of their union is contained in the outer of the two cycles.

3.2 Orientation and nesting of fundamental cycles

Recall that there are two classes of directed cycles in a planar drawing, because each is oriented either clockwise or counterclockwise. Since the intersection of overlapping fundamental cycles is a tree path containing at least one edge, the four possible configurations in Figure 3 can be summarized as follows.

Observation 1 *In a planar drawing of a DFS-oriented graph $G = (V, T \uplus B)$, two overlapping cycles are nested, if and only if they are oriented alike.*

By assigning orientations we essentially determine whether the inside is to the left or to the right of a directed cycle, but the above observation does not specify which of two nested cycles is enclosed by the other.

For disambiguation we use the convention that roots of DFS trees are incident to the outer face and measure the nesting depth of a fundamental cycle using the following concepts.

The *return points* of a tree edge $v \to w \in T$ are the ancestors u of v with $u \xrightarrow{+} v \to w \xrightarrow{*} x \hookrightarrow u$ for some descendant x of w. A back edge $v \hookrightarrow w$ has exactly one return point, its target w. The return points of a vertex $v \in V$ are formed by the union of all return points of outgoing edges $(v, w) \in E^+(v) \subseteq T \uplus B$. A back edge $x \hookrightarrow u$ is a *return edge* for every tree edge $v \to w$ with $u \xrightarrow{+} v \to w \xrightarrow{*} x \hookrightarrow u$, and for itself.

The *lowpoint* of an edge is its lowest return point, if any, or its source if none exists. Note that the lowpoint of a back edge is the lowest vertex of its fundamental cycle, and therefore also called the lowpoint of that cycle.

The second important observation establishes nesting constraints induced by lowpoints of cycles. It is justified by noting that if the root is on the outer face and there is a proper tree path from the lowpoint of one cycle to that of another cycle, this path can not be part of the inner cycle.

Observation 2 *In a planar drawing of a DFS-oriented graph $G = (V, T \uplus B)$ with all roots of DFS trees on the outer face, overlapping fundamental cycles are nested according to their lowpoint order.*

3.3 Relation to planar embeddings

The above two observations about orientations have immediate consequences for planar embeddings. This becomes obvious by considering the single fork in each of the four configurations in Figure 3.

First consider the fork of a pair of differently oriented cycles. Clearly, the outgoing edge of the left cycle is before the outgoing edge of the right cycle in the linearized order at branching point v.

Next consider the fork of a pair of likewise oriented cycles. In case they are right cycles and one contains a vertex that is strictly lower than those in the other cycle, the cycles outgoing edge (e_1 in Figure 3) comes first in the linearized order at branching point v. The converse is true when v is the branching point of left cycles.

A vertex may be the branching point for several pairs of overlapping cycles. Combining both observations yields a (for now partial) embedding at branching points: outgoing edge of left cycles need to be before those of right cycles, and the internal ordering in each subset is determined by lowpoints. Note that there may be ties, and that outgoing tree edges may be part of several, differently oriented cycles. We will have to resolve these ambiguities, but otherwise the whole approach rests entirely on Observations 1 and 2.

4 The Left-Right Planarity Criterion

With the above motivation in mind, we say that the *side* of a back edge in a planar drawing is *right*, if its fundamental cycle is oriented clockwise, and *left* otherwise. Assigning a side to a back edge thus corresponds to orienting a fundamental cycle, and this will be all that needs to be done.

The following definition summarizes all constraints resulting from sets of overlapping fundamental cycles in terms of their respective back edge. It is worth

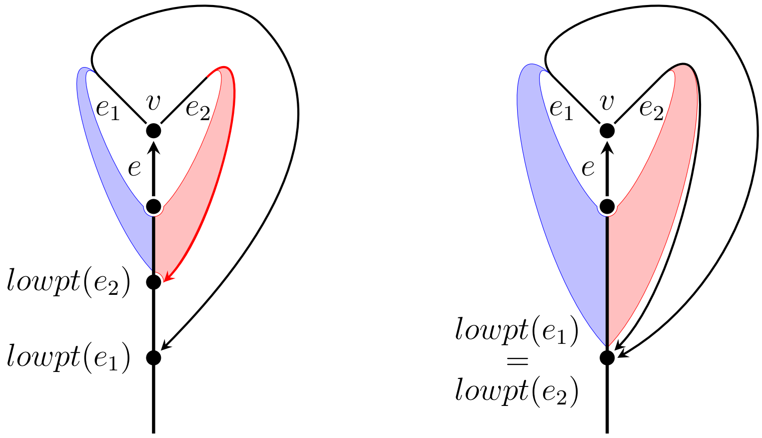


Fig. 4. LR constraints associated with $e = u \to v$.

noting that all constraints are generated by a single type of configuration associated with forks.

Definition 2 (LR partition) *Let $G = (V, T \uplus B)$ be a DFS-oriented graph. A partition $B = L \uplus R$ of its back edges into two classes, referred to as* left *and* right*, is called* left-right partition*, or* LR partition *for short, if for every fork $u \to v \in T$ and $e_1, e_2 \in E^+(v)$*

(1) all return edges of e_1 ending strictly higher than $lowpt(e_2)$ belong to one class and
(2) all return edges of e_2 ending strictly higher than $lowpt(e_1)$ to the other.

The LR partition constraints are illustrated in Figure 4. Each group of constraints is *associated* with a tree edge $u \to v$, and the system of constraints can be broken down into two sets of pairwise requirements: *same-constraints* forcing two back edges to be on the same side, and *different*-constraints forcing them to be on opposite sides. Note that two back edges are subject to a constraint only if their fundamental cycles overlap.[1] It is rather striking that these partition constraints (based on an arbitrary DFS orientation) are equivalent to planarity.

Theorem 3 (Left-Right Planarity Criterion) *A graph is planar if and only if it admits an LR partition.*

While necessity of the LR constraints is straightforward, we prove sufficiency in the next section by constructing a planar embedding from a given LR partition. The construction is guided by the constraints that orientation and nesting of fundamental cycles impose on an embedding.

[1] The configurations inducing either type of constraint are considered in Section 6.

Removing the following ambiguity will simplify both argumentation and algorithm. An LR partition is called *consistent*, if all back edges of a tree edge that end at its lowpoint are on the same side.

Lemma 4 *Any LR partition can be made consistent.*

PROOF. Consider two return edges b_1, b_2 of a tree edge $e = u \rightarrow v$ that end at $lowpt(e)$. If one of them is involved in any LR constraint as specified in Definition 2, this constraint must be associated with a tree edge $e' = u' \rightarrow v'$ such that $v' \xrightarrow{*} v$ and $lowpt(e')$ is strictly lower than $lowpt(e)$. Since b_1, b_2 originate from a common subtree entered by e and have the same return point, actually both are involved in this constraint and even required to be on the same side. Thus, consistency does not lead to contradictions. $\square$

4.1 Combinatorial embedding

Consider Figure 3 again and recall that the orientation of overlapping fundamental cycles induces a partial ordering of edges around forks.

We linearize clockwise cyclic orderings of edges around non-root vertices by starting from the unique incoming tree edge. Outgoing edges belonging to a counterclockwise cycle then need to appear before those belonging to a clockwise cycle branching at the respective fork. Moreover, outgoing edges of clockwise (counterclockwise) cycles must be ordered outside in (inside out) around their branching point.

Given a DFS-oriented graph $G = (V, T \uplus B)$ together with an LR partition of all back edges, a planar embedding can be obtained from extending the partition to cover tree edges as well, and a linear order defined on the outgoing edges of each vertex. This order will represent the nesting of cycles outside in (with the root fixed to be on the outer face). It is used without modification as the embedding order for right outgoing edges, and reversed for left outgoing edges when flipping them to appear before any right edges. In an implementation, this can be realized by assigning its order rank to each edge, changing the sign of left edges to minus, and a final sorting.

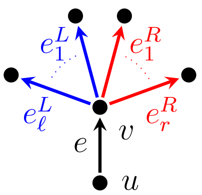
Extension of LR partitions to tree edges is straightforward. If a tree edge has any return edges (i.e., its source is neither the root nor a cut vertex), it is assigned to the same side as one of its return edges ending at the highest return point (i.e., according to an innermost fundamental cycle it is part of). Otherwise, the side is arbitrary.

To define a partial *nesting order* $\prec$, assume for a moment that all edges are on the right side and consider a fork consisting of $u \to v$ and outgoing edges e_1, e_2 of v. If both have return edges, v is a branching point of overlapping fundamental cycles sharing $u \to v$. Since both cycles are clockwise for now, we must properly nest them to avoid edge crossings. Since we fixed the root of the DFS tree to be in the outer face, we have to define $e_1 \prec e_2$ if and only if the lowpoint of e_1 is strictly lower than that of e_2. If both have the same lowpoint, but, say, only e_2 has another return point, we say that e_2 is *chordal* and let $e_1 \prec e_2$, because cycles containing e_2 and a return edge ending higher than $lowpt(e_2)$ can only lie inside of cycles containing e_1 and a return edge ending at $lowpt(e_1) = lowpt(e_2)$. If both e_1 and e_2 are chordal, the tie is broken arbitrarily, because eventually these two edges must be on different sides anyway.

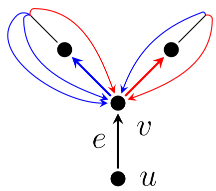
In the planarity testing algorithm, $\prec$ will be mimicked by defining the *nesting depth* of an edge e to be twice the height of the highest lowpoint of any cycle containing e, plus one if e is chordal.

The partial nesting order $\prec$ is extended to a combinatorial embedding by *LR ordering*, i.e. by flip-reversing left edges before right ones and placing incoming back edges on the appropriate side of the tree edge leading into the subtree of their source. Some care is needed to avoid crossings of back edges, but we will see that, algorithmically, this embedding is almost trivial to realize.

Definition 5 (LR Ordering) *Given an LR partition, let $e_1^L \prec \cdots \prec e_\ell^L$ be the left outgoing edges of a vertex v, and $e_1^R \prec \cdots \prec e_r^R$ its right outgoing edges. If v is not the root, let u be its parent. The clockwise* left-right ordering*, or* LR ordering *for short, of the edges around v is defined as follows:*



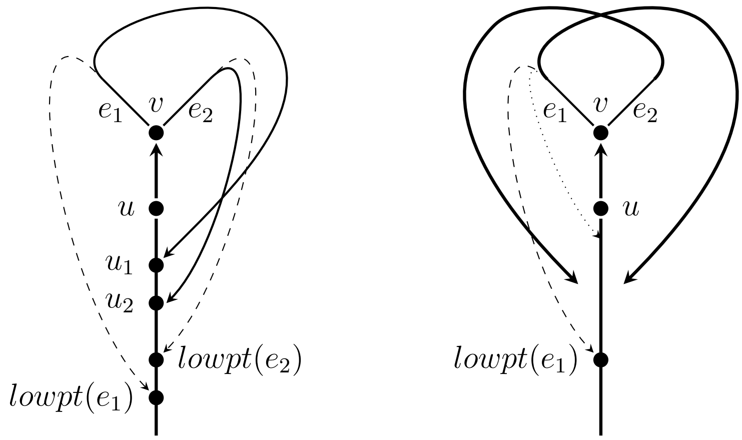
$$\begin{aligned}&(u,v),\\ &L(e_\ell^L), e_\ell^L, R(e_\ell^L), \ldots, L(e_1^L), e_1^L, R(e_1^L),\\ &L(e_1^R), e_1^R, R(e_1^R), \ldots, L(e_r^R), e_r^R, R(e_r^R)\end{aligned}$$



where (u,v) is absent if v is the root, and $L(e)$ and $R(e)$ denote the left and right incoming back edges whose cycles share e. For two back edges $b_1 = x_1 \hookrightarrow v, b_2 = x_2 \hookrightarrow v \in R(e)$ let $z \to x, (x, y_1), (x, y_2)$ be the fork of $C(b_1)$ and $C(b_2)$. Then, b_1 comes after b_2 in $R(e)$ if and only if $(x, y_1) \prec (x, y_2)$. If $b_1, b_2 \in L(e)$, the order is reversed.

Lemma 6 *Given any LR partition, LR ordering yields a planar embedding.*

PROOF. Let $G = (V, T \uplus B)$ be a DFS-oriented graph with an LR partition $B = L \uplus R$. We assume that it is consistent and extend it to also cover the tree edges as described above. Now consider the embedding defined by LR ordering



(a) same side (b) different sides

Fig. 5. Two types of crossings in proof of Lemma 6.

the edges around each vertex.

Since a graph with a spanning tree can always be drawn in such a way that a given embedding is respected, no two edges cross more than once, and none of the crossings involves a tree edge, the embedding is either planar, or any such drawing yields a simple crossing of two back edges (a crossing of more than two edges can be resolved into pairwise crossings). Only two cases are possible.

Case 1: (crossing back edges in same class)

Assume $x_1 \hookrightarrow u_1, x_2 \hookrightarrow u_2 \in R$ cross (the other case is symmetric). If $u_1 = u_2$, the crossings contradicts our definition of LR ordering the edges around that vertex.

W.l.o.g. we may therefore assume that u_1 is strictly higher than u_2, and u_2 therefore outside of the clockwise cycle $u_1 \xrightarrow{+} x_1 \hookrightarrow u_1$. Since the crossing is simple, x_2 in turn must be inside this cycle, and $u_1 \xrightarrow{+} x_1$ and $u_2 \xrightarrow{+} x_2$ cannot be disjoint (because we must enter the cycle somewhere along $u_2 \xrightarrow{+} x_2$). Let v be their highest common vertex, and e_1, e_2 the first edges on $v \xrightarrow{*} x_1$ and $v \xrightarrow{*} x_2$.

Since x_2 is inside of the clockwise cycle, e_1 comes before e_2 in the order around v. On the other hand, the LR partition requires that all return edges of e_1 ending higher than u_2 are on the same side as $x_1 \hookrightarrow u_1$, so that also e_1 is a right edge. LR ordering at v then implies that e_2 must be a right edge as well with $e_1 \prec e_2$.

By definition of $\prec$, either $lowpt(e_1)$ is strictly lower than u_2, or $lowpt(e_1) = u_2 = lowpt(e_2)$ and e_2 is chordal as well. In the former case, $x_1 \hookrightarrow u_1$ and $x_2 \hookrightarrow u_2$ had to be assigned different sides. In the latter case, the highest ending return edge of e_2 is right as is e_2, but conflicting with $x_1 \hookrightarrow u_1$ which is also right. In either case a contradiction.

Case 2: (crossing back edges in different classes)
Assume $x_1 \hookrightarrow u_1 \in R$ and $x_2 \hookrightarrow u_2 \in L$ (the other case is symmetric). Since the crossing is simple, the tree paths $u_1 \xrightarrow{+} x_1$ and $u_2 \xrightarrow{+} x_2$ cannot be disjoint and we define v, e_1, e_2 as in Case 1.

Again, e_1 must be before e_2 in the LR ordering of v for the back edges to cross. If $u_1 = \mathit{lowpt}(e_1) = \mathit{lowpt}(e_2) = u_2$, the LR partition is not consistent.

Otherwise, we may assume that $\mathit{lowpt}(e_1)$ is strictly lower than u_2 (the case that $\mathit{lowpt}(e_2)$ is strictly lower than u_1 is symmetric). Then, all return edges of e_2 ending at u_2 or higher must be on the same side as $x_2 \hookrightarrow u_2 \in L$, so that e_2 is left as well. Since e_1 comes before e_2, it must also be left and $e_2 \prec e_1$.

Due to the way we define sides for tree edges, e_1 is left only if it has a left return edge ending strictly higher than $\mathit{lowpt}(e_1)$ (because it must end at least as high as $x_1 \hookrightarrow u_1 \in R$ and the LR partition is consistent). On the other hand, $e_2 \prec e_1$ implies that $\mathit{lowpt}(e_2)$ is lower than or equal to $\mathit{lowpt}(e_1)$. This is a contradiction, since the LR constraints rule out that e_1 and e_2 have return edges ending strictly higher than $\mathit{lowpt}(e_2)$ and $\mathit{lowpt}(e_1)$ that are both on the left.

Since both types of crossings contradict our assumptions, the embedding is planar. $\square$

We have thus proved constructively the non-obvious implication of the Left-Right Planarity Criterion (Theorem 3).

5 Algorithm

We can now give the linear-time algorithm for testing planarity and for determining a planar embedding or a minimal non-planar subgraph. After a high-level description of its three main phases shown in Algorithm 1, full implementation details are provided for all operations but those concerning the specific data structure used to represent a graph and its embedding.

Orientation. The algorithm is based on the Left-Right Planarity Criterion and therefore starts with a depth-first search (DFS) to orient the input graph. For each connected component, the root of its spanning DFS tree is stored in a list, *Roots*. The tree-path distance of a vertex from its root is stored in an array *height*, so that roots of unexplored components are identified by still having the initial value 0. Different from other planarity algorithms, there is no need to worry about biconnected components.

variable	**type**	**purpose**	**initially**
height	integer node array	tree-path distance from root	∞
lowpt	integer edge array	*height* of lowest return point	n.a.
*lowpt*2	integer edge array	*height* of next-to-lowest return point (tree edges only)	n.a.
nesting_depth	integer edge array	proxy for nesting order $\prec$ given by twice *lowpt* (plus 1 if chordal)	n.a.

(a) orientation phase

variable	**type**	**interpretation**	**initially**
ref	edge array of edges	edge relative to which side is defined	$\perp$
side	edge array of signs $\{-1, 1\}$	side of edge, or modifier for side of reference edge	1
$I = [low, high]$	pair of edges	interval of return edges represented by its two edges with extremal lowpoints	n.a.
$P = (L, R)$	pair of intervals	overlapping intervals, i.e., a conflict pair	n.a.
S	stack of conflict pairs	conflicting intervals formed by current return edges	$\emptyset$
stack_bottom	edge array of conflict pairs	top of stack S when traversing the edge (tree edges only)	n.a.
lowpt_edge	edge array of edges	next back edge in traversal (with lowest return point)	n.a.

(b) testing phase

variable	**type**	**interpretation**
leftRef	vertex array of edges	leftmost back edge from current DFS subtree (i.e. after next incoming left back edge)
rightRef	vertex array of edges	tree edge leading into current DFS subtree (i.e. before next incoming right back edge)

(c) embedding phase

Fig. 6. Main variables used in the algorithm.

Algorithm 1: Left-Right Planarity Algorithm

input: simple, undirected graph $G = (V, E)$
output: planar embedding (halts if graph is not planar)

if $|E| > 3|V| - 6$ **then HALT: not planar**

▼ **orientation**
- **for** $s \in V$ **do**
  - **if** $height[s] = \infty$ **then**
    - $height[s] \leftarrow 0$;   append $Roots \leftarrow s$
    - DFS1(s) `/* see Algorithm 2 */`

▼ **testing**
- sort adjacency lists according to non-decreasing *nesting_depth*
- **for** $s \in Roots$ **do** DFS2(s) `/* see Algorithm 3 */`

▼ **embedding**
- **for** $e \in E$ **do** $nesting\_depth[e] = \underline{\text{sign}}(e) \cdot nesting\_depth[e]$
- sort adjacency lists according to non-decreasing *nesting_depth*
- **for** $s \in Roots$ **do** DFS3(s) `/* see Algorithm 6 */`

where
integer sign(edge e)
- **if** $ref[e] \neq \perp$ **then**
  - $side[e] \leftarrow side[e] \cdot \underline{\text{sign}}(ref[e])$
  - $ref[e] \leftarrow \perp$
- **return** $side[e]$

During DFS, the partial nesting order $\prec$ is determined by assigning to each edge an integer value *nesting_depth* such that $e_1 \prec e_2 \implies nesting\_depth[e_1] < nesting\_depth[e_2]$.

Testing. To determine whether there exists a consistent LR partition, the DFS forest is traversed for a second time. The traversal is modified, however, such that outgoing edges are visited in the order induced by *nesting_depth*. The second traversal halts if the graph is not planar, and we discuss at the end of Section 5.2 how to extract one or more Kuratowski subgraphs in that case.

The tentative side of edges may change often during the test, so that the bipartition is returned only implicitly for efficiency reasons. An edge array ref specifies for each edge a reference edge relative to which its side is defined, and in an edge array $side$ a value of 1 or -1 indicates whether the side of the edge is the same as, or different from, the side of its reference edge. If the reference edge of e is undefined, i.e. $ref[e] = \perp$, the value of $side[e]$ specifies the side directly, where -1 is for left and 1 is for right.

Embedding. Given an LR partition, flip-reversal of left edges is performed by sorting the outgoing edges in all adjacency lists once again according to their nesting order, though now modified by the side sign. Since the multiplication of *nesting_depth* with *side* only changes the sign of left edges to negative, they are effectively placed before all right edges, in reverse order. To complete the LR ordering, incoming edges are placed during a third traversal of the DFS forest that is guided by the order of outgoing edges.

For each of the three main phases, we provide detailed pseudo-code with ample comments in the subsequent sections.

5.1 Phase 1 – Orientation

The purpose of the first DFS is to orient the graph, and to determine lowpoints and nesting order $\prec$. It is therefore a standard DFS computing the auxiliary variables given in Table 6(a). Except for *height*, all are determined during backtracking.

Our use of lowpoints is slightly non-standard in two ways. Firstly, we have defined lowpoints for edges rather than vertices, and, secondly, we do not assign DFS numbers, but heights. The latter induce the same ordering of ancestors as DFS numbers, but are related to the tree more intuitively and in general results in a smaller range of values which may in turn speed up the subsequent sorting of adjacency lists according to *nesting_depth*.

Second lowpoints stored in $lowpt2$ only serve to determine whether an edge has more than one return point (i.e., it is chordal), and are not needed by themselves.

The rationale for representing $\prec$ via *nesting_depth* is two-fold: firstly, we can apply a linear-time sorting algorithm such as counting sort on the set of edges, because the range of values is linear in the size of the graph, and secondly, flip-reversal of left edges after the second phase can be performed by sign changes with subsequent re-sorting.

Algorithm 2: Phase 1 – DFS orientation and nesting order

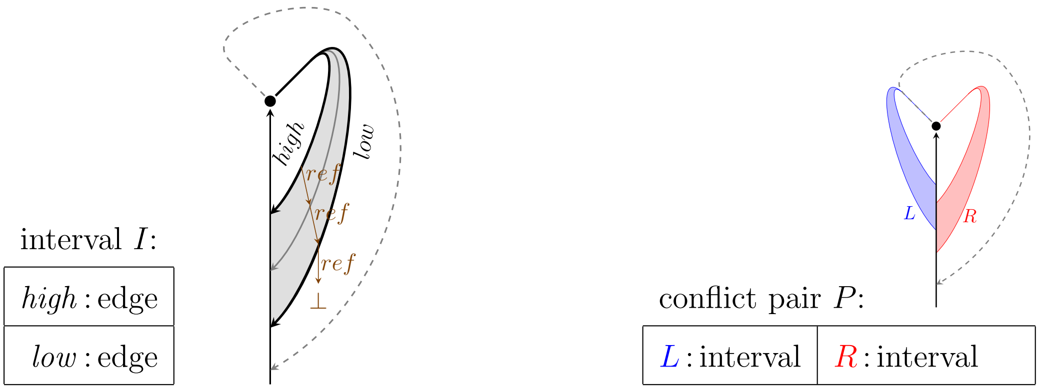
```
DFS1(vertex v)
  e ← parent_edge[v]
  while there exists some non-oriented {v, w} ∈ E do
    orient {v, w} as (v, w)
    lowpt[(v, w)] ← height[v];   lowpt2[(v, w)] ← height[v]
    if height[w] = ∞ then /* tree edge */
      parent_edge[w] ← (v, w)
      height[w] ← height[v] + 1
      DFS1(w)
    else /* back edge */
      lowpt[(v, w)] ← height[w]
    ▼ determine nesting depth
      nesting_depth[(v, w)] ← 2 · lowpt[(v, w)]
      if lowpt2[(v, w)] < height[v] then /* chordal */
        nesting_depth[(v, w)] ← nesting_depth[(v, w)] + 1
    ▼ update lowpoints of parent edge e
      if e ≠ ⊥ then
        if lowpt[(v, w)] < lowpt[e] then
          lowpt2[e] ← min{lowpt[e], lowpt2[(v, w)]}
          lowpt[e] ← lowpt[(v, w)]
        else if lowpt[(v, w)] > lowpt[e] then
          lowpt2[e] ← min{lowpt2[e], lowpt[(v, w)]}
        else
          lowpt2[e] ← min{lowpt2[e], lowpt2[(v, w)]}
```

5.2 Testing

The second phase is the working horse of the algorithm. It determines a consistent LR partition extended to all edges, if one exists; the code can be extended to otherwise identify fundamental cycles whose union yields a Kuratowski subgraph as sketched in Section 6.

The main challenge is to detect and maintain all pairwise constraints among back edges. Recall that constraints are associated with a tree edge and that there are only two types of constraints. According to Definition 2, a pair of back edges with overlapping fundamental cycles is placed on the same or on different sides.

The following observation may help in building a better intuition. Consider a signed *constraint graph*, in which vertices represent back edges of the original DFS-oriented graph, and edges are introduced and labeled $+1$ or -1 when two



(a) Return edges subject to pairwise *same*-constraints are represented in a list linked by ref-pointers and ordered by height of return point

(b) Two intervals of return edges subject to pairwise *different*-constraints

Fig. 7. The main data structure is a stack S storing overlapping pairs of intervals.

back edges are constrained to be on the same or on different sides. Finding an LR partition that satisfies all LR constraints is equivalent to testing whether this graph is *balanced*, i.e. there is a bipartition such that the corresponding cut is crossed exactly by the edges labeled -1. Balancedness of signed graphs is introduced in Harary (1953).

Clearly, we cannot afford to maintain all pairwise constraints explicitly, because their number may be quadratic in the size of the original graph. We will hence represent them implicitly to test for contradictions. To be able to realize a bipartition it is sufficient to maintain a spanning forest of the constraint graph. We therefore construct a rooted tree for each of its component using a reference pointer ref for every edge. This is including tree edges, since their side is determined by reference to a return edge ending at the highest return point, anyway.

A second array, *side*, is used to store the side of all edges that are roots in the spanning forest of the constraint graph. For all other edges the array holds the sign of the unique outgoing constraint-graph edge linking them to their corresponding reference edge. As indicated earlier, values $+1$ and -1 will therefore be interpreted either as *right* and *left*, or as *same* and *different*.

To reduce the number of constraints that need to be represented explicitly, observe that the *same*-constraints induced by a fork $u \to v$, $e_1, e_2 \in E^+(v)$ in Definition 2 involve two sets of return edges with a simple structure. For, say, e_1 let $h = x_h \hookrightarrow u_h$ and $\ell = x_\ell \hookrightarrow u_\ell$ be the two (possibly equal) return edges ending at the highest and lowest return point of e_1 that is strictly higher than $lowpt(e_2)$. Then we know that h_ℓ and all return edges $x' \hookrightarrow u'$ of e_1 with a return point in $u_\ell \xrightarrow{*} u_h$ are in the same group of *same*-constraints. This *interval* of edges can thus be represented by its two extreme members, h and ℓ, as shown in Figure 7(a). Return edges belonging to an interval are maintained in a singly-linked list, from highest to lowest return point, using the ref-array.

Different-constraints can be summarized similarly, because by transitivity they always involve all pairs of edges in a pair of intervals. A *conflict pair* therefore consists of two intervals of edges subject to *different*-constraints as shown in Figure 7(b). It represents their tentative assignment to the left and right, and thus a partial bipartition.

The second DFS traversal is designed to build the entire bipartition of edges incrementally by merging conflict pairs. Its main data structure is a stack S of conflict pairs representing all constraints associated with a tree edge that has been traversed, but not yet backtracked over. Note that these constraints involve only back edges that have already been traversed, but return to a vertex below the current one. In other words, each back edge in the stack is a return edge for at least one tree edge in the current DFS path.

By processing DFS trees bottom-up, the constraints associated with an edge can be determined by merging those associated with its outgoing edges. Two main invariants are maintained. We will not prove them explicitly, but rather let them serve as an orientation for understanding the implementation. The first invariant eventually yields correctness of the implementation,

> **Partitioning Invariant:** The additional conflict pairs accumulated at the top of the stack between traversing a tree edge and backtracking over it represent a partial bipartition satisfying all non-crossing constraints associated with that edge.

and the second one ensures that constraint merging can be carried out efficiently.

> **Ordering Invariant:** Return edges forming an interval are represented in singly-linked lists ordered from highest to lowest return point, and the lowest edge in a conflict pair is not lower than the highest edge in another conflict pair deeper in S.

5.2.1 Ordered traversal

Pseudo-code for the second DFS is given in Algorithms 3–5. Since all edges have been oriented during the first DFS, they are again traversed in the same direction. The traversal order differs, though, since adjacency lists have been rearranged according to *nesting_depth*, so that outgoing edges with lower lowpoints are traversed first. This reordering is crucial for the ordering invariant.

Algorithm 3: Phase 2 – Testing for LR partition

```
DFS2(vertex v)
  e ← parent_edge[v]
  for e_i ∈ E⁺(v) = ⟨e_1, ..., e_d⟩ do /* ordered by nesting_depth */
    stack_bottom[e_i] ← top(S)
    if e_i = parent_edge[target(e_i)] then /* tree edge */
      DFS2(target(e_i))
    else /* back edge */
      lowpt_edge[e_i] ← e_i;   push (∅, [e_i, e_i]) → S
    if lowpt[e_i] < height[v] then /* e_i has return edge */
      if e_i = e_1 then
        lowpt_edge[e] ← lowpt_edge[e_1]
      else
        ▶ add constraints of e_i (Algorithm 4)

  if e ≠ ⊥ then /* v is not root */
    u ← source(e)
    ▶ trim back edges ending at parent u (Algorithm 5)
    ▼ side of e is side of a highest return edge
      if lowpt[e] < height[u] then /* e has return edge */
        h_L ← top(S).L.high;   h_R ← top(S).R.high
        if h_L ≠ ⊥ and (h_R = ⊥ or lowpt[h_L] > lowpt[h_R]) then
          ref[e] ← h_L
        else
          ref[e] ← h_R
```

When visiting a vertex v during the DFS traversal, the high-level task is to recursively determine the constraints for all outgoing edges and integrate them into those associated with parent edge $e = u \rightarrow v$ (if v is not a DFS root).

Before traversing an outgoing edge $e_i \in E^+(v)$, we therefore remember the top conflict pair *stack_bottom*$[e_i]$ on S (where $top(S) = \perp$ if S is empty). If e_i was a tree edge in the first traversal, all constraints associated with e_i are recursively determined and pushed onto S. If e_i is a back edge, it is pushed onto S in a conflict pair of its own because it may be involved in later constraints. Recall that our goal is to determine a *consistent* LR partition. We therefore store in an edge array *lowpt_edge* the first back edge not traversed earlier. For edges that have return edges, this is the first return edge to their lowpoint and can thus be used as a reference for other return edges that have to be assigned to the same side to meet the consistency requirement. A back edge e_i is its own unique return edge to its lowpoint so that we let *lowpt_edge*$[e_i] = e_i$.

From the partitioning invariant we known that when returning from the traversal of e_i, the conflict pairs above $stack\_bottom[e_i]$ represent a partial LR partition of all return edges of e_i. While processing the first outgoing edge e_1 we simply leave them on the stack, if any, and pass on $lowpt\_edge[e_1]$ to $lowpt\_edge[e]$. Note that, since e_1 has a return edge, $parent\_edge[v] = e \neq \perp$, i.e. v is not a root. For each of the other outgoing edges $e_i = e_2, \ldots, e_d \in E^+(v)$, the constraints above $stack\_bottom[e_i]$ are merged into those which have already been accumulated for e and are directly beneath in S. Constraint integration is the most essential step and described separately in Algorithm 4 and below.

After all outgoing edges have been traversed, we trim all those back edges from the top of S that are return edges of some $e_i \in E^+(v)$, but not of e, i.e. which end at u. This requires some annoyingly lengthy but simple case distinctions given in Algorithm 5 and explained below. Observe that, if v is a DFS root, then there is no parent edge $e = u \to v$, but there are also no remaining constraint pairs on S, since a DFS root does not have outgoing back edges and there is more than one outgoing tree edge only if each leads into a different biconnected component.

If existent, parent edge e is finally assigned to the side of a back ending at the highest return point as suggested by the LR ordering procedure of Section 4.1. By the ordering invariant, this edge is the highest return edge in one of the two intervals in the top conflict pair, and we have already removed all non-return edges. Observe that the stack cannot be empty if there is a return edge.

5.2.2 Adding constraints associated with the next outgoing edge

We have to merge all constraints associated with the next outgoing edge, e_i, with those already accumulated from $e_1, \ldots, e_{i-1}$. The involved intervals are therefore gathered one by one in an initially empty conflict pair P as illustrated in Figure 8.

Merge return edges of $\boldsymbol{e_i}$ into right interval. All return edges of e_i have been traversed since traversing e_i, and they are represented in the top conflict pairs on stack S down to, but not including, $stack\_bottom[e_i]$. All of these intervals have to be merged on one side because of the fundamental cycle of $lowpt\_edge[e]$. If there is a conflict pair with two non-empty intervals, merging on one side violates an earlier constraint and the graph is not planar.

There is at least one conflict pair above $stack\_bottom[e_i]$ for otherwise we would not have entered this section. The non-empty interval of each such pair is merged in the right interval $P.R$ of P without changing their order by having the lowest edge of $P.R$ refer to the highest edge of the next conflict pair and replacing it accordingly. An exception is the interval containing a return edge

Algorithm 4: Adding constraints associated with e_i (part of Alg. 3)

```
▼ add constraints of e_i
    P ← (∅, ∅)
    ▼ merge return edges of e_i into P.R
        repeat
            Q ← pop(S)
            if Q.L ≠ ∅ then swap Q.L, Q.R
            if Q.L ≠ ∅ then
                HALT: not planar
            else
                if lowpt[Q.R.low] > lowpt[e] then /* merge intervals */
                    if P.R = ∅ then P.R.high ← Q.R.high
                      else ref[P.R.low] ← Q.R.high
                    P.R.low ← Q.R.low
                else /* make consistent */
                    ref[Q.R.low] ← lowpt_edge[e]
        until top(S) = stack_bottom[e_i]
    ▼ merge conflicting return edges of e_1, ..., e_{i-1} into P.L
        while conflicting(top(S).L, e_i) or conflicting(top(S).R, e_i) do
            Q ← pop(S)
            if conflicting(Q.R, e_i) then swap Q.L, Q.R
            if conflicting(Q.R, e_i) then
                HALT: not planar
            else /* merge interval below lowpt(e_i) into P.R */
                ref[P.R.low] ← Q.R.high
                if Q.R.low ≠ ⊥ then P.R.low ← Q.R.low
            if P.L = ∅ then P.L.high ← Q.L.high
              else ref[P.L.low] ← Q.L.high
            P.L.low ← Q.L.low
    if P ≠ (∅, ∅) then push P → S

where
boolean conflicting(interval I, edge b)
    return (I ≠ ∅ and lowpt[I.high] > lowpt[b])
```

to the lowpoint of e; to make the LR partition consistent, we make it refer to the *lowpt_edge* directly.

Merge conflicting return edges of $e_1, \dots, e_{i-1}$ into left interval. Return edges of $e_1, \dots, e_{i-1}$ with lowpoints higher than $lowpt[e_i]$ are subject to pairwise *same*-constraints and to a *different*-constraint with respect to some return edge of e_i. (If $lowpt[e_i] = lowpt[e]$ this is not $lowpt\_edge[e_i]$ but, e.g., a back edge returning to $lowpt2[e_i]$ which must exist, because apparently $lowpt2[e_{i-1}]$ exists as well by the way outgoing edges are ordered).

So while there are conflict pairs on the stack that contain return edges with lowpoints higher than $lowpt[e_i]$, these have to be merged on one side. If such

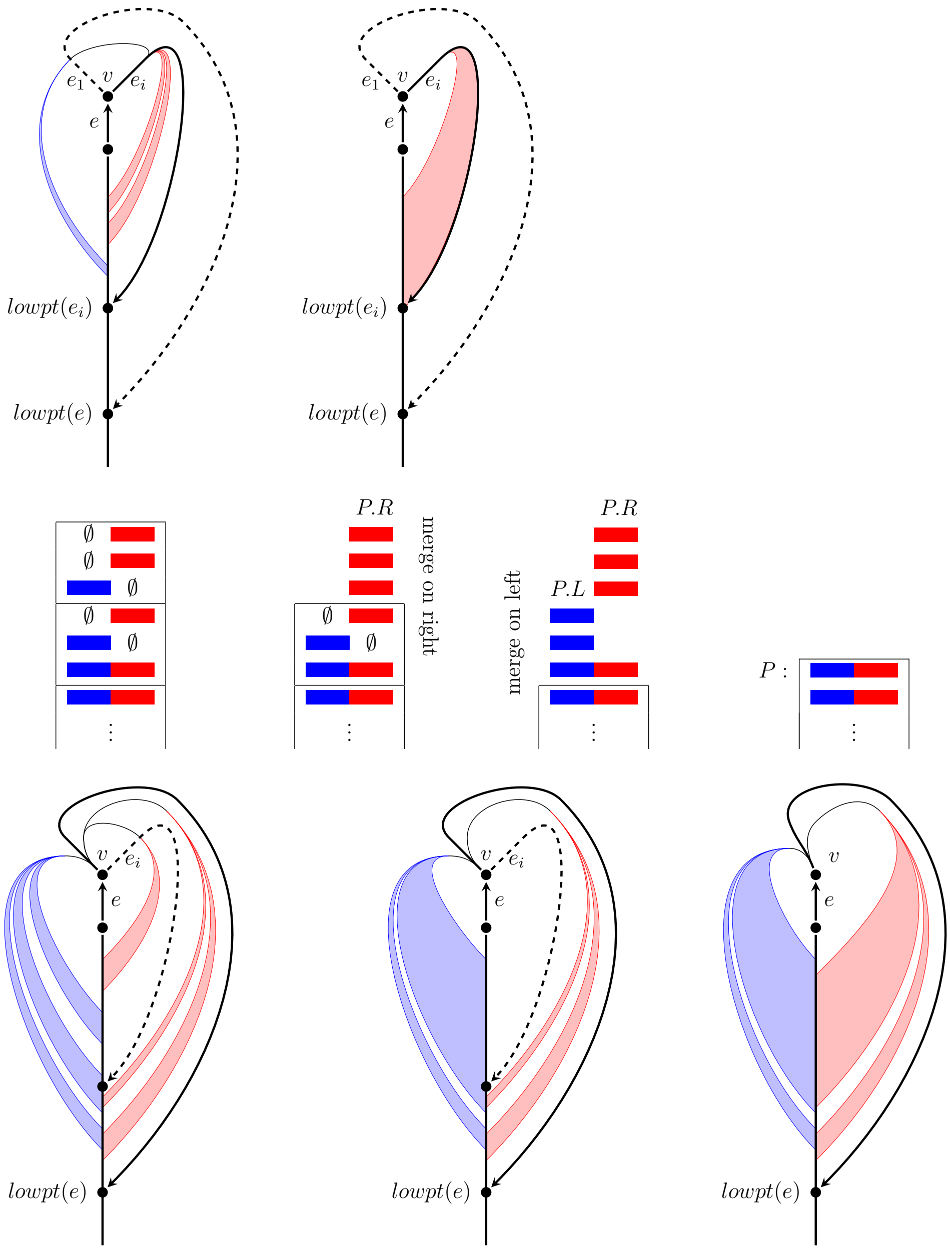


Fig. 8. In the core step of the algorithm, the constraints of e_i are merged into those of $e_1, \ldots, e_{i-1}$. Horizontal lines indicate where the top of stack S is divided by $stack\_bottom[e_i]$ and the topmost pair that is not conflicting with $lowpt\_edge[e_i]$. If $lowpt(e_i) = lowpt(e)$, the pair containing only $lowpt\_edge[e_i]$ is discarded and the bipartition is made consistent by assigning $ref[lowpt(e_i)] \leftarrow lowpt(e)$. Note that in this case, $P.R$ is not growing when merging on the left.

a pair contains two intervals ending above $lowpt[e_i]$, we again have a contradiction with a previous constraint and thus non-planarity. If only one side ends above $lowpt[e_i]$, we merge the other into $P.R$ (effectively closing these constraints under transitivity).

The actual merging of intervals is performed in the same way as above, and the final pair can be placed on the stack.

5.2.3 Trimming back edges

The purpose of Algorithm 5 is to remove all those back edges from conflict pairs on the stack that have the parent of the current tree edge $e = u \to v$ as their lowpoint, because they are no return edges of e or any lower tree edge, and therefore not subject to any constraint associated with a tree edge still to be processed.

Algorithm 5: Removing back edges ending at parent u (part of Alg. 3)

▼ **trim back edges ending at parent u**
- ▼ **drop entire conflict pairs**
  - **while** $S \neq \emptyset$ **and** lowest$(top(S)) = height[u]$ **do**
    - $P \leftarrow \text{pop}(S)$
    - **if** $P.L.low \neq \bot$ **then** $side[P.L.low] \leftarrow -1$
- **if** $S \neq \emptyset$ **then** `/* one more conflict pair to consider */`
  - $P \leftarrow \text{pop}(S)$
  - ▼ **trim left interval**
    - **while** $P.L.high \neq \bot$ **and** $\text{target}(P.L.high) = u$ **do**
      - $P.L.high \leftarrow ref[P.L.high]$
    - **if** $P.L.high = \bot$ **and** $P.L.low \neq \bot$ **then** `/* just emptied */`
      - $ref[P.L.low] \leftarrow P.R.low$
      - $side[P.L.low] \leftarrow -1$
      - $P.L.low \leftarrow \bot$
  - ▶ **trim right interval**
  - push $P \to S$

where
integer lowest(conflictpair P)
- **if** $P.L = \emptyset$ **then return** $lowpt[P.R.low]$
- **if** $P.R = \emptyset$ **then return** $lowpt[P.L.low]$
- **return** $\min\{lowpt[P.L.low], lowpt[P.R.low]\}$

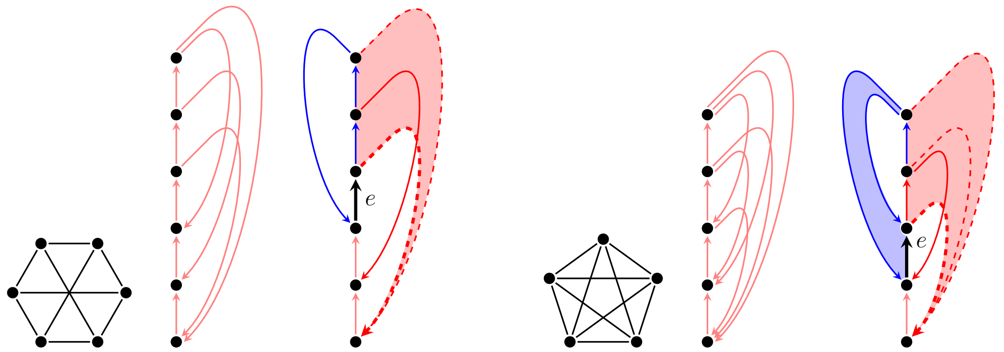
Dropping entire conflict pairs. If the lowest lowpoint on either side of a conflict pair P is the source of the current tree edge $u \to v$, all lowpoints of back edges in P are the same and the edges will not be involved in any future constraints. The pair is finalized by assigning the lowest back edge of the left interval to the left side. Since *side* is initialized with 1, the lowest back edge in the right interval $P.R$ is already assigned correctly to the right side, and all other back edges b in P to the same side as $ref[b]$.

Trimming a left interval. Since back edges in an interval are concatenated by ref-pointers in an order monotonic in the *height* of their lowpoints, we can simply remove back edges from the upper end of the left interval until the highest lowpoint is no longer u, or the interval has become empty. In the latter case the lower end of the interval is still defined and made to refer to an edge on the other side, setting its *side* to -1 accordingly. Note that the right interval cannot be empty for otherwise the entire conflict pair had been removed in the first while loop. All other removed back edges still refer to a back edge on the same side, so that the initial 1 of their *side*-entry must not be changed.

Trimming a right interval. This is symmetric to the previous operation. Note, however, that the assigned *side* in case the right interval becomes empty is -1 as well, because this indicates that the side of the lowest back edge is different from the side of the lowest back edge in the left interval. Again, the left interval cannot be empty.

Assigning a side to a tree edge. After trimming all back edges ending at source u of the current tree edge $e = u \to v$ in Algorithm 3, the side of e is determined by reference to a highest return edge. There is a return edge only if $lowpt[e] < height[u]$. Otherwise the u is a cutvertex or root and it does not matter, which side e is assigned to. Since the existence of a return edge renders S non-empty, the ordering invariant asserts that we have to compare the lowpoints of the highest back edges in the two intervals of the top constraint pair on S (checking for existence).

At the end of the testing phase, a non-crossing LR partition is given implicitly by edge arrays ref and *side*, if and only if the graph is planar. These define the side of an edge e relative to another, where $side[e]$ indicates whether the side is the same or different from that of $ref[e]$. Since $ref[e]$ always has a strictly lower target than e, the referrals are acyclic and form a rooted spanning forest of the constraint graph. The roots of that forest refer to $\perp$, and their side is determined explicitly by *side*. After dereferencing all referrals at the beginning of the embedding phase, the LR partition is known explicitly.



(a) $K_{3,3}$ (b) K_5

Fig. 9. The algorithm testing $K_{3,3}$ and K_5 for planarity. In either case, the status before starting the second DFS is depicted in the middle, and the algorithm halts in the configuration on the right while processing e.

Two small examples are shown in Figure 9. Even though both graphs are non-planar, the workings of the algorithm are nicely illustrated, since coloring and embedding correspond to the current (implicitly represented) bipartition and LR ordering.

5.3 Embedding

Compared to other planarity algorithms, the embedding phase is extremely simple. LR ordering the outgoing edges of the DFS-oriented graph is achieved by sorting them according to their *nesting_depth* on both sides. It is known that a partial embedding like this already determines a complete combinatorial embedding (see, e.g., Cai 1993), but for completeness we provide full details in Algorithm 6.

The DFS forest is traversed for the third time. Since outgoing edges are already ordered in the desired way, back edges are encountered exactly as required in the definition of LR ordering. As described in Table 6(c) we therefore maintain, for each vertex v, the two positions next to which the next left or right back edge needs to be inserted.

Observe that the incoming back edges from the same subtree actually appear in counterclockwise order. If the data structure available for embedded graphs does not provide a constant-time method for direct neighbor insertion, the now obsolete array ref can be used to build a singly-linked list of all edges incident to a vertex in counterclockwise order.

Algorithm 6: Phase 3 – Embedding

```
DFS3(vertex v)
  for e_i ∈ E^+(v) = ⟨e_1, ..., e_d⟩ do
    w ← target(e_i)
    if e_i = parent_edge[w] then /* tree edge */
      make e_i first edge in adjacency list of w
      leftRef[v] ← e_i;   rightRef[v] ← e_i
      DFS3(w)
    else /* back edge */
      if side[e_i] = 1 then
        place e_i directly behind rightRef[w] in adjacency list of w
      else
        place e_i directly before leftRef[w] in adjacency list of w
        leftRef[w] ← e_i
```

5.4 Running time and implementation

Theorem 7 *Algorithm 1 can be implemented to test in $\mathcal{O}(n)$ time whether a graph is planar and return a planar combinatorial embedding if it is.*

PROOF. We have argued throughout this section that the algorithm correctly yields an LR ordering if the graph admits an LR partition. Hence, correctness is established by the Left-Right Planarity Criterion (Theorem 3). Since a graph cannot be planar if $m > 3n-6$, we may assume that the number of edges is at most linear in the number of vertices.

The algorithm performs three DFS traversals, and rearranges the edges twice in between. Both rearrangements are obtained from sorting the edges according to *nesting_depth*, which can be done in linear time using bucket sort because all entries are integers with absolute value less than $2n$.

The first DFS clearly requires constant time per edge traversal and backtracking step, and hence linear time overall.

During the second traversal, every back edge is pushed onto the stack exactly once (when it is traversed), so that the number of newly generated constraint pairs is bounded by the number of back edges. If more than a constant number of constraint pairs is inspected during the addition of constraints, a corresponding number of them is merged. Since also the total time spent on trimming back edges that return to the parent is linear in the number of edges, the overall running time is linear.

Dereferencing ref-pointers takes linear time, because it is performed only once before the third DFS traversal, which also requires linear time if the graph data structure provides a constant-time operation to move an edge next to another in the embedding order. If it does not, the algorithm can be altered to re-use ref-pointers for the embedding as described in Section 5.3. □

The left-right approach can be implemented as described above [2] and our experiences with its performance essentially confirm the results of Boyer, Cortese, Patrignani, and Di Battista (2004). A special edge numbering scheme used in PIGALE (de Fraysseix and Ossona de Mendez, 2002) serves to avoid repeated DFS traversals, but we have found the sorting of adjacency lists to be even more costly. Note, however, that both sorting and DFS traversal can be avoided during the testing phase by splitting the stack into singly-linked lists associated with edges and processing edges (i.e., merging their final list of constraints into that of another edge) in the order given by $nesting\_depth$. This order is determined by creating two buckets for each *height* and adding an edge to its respective bucket when its *lowpt* is known during the initial DFS, i.e. when it is backtracked over. Since *lowpt* is determined bottom-up, edges added to the same bucket end up being in the desired order.

6 Non-Planarity

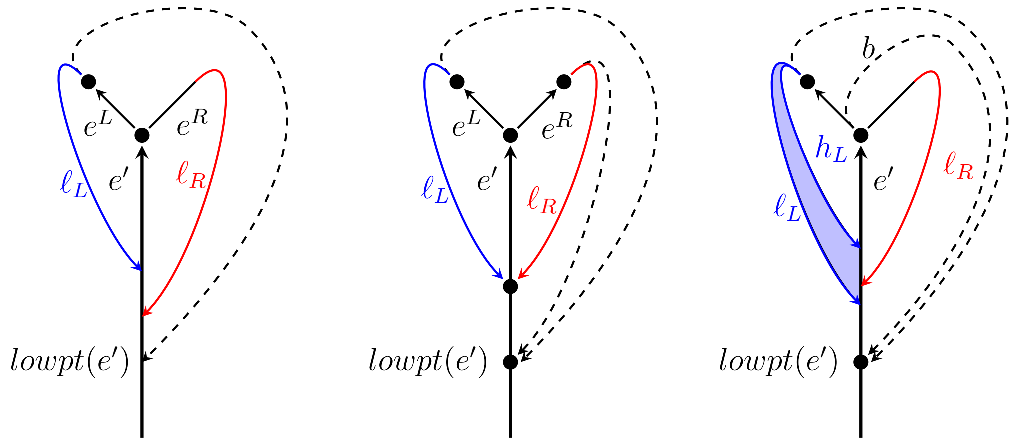
The algorithm halts, if and only if the input graph is non-planar. The initial test on the number of edges is justified by the fact that in any planar graph, $m \leq 3n-6$ (a well-known consequence of Euler's formula; see, e.g., Chiba and Nishizeki 1988).

If the algorithm does not halt, a planar embedding constructed as described in Section. 5.3 can serve as a certificate for planarity. If it does halt after the initial condition, though, the graph is non-planar and we would like a certificate for this case as well. The earliest characterization of planarity is due to Kuratowski (1930) and provides for exactly that. It is described in the following.

Recall the two non-planar graphs in Figure 9 and let us call a *subdivision* of a graph G any graph that can be obtained from G by repeatedly splitting edges using new vertices of degree two.

Theorem 8 (Kuratowski's Planarity Criterion) *A graph is planar, if and only if it does not contain a subdivision of* $K_{3,3}$ *or* K_5*.*

[2] The version described here has been implemented almost literally in C/C++ using LEDA by Daniel Kaiser, and in Java using yFiles by Martin Mader.



(a) Type 1: $lowpt(\ell_L) > lowpt(\ell_R)$

(b) Type 2: $lowpt(\ell_L) = lowpt(\ell_R)$, $lowpt(e^L) = lowpt(e^R)$

(c) Type 3: $lowpt(\ell_L) \leq lowpt(\ell_R)$, $lowpt(e^L) < lowpt(e^R)$

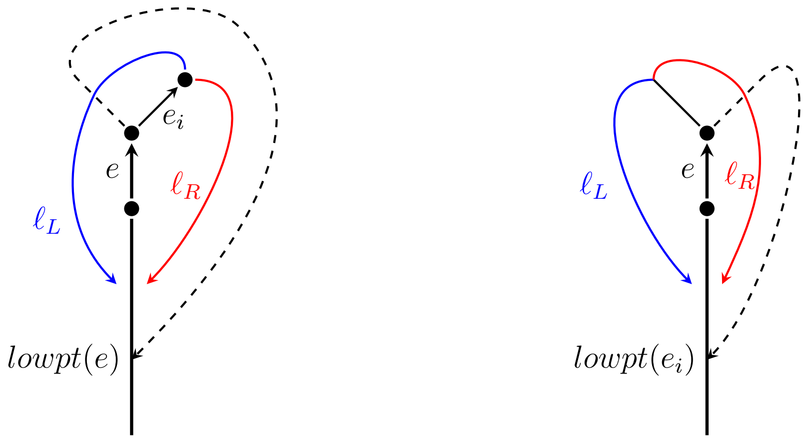
Fig. 10. A *different*-constraint between the two lowest edges ℓ_L, ℓ_R of a new constraint pair associated with e' is caused by one of three inclusion-minimal configurations (assuming w.l.o.g. that $lowpt(e^L) \leq lowpt(e^R)$).

Each subgraph that is a subdivision of $K_{3,3}$ or K_5 is called a *Kuratowski subgraph* (or *planarity obstruction*), and we show how to extract such a subgraph if the planarity test of the previous section fails.

Algorithm 1 halts when a pair $P = (L, R)$ at the top of S contains two conflicting intervals $L \neq \emptyset \neq R$ that prevent merging on the right or left side. In other words, a *same*-constraint associated with the current tree edge e contradicts a previously found *different*-constraint involving the same two intervals. Let $\ell_L = P.L.low$ and $P.R.low = \ell_R$ be the lowest edges in L and R. A Kuratowski subgraph can be determined from the union of the fundamental cycles of ℓ_L, ℓ_R and at most four others.

***Different*-constraint.** Since both ℓ_L, ℓ_R are return edges of e, their fundamental cycles are overlapping. Let e', e^L, e^R be the respective fork as shown in Figure 10, and let $lowpt(e^L) \leq lowpt(e^R)$ (otherwise exchange names). The *different*-constraint between ℓ_L, ℓ_R is introduced while backtracking over e', and we have $lowpt(e') < lowpt(\ell_R)$ because $lowpt(e') = lowpt(\ell_R)$ implies that ℓ_R is not involved in a constraint associated with e'. We distinguish three types of *different*-constraints (depicted in Figure 10) and show how to extract an inclusion-minimal subgraph preserving them.

Type 1: If $lowpt[\ell_L] > lowpt[\ell_R]$, then $lowpt\_edge[e^L] \neq \ell_L$ since $lowpt(e^L) < lowpt(\ell_R) \leq lowpt(\ell_L)$. Moreover, $lowpt(\ell_R) > lowpt(e^L) = lowpt(e')$, because if $lowpt(e^L) > lowpt(e')$, then ℓ_L and $lowpt\_edge[e^L]$ are subject to a *same*-constraint and thus ℓ_L is not the lowest edge in $P.L$. Hence, the funda-



(a) while merging on right side (b) while merging on left side

Fig. 11. One of these inclusion-minimal configurations is present when a subsequent *same*-constraint on ℓ_L, ℓ_R causes the planarity test to halt during constraint merging.

mental cycles of ℓ_L, ℓ_R and $lowpt\_edge[e^L]$ preserve the constraint forcing ℓ_L and ℓ_R on different sides; another cycle is added due to the *same*-constraint to form a subdivision of $K_{3,3}$.

Type 2: If $lowpt[\ell_L] = lowpt[\ell_R]$ and $lowpt(e^L) = lowpt(e^R)$, arguments symmetric to Case 1 imply that $\ell_R \neq lowpt\_edge[e^R]$ and that, in addition to the three cycles above, $C(lowpt\_edge[e^R])$ is needed to preserve the constraint. The overlap of $C(\ell_L)$ and $C(\ell_R)$ is removed to obtain a subdivision of $K_{3,3}$ when adding the final cycle.

Type 3: The final case is $lowpt[\ell_L] \leq lowpt[\ell_R]$ and $lowpt(e^L) < lowpt(e^R)$, since $lowpt[e^L] = lowpt[e^R]$ yields Case 2 if $lowpt(\ell_L) = lowpt(\ell_R)$, or Case 1 with the roles of L and R exchanged if $lowpt[\ell_L] < lowpt[\ell_R]$. It follows that $\ell_R = lowpt\_edge[e^R]$ for otherwise both would be subject to a *same*-constraint and ℓ_R would not be lowest in $P.R$. Hence, ℓ_L is in indirect conflict with ℓ_R via an edge $h_L \neq \ell_L$ which we can choose to be the highest in $P.L$. For ℓ_L and h_L to be forced into an interval, however, there must have been another back edge returning to $lowpt(e')$ (a back edge to a return point between $lowpt(e')$ and $lowpt(\ell_L)$ would replace ℓ_R as lowest in $P.R$). In addition to the four fundamental cycles of $\ell_L, \ell_R, lowpt\_edge[e']$ and h_L we therefore include $C(b)$, where b is the last edge that was made consistent with $lowpt\_edge[e']$. With the additional cycle added later due to the *same*-constraint we obtain a K_5 *minor*, i.e. a subgraph that can be turned into K_5 by contracting edges. Note that every K_5 minor that is not a subdivision of K_5 contains a subdivision of $K_{3,3}$.

All conditions distinguishing the three types can be tested in constant time using information obtained during the testing phase.

***Same*-constraint.** There are only two straightforward cases illustrated in Figure 11. Either the constraint is induced by the fundamental cycle of $lowpt\_edge[e]$ while merging the constraints associated with e_i on the right, or by $lowpt\_edge[e_i]$ while merging on the left. In either case only one cycle needs to be added to the subgraph extracted for the *different*-constraint.

The fundamental cycles forming a Kuratowski subgraph are extracted easily by traversing the DFS tree from the source of each of the at most six critical back edges downwards using *parent_edge* pointers. Forks of overlapping cycles are found along the way, and only the cycle with lower lowpoint needs to be continued. The time needed to extract a Kuratowski subgraph after the planarity test determined a constraint violation is thus proportional to the size of the union of at most six fundamental cycles.

Note that the failure configurations in Figure 9 are the prototypical combination of Type 1 and Type 3 *different*-constraints with a subsequent *same*-constraint that initiates merging on the left.

7 Discussion

We have reviewed the Left-Right Planarity Criterion (Theorem 3) and described a simple linear-time algorithm (Algorithm 1) based on it. While this is not a review of graph planarity, and many important references and developments are left out, some notes on closely related work seem in place.

7.1 Characterization

Historically, Kuratowski (1930) provides the first characterization of graph planarity. Given the discussion in Section 6, this characterization in terms of minimal forbidden subgraphs can be re-interpreted as identifying the cycle structures of $K_{3,3}$ and K_5 as the two minimal overlap configurations that prevent planar drawing.

Among various later characterizations, the seemingly most closely related criterion is due to Mac Lane (1937), because it is also formulated in terms of a representative set of cycles. Consider the set of all undirected cycles of a graph, and define the sum of two cycles as the symmetric difference of their edge sets. These together form a vector space, called *cycle space*. A *basis* of the cycle space is a minimum-cardinality set of cycles such that every cycle is the sum of some basis cycles.

Theorem 9 (Mac Lane's Planarity Criterion) *A graph is planar, if and only if it has a cycle basis in which every edge appears at most twice.*

For a better intuition, consider a planar drawing of a connected planar graph. Traversing each face in the drawing (say, inner faces clockwise, the outer face counterclockwise) yields the set of (directed) *facial cycles* forming a basis of the cycle space. As required, every edge is traversed exactly twice (once in each direction).

Any cycle basis for a graph G has cardinality $\mu(G) = m - n + \kappa(G)$, where $\kappa(G)$ is the number of connected components of G and $\mu(G)$ is called the *cyclomatic number* of G. This is exactly the number of non-tree edges of a spanning forest and, in fact, the fundamental cycles of any spanning forest induce a cycle basis.

The left-right criterion thus also asks for a cycle basis with a special property, namely that its elements, the (directed) fundamental cycles of a DFS orientation, can be bipartitioned such that all constraints associated with forks are satisfied.

The cycle bases considered in these criteria are therefore maximally distinct. While the basis cycles in Mac Lane's criterion are as different as possible (with each edge in at most two cycles), the basis cycles in the left-right criterion are as concentrated as possible (with their overlap forming a spanning forest).

7.2 Development

The earliest precursor of the left-right approach is a planarity characterization of Wu (1955), which states that a graph is planar, if and only if a certain system of linear equations has a solution. It was complemented by the concept of crossing chains in Tutte (1970), and refined to Boolean variables and fewer equations in the 1970s (see Wu 1985, 1986; Liu 1990). The variables in this smaller system are associated with the edges, and the equations represent constraints generated from configurations of overlapping cycles obtained from a spanning DFS forest. An alternative interpretation of the existence of a solution is that of balancing a constraint graph as in Section 5.2. Rosenstiehl (1980) gives an algebraic proof for this characterization.

This work was further developed in several papers, but the descriptions are rather incomplete, in particular with respect to linear-time implementation (de Fraysseix and Rosenstiehl, 1982, 1985; Xu, 1989; Cai, Han, and Tarjan, 1993).

Finally, de Fraysseix, Ossona de Mendez, and Rosenstiehl (2006) simplified the approach even further by concentrating on the single constraint-inducing configuration of Definition 2. While this paper is still incomplete and difficult to read, the linear-time implementation is described in just enough detail to provide a basis for replication. Among the differences to the present description is the maintenance and merging of constraints, since intervals are described as stacks rather than their extreme pairs of edges and there is a constraint stack for each edge rather than our global stack S. It turns out, however, that the most recent implementation in PIGALE (de Fraysseix and Ossona de Mendez, 2002) uses a similar representations.

The characterization of Kuratowski subgraphs in terms of configurations along a DFS spanning tree given in de Fraysseix and Ossona de Mendez (2003) and de Fraysseix (2008) leads to a linear-time extraction algorithm associated with the left-right approach. To the best of my knowledge, the version given here is original, and it is more directly based on the left-right characterization.

7.3 Algorithms

The first published polynomial-time planarity testing algorithm is due to Auslander and Parter (1961). It is based on an observation already noted above, namely that in a planar drawing of a graph every simple cycle forms a closed curve partitioning the plane into an inside and an outside region.

Consider the graph obtained by removing the edges of some simple cycle, but retaining copies of vertices on the cycle for every incident non-cycle edge. These vertices are called *attachments* and the connected components of the resulting graph are called *segments*. Clearly, each segment must be drawn completely inside or completely outside of the removed cycle, but a pair of segments must not be placed in the same region if their attachments interleave on the cycle. Planarity can thus be tested by recursively choosing cycles and sides. The related algorithm of Demoucron, Malgrange, and Pertuiset (1964) also starts from a simple cycle, but then iteratively chooses a path that can be added into one of the current faces. The algorithm is not only simple, but also has the unusual property to eagerly maintain a partial embedding that is not changed later on. Both algorithms require $\Omega(n^2)$ time, though.

In a graph-algorithmic milestone, the first linear-time planarity test was presented by Hopcroft and Tarjan (1974). Their approach is called *path-addition* because it refines that of Auslander and Parter (1961) by adding paths rather than segments, and in an order determined from a depth-first search of the graph. It took many years, though, until finally Mehlhorn and Mutzel (1996) complemented the algorithm with an $\mathcal{O}(n)$ embedding phase.

Recall how we observed in Section 3 that likewise-oriented cycles are nested if they overlap. Maybe because de Fraysseix and Rosenstiehl (1982) phrased the notions of left and right in terms of angles with the DFS tree rather than orientations of fundamental cycles, it has gone almost unnoticed that the left-right approach is yet another refinement of Auslander and Parter (1961) and Hopcroft and Tarjan (1974), progressing from segments to paths to edges. Together with Canfield and Williamson (1990) and Haeupler and Tarjan (2008) this observation instills hope that there may be a useful and elegant unification of path- and vertex-addition approaches including the two most efficient versions of de Fraysseix, Ossona de Mendez, and Rosenstiehl (2006) and Boyer and Myrvold (2004).

References

Auslander, L., Parter, S. V., 1961. On imbedding graphs in the sphere. Journal of Mathematics and Mechanics 10 (3), 517–523.

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

Boyer, J. M., Cortese, P.-F., Patrignani, M., Di Battista, G., 2004. Stop minding your P's and Q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. In: Liotta, G. (Ed.), Proc. Intl. Symp. Graph Drawing (GD '03). Vol. 2912 of LNCS. Springer-Verlag, pp. 25–36.

Boyer, J. M., Myrvold, W. J., 2004. On the cutting edge: Simplified $\mathcal{O}(n)$ planarity by edge additon. Journal of Graph Algorithms and Applications 8 (3), 241–273.

Cai, J., 1993. Counting embeddings of planar graphs using DFS trees. SIAM Journal on Discrete Mathematics 6 (3), 335–352.

Cai, J., Han, X., Tarjan, R. E., 1993. An $\mathcal{O}(m \log n)$-time algorithm for the maximal planar subgraph problem. SIAM Journal on Computing 22 (6), 1142–1162.

Canfield, E. R., Williamson, S. G., 1990. The two basic linear time planarity algorithms: Are they the same? Linear and Multilinear Algebra 26, 243–265.

Chiba, N., Nishizeki, T., 1988. Planar Graphs: Theory and Algorithms. North-Holland.

de Fraysseix, H., 2008. Trémaux trees and planarity. Electronic Notes in Discrete Mathematics 31, 169–180.

de Fraysseix, H., Ossona de Mendez, P., 2002. Pigale: Public implementation of a graph algorithm library and editor, software project at `pigale.sourceforge.net` (GPL License).

de Fraysseix, H., Ossona de Mendez, P., 2003. On cotree-critical and DFS cotree-critical graphs. Journal of Graph Algorithms and Applications 7 (4), 411–427.

de Fraysseix, H., Ossona de Mendez, P., Rosenstiehl, P., 2006. Trémaux trees and planarity. International Journal of Foundations of Computer Science 17 (5), 1017–1029.

de Fraysseix, H., Rosenstiehl, P., 1982. A depth-first characterization of planarity. Annals of Discrete Mathematics 13, 75–80.

de Fraysseix, H., Rosenstiehl, P., 1985. A characterization of planar graphs by Trémaux orders. Combinatorica 5 (2), 127–135.

Demoucron, G., Malgrange, Y., Pertuiset, R., 1964. Graphes planaires: Reconnaissance et construction de représentations planaires topologiques. Revue Fran cais Recherche Opérationnelle 8 (30), 33–47.

Haeupler, B., Tarjan, R. E., 2008. Planarity algorithms via PQ-trees. Electronic Notes in Discrete Mathematics 31, 143–149.

Harary, F., 1953. On the notion of balance of a signed graph. Michigan Mathematical Journal 2, 143–146 (with addendum).

Hopcroft, J. E., Tarjan, R. E., 1974. Efficient planarity testing. Journal of the ACM 21 (4), 549–568.

Kuratowski, K., 1930. Sur le problèm des courbes gauches en Topologie. Fundamenta Mathematicae 15, 271–283.

Lempel, A., Even, S., Cederbaum, I., 1967. An algorithm for planarity testing of graphs. In: Rosenstiehl, P. (Ed.), Proc. Intl. Symp. Theory of Graphs (Rome, July 1966). Gordon and Breach, pp. 215–232.

Liu, Y., 1990. A Boolean characterization of planarity and planar embeddings of graphs. Annals of Operations Research 24, 165–174.

Mac Lane, S., 1937. A combinatorial condition for planar graphs. Fundamenta Mathematicae 28, 22–32.

Mehlhorn, K., Mutzel, P., 1996. On the embedding phase of the hopcroft and tarjan planarity testing algorithm. Algorithmica 16 (2), 233–242.

Nishizeki, T., Rahman, M. S., 2004. Planar Graph Drawing. Vol. 12 of Lecture Notes Series on Computing. World Scientific.

Rosenstiehl, P., 1980. Preuve algèbrique du critère de planarite de Wu-Liu. Annals of Discrete Mathematics 9, 67–78.

Tutte, W. T., 1970. Toward a theory of crossing numbers. Journal of Combinatorial Theory 8, 45–53.

Wu, W., 1955. On the realization of complexes in Euclidean space I. Acta Mathematica Sinica 5, 505–552, English version in *American Mathematical Society Translations, Series 2* 78:137–184, 1968.

Wu, W., 1985. On the planar imbedding of linear graphs. Journal of System Sciences and Mathematical Sciences 5 (4), 290–302.

Wu, W., 1986. On the planar imbedding of linear graphs (continued). Journal of System Sciences and Mathematical Sciences 6 (1), 23–35.

Xu, W., 1989. An improved algorithm for planarity testing based on Wu-Liu's criterion. Annals of the New York Academy of Sciences 576, 641–652.