Dissertation

Circle Planarity of Level Graphs

Christian Bachmaier

Supervisor
Prof. Dr. Franz J. Brandenburg

14 May 2004

Dissertation for the aquisition of the degree of a doctor in natural sciences at the Faculty of Mathematics and Computer Science of the University of Passau.

1st referee: Prof. Dr. Franz J. Brandenburg, University of Passau
2nd referee: Prof. Dr. Ulrik Brandes, University of Konstanz

Abstract

In this thesis we generalise the notion of level planar graphs in two directions: track planarity and radial planarity. Our main results are linear time algorithms both for the planarity test and for the computation of an embedding, and thus a drawing. Our algorithms use and generalise PQ-trees, which are a data structure for efficient planarity tests.

A graph is a level graph, if it has a partition of the vertices in levels such that the vertices of each level can be placed on a horizontal line and the edges are strictly downwards. It is level planar if there are no edge crossings. Level planarity can be tested efficiently in linear time by sophisticated and complex algorithms.

Level graphs exclude horizontal edges between vertices on the same level. Such edges are allowed by our track graphs. In radial level graphs the vertices of each level are placed on concentric circles and the edges are outwards. We characterise essential differences between level and radial level planar graphs, which are expressed by level non-planar biconnected components called rings. The presence of rings introduces the particular problem of the nesting of non-connected components. Further, we study forbidden subgraphs which destroy radial level planarity. The track and circle extensions are combined to form circle graphs, which allow edges along the concentric circles.

Level planar graphs arise as a specialisation of directed acyclic graphs that are usually drawn by the Sugiyama algorithm, which avoids edge crossings. Applications of level or track planar drawings include for example biochemical pathways, entity relationship and UML class diagrams, or flow charts which occur for example in project management. Typical applications of radial drawings are social networks.

Preface

It is a pleasure to express my gratitude to those to whom I am indebted, directly or indirectly in writing this thesis. First of all, thanks are due to my supervisor, Professor Dr. Franz J. Brandenburg, who introduced me to graph drawing. Unlike me, he was sure even before I finished my diploma that I was capable of a work like this. Therefore he was the one who encouraged me to have a "look" at hierarchical graphs. I am very grateful to Franz Brandenburg for supporting my work in every aspect I could imagine. He gave me the opportunity and freedom to take part in the research in computer science, and to take part in conferences. It were the Graph Drawing Conferences that have been very important for me. Without his care and friendship, I would not have been able to complete this work.

Further I am very grateful to Professor Dr. Ulrik Brandes who always took the time to discuss various problems with me. He always had very helpful suggestions. My thanks also go to Dr. Sebastian Leipert for explaining his algorithm in detail to me. I wish to thank Andi Pick, Marcus Raitner, and Mike Forster for never getting tired of answering my questions about C++ and STL, and for providing technical support for the implementation of the prototype. I am especially grateful to Mike for fruitful cooperation on various problems related to level planarity and for writing various papers together with me. Last but not least I wish to thank Ruth Eades for carefully proofreading this thesis.

Christian Bachmaier

Contents

1 Introduction

In the mid 1980s graphic workstations became standard for information and software engineers. Since then, visualisation of relational information has became more and more indispensable. Such information is commonly modelled by graphs with vertices for the entities and edges for the relationships. A graph has the particular advantage of a natural visualisation. The human understanding of such a relational model depends heavily on whether the drawings can convey the information easily to the user. As the Chinese proverb says, "a good picture is worth a thousand words", but a poor one can be confusing and misleading. Therefore the graph drawing community is concerned with finding good drawings of graphs. The central problem in automated graph drawing is designing an algorithm which assigns a location to each vertex and computes a routing for each edge while optimising some aesthetic criteria. This is called the *graph drawing problem*.

There are many criteria for measuring the quality of drawings, e. g., few edge bends in straight line drawings, orthogonal line segments for the edges, small display area, representation of a common flow direction, good spatial and angular resolution, or recognisable symmetries. For an extensive list see [38]. Although their respective importance depends in most cases on the application and certain criteria cannot be optimised simultaneously, there are several which most drawings should have in common. One of the most important criteria, as indicated by the empirical studies of [125], is the edge crossing aesthetic criterion. The intersections of the curves used for drawing edges should be minimised. Planar graphs allow a drawing in the plane without any edge crossing.

In practice many network structures are hierarchically organised and contain some global direction like a time line, flow, or inheritance, e. g., in software engineering, project management, or database design. Such a network corresponds to a directed acyclic graph, where the vertices are restricted to placement on horizontal

lines (levels) according to their position in the hierarchy, and where the edge directions indicate the direction of information from lower to higher level. For drawing such structures the graph is usually layouted with the well known Sugiyama algorithm which reduces crossings in order to increase readability. Since all its phases are NP-hard problems [70], it uses heuristics. This is also true for its third phase, the reduction of the *crossing number*. The problem remains NP-hard even if there are only two levels and the ordering of the vertices on one level is fixed [52, 56]. As a consequence a crossing-free drawing cannot be guaranteed even if one exists. But exactly in this case it is especially desirable to obtain a planar drawing, because if a human looks at an automatically generated drawing, changing positions of only a few vertices to eliminate crossings seems to be very easy to her. As a consequence she might think that the unchanged layout is a bad layout. Fortunately, there is a linear time algorithm for efficiently detecting such kind of planarity, the so-called level planarity. Thus it is possible to check on level planarity in a preprocessing step and to avoid this dissatisfying situation.

Level planarity excludes horizontal edges between vertices in the same level. But horizontal edges occur in practice, e. g., as associations in UML diagrams. In this thesis we define track planarity, where these edges are allowed. We present a linear time algorithm which is based on a reduction of track planarity to level planarity. It can detect track planarity and construct a track planar embedding of track planar graphs, i. e., the topological structure of a planar drawing. Another new concept are radial level graphs, where the vertices of each level are placed on concentric circles instead of horizontal lines.

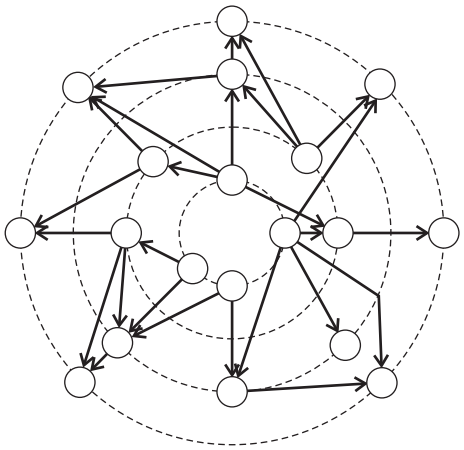


Figure 1.1. The radial drawing from the cover page of Kaufmann and Wagner [105].

A motivation for radial level planar graphs are the radial drawings of social networks studied in [18, 19], where the vertices are constrained to lie on radial levels according to their centrality. Here structural centrality is mapped to a geometric one. Simply speaking, a social network [105] is obtained by taking a group of people as vertices and inserting edges based on some abstract relationships between the

individuals. This research is motivated by the belief that visualising social structures and detecting patterns gives insight into how a society works, how individuals interact with society, and even why certain societies or individuals are more successful than others. Aspects taken into account may be sociological, economical, demographical, ethnical, or medical, and are used for various purposes on all scales.

We establish essential differences between level and radial level planar graphs, which are expressed by level non-planar biconnected components called rings. As our main result we present a linear time algorithm for detecting and embedding radial level planar graphs. The basis of this new algorithm is the new PQR-tree data structure which is a generalisation of PQ-trees. R-nodes represent biconnected components called rings, which are radial level planar but not level planar. Their presence entails the nesting of non-connected components. In a further step, the track and radial level extensions are combined to form circle graphs, which allow edges along the concentric circles. In analogy to Kuratowski's forbidden subgraphs for planarity, we study the forbidden subgraphs for radial level planarity. This gives a deeper insight into the combinatorial structures that are allowed in radial level planar graphs but forbidden in level planar ones.

1.1 Preliminaries

In this section we recall some basic concepts of graph theory from standard textbooks on graphs, e. g., from [30, 31, 44, 124]. Throughout the thesis, we freely adapt the terminology of [95, 97, 99, 112] to suit our needs. The established visualisation methods for different classes of graphs are summarised in [38, 105, 147], which cover the basics of graph drawing.

1.1.1 Graphs

A *graph* $G = (V, E)$ is a finite non-empty set of *vertices* V which are connected by *edges* E. It is *directed* (a *digraph*) if the edges are ordered pairs of vertices and *undirected* if the order does not matter. The vertices u and v are the *end vertices* of an edge $e = (u, v)$ and are called *adjacent* to each other. Then it is said that u and v are *neighbours*. An end vertex is called *incident* to its edge(s) and vice versa. A vertex is called *isolated* if it has no incident edges. Consider two directed edges (u, v) and (v, w). Then (u, v) is called an *incoming edge* of v and (v, w) is called *outgoing edge* of v. The *degree* of a vertex is the number of incident edges.

Usually graphs are visualised in such a way that vertices are drawn as points, circles, or squares and edges as curves between vertices. For a digraph an edge $e = (u, v)$ is drawn as an arrow from its *source vertex* u to its *target vertex* v. For an undirected graph an edge is drawn as simple line between its end vertices, not as an arrow. This should indicate the symmetry.

Parallel edges are two or more edges with the same end vertices. A *reflex edge* is an edge (v, v). A *simple graph* is a graph without parallel and reflex edges. Without

loss of generality, we consider only simple graphs from now on.

A graph $G' = (V', E')$ is called a *subgraph* of the graph $G = (V, E)$ if $V' \subseteq V$ and $E' = \{ (u, v) \mid u, v \in V' \} \subseteq E$ hold. A graph is said to be *complete* if every vertex $v \in V$ is adjacent to every other vertex $w \in V - \{v\}$. A complete graph with n vertices is denoted by K_n, see Figure 2.1(b) for an example. $K_{n,m}$ is a graph $(V \dot{\cup} W, E)$ with $E = V \times W$.

Every graph $G = (V, E)$ can be traversed by a *breadth first search* (BFS) using a queue or by a *depth first search* (DFS) using a stack or recursion in $\mathcal{O}(|V| + |E|)$ time.

1.1.2 Connectivity

A *directed* or an *undirected path* $P = (v_1, \ldots, v_k)$ is a sequence of vertices v_i with $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. P is *simple* if all vertices except the ends are distinct. P is a *cycle* if $v_1 = v_k$ for $k > 2$. It is common to write $v_1 \rightarrow^* v_k$ for P. The *(graph theoretical) length of a path* is the number of edges on that path. A *chain* is a path containing only vertices of degree 2 except the end vertices. A directed edge $e = (u, v)$ is a *transitive edge* if there is a directed path from u to v not containing e.

Two vertices u, v are *connected* in $G = (V, E)$ if there exists a path between them. G is connected if any pair of its vertices is connected. Every maximum subgraph that is connected is called a *connected component* or simply *component* of G. Thus a *disconnected graph* has at least two components. A *cut vertex* is a vertex whose removal increases the number of components. Thus if G is connected, at least one vertex has to be removed in order to disconnect it. If no such vertex exists, G is called *biconnected*. A pair of vertices $u, v \in V$ in a biconnected graph is called a *split pair* if their removal disconnects the graph. G is called *triconnected* if there is no split pair.

1.1.3 Trees

A digraph $T = (V, E)$ is a *(rooted) tree* if it contains no cycle, there is exactly one vertex called the *root* which is not the target of any edge in E, and each other vertex is the target of exactly one edge. The vertices of T which are not a source vertex of any edge are called *leaves*. All other vertices are called *internal vertices*. The source vertex u of an incoming edge (u, v) of a target vertex v is called the *parent* of v. On the other hand, the target vertex w of every outgoing edge (v, w) of a vertex v is called a *child* of v. Vertices with the same parent are called *siblings*. Any vertex u for which a path $u \rightarrow^* v$ exists is called an *ancestor* of v. Similarly, any vertex w for which a path $v \rightarrow^* w$ exists is called a *descendant* of v. The length of a path from the root to a vertex v is called the *depth* of v. The largest depth of any vertex in T is called the *height* of T. By definition a tree is always connected. A *forest* is a set of trees.

An *ordered tree* is a tree in which the children of each vertex are ordered from left to right. Two children of the same parent are said to be *direct siblings* if they are siblings and appear consecutively in the children order of their parent.

1.1.4 DAGs

A DAG is a *directed acyclic graph*. Further, more than one root vertex with no incoming edges is allowed. In a DAG, a vertex with no incoming edges is called a *source* and a vertex with no outgoing edges is called a *sink*. Each DAG $D = (V, E)$ has a *topological sorting* of its vertices, which is a linear ordering of all its vertices $v \in V$ such that if E contains a directed edge (u, v) then u appears previous to v in this ordering. Such a vertex ordering is not unique and can be found efficiently in $\mathcal{O}(|V|)$ time by successively removing all sources.

1.1.5 Level Graphs

Now we specialise to k-level graphs, which is one of the central notions of this thesis. A *k-level graph* $G = (V, E, \phi)$ with $k \leq |V|$ is a graph with a level assignment $\phi\colon V \to \{1, 2, \ldots, k\}$ that partitions the vertex set into k pairwise disjoint subsets $V = V^1 \dot\cup V^2 \dot\cup \cdots \dot\cup V^k$, $V^j = \phi^{-1}(j)$, $1 \leq j \leq k$, such that $\phi(u) \neq \phi(v)$ for each edge $(u, v) \in E$. A k-level graph is *proper* if each edge $(u, v) \in E$ is *short*, i. e., $|\phi(u) - \phi(v)| = 1$. Otherwise, it has a *long edge* which spans several levels. Level graphs are a generalisation of *bipartite graphs* which have only two levels, e. g., a $K_{n,m}$ is a *complete bipartite graph*. Sources and sinks in a level graph are defined analogously to DAGs using the implicit edge direction from lower to higher levels. A *hierarchy* $G = (V, E, \phi)$ as defined by [39] is a level graph with all sources on the first level V^1. If G is a hierarchy with more than one vertex in V^1, we can add a new level V^0 containing exactly one dummy source vertex which is connected to every $v \in V^1$. Such a transformation does not modify the planarity of the graph and its size remains $\mathcal{O}(|V|)$. Thus we consider only hierarchies with $|V^1| = 1$. Please do not confuse hierarchy with *hierarchical graph*. Some people use this term as a synonym for level graph, e. g., [51, 53].

1.2 Sugiyama Algorithm

The display of hierarchical structures is an important issue in automatic graph drawing. DAGs and trees are usually drawn such that the vertices are placed on horizontal levels, and the edges are straight lines or y-monotone polylines. This technique is used by the Sugiyama[1] algorithm, the most common algorithm for drawing DAGs and level graphs [38, 54, 105, 148]. The algorithm operates in four phases: In the

[1] Although there was some initial work on level drawings by Warfield [157] and Carpano [24] before, this approach is commonly attributed to Sugiyama et al. [148].

first phase, called *cycle removal*, the input graph is made acyclic by reversing appropriate edges. Reversing a minimum set of edges is known as the feedback arc set problem and is NP-hard [70, 104]. During the second phase, called *level assignment*, the vertices are assigned to horizontal levels. Thereby minimising both the height and the width is NP-hard as a simple reduction of the multiprocessor scheduling problem shows, see [70]. The first two steps are not necessary for level graphs. Before the third phase, long edges are subdivided into short edges by the introduction of up to $\mathcal{O}(|V|^2)$ dummy vertices. These dummy vertices represent potential bends of long edges. In the third phase, called *crossing reduction*, an ordering of the vertices within a level is computed such that the number of crossings is reduced. The fourth phase, called *horizontal coordinate assignment*, computes an x-coordinate for every vertex. The fourth phase is usually constrained to preserve the ordering determined in the third phase, and to introduce a minimum separation space between vertices within a level. The y-coordinates are given by the levels. Finally, the dummy vertices introduced before the crossing reduction are removed and replaced by edge bends.

However, crossing minimisation in phase three is also NP-hard, see Garey and Johnson [71]. This is even the case if there are only two levels [94, p. 97] and the vertices of one level are fixed [52, 56]. The two level crossing problem is fundamental and has received great attention in literature [38]. Therefore the Sugiyama algorithm uses one (or more) of the many and intensively investigated heuristics in its third phase. In the best case no crossings remain at all and the graph is drawn level planar. But a heuristic does not guarantee a planar drawing even if one exists, although in this case it is especially desirable to avoid crossings. Fortunately, there are efficient algorithms for testing this property and for constructing a drawing without a crossing, see Chapter 3.

1.3 Overview

In the next chapter we discuss the planarity of graphs. This section is intended as an introduction to the linear level planarity testing and embedding algorithm of Jünger, Leipert, and Mutzel [95–97, 99, 100, 112] which is summarised in Chapter 3 beside a survey of other results related to level planarity. This algorithm is the basis of our radial level planarity testing and embedding algorithm in Chapter 5. We show in Chapter 4 how track planar graphs can be recognised, i. e., level graphs with additional edges between vertices on the same level. In Chapter 6 we combine track and radial level planarity to circle planarity, where level graphs can have edges within a radial level. In Chapter 7 we give combinatorial characterisations of graphs that are not radial level planar and elaborate the essential differences between level and radial level planarity. The discussions of Chapter 8 are intended to summarise the results of this work and to suggest directions for further investigation in the wider sense of the topic level planarity. The appendix shows technical details of symmetrical lists and of our prototype implementation.

2 Planarity

The study of planar graphs has a long history in mathematics, going back to Leonard Euler in the 18th century. As a consequence, planar graphs are well investigated and there are many known facts about them and their drawings, e. g., see [102, 123]. Since the notions of graph planarity and graph embedding are fundamental here, they are discussed in detail.

2.1 Definition of Planarity

Consider a graph $G = (V, E)$ drawn in the plane such that each vertex is represented by a distinct point. Each edge is drawn as a continuous curve between its two end points. If no two edges share any point except their possible common ends, G is said to be planar. Before we treat the question of efficiently testing whether a given graph is planar in the next sections, we use the rest of this section to discuss some classical work concerning planar graphs.

Theorem 2.1 (Euler 1736). *Let $G = (V, E)$ be a non-empty connected planar graph. Then the number of faces f satisfies*

$$|V| + f - |E| = 2. \tag{2.1}$$

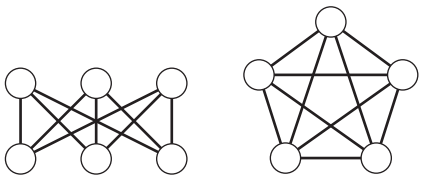
Corollary 2.1 (Euler). *If $G = (V, E)$ is a non-empty connected planar graph with no parallel and reflex edges and $|V| > 2$ then*

$$|E| \leq 3|V| - 6. \tag{2.2}$$

One of the most outstanding result is Kuratowski's theorem, which gives a simple certificate whether a graph is planar.

Theorem 2.2 (Kuratowski). *A graph G is non-planar if and only if there is a subgraph of G which is homeomorphic to either $K_{3,3}$ or K_5.*

Two graphs are *homeomorphic* if they can both be obtained from a common graph by a sequence of replacing edges by paths. In appearance, homeomorphic graphs look like ones that have extra vertices added or removed from edges.



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

Figure 2.1. Kuratowski subgraphs

For planarity testing it is useful to have a special numbering of the vertices. Given a graph $G = (V, E)$ with an edge $(s, t) \in E$ with $s \neq t$. An st-numbering of G is a bijective numbering $st : V \to \{1, 2, \ldots, |V|\}$ of its vertices such that $st(s) = 1$, $st(t) = |V|$, and that for every vertex $v \in V - \{s, t\}$ there are two adjacent vertices $u, w \in V$ with $st(u) < st(v) < st(w)$. A graph with such a numbering is called an st-graph. In [113] it is shown that every biconnected graph has an st-numbering. In [59] an $\mathcal{O}(|V| + |E|)$ time algorithm for finding such a numbering is given for biconnected graphs.

After planarity of a graph is approved, a planar drawing visualises the correctness of this result. A *straight-line drawing* of a graph is a drawing where every edge is drawn as a straight line. Fáry [60], Stein [145], Steinitz and Rademacher [146], and Wagner [156] have shown independently that every planar graph admits a straight-line drawing. For producing (straight-line) drawings of planar graphs see for example [27, 28, 35, 79, 102, 137, 153, 154].

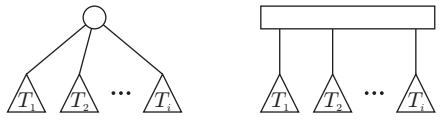
2.2 PQ-Trees

In this section we explain in detail *PQ-trees* which were introduced by Booth and Lueker [13] for the consecutive ones property in matrices. This data structure is the basis of the planarity testing algorithm described in Section 2.3 and all subsequent algorithms.

A PQ-tree represents the set of permutations of a finite set S, where the members of specified subsets $S' \subseteq S$ occur consecutively. It is a rooted, ordered tree with the leaves representing the elements of S. The possible permutations of the leaves are encoded by the combination of the two types of internal nodes, *P-nodes* and *Q-nodes*. P-nodes are drawn as circles and Q-nodes are drawn as rectangles. For PQ-leaves only their label or the element they represent is drawn. Sub-PQ-trees are

abstracted by triangles. Each of the following three operations will construct a valid PQ-tree.

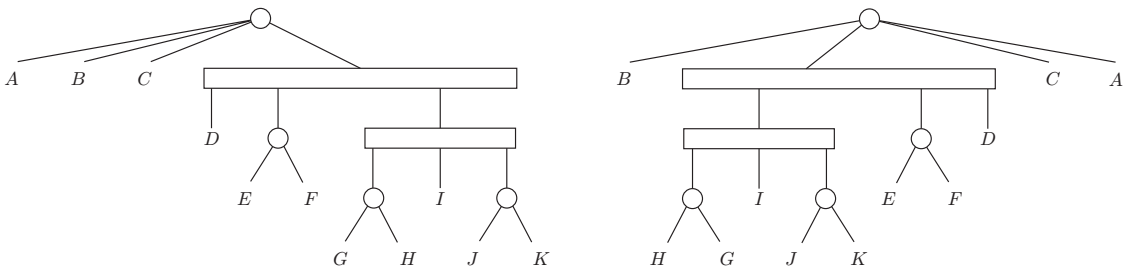
1. Every element $s \in S$ is a PQ-tree whose root is s.
2. If $T_1, T_2, \ldots, T_i$, $i \geq 2$, are PQ-trees then the structure shown in Figure 2.2(a) is a PQ-tree whose root is a P-node.
3. If $T_1, T_2, \ldots, T_i$, $i \geq 2$, are PQ-trees then the structure shown in Figure 2.2(b) is a PQ-tree whose root is a Q-node.



(a) A P-node (b) A Q-node

Figure 2.2. Drawing of internal PQ-nodes

Two PQ-trees are equivalent if and only if one can be transformed into the other by applying equivalence transformations. There are two types of equivalence transformations: The children of a P-node can be permuted arbitrarily, which means that there is no left-to-right-order among them. The children of a Q-node are ordered and only reversion is allowed. Hence, the same two children will always remain *endmost*, i. e., *leftmost* or *rightmost*, and all others will remain *interior*, i. e., not endmost. In addition, each interior child of a Q-node always has the same two direct siblings. Since every internal node has at least two children, the number of internal nodes is at most the number of leaves. Reading the leaves of a PQ-tree T from left to right yields its *frontier*, frontier(T). The frontier of a PQ-tree is one admissible permutation of the set S. See Figure 2.3 for an example.



(a) (b)

Figure 2.3. Two equivalent PQ-trees over a set $S = \{A, B, \ldots, K\}$

2.2.1 Reduce

As the elements of each new subset $S' \subseteq S$ are constrained to appear together, the number of admissible permutations is reduced. Given S' and a PQ-tree T, a reconstruction T' of T is needed such that its admissible permutations are exactly the original permutations in which the leaves selected by S' occur consecutively. This is achieved by the main operation on PQ-trees which is called *reduction* with respect to S'.

PQ-leaves representing elements of S' are called *pertinent*. After the reduction, the so-called *pertinent subtree* is the subtree of minimum height containing all pertinent PQ-leaves. This pertinent subtree is unique with respect to S'. Its root is called the *pertinent root*. A PQ-node with at least one pertinent child different from the pertinent root is called pertinent too. In illustrations of PQ-trees a grey shading indicates that a node or a subtree is pertinent. Nodes which are detected during reduction as having only pertinent children are marked as *full* and nodes having only non-pertinent children are marked as *empty*, actually those remain empty. While pertinent PQ-leaves are always marked as full, the non-pertinent ones are always marked as empty, accordingly. All other nodes, i. e., nodes with pertinent and non-pertinent children at the same time, are marked as *partial*. The method REDUCE performs the reduction. Its details are listed in Algorithm 2.1.

REDUCE is a bottom up strategy from the pertinent leaves to the pertinent root that uses a queue. X is the currently treated node. For every newly detected X, one of the later defined templates must fit to realise local changes within the tree or REDUCE fails. A failure means that it is impossible to make members of S' consecutive due to other constraints. Then the resulting PQ-tree T' is empty, i. e., there is no permutation with the given restrictions. P0, Q0, and marking non-pertinent leaves as empty are actually not executed because the algorithm starts only with pertinent leaves.

For efficiency reasons only the endmost children of a Q-node know their current parent whereas the interior children do not in general, cf. [13, 127]. Therefore, before each application of REDUCE, the method BUBBLE [13, p. 358] updates the father pointer of each child in the pertinent subtree to its correct value. This does not violate the time complexity of REDUCE, except for a single problem where efficiency does not allow a search of the pertinent root, see Figure 2.4. There the pertinent root is a Q-node where all its pertinent children are interior children and none of them knows their parent. But in this particular case the knowledge over the real parent is not necessary at all because a *pseudo Q-node* Z is introduced as a known parent to the children. Z does not know its father either. But even that is not necessary because Z is the new pertinent root. BUBBLE knows when it has reached the pertinent root by sophistically counting the pertinent leaves which are at the frontier of the subtree induced by the currently treated node.

The main part of Algorithm 2.1 is the pattern matching step which uses *templates*, cf. [13]. The left piece of a template is the *pattern* and the right piece is the *replacement*. Prior to the application of a template it may be necessary to permute

Algorithm 2.1. REDUCE

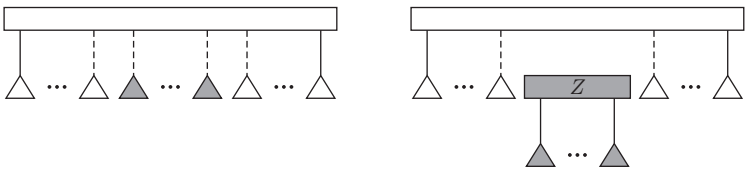
Input: A PQ-tree T and a subset $S' \subseteq S$
Output: The PQ-tree T'

Queue Q

foreach $s \in S'$ **do**
  $insert(Q, s)$
end

while Q not empty **do**
  $X = delete\_first(Q)$
  if some template applies to X **then**
    substitute the replacement for the pattern in T
  else
    return $T' \leftarrow \emptyset$ *// reduction impossible*
  end
  if $S' \subseteq \{ Y \mid X$ is an ancestor of $Y \}$ **then**
    exit from while loop *// reduction completed*
  end
  if every sibling of X has been matched **then**
    $insert(Q, parent(X))$
  end
end

return $T' \leftarrow T$



(a) The full children do not know their parent

(b) The new pseudo Q-node Z is the pertinent root

Figure 2.4. Introduction of a pseudo Q-node

the children of P-nodes or to reverse children of Q-nodes. All templates have special sub-cases in order to avoid creating chains and therefore to follow the definitions of PQ-trees. This is to ensure time complexities.

2.2.1.1 Templates for the Leaves

The node X is a leaf. If X is pertinent, i. e., $X \in S'$, then X is marked as full. Otherwise X is marked as empty. In both cases there are no structural changes in the PQ-tree.

2.2.1.2 P-Templates

Template P0 The node X is a P-node which has only empty children. Then X remains empty. There are no structural changes in the PQ-tree.

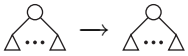


Figure 2.5. Template P0

Template P1 The node X is a P-node which has only full children. Then X is marked as full. There are no structural changes in the PQ-tree.

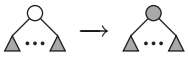


Figure 2.6. Template P1

Template P2 The node X is a P-node and the pertinent root. It has at least one empty and one full child. After grouping all full children with a new full P-node which is attached to X as a child, X is marked as partial.

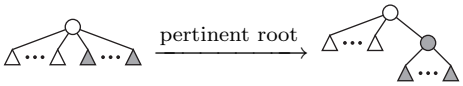


Figure 2.7. Template P2

Template P3 The node X is a P-node and not the pertinent root. It has at least one empty and one full child. Then the newly created Q-node Y which has two children is marked as partial. The empty P-node X grouping its empty children is attached to Y as the first child. A new full P-node which groups all full children of X is attached to Y as the second child.

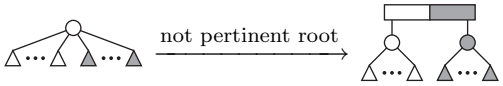


Figure 2.8. Template P3

Template P4 The node X is a P-node and the pertinent root. It has exactly one partial Q-node X' and an arbitrary number of other empty and full nodes as children. Then all full children of X are grouped by a new full P-node, which is attached to the pertinent end of X' as a child, and X is marked as partial.

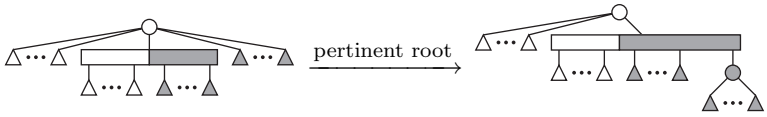


Figure 2.9. Template P4

Template P5 The node X is a P-node and not the pertinent root. It has exactly one partial Q-node X' and an arbitrary number of other empty and full nodes as children. Then the empty P-node X which groups all its empty children is attached to X' as a child at the empty end of X'. All full children are grouped by a new full P-node, which is attached as a child at the pertinent end of X'. The Q-node X' at the top of the replacement remains partial.

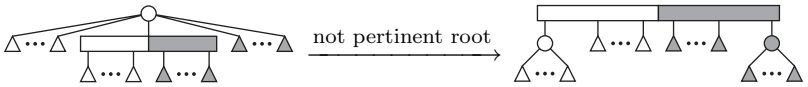


Figure 2.10. Template P5

Template P6 The node X is a P-node and the pertinent root. It has exactly two partial Q-nodes X' and X'' and an arbitrary number of other empty and full nodes as children. Then all full children of X are grouped by a new full P-node, which is attached as a child to the pertinent end of either X' or X''. Afterwards, X' and X'' are concatenated into one *doubly partial* Q-node which becomes the pertinent root.

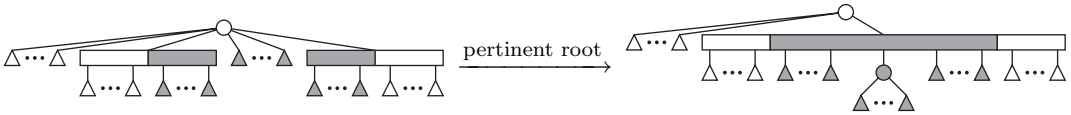


Figure 2.11. Template P6

2.2.1.3 Q-Templates

Template Q0 The node X is a Q-node which has only empty children. Then X remains empty. There are no structural changes in the PQ-tree.

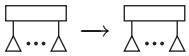


Figure 2.12. Template Q0

Template Q1 The node X is a Q-node which has only full children. Then X is marked as full. There are no structural changes in the PQ-tree.

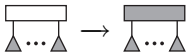


Figure 2.13. Template Q1

Template Q2 The node X is a Q-node which has at most one partial Q-node X' and an arbitrary number of other empty and full nodes as children. X' is located at the beginning of its consecutive pertinent sequence assuming that this sequence is located at the end of its children list. Then, after the children of X' are attached to X as children, X is marked as partial.

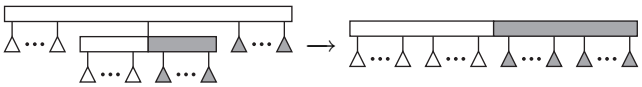


Figure 2.14. Template Q2

Template Q3 The node X is a Q-node and the pertinent root. It has at most two partial Q-nodes X' and X'' and an arbitrary number of other empty and full nodes as children. X' is located at the beginning and X'' is located at the end of its consecutive pertinent sequence. Thus all full children are located between them. Both endmost children must either be empty or partial, otherwise Template Q2 would apply. Then the children of X' and X'' are attached to X as children. X is marked as doubly partial and becomes the pertinent root.

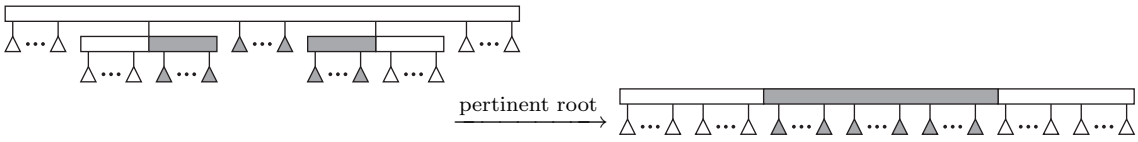


Figure 2.15. Template Q3

2.2.2 Replace Pertinent

There is another important operation on PQ-trees, replacing pertinent leaves. This task is performed by a method called REPLACE. It enlarges the set S by new members S^*, i. e., $S = S \cup S^*$. For this it replaces a consecutive subset $S' \subseteq S$ (actually the pertinent subtree(s) induced by S') after the reduction with respect to

S' with a newly created P-node whose children are leaves representing the members of S^*. If S^* contains only one element s then there is no P-node created and S' is directly replaced by a leaf representing s.

2.2.3 Improved Symmetric Lists

We store the set of children of a PQ-node in a *symmetric list*. Symmetric lists are an advanced data structure, which in addition to the standard operations on doubly connected lists supports both reversing the list and inserting a list into another one in constant time. This is essential for the linear running time of the vertex addition method [13] for planarity testing, i. e., for the REDUCE operation described above. Symmetric lists are described in detail in Appendix A.

2.3 Planarity Test

This section describes the well known linear time vertex addition method for planarity testing introduced by Lempel, Even, and Cederbaum (LEC) [13, 58, 113] in 1966. We recall this algorithm as described in [13] because it is the base for all subsequent algorithms. An implementation of this test can be found for example in GTL [76, 127]. Booth and Lueker [13] showed in 1976 how this test can be performed in $\mathcal{O}(|V|)$ time by the usage of PQ-trees. In 1985, Chiba, Nishizeki, Abe, and Ozawa [26] presented an extension of this method to compute a planar embedding of planar graphs.

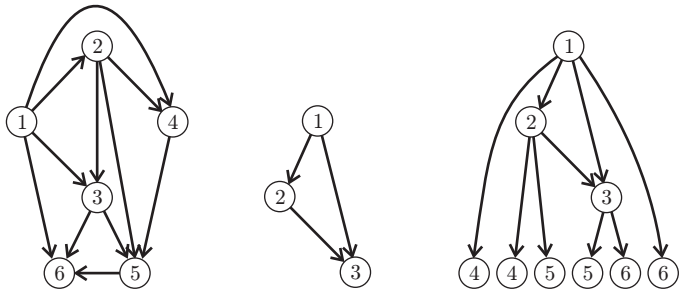
However, there are many other planarity tests, like Hopcroft and Tarjan's linear time path addition method [89] which is based in part on earlier work of Auslander and Parter [2] and Goldstein [73], online planarity testing with SPQR-trees [41], the $\mathcal{O}(|V|)$ time edge addition method of Boyer and Myrvold [16, 17] which is very fast in practice [14, 15], or tests for parallel machines with $|V|$ processors in $\mathcal{O}(\log^2 |V|)$ time [108, 109]. Further, there is the test of Williamson [158], the left-right algorithm of de Fraysseix and Rosenstiehl [36, 37], and the algorithm of Shih and Hsu [139, 140] based on the PC-tree data structure.

The idea of the vertex addition method is adding one additional vertex at each step. Previously drawn edges incident to this vertex are connected to it and newly discovered edges incident to it are drawn while their other endpoints are left unconnected.

We assume that the input graph $G = (V, E)$ is biconnected and has an st-numbering. If the graph is not biconnected then every biconnected component is treated separately or the algorithm of Mehlhorn, Mutzel, and Näher [119] is used to augment the graph with edges to achieve biconnectivity while potential planarity is not violated. It can be integrated in the test on biconnectivity which must be executed anyway. Recall that augmenting the graph with a minimum number of edges in order to make it biconnected is NP-hard [102]. Fortunately, this minimum number is not necessarily required here. The only restrictions are to add not more

than $\mathcal{O}(|E|)$ edges and the algorithm must not need more than $\mathcal{O}(|E|)$ time. Both is satisfied by the method described in [119].

Here edge direction means the implicit direction indicated by the *st*-numbering from the lower to the higher end vertex. Let $G_k = (V_k, E_k)$ be the directed subgraph of G induced by $V_k = \{ v \in V \mid st(v) \leq k \}$, $1 \leq k \leq |V|$. E_k consists of all edges of E whose both end vertices are in V_k. G_k is extended to B_k as follows: For each edge $(u, v) \in E$ with $u \in V_k$ and $v \in V - V_k$ the graph B_k gets a new *virtual vertex* and a *virtual edge* connecting u to this vertex. So there may be several separate virtual vertices in B_k that correspond to the same vertex in G. Virtual vertices are labelled as their counterparts in G. A representation of B_k with all virtual vertices drawn on a horizontal line is called a *bush form*, see Figure 2.16(c) for an example. Similar to level graphs, we draw a bush form with strictly downwards edges.



(a) G (b) G_3 (c) B_3

Figure 2.16. A graph and a bush form of it. The labels of the vertices indicate their name and their *st*-number at the same time

The algorithm proceeds successively "drawing" B_1, B_2, ..., $B_{|V|-1}$, $B_{|V|} = G$. If in the realisation of B_k all virtual vertices with *st*-number $k + 1$ are next to each other then it is easy to draw B_{k+1}. One joins all these virtual vertices into one vertex v and "pulls" it up from the horizontal line. Afterwards, all the virtual edges which emanate from v are added. See Figure 2.17.

It remains to explain how one can bring the virtual vertices with *st*-number $k+1$ next to each other, or if this is not possible to reject G as not planar. Therefore the PQ-tree data structure described in the last section is used. The leaves of the PQ-tree are the virtual vertices and the combination of its internal nodes saves their admissible permutations. This strategy leads to the fact that a P-node always represents a cut vertex and a Q-node represents a biconnected component of the currently scanned graph. By calling REDUCE on the set of leaves which correspond to the virtual vertices with number $k + 1$, the stored permutation set is restricted to those where these leaves lie side by side. If the reduction fails, i. e., a pattern occurs for which no template matches, G is not planar. The PQ-tree operation REPLACE extends the permutations with new leaves representing the new virtual vertices afterwards.

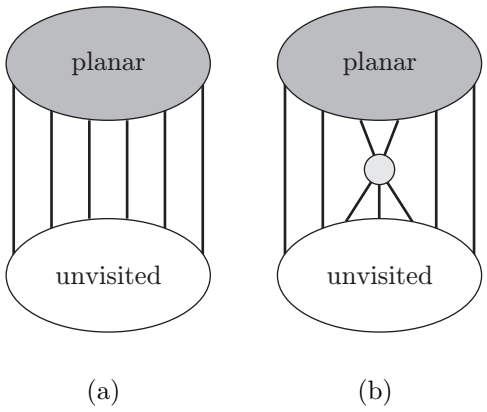


Figure 2.17. Vertices are reduced according to their *st*-numbering

2.4 Planar Embedding

In this section we describe the embedding algorithm of Chiba, Nishizeki, Abe, and Ozawa [26], which integrates seamlessly into the LEC algorithm described previously, whereby the $\mathcal{O}(|V|)$ time complexity is preserved.

2.4.1 Definition of a Planar Embedding

A planar drawing partitions the plane into topologically connected regions called *faces*. The *contour of a face* is the sequence of edges adjacent to the face. The unbounded face is called the *outer face* and all other faces are called *inner faces*. A planar drawing determines a cyclic ordering of the neighbours of each vertex v according to the clockwise sequence of the incident edges around v. It is irrelevant if the cyclic order is clockwise or counterclockwise as long as all adjacency lists are interpreted equally. To illustrate this, one can think of looking at a transparent slide from both sides on which a planar graph is drawn. Two planar drawings are equivalent if they determine the same cyclic ordering of the neighbour set. A *planar embedding* $\mathcal{E}$ is an equivalence class of planar drawings and is described by the cyclic order of the neighbours of each vertex v. That means it is defined by *ordered adjacency lists* $\mathcal{E}[v]$. A planar graph may have an exponential number of embeddings. If $\mathcal{E}$ is an embedding of an *st*-graph, it is also common to speak of an *st-embedding* $\mathcal{E}_{st}$.

Given an embedding, a generation of a drawing of the graph requires choosing an outer face, vertex positions and edge shapes. This is viewed as a separate problem, in part because it is application dependent. For example, the notion of what constitutes a suitable rendering of a graph may differ substantially if the graph represents an electronic circuit versus a hypertext book.

2.4.2 Computing a Planar Embedding

This section outlines how the LEC algorithm can be changed [26] such that it computes ordered adjacency lists of a planar input graph G. As in Section 2.3 it is assumed that G is biconnected and st-numbered.

In the first step an upward embedding $\mathcal{E}_u$ is computed, which is afterwards the base for computing $\mathcal{E}$. An *upward embedding* consists of ordered adjacency lists $\mathcal{E}_u[v]$ for each vertex $v \in V$ which only contain adjacent vertices with smaller st-number. For an example see Figure 2.18 which is an upward embedding of the graph shown in Figure 2.16(a).

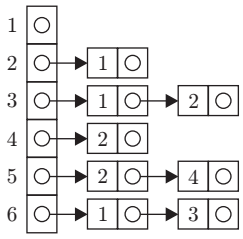
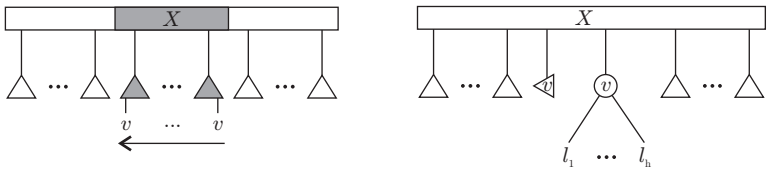


Figure 2.18. An upward embedding $\mathcal{E}_u$ of the graph in Figure 2.16(a)

2.4.2.1 Computation of an Upward st-Embedding $\mathcal{E}_u$

At the end of a successful reduction all leaves with the label $k+1$ lie in consecutive positions. Consider the corresponding bush form. Merging them into one vertex v determines the order of its incident edges to vertices with smaller st-number. That means the order of the virtual vertices $k+1$ corresponds to one admissible order of the incoming edges of v. Therefore traversing the leaves of the corresponding pertinent subtree from left to right with an ordered DFS delivers an upward embedding $\mathcal{E}_u[v]$. But in the course of the following reduction steps there may be one or more reversions of the biconnected component to which v is attached. Hence, it may be necessary to reverse $\mathcal{E}_u[v]$. This is exactly the case if the pertinent root is a partial Q-node X before the removal of the leaves $k+1$. Because of running time restrictions, the direction of the created list is only known relatively to the other children of X. Thus for saving this information a *direction indicator* is inserted as a child of X next to the newly created P-node which groups the new virtual edges of v. This indicator should reflect the direction in which the full children of X were traversed as $\mathcal{E}_u[v]$ was created. Of course all operations on PQ-trees must be updated to simply ignore the new node type direction indicator. In drawings of PQ-trees a direction indicator is depicted as a triangle pointing in its stored direction. Direction indicators representing the order of incoming edges of a vertex v are labelled with v.

Whenever a direction indicator d is found as an internal element of a pertinent sequence of a vertex w by REPLACE, it is removed together with this sequence from the PQ-tree and stored in $\mathcal{E}_u[w]$. The direction of d indicates the traversal direction of the ordered pertinent sequence at the time of its removal. At the end



(a) The arrow indicates the traversal direction of the pertinent sequence

(b) Figure 2.19(a) after REPLACE with $(v, l_i) \in E$, $1 \leq i \leq h$

Figure 2.19. Direction of traversing pertinent children and the resulting direction indicator

of such a modified planarity test, there is no direction indicator left in the PQ-tree. All indicators are stored in the upward embedding which has to be corrected now according to these indicators. Therefore all $\mathcal{E}_u[v]$ with v in descending st-order are traversed and each found direction indicator d is removed. Let u be the vertex which is the label of d. If the direction of d is opposite to the one of $\mathcal{E}_u[v]$ then the entire list $\mathcal{E}_u[u]$ is reversed.

2.4.2.2 Computation of an st-Embedding $\mathcal{E}_{st}$

As final step $\mathcal{E}_u$ is extended to an st-embedding $\mathcal{E}_{st}$. For this Chiba et al. use the method ENTIRE-EMBED shown in Algorithm 2.2, which is a backward DFS in $\mathcal{E}_u$. The DFS starts at t. Let w be a newly detected vertex. For each vertex $v \in \mathcal{E}_u[w]$, w is added at the front of the ordered list $\mathcal{E}_u[v]$.

Algorithm 2.2. ENTIRE-EMBED

Input: An upward embedding $\mathcal{E}_u$ of the graph $G = (V, E)$ and the sink vertex t
Output: An st-embedding $\mathcal{E}_{st}$

```
procedure DFS(w)
    visited[w] ← true
    foreach vertex v ∈ E_u[w] do
        insert w at the beginning of E_u[v]
        if visited[v] = true then
            DFS(v)
        end
    end
end
foreach v ∈ V do
    visited[v] ← false
end
DFS(t)
return E_st ← E_u
```

3

Level Planarity

The visualisation of hierarchical structures is an important issue in automated graph drawing. Such structures appear for example in project management (data flow or work flow charts), bioinformatics (biochemical pathways), database design (ER diagrams), and software engineering (UML diagrams) and are used to express dependencies due to time, reactions, or inheritance. They are modelled by directed acyclic graphs (DAGs). DAGs and trees are usually drawn such that the vertices are placed on horizontal levels, and the edges are drawn as straight lines or as y-monotone polylines. This technique is used by the Sugiyama algorithm, the most commonly used algorithm for drawing DAGs [38]. Because crossing minimisation is NP-hard, a possible planar embedding is not guaranteed, at least not in polynomial time. But exactly in this particular case it is especially desirable to obtain no crossings. Such drawings are easy to understand, as empirical experiments of Purchase [125, 126] have shown. Fortunately, there is a linear time algorithm to test level planarity, although the *level planarisation problem*, i. e., determining the minimum edge set whose removal eliminates crossings, is NP-hard even for two levels [52, 55, 152]. The level planarity algorithm can also compute an embedding in linear time if the graph is level planar. It is described after a few definitions and a section about the foundations of this topic in Section 3.3. Another important application of this algorithm is the detection of clustered level planar graphs [67]. There the vertices are not only constraint to lie on levels but also side by side within a rectangular box if they belong to the same cluster. For further information on clustered (planar) graphs see [21, 33, 63, 64, 78, 133–135].

3.1 Definition of Level Planarity

The level planarity problem [39, 85, 99] is the question whether or not a level graph G can be drawn in the plane such that all vertices of the j-th level V^j are placed on the j-th horizontal line $l_j = \{ (x, j) \mid x \in \mathbb{R} \}$, $1 \leq j \leq k$, and the edges are drawn as strictly y-monotone curves without crossings. Alternatively, level planar graphs are called *h-planar*, e. g., in [48, 51, 53]. An example of a level planar graph is given in Figure 3.1.

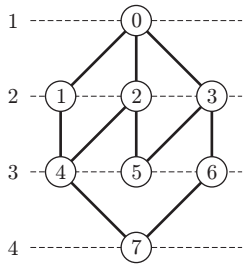


Figure 3.1. A level planar graph

For k-level graphs the partition of the set of vertices into levels is given. Finding a levelling is a different problem. Heath and Rosenberg [88] have shown that it is NP-hard whether a planar graph has a levelling into a proper level planar graph. In the non-proper case every planar graph has an $\mathcal{O}(|V|)$-level planar levelling with many levels and long edges. This follows for example from straight-line grid drawings of planar graphs [34, 35, 38, 137] or from visibility representations of planar graphs [38, 40, 42, 130, 149]. Both approaches, however, take into account neither the number of levels nor the length of the edges, e. g., for a minimisation. $\mathcal{O}(|V|)$ is also the lower bound for the number of levels, as the nested sequence of triangles of [34] show.

3.2 Foundations

The basis of our algorithm is the linear time algorithm of Jünger, Leipert, and Mutzel (JLM) [95–97, 99, 100, 112] for level planarity testing and embedding which in turn is based on the approach of Heath and Pemmaraju [85, 86]. These algorithms extend the level planarity testing algorithm for hierarchies of Di Battista and Nardelli [39] to arbitrary level graphs. The linear time algorithm of Chandramouli and Diwan [25] determines whether a triconnected DAG is level planar. Because the JLM algorithm is rather involved and difficult to implement, Healy and Kuusik [81] have presented a much simpler approach for the detection of level planarity. Their algorithm runs in $\mathcal{O}(|V|^2)$ time for proper level graphs and $\mathcal{O}(|V|^4)$ time in the general case. If an embedding is needed, the time complexity raises to $\mathcal{O}(|V|^3)$ and $\mathcal{O}(|V|^6)$, respectively. Dujmović et al. [48] have applied the concept of fixed parameter tractability to level planarity testing. They obtain linear running time if

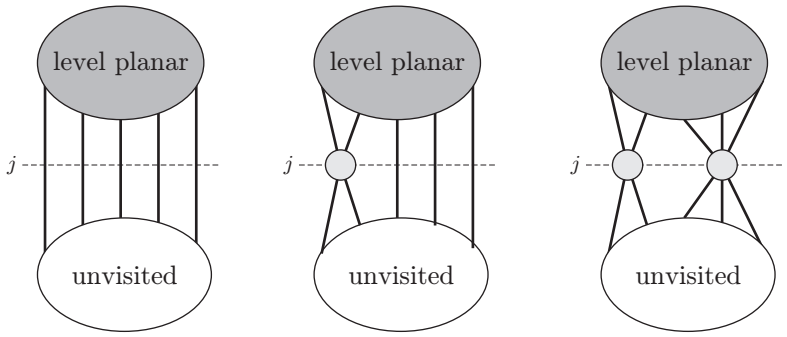
the number of levels is constant. Finally, Randerath et al. [128] presented a quadratic time reduction of level planarity of proper level graphs to the satisfiability problem of Boolean formulas in 2CNF. 2CNF formulas are solvable in linear time.

3.3 Level Planarity Testing

Since the JLM algorithm must be extended in various directions, its basic concepts are recalled. Let G be a k-level graph. The algorithm performs a top down sweep, processing the levels in ascending order. Let G^j be the subgraph induced by the vertices of the first j levels $V^1 \cup V^2 \cup \cdots \cup V^j$. For every G^j, $1 \leq j < k$, a set of admissible permutations of V^{j+1} is computed, which are the permutations of level planar embeddings of G^{j+1}. The input graph G is level planar if and only if the set of permutations of $G^k = G$ is non-empty.

In order to store and manipulate sets of admissible vertex permutations efficiently, the PQ-tree data structure of Booth and Lueker is used. As already mentioned in Section 2.3, a P-node represents a cut vertex and a Q-node represents a biconnected component of the visited part of the graph. The leaves represent edges to the unvisited part. Restrictions are introduced by edges towards the same vertex. If there are no permutations with the given restrictions, the PQ-tree is empty.

Figure 3.2 illustrates the sweep over one level. Consider the difference to Figure 2.17, where the reduction order is according to the *st*-ordering of the vertices. Here vertices are reduced from left to right for each level.



(a) Before reducing level j

(b) While reducing level j

(c) After reducing level j

Figure 3.2. Vertices are reduced level by level

The subgraph G^j induced by the first j levels is not necessarily connected. Thus a separate PQ-tree $T(F_i^j)$ is introduced for every component F_i^j of G^j with m_j such components and $1 \leq i \leq m_j$. $T(F_i^j)$ represents the set of admissible permutations of the vertices of F_i^j in V^j that appear in some level planar embedding of G^j. If two different components are adjacent to a common vertex v, their corresponding PQ-trees must be merged. $\mathcal{T}(G^j)$ denotes the set of all $T(F_i^j)$.

Algorithm 3.1. LEVEL-PLANARITY-TEST

Input: A level graph $G = (V^1 \dot\cup V^2 \dot\cup \ldots \dot\cup V^k, E, \phi)$

Output: A Boolean value indicating whether G is level planar

```
Initialise T(G^1)
for j ← 1 to k − 1 do
    T(G^{j+1}) ← CHECK-LEVEL(T(G^j), V^{j+1})
    if T(G^{j+1}) = ∅ then
        return false
    end
end
return true
```

A formal description of the LEVEL-PLANARITY-TEST algorithm is given by Algorithm 3.1. All operations are directly applied to the PQ-trees and not to the graph. The procedure CHECK-LEVEL in Algorithm 3.2 is a sweep over a single level j, divided into a first and a second reduction phase.

Algorithm 3.2. CHECK-LEVEL

Input: The PQ-trees $\mathcal{T}(G^j)$ and the vertices V^{j+1} of the next level

Output: The PQ-trees $\mathcal{T}(G^{j+1})$ of the next level

```
T(G^j) ← FIRST-REDUCTION-PHASE(T(G^j), V^{j+1})
if T(G^j) = ∅ then
    return ∅
end
T(G^j) ← SECOND-REDUCTION-PHASE(T(G^j), V^{j+1})
if T(G^j) = ∅ then
    return ∅
end
T(G^j) ← FINAL-UPDATES(T(G^j), V^{j+1})
return T(G^{j+1}) ← T(G^j)
```

The following describes the *first reduction phase* as formally shown in Algorithm 3.3. Define H_i^j to be the *extended form* of F_i^j which consists of F_i^j and some new *virtual vertices* and *virtual edges*. For every edge $(u, v) \in E$ with $u \in V(F_i^j) \cap V^j$ and $\phi(v) > j$, a new virtual vertex v' with label v and a virtual edge (u, v') are introduced into H_i^j. The set of all virtual vertices of H_i^j with label v is denoted by S_i^v. Note that there may be several virtual vertices with the same label, possibly adjacent to different components of G^j and each with exactly one entering edge. The extension of $T(F_i^j)$ to $T(H_i^j)$ is called the *vertex addition step* and is accomplished by the PQ-tree operation REPLACE. After REDUCE all PQ-leaves with the same label v appear consecutively in every admissible permutation. REPLACE replaces

every such consecutive set with a P-node labelled v. This is the parent of some new leaves representing the adjacent vertices of v in $V^{j+1} \cup V^{j+2} \cup \cdots \cup V^k$. Thereafter all PQ-leaves representing vertices in V^{j+1} with the same label are reduced to appear as a consecutive sequence in any permutation stored by the PQ-trees. Then REPLACE-SINGLE replaces them with a single representative PQ-leaf with the same label by a call of REPLACE. The resulting *reduced extended form* of H_i^j is denoted by R_i^j. If the graph is not a hierarchy, the replacement with a single representative is necessary for the correctness of the algorithm as Leipert [112, p. 71ff] has discovered.

Algorithm 3.3. FIRST-REDUCTION-PHASE

Input: The PQ-trees $\mathcal{T}(G^j)$ and the vertices V^{j+1} of the next level

Output: The PQ-trees $\mathcal{T}(G^j)$

```
foreach component F_i^j in G^j do
    construct H_i^j
    construct T(H_i^j)                    // from the PQ-tree of the previous iteration
end
foreach v ∈ V^{j+1} do
    foreach extended form H_i^j do
        if S_i^v ≠ ∅ then
            T(R_i^j) ← REDUCE(T(H_i^j), S_i^v)
            if T(R_i^j) = ∅ then
                return ∅
            end
            let ṽ be a single representative of S_i^v
            UPDATE(S_i^v, ṽ)              // update PML and QML
            T(R_i^j) ← REPLACE-SINGLE(T(H_i^j), S_i^v, ṽ)
        end
    end
end
return T(G^j)
```

Different PQ-trees may contain PQ-leaves with the same label. Thus a *second reduction phase* is needed to merge these trees, see Algorithm 3.4. A reduced extended form R_i^j is called *v-singular* if all its virtual vertices have the same label, i. e., $\bigcup_{w \in V, \phi(w) > j} S_i^w = \{v\}$. Whenever new inner faces are created by replacing all leaves labelled v with a single representative, a value PML or QML , which stores the lowest level of these faces, is maintained in the PQ-leaf representing v. Using this information it is possible to decide whether or not a v-singular component fits into an inner face above v. Otherwise, it is checked whether it can be placed into the outer face with the same mechanism as for non-singular forms.

Algorithm 3.4. SECOND-REDUCTION-PHASE

Input: The PQ-trees $\mathcal{T}(G^j)$ and the vertices V^{j+1} of the next level

Output: The PQ-trees $\mathcal{T}(G^j)$

foreach $v \in V^{j+1}$ **do**
  // lazy reductions
  foreach PQ-tree $T(R_i^j)$ containing a leaf labelled with v **do**
    if $S_i^v \geq 2$ **then**
      $T(R_i^j) \leftarrow$ REDUCE($T(R_i^j)$, S_i^v)
      if $T(R_i^j) = \emptyset$ **then**
        return $\emptyset$
      end
      let $\tilde{v}$ be a single representative of S_i^v
      UPDATE(S_i^v, $\tilde{v}$) *// update* PML *and* QML
      $T(R_i^j) \leftarrow$ REPLACE-SINGLE($T(R_i^j)$, S_i^v, $\tilde{v}$)
    end
  end
  eliminate all v-singular R_i^j except for the one with the lowest LL-value
  reorder indices such that S_1^v, S_2^v, …, $S_p^v \neq \emptyset$, $S_q^v = \emptyset$ for $q > p$,
    and $\mathrm{LL}(R_1^j) \leq \mathrm{LL}(R_2^j) \leq \ldots \leq \mathrm{LL}(R_p^j)$
  for $i \leftarrow 1$ **to** p **do**
    $T(R_1^j) \leftarrow$ INSERT($T(R_1^j)$, $T(R_i^j)$, v)
    $R_1^j \leftarrow R_1^j \cup_v R_i^j$
    if REDUCE($T(R_1^j)$, S_1^v) $= \emptyset$ **then**
      return $\emptyset$
    end
    let $\tilde{v}$ be a single representative of S_1^v
    UPDATE(S_1^v, $\tilde{v}$) *// update* PML *and* QML
    REPLACE-SINGLE($T(R_1^j)$, S_1^v, $\tilde{v}$)
  end
end
return $\mathcal{T}(G^j)$

Next we briefly describe these pairwise merge operations finally executed by the procedure INSERT. Define the *low indexed level* $\mathrm{LL}(F_i^j)$ of F_i^j to be the least d such that F_i^j contains a vertex in V^d. This value is maintained as an attribute of the corresponding PQ-tree $T(F_i^j)$. The *height* of a component F_i^j is $j - \mathrm{LL}(F_i^j)$. A merge operation is accomplished by using information that is stored at the nodes of the PQ-trees. For any set of virtual vertices $S \subseteq V^{j+1} \cup V^{j+2} \cup \cdots \cup V^k$ of a form H_i^j or R_i^j define the *meet level* $\mathrm{ML}(S)$ of S to be the largest $d \leq j$ such that $V^d \cup V^{d+1} \cup \cdots \cup V^j$ induces a subgraph of G where all $s \in S$ occur in the same connected component. For every P-node X a single value $\mathrm{ML}(X) = \mathrm{ML}(\mathrm{frontier}(X))$ is maintained, where $\mathrm{frontier}(X)$ is the sequence of its descendent leaves from left to right. For every Q-node Y with ordered children $Y_1, Y_2, \ldots, Y_t$ the values $\mathrm{ML}(Y_i, Y_{i+1}) = \mathrm{ML}(\mathrm{frontier}(Y_i) \cup \mathrm{frontier}(Y_{i+1}))$, $1 \leq i < t$, are stored. These indicators tell whether a PQ-tree with a given low indexed level fits into the indentations below a P-node or between two sons of a Q-node. The maintenance of the ML-values during template reductions and insertions in PQ-trees is straightforward. The definition of the meet levels imply the following Observation 3.1.

Observation 3.1. *The meet levels between a node and its siblings are always less or equal to those between its children.*

Let $T_1^v, T_2^v, \ldots, T_f^v$ be all PQ-trees containing a leaf $v \in V^{j+1}$ sorted[1] by descending height. All PQ-trees T_e^v, $2 \leq e \leq f$, are merged sequentially into the highest one, T_1^v. This corresponds to adding the root of the *guest PQ-tree* T_e^v as a child to a PQ-node of the *host PQ-tree* T_1^v. In order to find an appropriate location to insert T_e^v, the method starts with the leaf in T_1^v labelled with v and traverses upwards in T_1^v until a node X' and its parent X are encountered which satisfy one of the following merge conditions. These are checked in the order A to E.

Merge Condition A The node X is a P-node with $\mathrm{ML}(X) < \mathrm{LL}(T_e^v)$. Then attach T_e^v as a child of X in T_1^v.

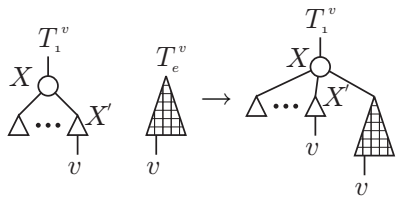


Figure 3.3. Merge condition A

Merge Condition B The node X is a Q-node with ordered children X_1, X_2, …, X_t, $X' = X_1$, and $\mathrm{ML}(X_1, X_2) < \mathrm{LL}(T_e^v)$. Then replace X' in T_1^v with a new Q-node Y having X' and T_e^v as children. The case where $X' = X_t$ and $\mathrm{ML}(X_{t-1}, X_t) < \mathrm{LL}(T_e^v)$ is symmetric.

[1] Sorting must be done in linear time, e. g., with bucket sort.

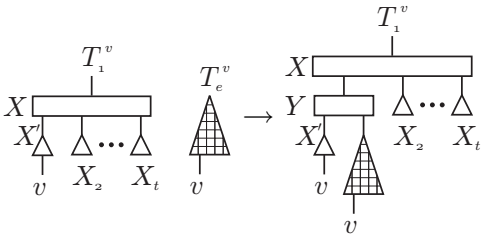


Figure 3.4. Merge condition B

Merge Condition C The node X is a Q-node with ordered children X_1, X_2, …, X_t, $X' = X_i$, $1 < i < t$, and $\mathrm{ML}(X_{i-1}, X_i) < \mathrm{LL}(T_e^v)$ and $\mathrm{ML}(X_i, X_{i+1}) < \mathrm{LL}(T_e^v)$. Then replace X' with a new Q-node Y having X' and T_e^v as children.

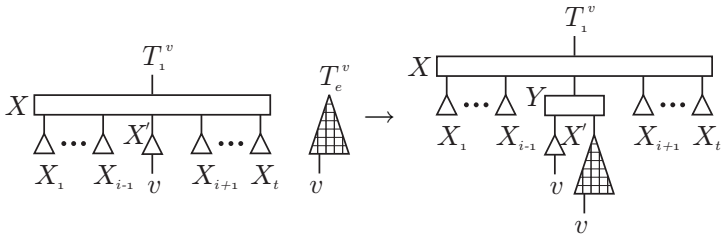


Figure 3.5. Merge condition C

Merge Condition D The node X is a Q-node with ordered children X_1, X_2, …, X_t, $X' = X_i$, $1 < i < t$, and

$$\mathrm{ML}(X_{i-1}, X_i) < \mathrm{LL}(T_e^v) \leq \mathrm{ML}(X_i, X_{i+1}).$$

Then attach T_e^v as a child of X between X_{i-1} and X_i. If

$$\mathrm{ML}(X_i, X_{i+1}) < \mathrm{LL}(T_e^v) \leq \mathrm{ML}(X_{i-1}, X_i)$$

then attach T_e^v as a child of X between X_i and X_{i+1}.

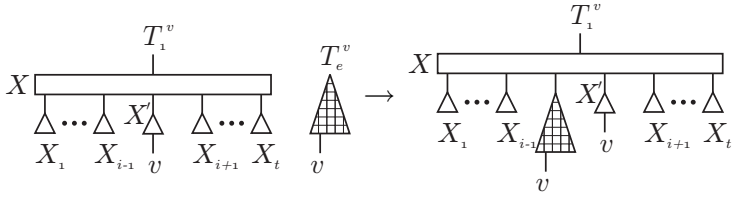


Figure 3.6. Merge condition D

Merge Condition E The node X' is the root of T_1^v. Then reconstruct T_1^v by inserting a new Q-node Y as the new root with X' and T_e^v as children.

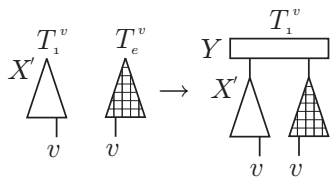


Figure 3.7. Merge condition E

After each merge operation REDUCE and REPLACE-SINGLE are called again to make all v-leaves consecutive and then to replace them with a single representative PQ-leaf. Afterwards, T_e^v is deleted from $\mathcal{T}(G^{j+1})$. In order to achieve linear running time there is no scan for other leaves with the same label after v-merging several reduced extended forms. However, this strategy results in improper reduced extended forms possibly with several virtual vertices with the same label. These are called *partially reduced extended forms* and are reduced on demand.

Finally, in a new sweep over the current level Algorithm 3.5 deletes all PQ-leaves representing sinks v in V^{j+1} from their corresponding PQ-tree and reconstructs the tree such that it obeys the properties of a valid PQ-tree again. Further, it is necessary to update the pointers of the leaves to the potentially new PQ-tree to which they belong. Afterwards, for every detected source vertex on level $j+1$ a new PQ-tree is created before the sweep over the next level $j+1$.

Algorithm 3.5. FINAL-UPDATES

Input: The PQ-trees $\mathcal{T}(G^j)$ and the vertices V^{j+1} of the next level

Output: The PQ-trees $\mathcal{T}(G^j)$

delete leaves representing sink vertices of V^{j+1} from the PQ-trees
update the pointers of the leaves to their PQ-tree
add for every source in V^{j+1} a new PQ-tree to $\mathcal{T}(G^j)$

return $\mathcal{T}(G^j)$

Remember, LEVEL-PLANARITY-TEST also works on non-proper level graphs within $\mathcal{O}(|V|)$ time and without inserting up to $\mathcal{O}(|V|^2)$ dummy vertices for long edges by adding all children on higher levels and not only those on the next level.

3.4 Level Planar Embedding

Level planar embeddings are characterised by a family of linear orderings $(\leq_j)_{1\leq j\leq k}$ of the vertices on each level V^j, which in our case is the ordering from left to right. For a witness after the positive level planarity test and for a level planar drawing the algorithm computes a level embedding in two passes. This is outlined by Algorithm 3.6. First G is augmented to a planar st-graph. An st-numbering for G can be computed by topologically sorting the vertices using implicit edge directions from lower to higher levels. This corresponds to numbering the vertices level by

level in ascending order. Then a planar st-embedding is obtained by the algorithm of Chiba et al. [26], from which a level planar embedding is directly computed by an ordered DFS.

Algorithm 3.6. LEVEL-PLANAR-EMBED

Input: A level graph $G = (V^1 \dot\cup V^2 \dot\cup \ldots \dot\cup V^k, E, \phi)$
Output: A level embedding $\mathcal{E}_l$ of G if it is level planar, $\emptyset$ otherwise

expand G to G_{st} by adding $V^0 \leftarrow \{s\}$ and $V^{k+1} \leftarrow \{t\}$

AUGMENT(G_{st})
if AUGMENT fails **then**
  return $\mathcal{E}_l \leftarrow \emptyset$
end

// G_{st} is now a hierarchy

reverse the level numbering G_{st} from bottom to top
AUGMENT(G_{st}) *// cannot fail*
reverse level numbering G_{st} from the top to the bottom
$E_{st} \leftarrow E_{st} \cup (s, t)$ *// add st-edge*

// G_{st} is now an st-graph

TOPSORT(V_{st})
compute a planar embedding $\mathcal{E}_{st}$ according to Chiba et al. [26]
  using the topological sorting as an st-ordering

$\mathcal{E}_l \leftarrow$ CONSTRUCT-LEVEL-EMBED($\mathcal{E}_{st}$, G_{st})

return $\mathcal{E}_l$

Augmenting a level graph G to an st-graph G_{st} is divided into two phases. After adding a new source s and a new sink t, in the first phase an outgoing edge is added to every old sink of G by the application of a modified LEVEL-PLANARITY-TEST algorithm from level 1 to k. Using the same algorithmic concept bottom up from level k to 1, an incoming edge is added to every old source of G in the second phase. To add these edges without violating level planarity, every PQ-leaf representing a sink in G is replaced with a *sink indicator* as a leaf in its corresponding PQ-tree. This indicator is ignored throughout the application of the algorithm. If all siblings of a node are ignored, its parent is ignored, too. Thus entire PQ-trees can be ignored. Sink indicators are removed either together with the leaves representing the incoming edges of some vertex $w \in V^l$, $l > j$, where j is the current level, or they are still left in the final PQ-trees. In the first case vertices which are represented by sink indicators are connected to w after its reduction by the subsequent REPLACE on w. In the second case they are connected to t at the end of the augmentation phase. Sink indicators in PQ-trees representing a v-singular form are connected to v if they are inserted into an inner face above v.

Algorithm 3.6 computes an st-embedding $\mathcal{E}_{st}$ according to the technique of Chiba et al. [26] using a topological sorting of the augmented graph as the st-numbering. The algorithm CONSTRUCT-LEVEL-EMBED computes a level planar embedding $\mathcal{E}_l$ of G from the planar embedding $\mathcal{E}_{st}$. For this it traverses the graph with DFS from t and proceeds from every visited vertex v to the unvisited neighbour w on a smaller level that appears first in the clockwise ordering of v's adjacency list in $\mathcal{E}_{st}$. Initially, all levels in $\mathcal{E}_l$ are empty. If a vertex $w \notin \{s, t\}$ is visited, it is appended at the end of the ordered list of the vertices assigned to $\phi(w)$, i. e., at the end of $\mathcal{E}_l[\phi(w)]$. Since the DFS starts at t and uses only edges to vertices with a smaller st-number, the DFS in Chiba's method ENTIRE-EMBED in Algorithm 2.2 extending the obtained directed upward embedding $\mathcal{E}_u$ into a complete and undirected st-embedding $\mathcal{E}_{st}$ can be omitted.

In order to achieve linear running time a search for sink indicators which can be considered for augmentation must be avoided. But sink indicators must be treated correctly by merge operations. Therefore a new ignored node type called *contact* is introduced in the PQ-trees during the merge operations B and C. The contacts store which sinks have to be augmented if the newly introduced Q-node is inserted into its parent Q-node by an application of a template later in the algorithm. For details see [95, 97, 112].

3.5 Straight-Line Drawings

It is clear that every proper level planar graph has a straight-line drawing. Moreover, Eades, Feng, and Lin have shown in [51, 53] that every level planar graph, even with long edges, has a straight line drawing. Further, they have presented an $\mathcal{O}(|V|^2)$ time algorithm for computing such a drawing from a level planar embedding. However, the drawings produced by this algorithm require up to exponential area.

4 Track Planarity

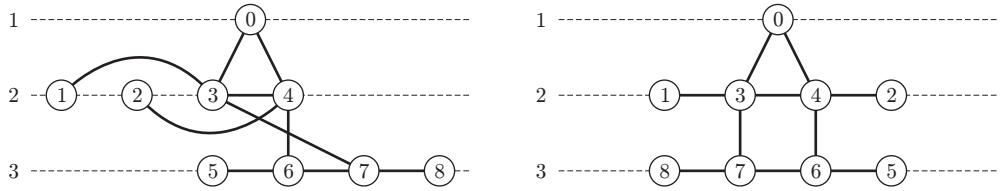
Next we generalise level planar graphs to obtain an even larger class of hierarchical graphs for which hierarchically planar drawings can be generated efficiently. We introduce track graphs, where edges connecting consecutive vertices on the same hierarchical level are allowed.

This generalisation is similar to the extension of strictly upward drawings of binary trees to upward drawings. In strictly upward drawings the edges are strictly monotone, whereas in upward drawings horizontal edges are also allowed. It is well known [38] that $\Theta(|V| \log |V|)$ is the upper and lower bound for the area of strictly upward drawings of binary trees while only $\mathcal{O}(|V|)$ area is needed for upward drawings [72].

4.1 Definition of Track Planarity

A *k-track graph* $G = (V, E \cup E', \phi)$ is a k-level graph with additional edges $E' \subseteq \{ (u, v) \mid u, v \in V, \phi(u) = \phi(v) \}$ on the same level (*track*[1]). It is *k-track planar* if there are linear orderings $\leq_i$, $1 \leq i \leq k$, of the vertices on each level such that edges are drawn as weak monotone curves without edge or vertex intersections. Thus in a track planar embedding $\mathcal{E}_l = (\leq_i)_{1 \leq i \leq k}$ all edges $(u, v) \in E'$ connect consecutive vertices on the same level $i = \phi(u)$, i. e., $u \leq_i v$ implies $\forall t \in V^i - \{u, v\}\colon \neg(u \leq_i t \leq_i v)$. Figure 4.1 shows a 3-track planar graph and a track planar drawing. Note that an additional edge $(6, 8)$ is not allowed because it violates edge monotony or planarity by overlapping edges.

[1] The term "track" stems from the two layer planarisation approach of [61, 62]. However, these papers consider straight line drawings, which is not guaranteed here, especially under the presence of long edges.



(a) A 3-track planar graph

(b) A 3-track planar drawing of Figure 4.1(a)

Figure 4.1. A track planar graph and a track planar drawing of it

Di Giacomo [43, Chapter 6–8] describes families of graphs which admit a 3-dimensional straight-line track planar drawing, see Figure 4.2 for an example. Contrary to our track graphs, the vertices are not preassigned to the tracks. Further, there no three distinct tracks are allowed to be co-planar, i. e., no three tracks are allowed to be drawn on a plane. All tracks are parallels of the x-axis. Particularly, Di Giacomo examines upper and lower bounds of the *track number* which is the minimum number of the tracks needed for a 3-dimensional straight-line track planar drawing.

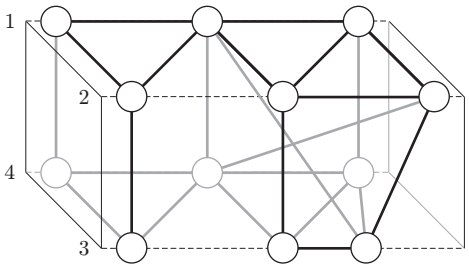


Figure 4.2. The $3d$ track planar graph of [43, p. 100]

However, we consider 2d-drawings of track graphs with a given partition of the vertices into tracks. All tracks are parallels on the xy-plane.

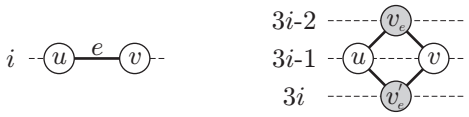
Remark 4.1. *For a k-track graph G:*

$$G \text{ is } k\text{-level planar} \Rightarrow G \text{ is } k\text{-track planar} \Rightarrow G \text{ is planar.}$$

4.2 Reduction to Level Planarity

Our basic idea for solving the track planarity problem is a linear time reduction to level planarity. We transform the track graph $G = (V, E, \phi)$ into a level graph $G' = (V', E', \phi')$ such that G' is level planar if and only if G is track planar. After initialising V' with V and E' with E we triple the number of levels by defining $\phi'(v) = 3\phi(v) - 1$ for all $v \in V'$. Then every *horizontal edge* $e = (u, v) \in E'$

with $\phi'(u) = \phi'(v)$ is replaced by a *diamond* subgraph, see Figure 4.3. Two new vertices v_e and v'_e on the two adjacent levels are introduced and are connected to both end vertices of e. Afterwards, e is removed from E. We obtain $|V'| \in \mathcal{O}(|V|)$ and $|E'| \in \mathcal{O}(|E|)$. The transformation preserves planarity and the embedding of the graph.

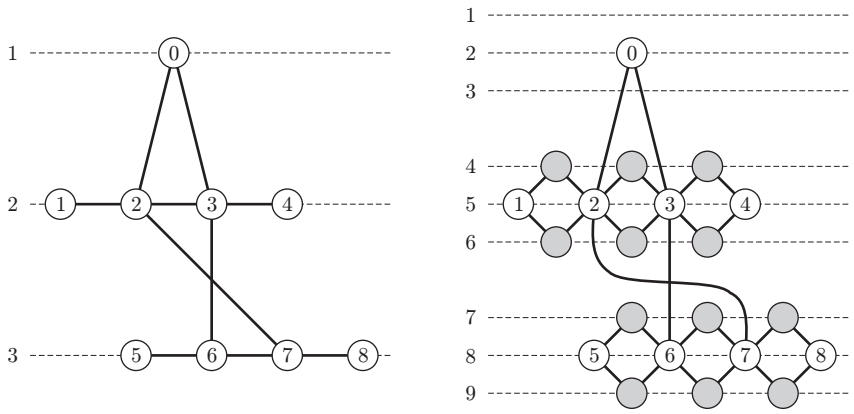


(a) A horizontal edge

(b) A diamond for Figure 4.3(a)

Figure 4.3. Transformation of the horizontal edges into diamonds

Example 4.1. *Figure 4.4 shows the transformation of a* 3*-track graph into an equivalent* 9*-level graph which contains two* diamond chains*. Note that the new vertices are on levels which do not contain original vertices and vice versa.*



(a) A 3-track graph

(b) The 9-level transformation of Figure 4.4(a)

Figure 4.4. Example of a track graph transformed into a level graph

Theorem 4.1. *Let G be a k-track graph and G' its transformation. Then*

$$G \text{ is } k\text{-track planar} \Leftrightarrow G' \text{ is } 3k\text{-level planar.}$$

Proof. Since isolated vertices affect neither track nor level planarity, we consider only graphs not containing isolated vertices.

For the only if direction let $G = (V, E, \phi)$ be k-track planar. Thus there exists a k-track planar embedding $\mathcal{E}_l$ of G. We construct an embedding $\mathcal{E}'_l$ of $G' = (V', E', \phi')$ from $\mathcal{E}_l$ by tripling the number of levels and by defining $\leq'_{3i-1} = \leq_i$, $1 \leq i \leq k$. The remaining relations $\leq'_{3i-2}$ and $\leq'_{3i}$ are given by ordering the dummy vertices on levels $3i-2$ and $3i$ according to their adjacent non-dummy vertices. The two new vertices of diamonds are always placed with respect to horizontal ordering between the end vertices of the corresponding horizontal edge e without violating planarity. Suppose that $\mathcal{E}'_l$ is not level planar. Then at least two edges e_1 and e_2 cross in $\mathcal{E}'_l$. If they are both non-diamond edges, they also cross in $\mathcal{E}_l$, a contradiction to the track planarity of $\mathcal{E}_l$. If exactly one of them is a diamond edge, suppose e_2, then e_1 crosses the horizontal edge in $\mathcal{E}_l$ due to which e_2 was introduced. Again a contradiction to the track planarity of $\mathcal{E}_l$. If both edges are diamond edges, we obtain a contradiction again because their corresponding horizontal edges overlap in $\mathcal{E}_l$ which is not allowed for track planar embeddings. Since $\mathcal{E}'_l$ is level planar, G' is level planar, too.

For the if direction let G' be $3k$-level planar. Thus G' has a $3k$-level planar embedding $\mathcal{E}'_l$. In $\mathcal{E}'_l$ the inner face of every diamond is empty. Apart from isolated vertices, which don't exist at this step, no vertex can be inside without violating level planarity, see Figure 4.5.

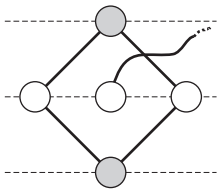


Figure 4.5. No vertex can be inside a diamond

Thus it is possible to draw an edge in G' between the two original vertices of every diamond without violating planarity. Every level i with $i \not\equiv 2 \pmod 3$ and all vertices on that level i together with their adjacent edges are deleted. Deletions never violate planarity. After renumbering the levels from 1 to k we obtain a k-track planar embedding $\mathcal{E}_l$ of G. □

We summarise the results of this chapter: Due to Theorem 4.1, the given transformation can be used to reduce track planarity to level planarity in linear time.

Theorem 4.2. *There is an $\mathcal{O}(|V|)$ time reduction of track planarity to level planarity.*

4.3 Algorithm

From Theorem 4.2 we obtain a straightforward track planarity testing algorithm. We use the method TRANSFORM for transforming G into G' and apply any level planarity test afterwards. Because TRANSFORM runs in linear time, the overall

algorithm has the same complexity as the embedded level planarity test, $\mathcal{O}(|V|)$ time for the JLM algorithm and up to $\mathcal{O}(|V|^4)$ time for Healy and Kuusik's algorithm for non-proper graphs. Note that our construction has to be made proper for usage with the algorithm of Healy and Kuusik.

Since the transformation of horizontal edges to diamonds and the shrinking of diamonds to edges preserves the embedding, we can use Algorithm 4.1 to obtain a track planar embedding $\mathcal{E}_l$ of a track planar graph G.

Algorithm 4.1. TRACK-PLANAR-EMBED

Input: A track graph $G = (V, E, \phi)$

Output: A track embedding $\mathcal{E}_l$ if G is track planar, $\emptyset$ otherwise

remove all isolated vertices from G
$G' \leftarrow$ TRANSFORM(G)
$\mathcal{E}_l \leftarrow$ LEVEL-PLANAR-EMBED(G')
if $\mathcal{E}_l = \emptyset$ **then**
  return $\emptyset$
end
remove all levels i from $\mathcal{E}_l$ for which $i \not\equiv 2 \pmod 3$
renumber the levels from 1 to k
insert each removed isolated vertex v of G at the end of $\mathcal{E}_l[\phi(v)]$
  in arbitrary order
return $\mathcal{E}_l$

Again, the running time of the level embedding algorithm, $\mathcal{O}(|V|)$ time for JLM's up to $\mathcal{O}(|V|^6)$ time for Healy and Kuusik's, dominates the overall complexity. In order to prevent occurrences of isolated vertices within inner faces of diamonds we handle these in a special way. They are removed at the beginning and reinserted in arbitrary order afterwards. Placing them at the end of their respective ordered level in $\mathcal{E}_l$ does not violate planarity in any case.

Corollary 4.1. *The algorithm TRACK-PLANAR-EMBED returns a valid track planar embedding if and only if the input track graph G is track planar.*

Proof. Assuming that the level embedding algorithm applied in Algorithm 4.1 returns a valid level embedding $\mathcal{E}'_l$, we obtain a valid vertex ordering of each level i with $2 \equiv i \pmod 3$. These are exactly the vertex orderings of each level in a valid track planar embedding $\mathcal{E}_l$ of G according to the proof of Theorem 4.1. □

5 Radial Level Planarity

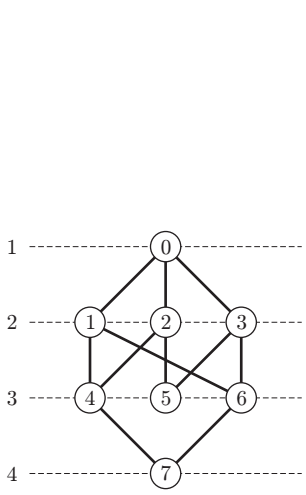
Our innovation in this chapter is a generalisation of level planarity to radial level planarity. In contrast to Chapter 3, the vertices are not drawn on k horizontal lines but on k concentric circles $l_j = \{ (j \cos\theta, j \sin\theta) \mid \theta \in [0, 2\pi) \}$, $1 \leq j \leq k$. Such drawings extend the radial tree drawings of Eades [49, 50], where the levels of the vertices are given by their depth. Another motivation for radial level planar graphs are radial drawings of social networks in [18, 19], where the vertices are constrained to lie on radial levels according to their centrality. This maps structural centrality to a geometric one and thus a traditional sociogram [121] is obtained. There are also the ring diagrams introduced by [129] which, e. g., eliminate some disadvantages of the Hasse diagrams[1].

5.1 Definition of Radial Level Planarity

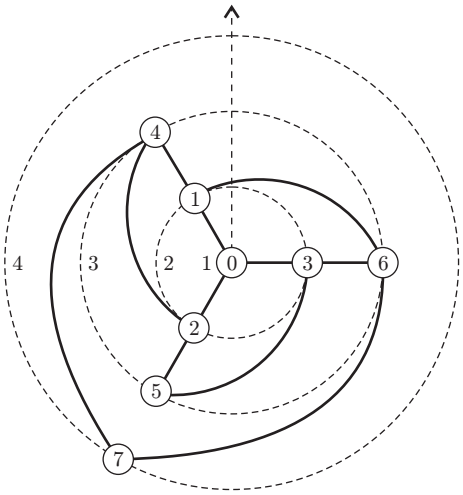
A k-level graph is *radial k-level planar* if there are orderings $(\leq_j)_{1 \leq j \leq k}$ of the vertices on each radial level such that the edges can be drawn as strictly monotone curves from inner to outer levels without crossings. This ensures that edges between two levels do not cross inner level lines. Thus the drawing is a so-called *outward drawing*. Every level planar embedding can be transformed into a radial level planar embedding by connecting the ends of each level to concentric circles. This introduces new possibilities to add edges as monotone curves from the end of one level to the beginning of another. These *cut edges* cross a *ray* from the *centre* of the concentric levels to infinity through the connection points of the level lines exactly once. Hence, there are two directions for routing cut edges around the centre. As an extension to level planar embeddings, *radial level planar embeddings* need additional information

[1] A Hasse diagram is an an upward drawing of a DAG without reflex and transitive edges.

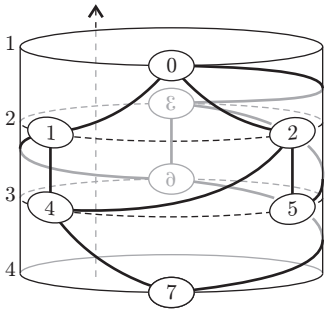
about cut edges and their direction. Figure 5.1(b) shows a radial level planar drawing of the graph in Figure 5.1(a) which is not level planar. The edge $(1, 6)$ crosses the ray and thus is a clockwise cut edge, following its implicit direction from lower to higher levels. Another simple example is a levelled $K_{2,2}$ which is proper radial 2-level planar and not proper 2-level planar.



(a) A radial 3-level planar graph



(b) A radial level planar drawing of Figure 5.1(a)



(c) A 3d-drawing of Figure 5.1(a) on a cylinder

Figure 5.1. A radial level planar graph with radial level planar drawings

Figure 5.1(c) is a 3-dimensional planar drawing of Figure 5.1(a). To obtain such a drawing, which is another nice representation of radial level planar graphs, one can imagine putting the drawing of Figure 5.1(c) on a cylinder. This is different to the representation of graphs on a cylinder in [150], where visibility representations are considered. The opposite direction is a projection onto the plane where the radii of the level lines grow by a constant factor. Note that our radial drawings are different to the *recurrent hierarchies* of Sugiyama et al. [148], where the levels are drawn as

consecutive rays emanating from the centre, see Figure 5.2 for an example. There (directed) edges are allowed from the bottom levels to the top levels. In analogy to Figure 5.1(c), such graphs can be be drawn in three dimensions with horizontal level lines on a horizontal cylinder where the "cut" between the last and the first level is a horizontal line, too. However, in the following only 2d-drawings and concentric levels are considered.

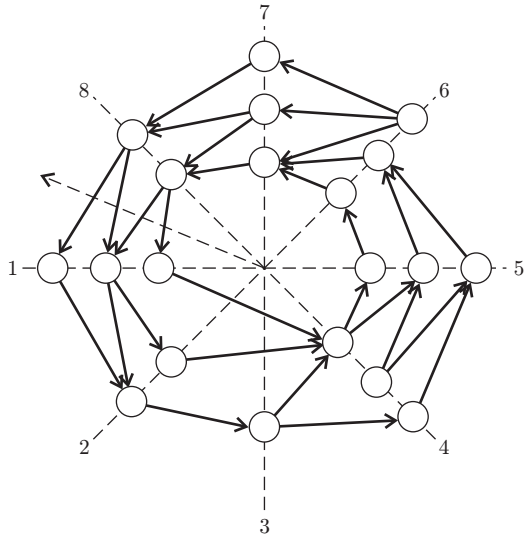
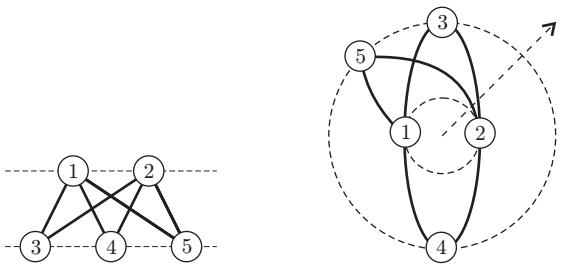


Figure 5.2. The recurrent hierarchy from the cover page of Kaufmann and Wagner [105]

Figure 5.3 shows a minimum 2-level graph which is not radial level planar. It is a levelled variant of $K_{2,3}$ with the two vertices fixed to one level and the three vertices fixed to another level. Note that there are also radial level non-planar graphs which do not contain a cycle, cf. Section 7.3.1.



(a) A levelled $K_{2,3}$

(b) A radial drawing of Figure 5.3(a)

Figure 5.3. A minimum radial level non-planar graph

Remark 5.1. *For a k-level graph G:*

$$G \text{ is level planar} \Rightarrow G \text{ is radial level planar} \Rightarrow G \text{ is planar.}$$

5.2 Related Work

It is well known that every planar graph has a concentric representation based on a BFS traversal [155]. There the vertices are placed on concentric circles and the edges are routed as not necessarily monotone curves without crossings. Here we take the opposite view (for monotone edge routing) and consider the problem whether a graph with a given partition of its vertices on k concentric levels is radial k-level planar. Another type of radial drawings are the circular visibility layouts of [91], which also do not accommodate a predefined vertex partition.

In addition to level planarity, the concept of fixed parameter tractability can also be applied to radial level planarity. Dujmović et al. [48] have shown that the radial k-level planarity problem is fixed parameter tractable. However, k must be bounded by a constant. As a consequence an $\mathcal{O}(|V|)$ running time is obtained for a fixed number of levels, but the $\mathcal{O}$-notation hides large constants, i. e., 2^{32k^3} is an upper bound even if the levelling is not given. We pursue a direct approach and improve the result of Dujmović et al. to linear time for an arbitrary number of levels by giving a practical algorithm based on the JLM algorithm described in Section 3.3.

5.3 Radial Level Planarity Testing

Our extensions made in this section are the new data structure PQR-trees with advanced merging and the detection of nested rings. But before we explore that, we establish some fundamental properties of radial level planar graphs.

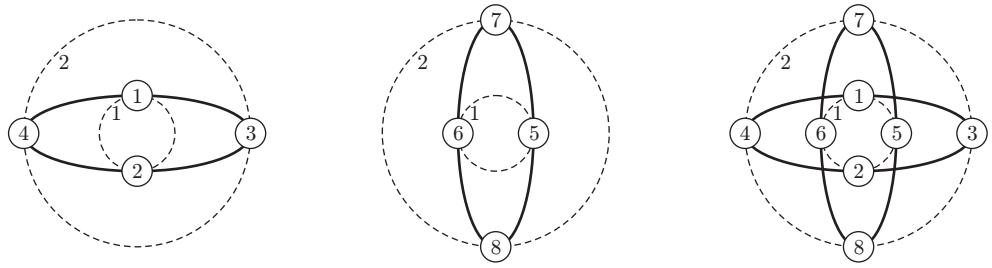
5.3.1 Fundamental Properties

First we elaborate distinctions between level and radial level planar graphs. Our first result is obvious.

Lemma 5.1. *A radial level planar graph is level planar if there are no cut edges.*

Proof. The correctness follows directly from the transformation of a level planar graph into a radial planar graph in Section 5.1. □

Next consider connectivity. The JLM algorithm relies on the fact that a level graph is level planar if and only if each connected component is level planar. Therefore it tests each connected component separately, which is no restriction since separate components can be placed next to each other. This is no longer true for radial level planarity as Figure 5.4(c) illustrates. There two disjoint ovals interleave.



(a) A radial level planar component

(b) A radial level planar component

(c) Figures 5.4(a) and 5.4(b) combined

Figure 5.4. Two radial planar connected components and a radial level non-planar graph which is a combination of them

Obviously a graph is radial level planar if it consists only of level planar components. Hence, we must consider those components of a level graph that are radial level planar and level non-planar.

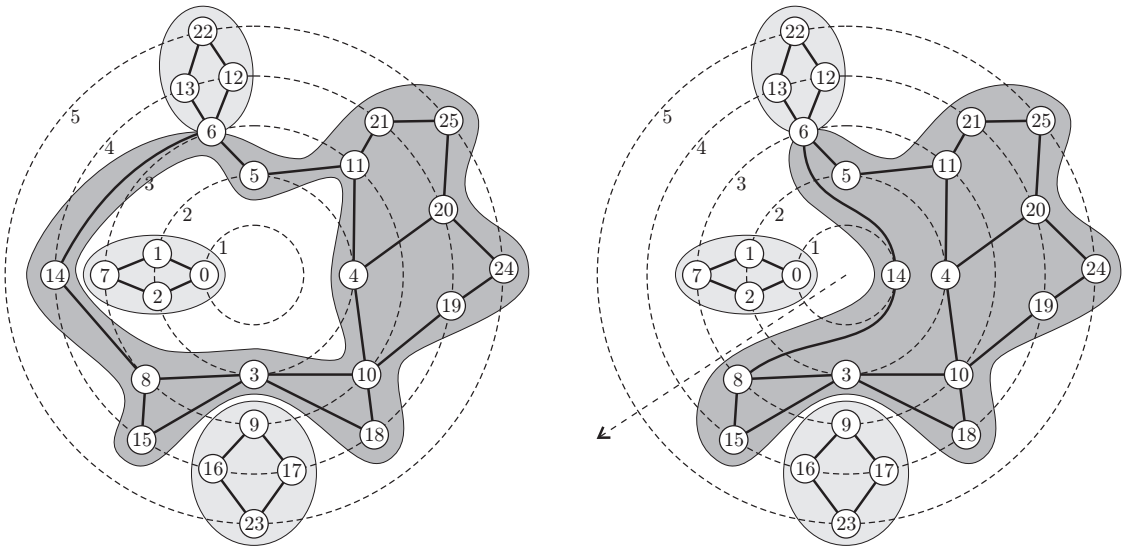
Definition 5.1 (Ring). *A* ring *is a biconnected component of a level graph which is radial level planar and not level planar. A level graph containing a ring is called a* ring graph.

It is not immediate whether a biconnected component is a ring. We will see later how rings are detected. Nevertheless, we investigate some properties of rings first. The graph in Figure 5.5(a) consists of four biconnected components with a darker shaded ring. Observe that a component can and sometimes must be nested in another one. This occurs if a component contains a ring. Clearly, a ring must contain a cycle, but a cycle does not necessarily induce a ring. In fact every biconnected component with at least three vertices contains a cycle, but whether it is a ring depends on the levelling. If vertex 14 in Figure 5.5(a) was on level 1, this graph would not contain a ring because according to the ray in Figure 5.5(b) there are no cut edges and every component is level planar.

Lemma 5.2. *If a level graph G does not contain a ring, the following are equivalent:*

1. *G is radial level planar.*
2. *G is level planar.*
3. *Each connected component of G is level planar.*
4. *Each connected component of G is radial level planar.*

Hence, if a graph does not contain a ring, we can use JLM's level planarity testing algorithm to test for radial level planarity. For ring graphs the algorithm needs an extension.



(a) A ring graph

(b) Not a ring graph

Figure 5.5. Rings depend on the levelling

5.3.2 Properties of Rings

Before we describe how our algorithm stores the admissible permutations of the vertices on each radial level, we discuss some more properties of rings.

Lemma 5.3. *In every radial level planar embedding of a ring graph the centre of the concentric levels lies in an inner face. This face is called the* centre face.

Proof. Suppose there exists a ring graph G that has a radial level planar embedding with the centre lying in the outer face. Then there is a ray from the centre to infinity which crosses no edges. Hence, there are no cut edges and every biconnected component of G is level planar. Thus G does not contain a ring contradicting the assumption. □

Lemma 5.4. *A ring contains at least four vertices and four edges, and there is a ring of that size.*

Proof. A ring is not level planar. Thus every level embedding contains at least two crossing edges (u, v) and (w, x) with mutually different vertices u, v, w, and x. To ensure biconnectivity at least four edges are needed. The $K_{2,2}$ on two levels is a *minimum ring*, see Figure 5.4(a). □

Another important property of rings is the nesting, which is determined by some characterising parameters:

Definition 5.2 (Ring extremes). *For a k-level graph G containing a ring R let α_R and δ_R be the minimum and maximum levels of R containing a vertex of R,*

respectively. Let the inner radius β_R *of* R *be the maximum level with a vertex of the centre face of* R *in any radial level planar embedding of* G*, and let the* outer radius γ_R *of* R *be the minimum level with a vertex of the outer face of* R *in any radial level planar embedding of* G*.*

These parameters are illustrated by Figure 5.6. The minimum level α_R and the maximum level δ_R of a ring are independent of the embedding because they are given by the levelling of the graph. This is not necessarily true for the inner radius β_R and the outer radius γ_R. This can be seen in Figure 5.6 by moving vertex 5 to the right and by moving vertex 10 to the left.

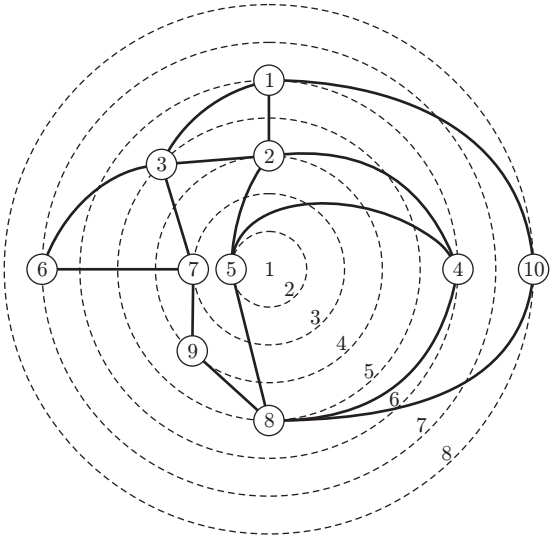


Figure 5.6. Extreme levels of a ring. $\alpha_R = 2$, $\beta_R = 6$, $\gamma_R = 3$, $\delta_R = 8$

Definition 5.3 (Level optimal). *A radial level planar embedding of a ring* R *is* level optimal *if* β_R *and* γ_R *are the extremes of the centre face and of the outer face in any radial level planar embedding of* R*.*

Our algorithm always constructs level optimal embeddings, as will be shown later by Lemma 5.11.

Lemma 5.5. *Every level planar graph has an embedding that is level optimal for each contained ring.*

Lemma 5.6. *Every ring* R *spans at least two levels and its characterising parameters relate by* $\delta_R > \gamma_R \geq \alpha_R$ *and* $\delta_R \geq \beta_R > \alpha_R$*.*

Proof. The end vertices of an edge always lie on different levels since inner level edges are not allowed. By Lemma 5.4 a ring always contains edges and thus has vertices on at least two levels. Therefore the four relations follow directly from the definitions. □

Lemma 5.7. *Let G be a level graph consisting of two disjoint rings R and S. G is radial level planar if and only if there are radial level planar embeddings of R and S satisfying*

$$(\alpha_S > \gamma_R \text{ and } \beta_S > \delta_R) \quad \text{or} \quad (\alpha_R > \gamma_S \text{ and } \beta_R > \delta_S),$$

i. e., R can be embedded in the centre face of S or vice versa.

Proof. For the only if direction let G be a radial level planar graph consisting of two disjoint rings R and S. Since subgraphs of radial level planar graphs are radial level planar, R and S are radial level planar. Each ring is biconnected and encloses the centre according to Lemma 5.3. Thus in any radial level planar embedding one ring is completely contained within the centre face of the other. Let R be contained in S. If $\alpha_S \leq \gamma_R$ or $\beta_S \leq \delta_R$ then the contour C_S of the centre face of S intersects the contour C_R of the outer face of R. C_R and C_S are cycles, each with one inner face containing the centre. For their computation we assume w. l. o. g. that G and thus both rings are proper. The ordering of incident edges around a vertex is determined by the ordering of the adjacent edges in the radial level planar embedding of G together with its information about cut edges and their direction. This allows the traversal of the contour of each face and thus the computation of C_R and C_S. Assume $\alpha_S \leq \gamma_R$. The case $\beta_S \leq \delta_R$ is symmetric. Then the vertex v of C_S which defines α_S lies in the inner face of C_R because γ_R defines the least level of a vertex u of C_R. S has a vertex w in the outer face of C_R. Otherwise S would be contained in R. Since S is connected, there exists a path $P = v \rightarrow^* w$. P crosses C_R, which is a contradiction to the radial level planarity of G. This is also true in the extreme case $\alpha_S = \gamma_R$. Vertex v cannot be placed in the outer face of C_S since both incident edges of u in C_R connect u with vertices on a higher level. For an illustration see Figure 5.7.

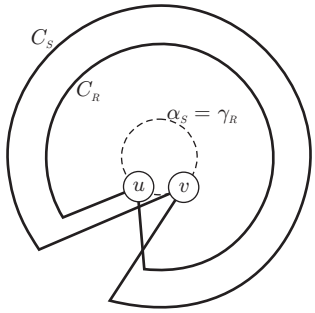


Figure 5.7. Two intersecting contour cycles C_R and C_S with $\alpha_S = \gamma_R$

For the if direction let G be a level graph consisting of two disjoint radial level planar rings R and S with $\alpha_S > \gamma_R$ and $\beta_S > \delta_R$. The case $\alpha_R > \gamma_S$ and $\beta_R > \delta_S$ is symmetric. We show the radial level planarity of G by constructing an embedding $\mathcal{E}_l^{R \cup S}$ of G which combines the radial level planar embeddings $\mathcal{E}_l^R$ and $\mathcal{E}_l^S$ of R and S. These embeddings have only the levels between α_S and δ_R in common, whereas the others remain unchanged. Thus we copy $\mathcal{E}_l^S$ completely and $\mathcal{E}_l^R$ from level 1 to

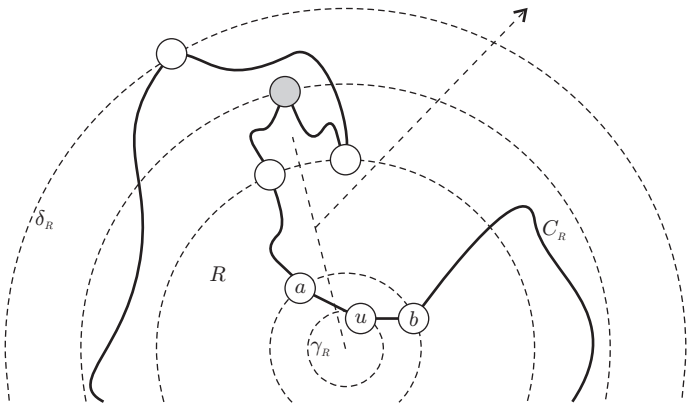
$\alpha_S - 1$ to $\mathcal{E}_l^{R \cup S}$ maintaining the original vertex orderings. Since $\alpha_S > \gamma_R$ all vertices of S are on higher levels than γ_R and thus can be placed beyond the contour of the outer face of R. Since $\beta_S > \delta_R$ all vertices of R fit inside the contour of the centre face of S. Observe that it may be necessary to rotate and squeeze R, so that all vertices fit inside the largest cavity of S and vice versa.

It remains to show where to insert the vertices of R from level α_S to δ_R between the vertices of S in $\mathcal{E}_l^{R \cup S}$. We assume that R and S are proper and that in $\mathcal{E}_l^R$, $\mathcal{E}_l^S$, and $\mathcal{E}_l^{R \cup S}$ the position of the ray among the vertices of each ordered level is implicitly after the last vertex. Let C_R be the contour of the outer face of R and let C_S be the contour of the centre face of S. They are computed as described in the only if part of this proof. Thus we know for each edge whether it is an edge on one of the two contours or not. First we rotate R: Let u be the vertex of R which defines γ_R. It has exactly two incident edges (u, a) and (u, b) in C_R. The levels of a and b relate as $\phi(a) = \phi(b) = \gamma_R + 1$. We rotate the level $\gamma_R + 1$, i. e., move vertices of $\mathcal{E}_l^R[\gamma_R + 1]$ from the front to the back or vice versa, such that exactly one of the edges (u, a) and (u, b) is a cut edge. After this intersection, the ray leaves R to R's outer face. From this point outwards no further intersection of the ray with an edge is necessary. We rotate the levels $\gamma_R + 2$ to δ_R in ascending order such that there are no edges which cross the ray. Thereby it may be possible to run into a local maximum since C_R is not necessarily monotone and thus a rotation of one of the next levels without producing a cut edge is impossible, see Figure 5.8(a). To avoid this we use a backtracking strategy. Now, analogously to R, we rotate S: Let v be the vertex of S which defines β_S. It has exactly two incident edges (v, c) and (v, d) in C_S. The levels of c and d relate as $\phi(c) = \phi(d) = \beta_S - 1$. We rotate the level $\beta_S - 1$ such that exactly one of the edges (v, c) and (v, d) is a cut edge. After this intersection, the ray enters the centre face of S. From this point inwards the ray needs not to cross any more edges of S. We rotate the levels $\beta_S - 2$ to α_S in descending order such that there are no edges which cross the ray. Again, we need a backtracking strategy, here to get out of local minima, see Figure 5.8(b). The described operations are done before copying $\mathcal{E}_l^S$ and $\mathcal{E}_l^R$ from level α_R to α_S to $\mathcal{E}_l^{R \cup S}$. At the end, the ordered sets of the remaining levels of $\mathcal{E}_l^R$ are inserted at the back of the corresponding levels of $\mathcal{E}_l^{R \cup S}$. □

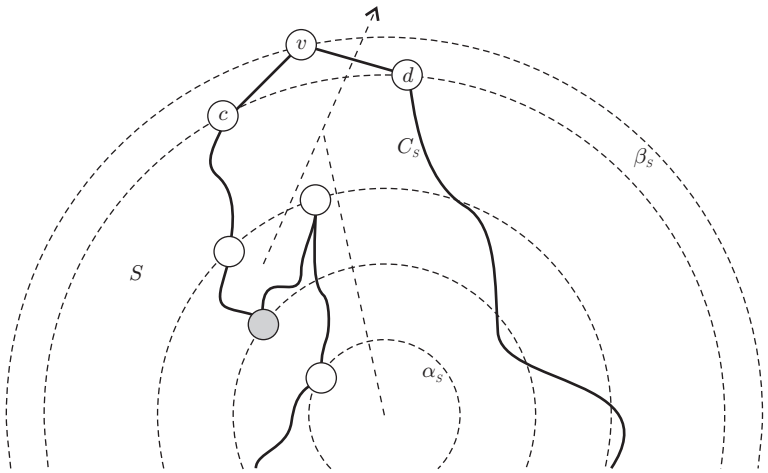
The nesting of disjoint rings is illustrated in Figure 5.9. This is an essential difference to disjoint components of level planar graphs, which are usually placed side by side and which can be treated separately.

5.3.3 R-Nodes

We now come to the main results of this chapter and extend the JLM algorithm to test for radial level planarity, see Algorithm 3.1. The input graph is traversed in a top down sweep, which now becomes a wavefront sweep outwards from the centre. The processed part of the graph is represented by a collection of trees which is denoted by $\mathcal{T}$. We need a new data structure PQR-trees to deal with rings. PQR-trees store

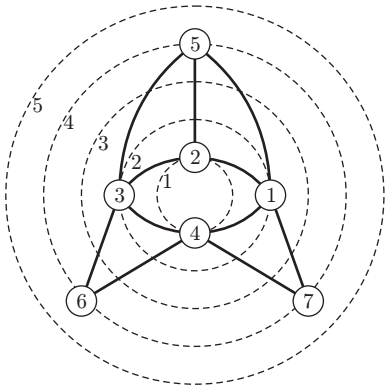


(a) A local maximum while traversing a path from u in C_R outwards

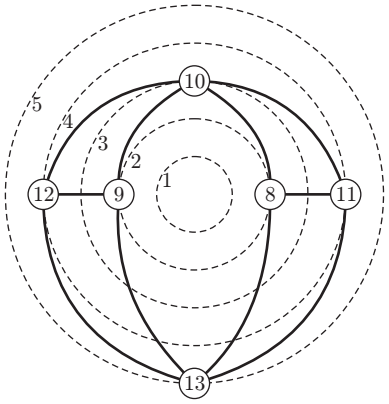


(b) A local minimum while traversing a path from v in C_S inwards

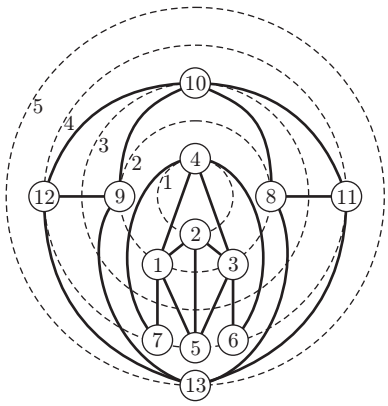
Figure 5.8. The rotations to avoid cut edges may fall for local extrema



(a) The inner ring R, $\gamma_R = 1$, $\delta_R = 4$



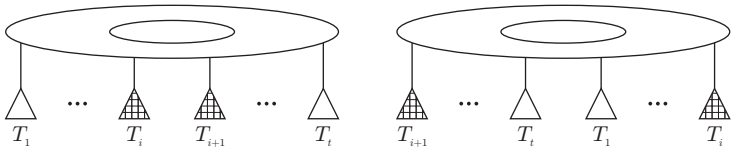
(b) The outer ring S, $\alpha_S = 2$, $\beta_S = 5$



(c) R nested in the centre face of S

Figure 5.9. Nesting of rings

the admissible edge permutations of radial level planar graphs. They are based on PQ-trees and contain a new "R" node type for the rings. PQR-trees are not related to SPQR-trees that are used for example in incremental planarity testing [41]. *R-nodes* are similar to Q-nodes. Their new properties express the differences between rings and other biconnected components. An R-node is drawn as an elliptical ring. The admissible equivalence operations on an R-node are reversion, i. e., inverting the iteration direction of its children in the same way as for Q-nodes, and a new one, *rotation*. Since rings always contain the centre, it is possible to rotate a ring. This corresponds to rotating the graph around the centre and moves a subsequence of the children of the R-node from the beginning of the children list to its end, or vice versa, maintaining their relative order. See Figure 5.10 for an example. This happens implicitly on circular lists. Therefore R-nodes (as well as Q-nodes) can be efficiently implemented with the *improved symmetric list* data structure explained in Appendix A. This is an encapsulated data structure where insertions, reversions and rotations can be done in constant time. This is crucial for the linear running time of the test.



(a) An R-node with children $T_1, T_2, \ldots, T_t$

(b) Figure 5.10(a) after rotation

Figure 5.10. Rotation of an R-node

Lemma 5.8. *During the radial planarity test the admissible edge permutations can be stored such that R-nodes only occur as the root of PQR-trees.*

Proof. At any time in the wavefront sweep the leaves of a PQR-tree represent edges to vertices of the unvisited part of the graph. When a ring R is encountered, radial planarity implies that there are no such edges left originating from a component nested within the centre face of R. Otherwise, an edge would cross a cycle of R which encloses the centre. Hence, it is sufficient that R-nodes never have siblings and thus they only occur at the root of PQR-trees. This follows from the definition of PQ-trees, since P-nodes or Q-nodes must have at least two children, see [13, p. 339]. The same holds in PQR-trees. A P-node encodes arbitrary permutations of its children. If it has a single child, there is only one permutation as it is the case if the child is on the position of its P-parent. The same argument holds for a Q-node encoding reversion of its children. As we see later, R-nodes can have a single child, and thus chains of R-nodes representing nested rings would be possible. This is unnecessary because it suffices to keep only the outermost ring in the PQR-tree,

since the embedding of the inner components can be left unchanged. The contours of the inner rings are cycles, there are no crossings, and they are only connected via cut vertices to the next outer ring. □

5.3.4 New Templates

For the R-nodes twelve new templates are needed to implement REDUCE on PQR-trees, some of them being analogous to Q-node templates. They are given in Figures 5.11 to 5.22. Similar to P0 and Q0, R0 is actually not used because we initialise the bottom up strategy of REDUCE only with full leaves. REDUCE has to test the templates in the order P1, ..., P9, Q1, ..., Q7, R1, ..., R4. The descriptions of the templates assume implicitly that previously tested templates do not fit and that there are at least two children of every P-node and Q-node.

5.3.4.1 P-Templates

Template P7 The node X is a P-node and not the root or the pertinent root. It has exactly two partial Q-nodes X' and X'' and an arbitrary number of other empty and full nodes as children. Then all empty children of X are grouped by the empty P-node X, which is attached as a child at the empty end either of X' or X''. All full children of X are grouped by a new full P-node, which is attached as a child at the pertinent end of either X' or X''. Afterwards, X' and X'' are concatenated into one boundary partial Q-node. Observe that P7 is the variant of P6 for nodes different to the pertinent root.

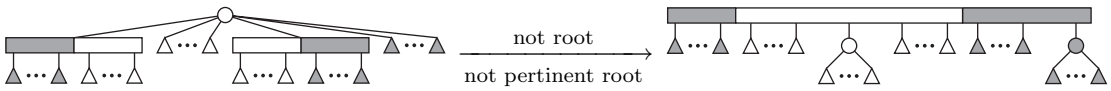


Figure 5.11. Template P7

Template P8 The node X is a P-node and the root. As a consequence X is also the pertinent root. It has exactly one boundary partial Q-child X' and an arbitrary number of other full nodes as children. It has no empty children. Then all full children of X are grouped by the full P-node X, which is attached to X' as a child at an arbitrary end of X'. Afterwards, X is replaced by a new partial R-node which becomes the pertinent root.

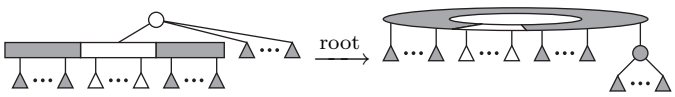


Figure 5.12. Template P8

Template P9 The node X is a P-node and not the root or the pertinent root. Besides, this template has the same pattern as P8. X has exactly one boundary partial Q-child X' and an arbitrary number of other full children. X has no empty children. Then all full children are grouped by the full P-node X, which is attached at an arbitrary end of X' as a child. X' remains boundary partial.

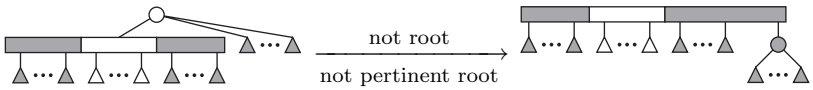


Figure 5.13. Template P9

5.3.4.2 Q-Templates

Template Q4 The node X is a Q-node and the root. As a consequence X is also the pertinent root. It has at most two partial Q-nodes X' and X'' and an arbitrary number of other empty or full nodes as children. X' and X'' cannot be the only children, because then template Q3 would apply. X' and X'' are located at the beginning and at the end, respectively, of X's consecutive sequence of empty children. Thus all empty children are enclosed between them. Then the children of X' and X'' are attached to X as children. X is afterwards replaced by a new partial R-node which becomes the pertinent root. Observe that Q4 is the outwards turned variant of Q3.

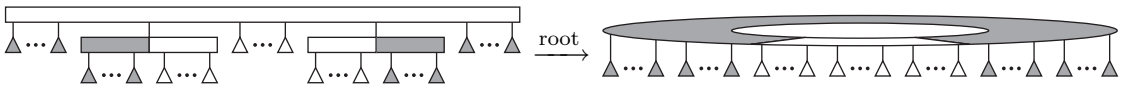


Figure 5.14. Template Q4

Template Q5 The node X is a Q-node and not the root or the pertinent root. As a consequence it is not the root. Besides, this template has the same pattern as Q4. X has at most two partial Q-nodes X' and X'' and an arbitrary number other empty and full nodes as children. The partial Q-children are located at the beginning and at the end, respectively, of X's consecutive sequence of empty children. Thus all empty children are enclosed between them. Then the children of X' and X'' are attached to X as children. X is marked as boundary partial.

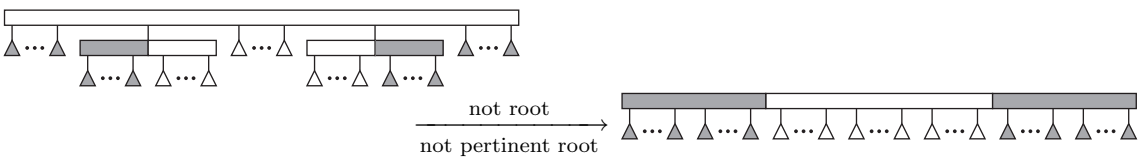


Figure 5.15. Template Q5

Template Q6 The node X is a Q-node and the root. As a consequence X is also the pertinent root. It has exactly one boundary partial Q-node X' and an arbitrary number of other full nodes as children. X has no empty children. Then, after the children of X' are attached to X as children, X is replaced by a new partial R-node which becomes the pertinent root.

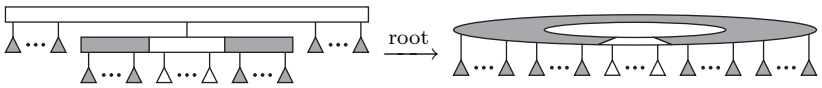


Figure 5.16. Template Q6

Template Q7 The node X is a Q-node and not the root or the pertinent root. Besides, this template has the same pattern as Q6. X has exactly one boundary partial Q-node X' and an arbitrary number of other full nodes as children. It has no empty children. Then, after the children of X' are attached to X as children, X is marked as boundary partial.

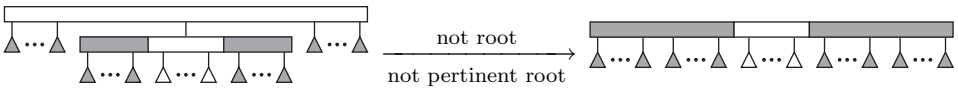


Figure 5.17. Template Q7

5.3.4.3 R-Templates

Template R0 The node X is an R-node and thus the root. Besides, this template has the same pattern as Q0. X has only empty children. X remains empty. There are no structural changes in the PQR-tree.

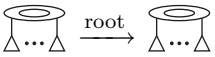


Figure 5.18. Template R0

Template R1 The node X is an R-node and thus the root. Besides, this template has the same pattern as Q1. X has only full children. Then all children of X are grouped by a new full Q-node Y, which is attached to X as a child. X remains empty and Y becomes the pertinent root.

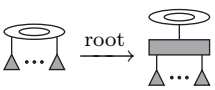


Figure 5.19. Template R1

Template R2 The node X is an R-node and thus the root. Besides, this template has the same pattern as Q2. X has at most one partial Q-node X' and an arbitrary number of other empty and full nodes as children. X' is located at the beginning of its consecutive pertinent sequence. Then, after the children of X' are attached to X as children, X is marked as partial and X becomes the pertinent root.

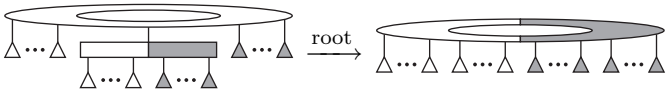


Figure 5.20. Template R2

Template R3 The node X is an R-node and thus the root. Besides, this template has the same pattern as Q3. X has at most two partial Q-nodes X' and X'' and an arbitrary number of other empty and full nodes as children. X' and X'' are located at the beginning and at the end, respectively, of its consecutive pertinent sequence. Thus all full children are enclosed between them. Then, after the children of X' and X'' are attached to X as children, X is marked as partial and X becomes the pertinent root.

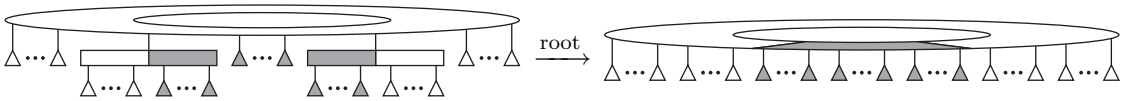


Figure 5.21. Template R3

Template R4 The node X is an R-node and thus the root. Besides, this template has the same pattern as Q7. X has exactly one boundary partial Q-node X' and an arbitrary number of other full nodes as children. It has no empty children. Then, after all children of X' are attached to X as children, X is marked as partial and X becomes the pertinent root.

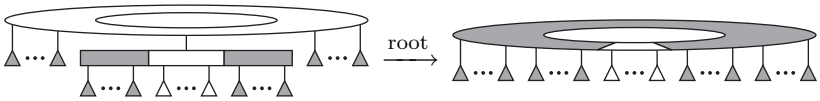


Figure 5.22. Template R4

A new R-node is generated only by the templates P8, Q4, or Q6 illustrated in Figures 5.12, 5.14, and 5.16. The displayed children are optional, as long as the child sequence of the resulting R-node starts and ends with pertinent children and has at least one empty child. Obviously, in these cases it is not possible to apply any of the standard templates, i. e., the graph is not level planar anymore. An R-node is created only when needed, i. e., if newly encountered edges transfer a represented biconnected component from level planar into a ring. By Lemma 5.8 templates P8,

Q4, and Q6 may only be applied to the root of a PQR-tree. This is different from the restriction that some PQ-tree templates may only be applied to the pertinent root.

The meet level between two children of an R-node which are direct siblings or are both endmost is defined and maintained analogously to the meet levels between children of a Q-node, cf. p. 27. To determine which graph components fit below a ring component we define the following:

Definition 5.4 (minML)**.** *For an R-node X with children X_1, X_2, $\dots X_t$ let*

$$\text{minML} = \min\{\, \text{ML}(X_i, X_{i+1}) \mid 1 \leq i \leq t, X_{t+1} = X_1 \,\}.$$

To treat patterns with an R-node as the root it is necessary to provide new templates R0–R4, shown in Figures 5.18 to 5.22, that are similar to the templates Q0–Q3 and Q6. Before an R-template can be applied it may be necessary to rotate the R-node. R0, R2, R3, and R4 are the straightforward transformations of Q0, Q2, Q3, and Q6, respectively. For technical reasons we introduce a new pseudo Q-node Y in R1 as the parent of all full children. This R-node preserves the information that the PQR-tree represents a ring component and allows a later computation of minML, i. e., the least level on which a component fitting below this ring can have a vertex. The single meet level $\text{ML}(Y, Y)$ at the root of the replacement is set to minML of the R-root X of the pattern, which can be done after an appropriate rotation of X. This rotation is not done explicitly because REPLACE will remove the pertinent subtree anyway.

In P8 and Q6 a Q-node may be *boundary partial*, i. e., it may have pertinent children at the boundaries, enclosing some empty children in the middle. In radial level planar graphs this can occur if the root of the PQR-tree is already an R-node or becomes an R-node during the current reduction step and thus if a rotation is possible thereafter. Then the front and the back can be connected by cut edges. Clearly, for that each child of an ancestor of the boundary partial Q-node Z must be full if it is not on the path from Z to the root. For an example see Figure 5.23. All pertinent children become children of the R-node and can be made consecutive by a rotation. If no template matches for a boundary partial Q-node during REDUCE, the graph is not radial level planar, because its PQR-tree contains non-consecutive pertinent nodes. Observe that the templates prohibit a boundary partial Q-node being created at the pertinent root, because this always results in a non consecutive pertinent sequence, except in one special case: Because of R1, an R-root is the only internal node which may have a single child in a valid PQR-tree. If this single child later becomes boundary partial and would be the pertinent root during REDUCE, we must explicitly set the pertinent root to its father R-node to allow the application of R4 and a rotation thereafter. In contrast, in (level) planarity testing the graph is immediately rejected as non-planar as soon as a Q-node is known to be boundary partial because then a consecutive pertinent sequence cannot be formed.

For the boundary partial Q-nodes we must provide the additional templates P7–P9 and Q5–Q7. P7 is the straightforward transformation of P6 if P6 is not applied to

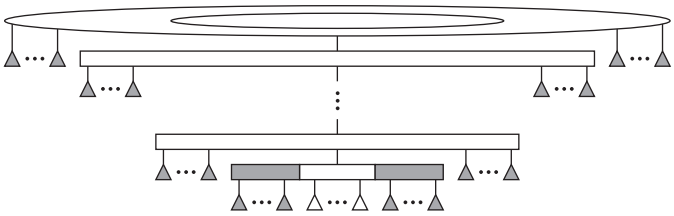


Figure 5.23. Iterative merges of boundary partial Q-nodes

the pertinent root. The full children are grouped by a new P-node which is inserted into the Q-node. It is admissible to place it at either boundary of the Q-node. The only difference between these two positions is whether or not the edges represented by the descendant leaves later become cut edges. This holds accordingly for the new P-node created in P8 or P9. P8 can only be applied to the root, otherwise P9 is applied. Template Q5 is basically the same as Q4 but it treats non-roots. Q4 and Q5 are the inversions of Q3. Templates Q6 and Q7 are used for Q-nodes with only full children except for one boundary partial child. The former is for the root and the latter for a non-root. Now we are ready to establish another important property of R-nodes. Further, we see that no template is destructive because they are specifically constructed that way.

Lemma 5.9. *If an R-node is created, it is preserved until the PQR-tree containing it is deleted.*

Proof. There is no template which destroys or replaces an R-node. Furthermore, R1 ensures that an R-node never becomes full, which means that it is never replaced by an application of REPLACE. □

Observation 5.1. *None of the templates P1, ..., P9, Q1, ..., Q7, R0, ..., R4 destroys radial level planarity.*

5.3.5 Merge Operations on PQR-Trees

Since radial level planarity works on graphs which are not necessarily hierarchies, merges of PQR-trees are needed for the same reason as for PQ-trees. If there is no R-node, the merge conditions for PQR-trees are essentially the same as those for PQ-trees described in Section 2.2. Because of Lemma 5.7 merge condition E cannot be applied if any of the trees has an R-root. As a consequence a merge operation may fail in contrast to the non-radial case, where condition E is always admissible if no other condition applies. For PQR-trees with an R-node as root we have to provide two additional merge conditions. If the root of the guest PQR-tree T_e^v is an R-node then the merge operation fails and the input graph is rejected as radial level non-planar, see proof of Lemma 5.8. For an R-root X of the host PQR-tree T_1^v, condition B and C collapse into the new condition C^{R}. This is because R-nodes can be rotated such that the merge can be done on its interior children. Similarly, if X is the root of the pattern of condition D and X is an R-node we obtain D^{R}.

Merge Condition C$^{\mathbf{R}}$ The root of T_e^v is not an R-node. The node X is an R-node with ordered children $X_1, X_2, \ldots, X_t$, $X' = X_i$, $1 < i < t$, and $\mathrm{ML}(X_{i-1}, X_i) < \mathrm{LL}(T_e^v)$ and $\mathrm{ML}(X_i, X_{i+1}) < \mathrm{LL}(T_e^v)$. Then replace X' with a new Q-node Y with X' and T_e^v as children.

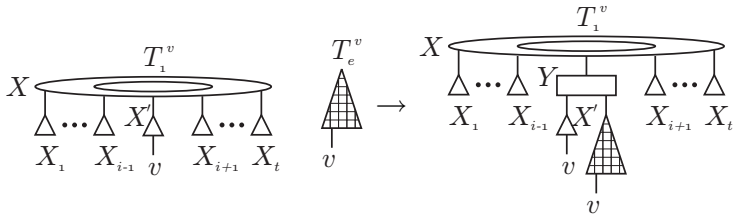


Figure 5.24. Merge condition C$^{\mathrm{R}}$

Merge Condition D$^{\mathbf{R}}$ The root of T_e^v is not an R-node. The node X is an R-node with ordered children $X_1, X_2, \ldots, X_t$, $X' = X_i$, $1 < i < t$, and

$$\mathrm{ML}(X_{i-1}, X_i) < \mathrm{LL}(T_e^v) \leq \mathrm{ML}(X_i, X_{i+1}).$$

Then attach T_e^v as a child of X between X_{i-1} and X_i. If

$$\mathrm{ML}(X_i, X_{i+1}) < \mathrm{LL}(T_e^v) \leq \mathrm{ML}(X_{i-1}, X_i)$$

then attach T_e^v as a child of X between X_i and X_{i+1}.

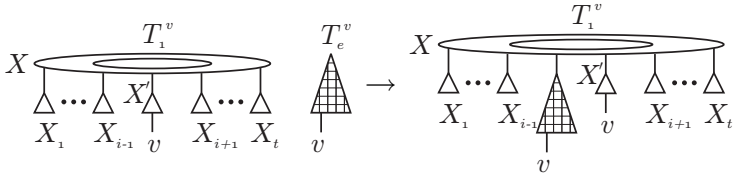


Figure 5.25. Merge condition D$^{\mathrm{R}}$

Merge Condition E The node X' is the root of T_1^v. X' and the root of T_e^v are not R-nodes. Then reconstruct T_1^v by inserting a new Q-node Y as the new root with X' and T_e^v as children.

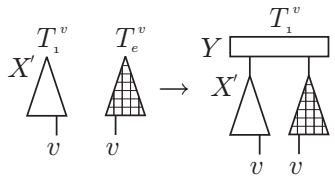


Figure 5.26. Merge condition E

5.3.6 Nesting of Processed Non-Rings

In level planar graphs separate components can always be placed next to each other without violating planarity. This is not necessarily true for radial level planar graphs. If a component of the input graph G contains a ring, it must be checked that each other component detected so far fits into an inner face of the ring or into its outer face. First we consider the case that the other components do not contain a ring. For the efficient execution of the additional checks, the algorithm maintains the lowest level minLL where an insertion of such a component is necessary.

Definition 5.5 (minLL)**.** *A* completely processed PQR-tree *is a PQR-tree representing a component of the graph with no vertices on the current or on higher levels.* $\text{minLL} = \min(\{\,\text{LL}(T) \mid T$ *is a* completely processed PQR-tree *without an R-root*$\} \cup \{\infty\})$*. If there is no completely processed tree* T *then* $\text{minLL} = \infty$*.*

The detection of a processed PQR-tree T works as follows: After every call of REPLACE-SINGLE we check whether T consists of a single leaf (or of an R-node with one leaf) and whether the vertex represented by this leaf is a sink of the graph. As soon as a PQR-tree T is classified as completely processed after REPLACE-SINGLE, minLL is updated by $\min\{\text{minLL}, \text{LL}(T)\}$. All processed PQR-trees are discarded as in the JLM testing algorithm. It suffices to check whether the component C of the completely processed PQR-tree starting at the lowest level fits into an internal face. For all other processed (non-ring) components there is enough space to embed them in the same face as C. Inner faces are always closed by a call of REPLACE-SINGLE for a vertex v. If there is a processed PQR-tree without an R-root, i. e., if $\text{minLL} < \infty$, we check if the newly created inner face starting at the lowest level can include C. For that we use the same mechanism as JLM do for v-singular forms and compare minML with the new PML/QML value, see p. 25. If $\text{minLL} > \text{PML}$ or $\text{minLL} > \text{QML}$, we set $\text{minLL} = \infty$. Otherwise, we need not care whether another processed component smaller than C and whose PQR-tree has already been discarded can be nested inside a face without violating planarity. These will fit later when a face for C is found. If no such face can be found, the graph is not radial level planar. Recall that a processed PQR-tree with an R-root cannot be included in this way. Their nesting is described in the next section.

5.3.7 Nesting of Processed Rings

Our algorithm maintains the invariant that at any time while testing a radial level graph there is at most one PQR-tree T^R with an R-root. This is no restriction of planarity as is described in the following. T^R may be processed. A *link vertex* v denotes the vertex for which the reduction of all leaves labelled with v transforms the component represented by the PQR-tree into a ring. At the start of the algorithm T^R is undefined and the invariant is obviously true. As the process continues, it is maintained as follows: If the algorithm detects a ring for the first time, T^R is defined. It remains defined until the end of the algorithm. However, the tree for T^R

may change. If another PQR-tree T gets an R-root by the application of template P8, Q4 or Q6 during the reduction of a link vertex v, we proceed as described by Algorithm 5.1.

Algorithm 5.1. TREAT-NEW-RING

Input: A PQR-tree T of a newly encountered ring, the link vertex v, T^R, and minLL

Output: A Boolean value for radial level planarity

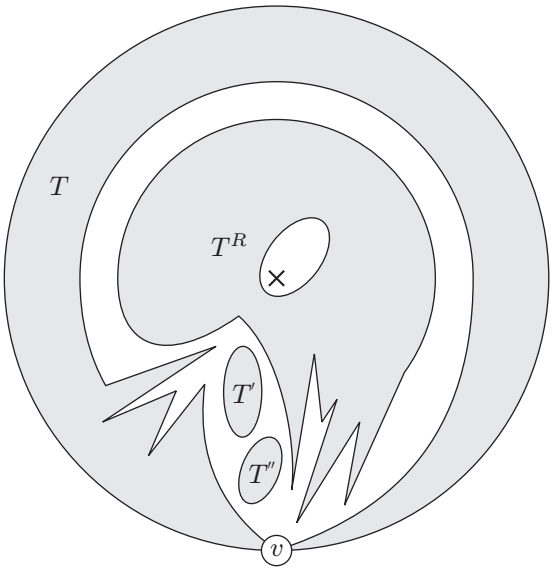
```
if T^R ≠ NULL then
    if T^R is not completely processed and T^R is not v-singular then
        return false
    end
    minML ← min{ML between the children of the root of T^R}
    if minML ≥ LL(T) or minML ≥ minLL then
        return false
    end
    delete T^R
end
T^R ← T

return true
```

If there is a PQR-tree T^R with an R-node as root, it must be either completely processed or v-singular. Otherwise, T^R has leaves which represent vertices different to v on level $\phi(v)$ or higher, i. e., G is not radial level planar as we have already seen in the proof of Lemma 5.8. The algorithm checks whether minML is small enough for T to fit below T^R. Moreover, the tree with the smallest low indexed level minLL and thus all other trees must fit between T and T^R. Recall that before the nesting T^R must be rotated and squeezed such that all its jags are embedded into the space above v and such that the indentation of T^R with the minML meet level encloses all inner jags of T. See Figure 5.27 for an illustration. The rotation of T^R is not done explicitely because T^R is discarded anyway.[2]

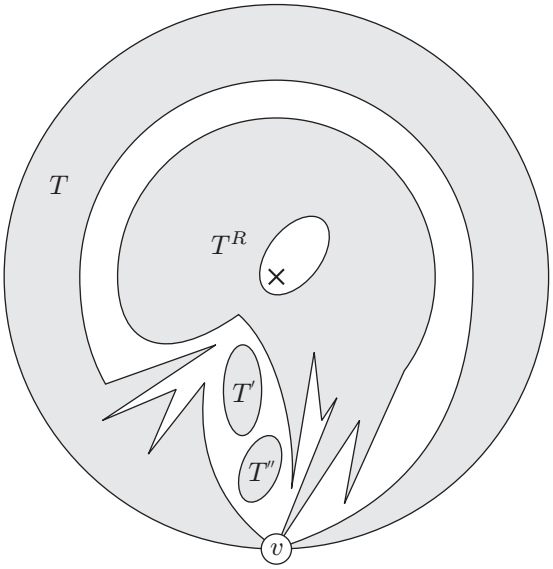
If any of the checks fails then by Lemma 5.7 G is not radial level planar. That means if R and S are the rings represented by T^R and T, respectively, either $\delta_R \geq \beta_S$, $\gamma_R \geq \alpha_S$, or $\gamma_R \geq$ minLL. We will see later in the proof of Lemma 5.11 that γ_R corresponds to minML of T^R and that β_R corresponds to $\phi(v)$. If $\gamma_R \geq$ minLL, there is a component for which it was not possible to embed it in a prior step into an inner face of R. Now it turns out that it does not fit in the outer face of R either. Finally in the algorithm, T^R is updated. By the construction of Algorithm 5.1 the invariant shown by Lemma 5.10 is preserved.

Lemma 5.10. *At any time while testing a radial level graph, the collection of trees $\mathcal{T}$ contains at most one PQR-tree with an R-node as its root.*

[2]When computing an embedding this has already been done by an earlier application of the embedding variant of template R1 as we will see later in Section 5.4.3.



(a) T^R is completely processed



(b) T^R is v-singular

Figure 5.27. Schematic nesting of a processed ring. T' and T'' correspond to two other completely processed components

5.3.8 Completion

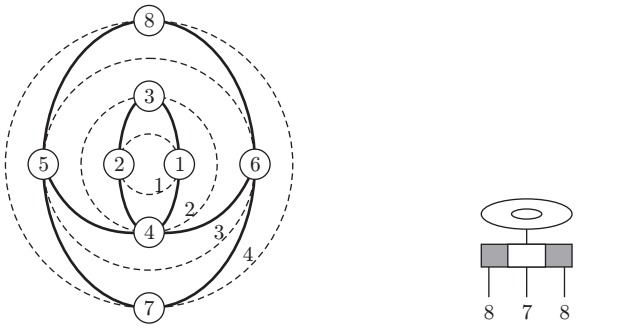
Finally, if there is no PQR-tree T^R representing a ring graph then the graph is level planar. Otherwise, if no other trees have occurred after T^R has been detected, the graph is radial level planar. This is the case if minLL $= \infty$. If $minLL < \infty$ it remains to check whether the other PQR-trees fit below T^R, i. e., minML $<$ minLL. Otherwise, G is not radial level planar since there is no face including the outer face which spans enough levels to completely contain the component which defines minLL.

5.3.9 Correctness

For the correctness of the algorithm every computed embedding of a ring must be level optimal, and this property is granted by our algorithm.

Lemma 5.11. *Our radial level planarity test induces a level optimal embedding for every ring.*

Proof. Let R be a ring of the given graph. As long as the corresponding PQR-tree does not contain an R-node, the centre of the concentric levels lies in the outer face. Only the templates P8, Q4, and Q6 introduce a new R-node which closes the centre face. This does not cover the case shown in Figure 5.28, where two nested rings share a common vertex on a lower level than the link vertex of the outer ring. Then the centre face of the outer ring is closed by the application of template R4.



(a) Vertex 4 links two rings

(b) Before closing the outer ring

Figure 5.28. Linked and nested rings

As these four templates are only applied if no other template matches, there is no admissible permutation which allows the closure of the centre face on a higher level. Hence, the centre face ends on level β_R, which is the radius of the centre face. Note that inserting v-singular forms into the centre face of R does not influence β_R.

Each PQR-tree representing a ring R has an R-node as its root. At any time during the application of the algorithm the indentations of the outer face are represented

by the ML-values between two siblings in the PQR-tree. Since Observation 3.1 is also true for PQR-trees, the least ML-values are stored between children of the root. Thus minML represents the highest indentation of the outer contour of R, i. e., the contour of the outer face of R in the embedding. The value of minML can only change if an inner face is closed by REPLACE-SINGLE. Then there may be several faces which can be closed due to the freedom of rotation. Which one is taken only depends on the templates applied in REDUCE. Only template R1 has multiple options. Since R1 always preserves the minimum meet level as shown in Section 5.3.4, it is guaranteed that the highest possible indentation is preserved whenever this is possible. Thus the outer radius $\gamma_R = \text{minML}$ is level optimal in the induced embedding. □

Lemma 5.12. *The REDUCE operation, extended with the new templates from Figures 5.11 to 5.22, correctly computes the new set of admissible permutations for radial level planarity.*

Proof. We follow the corresponding arguments for PQ-trees in [13, p. 348f]. First, no template violates radial level planarity, see Observation 5.1. Second it must be shown that any radial planar graph can be processed successfully, i. e., no further templates are necessary. Consider the complete case analysis of node types, their position in the tree, and the order of their empty, full, and (doubly/boundary) partial children in Tables 5.1 and 5.2. Note that partial child in both tables always means partial Q-child. We do not consider partial P-nodes as pertinent children because they never occur according to the bottom up traversal of REDUCE and Template P3. For example consider line 8 of Table 5.1. If any vertex has three or more partial children (and an arbitrary number of all kinds of other children) there is no way to construct a consecutive pertinent sequence without violating planarity since each of the partial children corresponds to a biconnected component which can only be reversed. The consequence of this complete enumeration of all cases in a PQR-tree is that either a presented template fits or radial level non-planarity follows, i. e., no further templates are needed. □

In analogy to Jünger et al. [95, 97, 99, 112] this implies our first main theorem of this chapter:

Theorem 5.1. *There is an $\mathcal{O}(|V|)$ time algorithm for testing radial k-level planarity.*

Proof. The linear running time follows directly from the linear running time of the JLM algorithm. We have only a constant number of new templates to test for each pertinent node during REDUCE. The height of the PQR-trees is the same as for PQ-trees, except for the technical exception that an R-node may have only one child which results in a difference of at most one. The number of merges of PQR-trees stays also linear to the number of vertex reductions. The nesting of components needs constant time per face. □

Table 5.1. Situations in a PQR-tree with an special set of children of the currently treated node. The abbreviation n. p. stands for radial level non-planar

children					not pertinent root		pertinent root and not root		pertinent root and also root		
empty	full	partial	doubly partial	boundary partial	P	Q	P	Q	P	Q	R
0	0	0	0	0	—	—	—	—	—	—	—
1	0	0	0	0	—	—	—	—	—	—	—
> 1	0	0	0	0	P0	Q0	P0	Q0	P0	Q0	R0
0	1	0	0	0	—	—	—	—	—	—	—
0	> 1	0	0	0	P1	Q1	P1	Q1	P1	Q1	R1
0	0	1	0	0	—	—	—	—	—	—	—
0	0	2	0	0	P7	Q5	P6	Q3	P6	Q3	R3
		> 2			n. p.	n. p.	n. p.	n. p.	n. p.	n. p.	n. p.
0	0	0	0	1	—	—	—	—	—	—	R4
				> 1	n. p.	n. p.	n. p.	n. p.	n. p.	n. p.	n. p.
			> 0		n. p.	n. p.	n. p.	n. p.	n. p.	n. p.	n. p.
		> 0		> 0	n. p.	n. p.	n. p.	n. p.	n. p.	n. p.	n. p.
	> 0			1	n. p.	n. p.	n. p.	n. p.	n. p.	n. p.	n. p.

Table 5.2. Situations in a PQR-tree with arbitrary children sets of the currently treated node. The special cases of Table 5.1 are regarded as already filtered out

children				not pertinent root		pertinent root and not root		pertinent root and also root		
consecutiveness	position	partial	boundary partial	P	Q	P	Q	P	Q	R
pertinent children not consecutive	pertinent children not consecutive at boundaries	0	0	P3	n. p.	P2	n. p.	P2	n. p.	n. p.
		1		P5	n. p.	P4	n. p.	P4	n. p.	n. p.
		2		P7	n. p.	P6	n. p.	P6	n. p.	n. p.
	pertinent children consecutive at boundaries	0	0	P3	Q5	P2	n. p.	P2	Q4	R3
		1 at empty begin or end		P5	Q5	P4	n. p.	P4	Q4	R3
		1 not at empty begin or end		P5	n. p.	P4	n. p.	P4	n. p.	n. p.
		2 at empty begin and end		P7	Q5	P6	n. p.	P6	Q4	R3
		2 not at empty begin and end		P7	n. p.	P6	n. p.	P6	n. p.	n. p.
pertinent children consecutive	pertinent children at (w. l. o. g.) right boundary	0	0	P3	Q2	P2	Q2	P2	Q2	R2
			1	P9	Q7	n. p.	n. p.	P8	Q6	R4
		1 at pertinent begin	0	P5	Q2	P4	Q2	P4	Q2	R2
		1 at pertinent end		P5	n. p.	P4	Q3	P4	Q3	R3
		1 not at pertinent begin or end		P5	n. p.	P4	n. p.	P4	n. p.	n. p.
		2 at pertinent begin and end		P7	n. p.	P6	Q3	P6	Q3	R3
		2 not at pertinent begin and end		P7	n. p.	P6	n. p.	P6	n. p.	n. p.
	pertinent children in the middle	0	0	P3	n. p.	P2	Q3	P2	Q3	R3
			1	P9	Q7	n. p.	n. p.	P8	Q6	R4
		1 at pertinent begin	0	P5	n. p.	P4	Q3	P4	Q3	R3
		1 not at pertinent begin		P5	n. p.	P4	n. p.	P4	n. p.	n. p.
		2 at pertinent begin and end		P7	n. p.	P6	Q3	P6	Q3	R3
		2 not at pertinent begin and end		P7	n. p.	P6	n. p.	P6	n. p.	n. p.

5.4 Radial Level Planar Embedding

Algorithm 3.6 describes the algorithm of Jünger et al. [95–97, 112] for computing level planar embeddings of level planar graphs. This algorithm is extended to compute radial level planar embeddings of radial level planar graphs consisting of an ordering of the vertices and a specification of clockwise and counterclockwise cut edges. We extend the upward embedding algorithm of Chiba et al. [26] to work with PQR-trees instead of PQ-trees and present a new algorithm for constructing a radial level planar embedding from the upward embedding.

5.4.1 Meet Levels between Ignored Siblings

When computing an embedding, PQ-trees can contain ignored nodes. Since we use the same strategy for computing radial embeddings, we have to treat ignored nodes. This is particularly important when minML is computed because we have to consider ML-values between any pair of children of the R-node which are direct siblings. This includes ignored children. Therefore we have to ensure that the ML-values of ignored nodes are computed correctly. Particularly, the outer ML-values have to be initialised when a Q-node with outermost ignored children is inserted into another Q-node. This is straightforward and can be done in constant time.

5.4.2 Contacts as Children of R-nodes

At the end of Section 3.4 we mentioned that a search for sink indicators after inserting a Q-node into its father Q-node must be avoided to obtain linear running time. This is also true for inserting a Q-node into an R-father. Thus contacts are also necessary as children of an R-node. Their treatment is analogous to that for children of a Q-node in JLM's algorithm for creating a level planar embedding. For R-nodes contacts are introduced by the merge operation C^R.

5.4.3 Embedding the Edges

We not only have to compute a vertex ordering $\leq_j$ on each level j but also the edge routing. It is not necessary to sort the adjacent edges of each vertex as it has been done in [26], but it suffices to determine cut edges. The detection of cut edges is done by the *st*-embedding creation step described in Section 5.4.5, and not during the augmentation phase since we need to know the cut edges and their types for the *st*-embedding to generate a radial level embedding.

Initially there are no edges marked as cut edges. They are recognised as follows: For an R-node we introduce a new child denoted by *ray indicator* and labelled with \$ which marks where the ray splits the children. Like the sink indicators the ray indicator is ignored throughout the algorithm, and it remains always a child of the R-node. It is created with every R-node by modified templates P8, Q4, and Q6, see Figures 5.29 to 5.32.

Template R1 with Ray Indicator The node X is an R-node and thus the root. Besides, this template has the same pattern as Q1. Except for its ray indicator \$, X has only full children. Then all children of X except \$ are grouped by a new full Q-node Y which is attached to X as a child. X remains empty and Y becomes the pertinent root. But before the children of X are attached to Y they are rotated such that the minimum meet level is between the first and the last child. This value is used for the meet level between Y and \$ which remains a child of X.

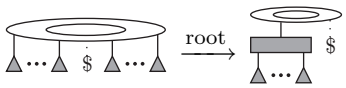


Figure 5.29. Template R1 with ray indicator

Template P8 with Ray Indicator The node X is a P-node and the root. As a consequence X is also the pertinent root. It has exactly one boundary partial Q-child X' and an arbitrary number of other full nodes as children. It has no empty children. Then all full children of X are grouped by the full P-node X, which is attached to X' as a child at an arbitrary end of X'. After the new ray indicator \$ is inserted at an arbitrary end of X as a child, X is replaced by a new partial R-node which becomes the pertinent root.

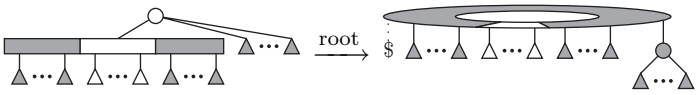


Figure 5.30. Template P8 with ray indicator

Template Q4 with Ray Indicator The node X is a Q-node and the root. As a consequence X is also the pertinent root. It has at most two partial Q-nodes X' and X'' and an arbitrary number of other empty or full node as children. X' and X'' cannot be the only children because then template Q3 would apply. X' and X'' are located at the beginning and at the end, respectively, of its consecutive sequence of empty children. Thus all empty children are enclosed between them. Then the children of X' and X'' are attached to X as children. After the new ray indicator \$ is inserted at an arbitrary end of X as a child, X is replaced by a new partial R-node which becomes the pertinent root. Note that Q4 is the outwards turned variant of Q3.

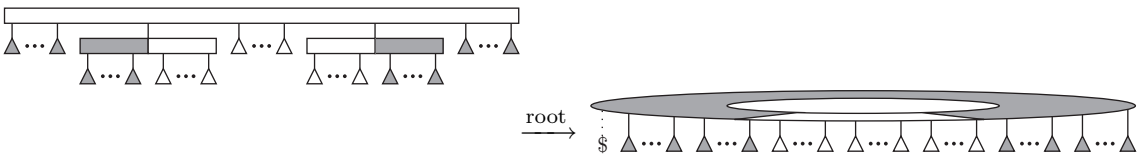


Figure 5.31. Template Q4 with ray indicator

Template Q6 with Ray Indicator The node X is a Q-node and the root. As a consequence X is also the pertinent root. It has exactly one boundary partial Q-node X' and an arbitrary number of full nodes as children. X has no empty children. Then the children of X' are attached to X as children. After the new ray indicator \$ is inserted at an arbitrary end of X as a child, X is replaced by a new partial R-node which becomes the pertinent root.

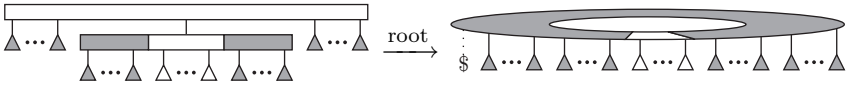
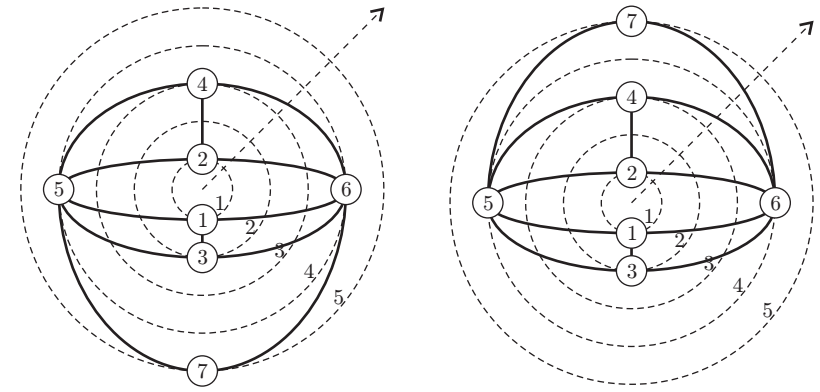


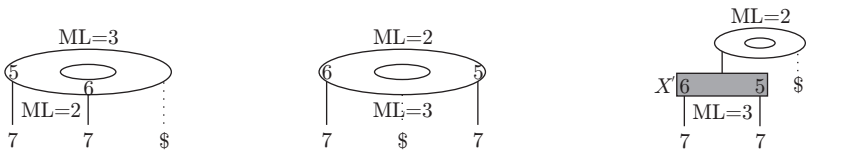
Figure 5.32. Template Q6 with ray indicator

R1 has to be modified, too. Recall that R1 creates a pseudo Q-node Y. Before this is done the R-node is rotated such that the two siblings with minML in between become the end vertices of Y. Otherwise, level optimality may be lost. See Figure 5.33 for an illustration.



(a) Not level optimal

(b) Level optimal



(c) Initial PQR-tree on level 4

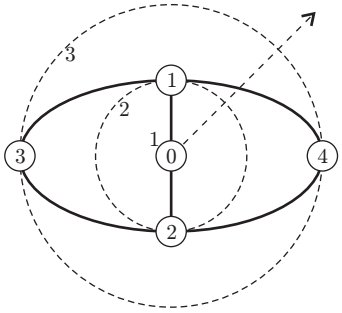
(d) After rotation

(e) Final PQR-tree

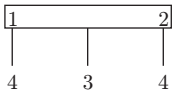
Figure 5.33. Preserving level optimality

The ray indicator \$ may divide the children of Y into two parts. Thus before creating Y it is necessary to drag one part over \$ because \$ must remain a child of the R-node. This is an additional step which is different to the previously described rotation to get minML between the outer children. Remember that rotation is only

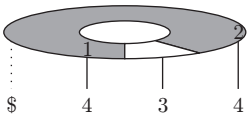
moving children from the front to the back and vice versa, cf. Section 5.3.3. The leaves of all pertinent subtrees that are dragged over the ray indicator represent cut edges. They are computed by DFS without violating the $\mathcal{O}(|V|)$ time bound since REPLACE removes the traversed pertinent subtrees from the PQR-tree after each drag operation. Accordingly, if the ray indicator in REPLACE lies within the pertinent sequence, one part of the pertinent sequence is dragged over the ray indicator before the pertinent sequence is replaced. As an example consider the graph shown in Figure 5.34(a). Figure 5.34(b) shows the corresponding PQR-tree before the reduction of all leaves with label 4, whereas Figure 5.34(c) shows the resulting PQR-tree after the reduction by template Q4. As shown in Figure 5.34(d) the leaf representing the edge $(1, 4)$ is dragged over \$ in REPLACE and thus the edge $(1, 4)$ becomes a cut edge. The number of drag operations is the same as the number of reductions. Thus the linear time complexity is preserved.



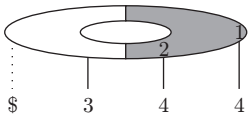
(a) The input graph



(b) The PQR-tree before Q4



(c) Figure 5.34(b) after REDUCE



(d) Edge $(1, 4)$ is dragged over \$

Figure 5.34. Detection of a cut edge while reducing the leaves of vertex 4

5.4.4 Augmenting G to an st-Graph G_{st}

The processed PQR-trees from Section 5.3 are now called *ignored PQR-trees* because they consist of ignored nodes only, cf. Section 3.4. However, the LL-value of the highest ignored PQR-tree minLL is not sufficient here. We must also store the whole ignored PQR-trees, because their sinks must later be augmented with edges if a ring is closed by templates P8, Q4, or Q6. Then all sinks are connected to the link vertex w on which REDUCE was called and which closes the ring. The embedding

of ignored components within a newly encountered ring never introduces crossings because $\phi(w) > \phi(u)$ for each ignored sink u. Furthermore, if inner faces are closed by REPLACE-SINGLE for a vertex v, it is necessary to connect sinks of components which are nested into these faces to v. Hence, we maintain a *collection* $\mathcal{T}^*$ which stores all ignored PQR-trees in addition to the *active collection* $\mathcal{T}$.

When an R-node is created, an existing PQR-tree with an R-root is nested into the centre face. This includes an ignored PQR-tree with an R-root. Only a single PQR-tree T^R is left. In analogy to Lemma 5.10, this leads directly to the following Lemma 5.13.

Lemma 5.13. *$\mathcal{T} \cup \mathcal{T}^*$ contains at most one R-rooted PQR-tree T^R.*

If a vertex v closes a face, it does not suffice to test whether the ignored PQR-tree starting at the lowest level fits into this face after REPLACE-SINGLE. If it fits, additionally all sinks in $\mathcal{T}^*$ are connected to v and $\mathcal{T}^*$ is emptied. Similar to the radial level planarity test, this step is omitted for a face different from the centre face of an outer ring if there exists an ignored R-rooted PQR-tree T^R. Rings cannot be embedded into faces not containing the centre. The other PQR-trees in $\mathcal{T}^*$ are embedded later in the same face as T^R. If they do not fit in this face, the graph is not radial level planar because previous tests after each REPLACE-SINGLE have shown that they do not fit in an inner face of the ring represented by T^R also.

The tests whether minLL and the LL-value of a newly detected ring are greater than the minML-value of an enclosed ignored ring in Algorithm 5.1 can be omitted as an optimisation. These checks are done automatically in the bottom up phase with the single hierarchy rooted at t. However, the sinks have to be connected to the link vertex.

If a PQR-tree contains ignored nodes, the templates P8, Q4, and Q6 can be applied to nodes other than the root of a PQR-tree. There may be a path from the PQ-node X to the root which is the only non-ignored path from the root downwards, i. e., all predecessors of X have only one non-ignored child. Then all vertices represented by nodes that are not descendants of X can be embedded within the ring represented by the new R-root. Therefore these nodes are removed and the corresponding sinks are connected to the link vertex. The $\mathcal{O}(|V|)$ time bound is preserved. If the test on the above situation fails, either the input graph is not radial level planar and the algorithm rejects, or there is a situation similar to the one shown in Figure 5.23 and other templates fit. This case can be checked in $\mathcal{O}(1)$ time since there is no node chain in a PQR-tree and thus the parent Q-node of X has at least one other non-ignored child. If the test does not fail, the traversed nodes are removed. Hence, the total computation time remains linear.

5.4.5 Computation of a Radial Upward *st*-Embedding $\mathcal{E}_u$

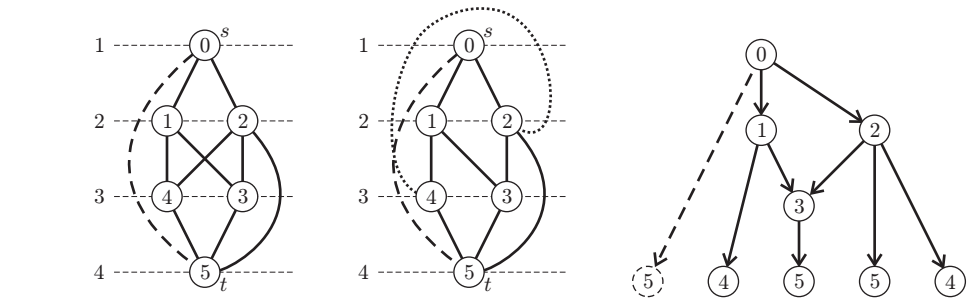
To compute an *st*-embedding $\mathcal{E}_{st}$ of the graph G_{st} (see Algorithm 3.6) the algorithm of Chiba et al. [26] is used. It is based on the vertex addition method of [58, 113] and needs an *st*-graph. But in our case G_{st} has no *st*-edge (s, t). If G is a ring graph, s

and t are not in the same face of any planar level embedding $\mathcal{E}_l$ of G, i. e., s does not lie in the outer face as t does, cf. Lemma 5.3. Therefore the introduction of a new edge (s, t), as in the JLM algorithm, is not possible since it may destroy radial level planarity and the st-embedding algorithm would fail. Thus we omit introducing the edge (s, t) and obtain only an induced st-numbering by numbering the vertices level by level in ascending order. After the radial edge augmentation each vertex except s and t has at least one incoming and at least one outgoing edge. There are no sources other than s and no sinks other than t. Without the st-edge, G_{st} may be not biconnected. The essential property of an st-numbering for embedding a graph with the vertex addition method is that there is a path of higher numbered vertices leading from every vertex to t, which has the highest number. Thus there must exist an embedding of the first i vertices such that the remaining vertices ($i + 1$ to t) can be embedded in a single face (the outer face) of the already embedded part. This property also holds for our induced st-numbering.

If an embedding is computed by the standard vertex addition method [26, 58, 113], the edge (s, t) behaves similarly to the ray in the radial level planarity test. The st-edge is real, however, and therefore no other edge is allowed to cross. Thus cyclic reductions, i. e., cut edges, are not allowed and need not be considered. Without (s, t) cyclic reductions are admissible. We adopt our ideas from extending the level planarity test to the radial case. The standard planar embedding algorithm is updated with the PQR-tree data structure to realise cyclic reductions. Again we omit ENTIRE-EMBED of Chiba's algorithm, shown in Algorithm 2.2, to compute an st-embedding $\mathcal{E}_{st}$ from the upward st-embedding $\mathcal{E}_u$, cf. Section 3.4. Here the reason is both efficiency and correctness. In the radial case the upward embedding $\mathcal{E}_u$ can be seen as an *inward embedding* because every vertex knows its ordered edges from vertices on a smaller and thus inner level. In our approach it is possible to route edges around s. The routing around t is not allowed because we consider only monotone level planar graphs. Figure 5.35(b) without the dashed st-edge is a radial level planar drawing of the graph shown in Figure 5.35(a). If cut edges exist, Chiba's ENTIRE-EMBED may provide an invalid edge ordering around each vertex. The adjacency lists of vertices 0 and 2 in Figure 5.35(h) are incorrect, while in $\mathcal{E}_u$ the orderings of the incoming edges are correct, see Figure 5.35(g). Therefore in Section 5.4.6 we use $\mathcal{E}_u$ instead of $\mathcal{E}_{st}$ to compute a radial level planar embedding $\mathcal{E}_l$.

For instance, Figures 5.35(c) to 5.35(f) show the now admissible cyclic reduction of vertex 4 by an illustration of the bushes, cf. Section 2.3, and their corresponding PQR-trees. With the edge (s, t) $T(B_4)$ cannot be reduced according to vertex 4 because no template fits. Otherwise, template Q4 can be applied and we obtain Figure 5.35(f). Afterwards, replacing 4 with 5 by an application of REPLACE determines one of the edges $(1, 4)$ or $(2, 4)$ as a cut edge.

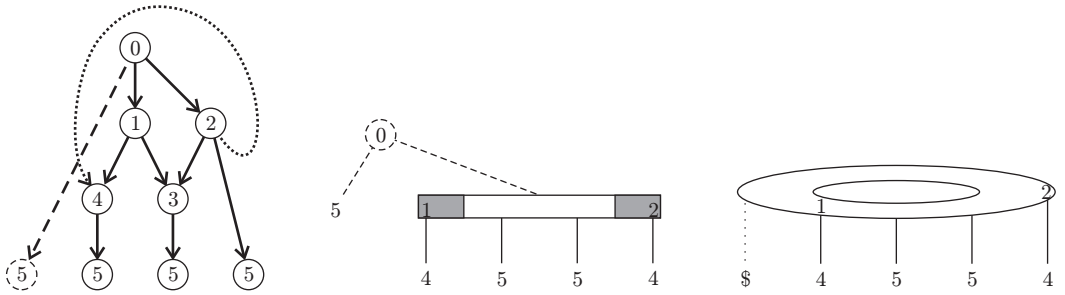
The procedure UPWARD-EMBED from [26, p. 67f] relies on the fact that the leaves which are removed from the PQR-tree by REPLACE for storing the represented edges in $\mathcal{E}_u$ are in an admissible order except for reversion. Possible subsequent reversions of a parent Q-node are handled by direction indicators [26]. Reversions of a parent R-node X are accomplished accordingly. However, if the ray



(a) A planar graph

(b) A planar drawing

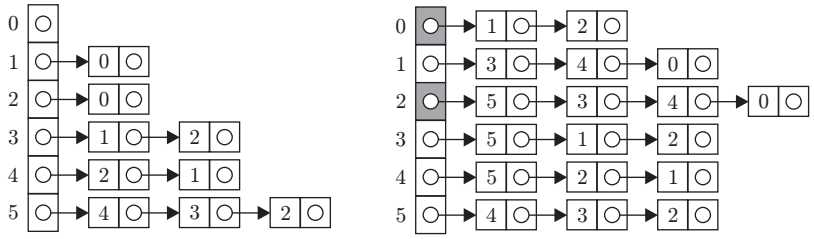
(c) The bush B_4



(d) The bush B_5

(e) $T(B_4)$, reducing 4 is not possible

(f) $T(B_4)$ after reduction of 4



(g) $\mathcal{E}_u$

(h) $\mathcal{E}_{st}$

Figure 5.35. Embedding an edge around s without the (dashed) st-edge. The numbers in the vertices not only show their label but also represent their induced st-numbers

indicator occurs within the pertinent sequence of X, we have to drag a part of the sequence over it. This is done before the removal of the pertinent sequence. Later in the algorithm there is the possibility of a rotation of X and thus of an implicit rotation of its children. However, this only means a rotation of the whole graph including the ray. Hence, the ordering of the stored sequence remains valid. If an R-node has only pertinent children then it is admissible to move the ray indicator arbitrarily, leading to different cut edges and thus to different embeddings. This is not significant because we are interested in a single admissible embedding. Thus, analogously to UPWARD-EMBED of Chiba et al., we obtain a valid inward embedding.

5.4.6 Computation of a Radial Level Embedding $\mathcal{E}_l$

In this section we assume that in the upward embedding $\mathcal{E}_u$ the incoming edges of every vertex are sorted in clockwise order. Before we present our algorithm for computing a radial level embedding $\mathcal{E}_l$ we establish further properties.

Lemma 5.14. *Let $G_{st} = (V_{st}, E_{st})$ be the augmented st-graph. Then every vertex $v \in V_{st} - \{s\}$ has at least one incoming non-cut edge.*

Proof. G_{st} is an induced st-graph without an st-edge. Thus every vertex $v \in V_{st} - \{s\}$ has at least one incoming edge. An edge is only marked as a cut edge in REPLACE or in template R1 if the ray indicator lies within the pertinent sequence. In both cases there are PQ-leaves representing edges on both sides of the ray. They must be placed on one side of the ray indicator. Thus the edges which are not dragged over \$ are non-cut edges. If \$ is already at the beginning or at the end of the pertinent sequence, there are no cut edges. □

Corollary 5.1. *There exists a path from s to every vertex $v \in G_{st}$ not containing a cut edge.*

Lemma 5.15. *In any upward embedding $\mathcal{E}_u$ the ordered adjacency list of a vertex v never contains a cut edge between two non-cut edges.*

Proof. Assume that v has adjacent incoming edges in the ordering e_1, e_c, e_2, where e_1 and e_2 are non-cut edges and e_c is a cut edge, see Figure 5.36.

Let v_1 be the source vertex of e_1 and v_2 the source vertex of e_2. Then $v_1 \neq v_2$. Thus there exist two paths p_1 and p_2 from s to v_1 and from s to v_2, respectively, which according to Corollary 5.1 differ in at least one edge. The cut edge e_c violates planarity by crossing the contour of the face between p_1 and p_2, which is a contradiction. □

This leads to two different types of cut edges according to their position in the adjacency list. We call them clockwise or counterclockwise according to the implicit direction from lower to higher levels.

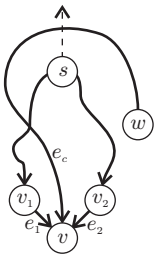


Figure 5.36. No cut edge can be between two non-cut edges

Definition 5.6 (Edge direction). *A cut edge is called* clockwise *with respect to* $\mathcal{E}_u$ *if it occurs at the right end of the incoming adjacency list of its target vertex. Otherwise, it is called* counterclockwise.

Lemma 5.16. *All cut edges of a radial level planar embedding that end on the same level have the same direction (clockwise or counterclockwise) and the same target vertex.*

Proof. First we show that all cut edges with their target vertex on the same level have the same direction. Assume two cut edges with different directions ending on the same level. Their target vertices need not be different. The obtained crossing, see Figure 5.37(a) for an illustration, contradicts radial level planarity. This crossing cannot be avoided because there are paths from s to the source vertices of the cut edges according to Corollary 5.1.

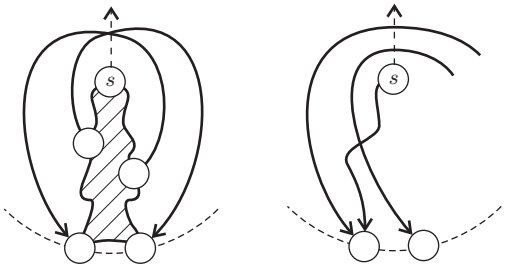


Figure 5.37. In a radial level planar graph all cut edges ending on the same level have the same direction and the same target vertex

It remains to show that there is at most one vertex with incoming cut edges on a level. Assume that there are two vertices on the same level which both have an incoming cut edge. We have already shown that they have the same direction. Then the inner cut edge crosses a path from s to the target of the other cut edge, see Figure 5.37(b) for an illustration. Such a path always exists because of Corollary 5.1. This contradicts radial level planarity. □

Algorithm 5.2. CONSTRUCT-LEVEL-EMBED

Input: The upward embedding $\mathcal{E}_u$ and the st-graph $G_{st} = (V_{st}, E_{st})$
Output: A radial level planar embedding $\mathcal{E}_l$

```
procedure DFS((v, dir))
    if visited[v] = false then
        visited[v] ← true
        if dir = left then
            insert v at the left end of E_l[φ(v)]
        else
            insert v at the right end of E_l[φ(v)]
        end
        foreach incoming non-cut edge e of v scanned in direction dir do
            DFS((source(e), dir))
        end
        if v has incoming clockwise cut edges then
            foreach incoming cut edge e of v scanned from right to left do
                insert(Q, (source(e), left))
            end
        else
            foreach incoming cut edge e of v scanned from left to right do
                insert(Q, (source(e), right))
            end
        end
    end
end
foreach v ∈ V_st do
    visited[v] ← false
end
Queue Q                                   // stores pairs
insert(Q, (t, right))
while Q not empty do
    DFS(delete_first(Q))
end
return E_l
```

Because of the above lemmata we can introduce Algorithm 5.2, CONSTRUCT-LEVEL-EMBED. $\mathcal{E}_l[j]$ denotes the vertex list of the radial level j, ordered by $\leq_j$. The algorithm is a sequence of ordered backward DFS traversals in $\mathcal{E}_u$ which use no cut edges. The first of these traversals starts at the sink vertex t and inserts every visited vertex v at the right end of $\mathcal{E}_l[\phi(v)]$. The part of the graph visited in this first step is called *trunk*, see Figure 5.38. Source vertices of discovered cut edges are placed into a queue together with the information on which side of the trunk they have to be placed later. It is important that these vertices are inserted into the queue in the correct order, from right to left for incoming clockwise cut edges and from left to right for incoming counterclockwise cut edges. The subsequent DFS traversals start at a vertex from the queue and insert visited vertices at the respective side of $\mathcal{E}_l$. Source vertices of newly detected cut edges are again inserted into the queue. The algorithm terminates when the queue is empty and thus all vertices have been visited.

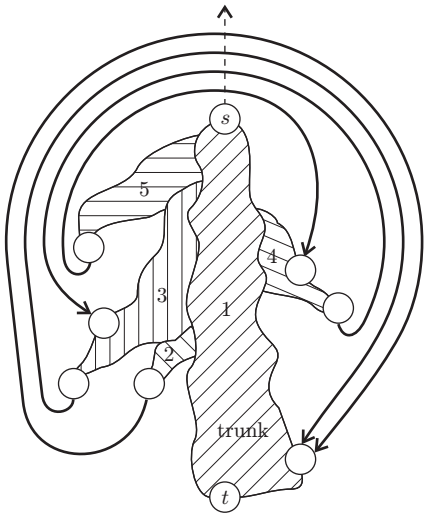


Figure 5.38. Successive and sorted attachments of faces to the sides of the trunk

The correctness of the algorithm relies on the following fact:

Lemma 5.17. *Let G be a level graph with a single sink assuming edge directions from lower to higher levels. Then an upward embedding of G induces a unique level embedding.*

Proof. Assume there are two different level embeddings of G. Then there are two vertices u and v on the same level whose relative positions differ in the two embeddings. From both vertices there exists a path to the sink. Let w be the first common vertex on these paths and let u' and v' be the direct predecessors on the respective paths, see Figure 5.39. Since the paths $u \rightarrow^* u'$ and $v \rightarrow^* v'$ are disjoint and do not cross, the edges (u', w) and (v', w) have different relative positions in the

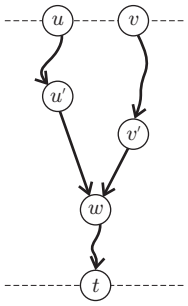


Figure 5.39. An upward embedding induces a unique level embedding

incoming adjacency list of w and thus contradict the common upward embedding. □

Theorem 5.2. *Algorithm 5.2 constructs a valid radial level planar embedding of the given upward embedding $\mathcal{E}_u$ in $\mathcal{O}(|V|)$ time.*

Proof. The running time of $\mathcal{O}(|V|)$ is obvious because the algorithm performs DFS only with different parts of the graph one after the other. To see the correctness, the algorithm starts at t and first traverses the trunk. This is the same mechanism as JLM use in Algorithm 3.6 for computing a level planar embedding from an st-embedding. A *branch* is a subgraph that is traversed with a single invocation of DFS. Since each branch is level planar and meets the requirements of Lemma 5.17, it has a unique level embedding. The side of the trunk on which the branches are placed, left or right, is predefined by the direction of the cut edges that led to them. Thus vertices of the left branches are stored at the front of their ordered level lists and vertices of the right branches are stored at the back of their ordered level lists. This is done recursively for each branch by the same algorithm as JLM use. Here the only difference is that incoming edges on a vertex in a left branch are traversed from right to left. For vertices in a right branch they are traversed as usual from left to right. It only remains to show that the branches are processed in the correct order. This is ensured by processing the cut edges from the outside inwards. Their source vertices and thus their branches are traversed on the left side of the trunk from right to left, on the right side of the trunk from left to right, and in each case from bottom to top. □

Example 5.1. *Figure 5.40 shows $\mathcal{E}_l$ computed by Algorithm 5.2 from $\mathcal{E}_u$ shown in Figure 5.35(g).*

5.5 Assigning Coordinates

By the computation of x-coordinates for the vertices a finalisation of the graph drawing is determined. The previous steps only define the order of the vertices on

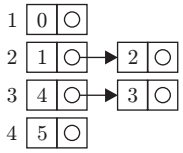


Figure 5.40. $\mathcal{E}_l$ computed from $\mathcal{E}_u$ in Figure 5.35(g)

each level. The y-coordinates are fixed by the given levelling of the graph. Now the vertices are fixed in the plane. This is similar to the fourth phase of the Sugiyama algorithm described in Section 1.2.

5.5.1 Radial Drawing

First we show that each radial level planar embedding has a corresponding radial level planar drawing. In a level drawing with horizontal levels let z be the maximum x-coordinate and k be the maximum y-coordinate. The origin of the used coordinate system is the upper left corner, $(z, 0)$ is the upper right corner, and $(0, k)$ is the lower left corner.

Every point of a straight-line edge e from (x_1, y_1) to (x_2, y_2) can be described with Equation 5.1 where $0 \leq t \leq 1$, $1 \leq x_1 \leq z$, $1 \leq x_2 \leq z$, and $1 \leq y_1 < y_2 \leq k$. See Figure 5.41 for an illustration.

$$\psi(t) = (1 - t)(x_1, y_1) + t(x_2, y_2) \tag{5.1}$$

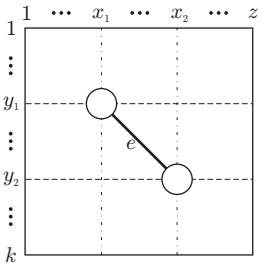


Figure 5.41. Geometrical description of an edge in the plane

The vertices of a level drawing with coordinates (x, y) can be transferred into a radial drawing by Equation 5.2 where $1 \leq x \leq z$ and $1 \leq y \leq k$.

$$\omega(x, y) = (y \cdot \cos \frac{2\pi x}{z}, y \cdot \sin \frac{2\pi x}{z}) \tag{5.2}$$

Usually we draw edges in a radial drawing as spiral segments, e. g., see Figures 5.1(b), 5.6, and 5.9. This has the advantage over straight lines that they never cross the lower radial level lines indicated by the dashed circles. Further, it ensures

that edges are always drawn monotonic from the centre outwards and that an outward drawing is created. Every point of such a spiral segment in a radial level planar drawing can be described by Equation 5.3.

$$\begin{aligned} \psi'(t) &= \omega(\psi(t)) = \\ &= \left[((1-t)y_1 + ty_2) \cdot \cos\left(\frac{2\pi((1-t)x_1 + tx_2}{z}\right),\right. \\ &\quad \left.((1-t)y_1 + ty_2) \cdot \sin\left(\frac{2\pi((1-t)x_1 + tx_2}{z}\right)\right] \end{aligned} \tag{5.3}$$

Observation 5.2. *Since both Equations 5.1 and 5.2 are bijective on the restricted ranges of their parameters, the composition of them shown in Equation 5.3 is bijective, too.*

Theorem 5.3. *For every radial level planar graph there exists a radial level planar drawing.*

Proof. With the algorithm described in this chapter we obtain a radial level planar embedding of the graph. For this proof we split each edge into proper edges with the insertion of up to $\mathcal{O}(|V|^2)$ dummy vertices. This is done before the radial embedding algorithm in order to determine where long edges are routed between the vertices of a level. Further, this detects the proper segment of a cut edge which crosses the ray. Hence, for the rest of this proof we assume w. l. o. g. that the radial level planar embedding is proper.

To simplify matters first, we assume that there are no cut edges in the embedding and thus it is a level planar embedding. From this we produce a planar drawing with the $\mathcal{O}(|V|)$ time[3] algorithm of Brandes and Köpf [20]. In the resulting drawing every originally long edge has at most two bends and the segments are drawn straight-line. Remember that every proper level planar graph has a straight-line drawing according to Section 3.5 and [51, 53]. After all vertices are moved to their radial levels by the coordinate transformation with Equation 5.2, the edge segments are transformed with Equation 5.3 to segments of a spiral. As a result each radial edge has at most two bends. Because of Observation 5.2, the drawing is planar.

In the general case we have cut edges in our radial embedding, see Figure 5.42(a) where the dotted line is a cut edge. We have to simulate each cut edge as shown in Figure 5.42(c) with two segments to avoid crossings. This picture is transformed as previously described to a radial drawing, where these two segments are melted into one again. We use the trick of doubling the drawing as shown in Figure 5.42(b) for determining the gradient of the segments of the cut edges. This is necessary to determine the parameters of Equation 5.3. □

[3] On non-proper level graphs the algorithm needs $\mathcal{O}(|V|^2)$ time.

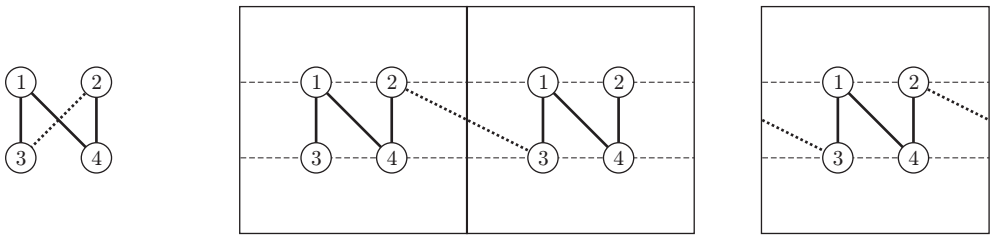


Figure 5.42. Drawing (segments of) cut edges

5.5.2 Drawing Algorithm

Of course we like to obtain nice drawings which follow certain aesthetic criteria and not only arbitrary planar drawings. If we use the method of [20] as suggested by the proof of Theorem 5.3 we have automatically optimised the following aesthetic criteria because the horizontal coordinate assignment algorithm ensures this on the fly and our radial transformation is bijective according to Observation 5.2.

- Edges should be short.
- Vertex positioning should be balanced between upper and lower neighbours.
- Long edges should be as straight as possible.

This $\mathcal{O}(|V|^2)$ time algorithm ensures that every cut edge has at most four bends, see Figure 5.43, because all dummy vertices are drawn among each other except both dummy end vertices of the cut-segment which is inserted at a later stage of the algorithm. Every other edge has at most two bends, which is especially useful in radial drawings. Here, bends tend to be even more confusing than in drawings with horizontal levels since they are connecting segments of a spiral and not straight lines.

If one likes as few cut edges as possible, because they can have up to four bends, there is the possibility of rotation. That means searching for the ray which crosses the fewest edges. Therefore one can use BFS on the dual graph, cf. Definition 5.7, to find the shortest graph theoretic way from the centre to the outer face. Afterwards, the graph is unfolded into a level planar graph by hiding the cut edges.

Definition 5.7 (Dual graph). *A* dual graph *of a planar graph G is a graph with a vertex for each face in a planar embedding of G and an edge for each pair of adjacent faces. The new edges cross exactly the edges of G which are the boundaries between the adjacent faces in the embedding of G. A dual of a planar graph is also planar. The original graph is the dual of the dual graph. That is, they are duals of each other.*

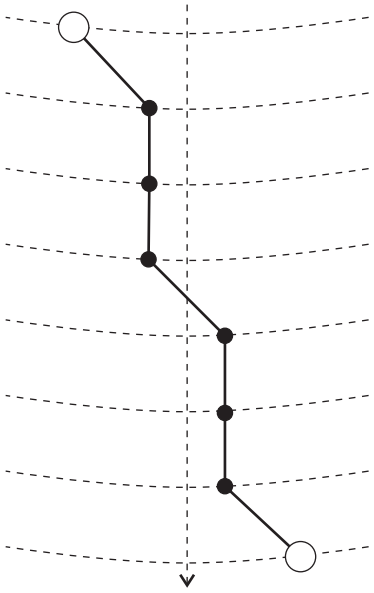


Figure 5.43. A cut edge can be drawn with at most four bends. The black vertices are dummy vertices

Algorithm 5.3 shows an outline of how a radial drawing can be generated out of a radial planar level embedding. It is being implemented in the context of the diploma thesis [65] as a plug-in of the Graph Visualization Toolkit (Gravisto) [75].

Algorithm 5.3. DRAW-RADIAL

Input: A proper level graph G and a radial planar level embedding $\mathcal{E}_l$ of G
Output: A radial level planar drawing

compute the dual graph G_D of G according to $\mathcal{E}_l$
search a shortest path P from the centre to the outer face in G_D with BFS
rotate $\mathcal{E}_l$ such that edges of G crossed by P are cut edges

hide cut edges
apply horizontal x-coordinate assignment algorithm
determine gradients of cut edges
insert cut edges

transform into a radial drawing
remove dummy vertices
optionally beautify bends with splines

For the x-coordinate assignment algorithm it is only necessary that the curves for the edges can be described with a bijective function. Besides of this, any other algorithm, e. g., [22, 23], [131, 132, 134], or the ones referenced in [38, Section 9.3], can be used to optimise any aesthetics criteria other than view edge bends. In any case there is always the option of smoothing bends with splines.

Since the circumference of level j with $1 < j \leq k$ is larger than its predecessor level $j-1$, there is space for more vertices on it. For a radial drawing to not become more and more sparse by traversing the levels outwards, the level graph should have the following property: For every level j there is a growth factor g_j with $V^j = g_j V^{j-1}$ which is approximately the same as the growth factor of the respective circumference. Equation 5.4 shows g_j for drawings with a constant distance d between the radial level lines.

$$g_j \approx \frac{2\pi jd}{2\pi(j-1)d} = \frac{j}{j-1} \tag{5.4}$$

5.5.3 Drawing Edges without Bends

Why do we not use the straight-line algorithm of [51, 53] already mentioned in Section 3.5 for generating a level planar drawing without cut edges before performing the radial transformation? There is a simple answer: The insertion of long cut edges with the trick of Figure 5.42 can introduce edge crossings, see Figure 5.44. This is also the deeper reason why long cut edges can get up to four bends and not only two as non-cut edges while using the method of [20] in Algorithm 5.3.

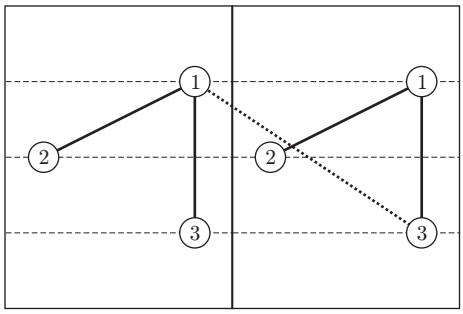


Figure 5.44. A long straight cut edge $(1, 3)$ introduces an edge crossing

Another advantage of the two bend algorithm over the straight-line algorithm is that it only needs linear width whereas the latter one needs up to exponential width in the worst case [53, p. 126], which means up to exponential area. Transforming this into a radial drawing means that the resolution would be very bad, e. g., the number of vertices in certain small areas would be high.

5.5.4 Force Directed Approach

Another possibility for generating nice radial level planar drawings (see [105, Section 4]) is to apply a force directed approach, i. e., the spring embedder method of Fruchtermann and Reingold [68] or the energy based method of Kamada and Kawai [101]. Again, this only applies to proper graphs in order to avoid crossings of long edges. Thus the graph has to be made proper by inserting up to $\mathcal{O}(|V|^2)$ dummy

vertices before the application of the radial embedding algorithm. The dummy vertices determine between which consecutive levels a long edge crosses the ray, i. e., which proper segment of a long edge is the proper cut edge. Further, they fix the routing of long edges between the vertices.

First we generate an initial placement. This can be done for example by the following strategy: The vertices on the first level are distributed equally on the innermost radial level line. Afterwards, all other vertices of the graph are placed level by level outwards by a barycentre or median heuristic. In order to obtain unique positions of the vertices in the radial case, both heuristics consider the angles between the ray and lines from the centre through the adjacent vertices on the inner level instead of their x-coordinates. The ray represents $0°$ and we take the positive angles in counterclockwise direction. We start on a new level always with the first vertex in counterclockwise direction after the ray which has no incident cut edge connecting it to a vertex on an inner level. With barycentre heuristic the angle spanned by such a cut edge is negative. The ordering of the vertices must not be violated. If this is required by the placement heuristic then the vertex is placed with a default minimum distance next to its predecessor in the embedding. This is also the strategy for placing vertices which have no adjacent vertices on a smaller level.

After that, we use a force directed approach [38, 68, 101] for a postprocessing of the vertex placement such that vertices on the same level repel themselves. For this it suffices that repelling forces are only considered between neighbours on a level. Vertices connected by an edge attract each other. Again there is the constraint that the vertex orderings must not be violated and the vertices must remain on their respective radial level line. Remember that the radial level lines are equidistant. For an implementation of the effects of the forces one may operate with polar coordinates and not with Euclidean distances. Due to the fixed radius only the direction, i. e., positive or negative angle in which a vertex is attracted or repelled, and the size of this angle need to be computed.

At the end all edges are drawn as segments of spirals. For this one should pay attention whether the current edge to draw is a cut edge. This determines the direction in which it has to be drawn around the centre. At the end, dummy vertices are removed. However, this approach creates many bends in long edges since there are no strict rules to place dummy vertices of a long edge on the same angle. Therefore it is actually only useful for “nearly proper” graphs.

6 Circle Planarity

In Chapter 5 level planarity has been generalised to radial level planarity. Now we generalise track planarity, see Chapter 4. For this we consider a combination of track planarity and radial level planarity and thus obtain an even larger class of levelled graphs that can be tested on planarity efficiently. The graphs treated here are a superset of planar level and track graphs.

6.1 Definition of Circle Planarity

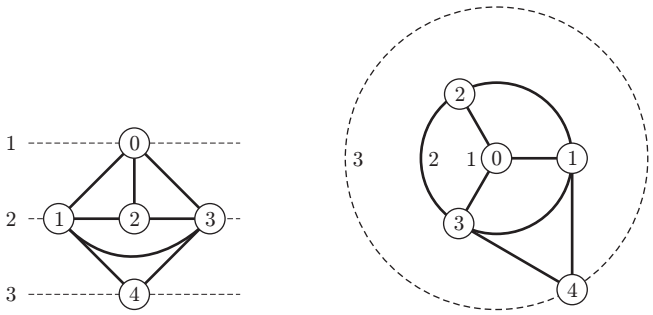
The *k-circle planar graphs* generalise radial level planar graphs in the same way as track planar graphs generalise level planar graphs, i. e., consecutive vertices on a radial level can be connected by a "horizontal" edge. Figure 6.1(b) shows a circle planar drawing of the track graph in Figure 6.1(a) which is neither level planar nor track planar.

Remark 6.1. *For a k-level graph G:*

$$G \text{ is level planar } \Rightarrow G \text{ is circle planar } \Rightarrow G \text{ is planar.}$$

6.2 Testing and Embedding

For the detection and the embedding of k-circle planar graphs we use the same technique as for k-track planar graphs. At the beginning all isolated vertices are removed from the graph G. The number of levels is tripled and the radial level of every vertex v is updated to $\phi'(v) = 3\phi(v) - 1$. Every horizontal chain is extended to a chain of diamonds. Therefore two new vertices, v_e on level $\phi'(v_e) = \phi'(u) - 1$



(a) A 3-track graph

(b) A circle planar drawing of Figure 6.1(a)

Figure 6.1. A circle planar graph and a circle planar drawing of it

and v'_e on level $\phi'(v'_e) = \phi'(u) + 1$ are created for every horizontal edge $e = (u, v)$, $\phi(u) = \phi(v)$. As a replacement for e we introduce four new edges (v_e, u), (v_e, v), (u, v'_e), and (v, v'_e). If in the original graph a horizontal chain closes to a *circle*, i. e., a single level cycle, we call the arising structure in the transformation a *diamond wheel*. For example, Figure 6.3 is generated from the graph shown in Figure 6.1. It contains a diamond wheel on levels 4 to 6. After transforming a k-circle graph $G = (V, E, \phi)$ into the radial $3k$-level graph $G' = (V', E', \phi')$ the linear time radial level planarity test algorithm described in Chapter 5 can be applied to G' in order to detect circle planarity of G according to the following lemma:

Lemma 6.1. *Let G be a radial level graph without isolated vertices and G' be the radial level graph obtained via the transformation of G described above. Then*

$$G \text{ is } k\text{-circle planar} \Leftrightarrow G' \text{ is radial } 3k\text{-level planar.}$$

Since G contains no isolated vertices, the proof of Lemma 6.1 is analogous to the proof of Theorem 4.1. It remains to show how to treat isolated vertices. If a circle occurs on level i, $1 \leq i \leq k$, we have no space left to place isolated vertices of i. Thus the graph under test has to be rejected as circle non-planar. This can be tested within linear time, e. g., in a preprocessing step.

A radial level embedding E'_l of G' can easily be transformed into a circle planar embedding E_l of G, see Section 4.2. But as described in Section 5.1, to compute a radial embedding the algorithm must identify cut edges. Therefore if exactly one of the artificial edges (u, v_e) and (v, v_e) of $\mathcal{E}'_l$ is a cut edge, their original horizontal edge e is a cut edge in $\mathcal{E}_l$, too. Note that if both edges (u, v_e) and (v, v_e) are cut edges, e is not a cut edge, see Figure 6.2. For example, in Figure 6.3 the edge $(9, 1)$ is a cut edge and therefore edge $(3, 1)$ in Figure 6.1 is a cut edge, too. Edge $(4, 1)$ is also a cut edge but an original one.

Theorem 6.1. *There is an $\mathcal{O}(|V|)$ time reduction of circle planarity to radial level planarity.*

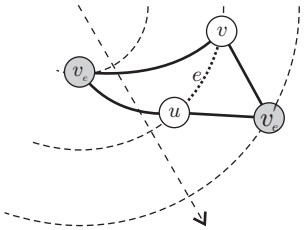


Figure 6.2. Both artificial edges (v_e, u) and (v_e, v) are cut edges but the original circle edge e is not

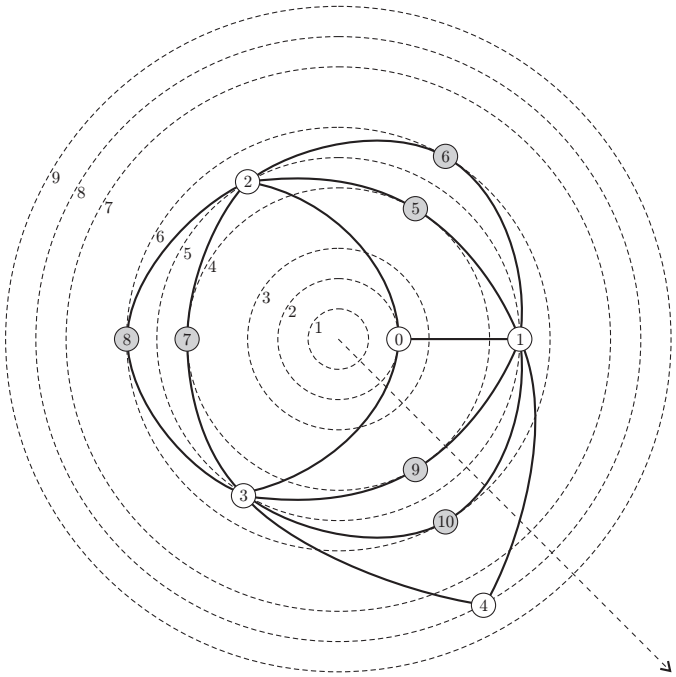


Figure 6.3. The radial 9-level planar graph G' computed from Figure 6.1

6.3 Generating a Drawing

In order to draw a circle planar graph with the help of a horizontal x-coordinate assignment, the procedure is nearly the same as described for radial level planar graphs in Section 5.5. The only difference is that horizontal edges are not considered for the computation of the x-coordinates. These edges can be easily integrated in an existing drawing. They follow the radial level line.

The force directed approach sketched in Section 5.5.4 can also be used to generate a circle planar drawing. Then the optimal length of a horizontal j edge on level j is $l_o^j = \frac{2\pi jd}{|V^j|}$, where d is the constant distance between the radial level lines. If the length of a horizontal edge e in the current drawing is higher than the optimal length, i. e., $l_e^j > l_o^j$, there is an attraction between the two end vertices. Otherwise, $l_e^j < l_o^j$ and there is a repulsion between them. This is handled by choosing the functions of edge attraction and of vertex repulsion such that the attraction of the edge and the repulsion of the vertices level off at l_o^j.

7 Forbidden Subgraphs

There is the famous characterisation of planar graphs by Kuratowski's Theorem 2.2 stating that a graph is not planar if and only if it contains a subgraph homeomorphic to the $K_{3,3}$ or the K_5. Healy, Kuusik, and Leipert [80, 82, 83] have presented the complete set of *minimum level non-planar subgraph patterns (MLNPs)* for proper level non-planar graphs. These are the counterparts of the Kuratowski graphs for level planarity. Minimum means deleting an arbitrary edge leads to level planarity. The steps provided in this chapter are intended to realise a theorem similar to Theorem 2.2 for radial level planarity, i. e., to give a combinatorial characterisation of graphs which are not level planar. This would be helpful to make some proofs easier and to present a small witness to the (radial) level non-planarity of a graph.

7.1 Level Non-Planar Patterns for Hierarchies

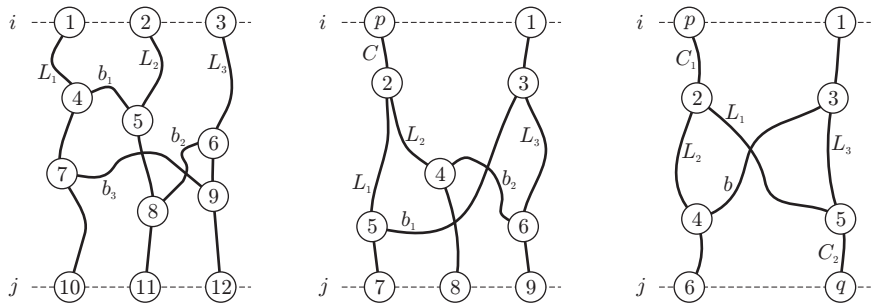
Since [80, 82, 83] are based on the three (not necessarily minimum) level non-planar subgraph patterns (LNP) for hierarchies, we summarise the results of Di Battista and Nardelli [39] in Theorem 7.1. This needs some terminology. Let i and j, $i < j$, be two levels of a level graph $G = (V, E, \phi)$. Then $LACE(i, j)$ denotes the set of paths C connecting any two vertices $x \in V_i$ and $y \in V_j$ such that $z \in C \Rightarrow z \in V_l$ with $i \leq l \leq j$. If C_1 and C_2 are disjoint paths belonging to LACE(i, j) then a *bridge* is a path connecting vertices $x \in C_1$ and $y \in C_2$. Vertices x and y are thus called the *end vertices* of a bridge.

Theorem 7.1 (Di Battista and Nardelli). *Let $G = (V, E, \phi)$ be a hierarchy with $k > 1$ levels. G is level planar if and only if there is no triple $L_1, L_2, L_3 \in$ LACE(i, j), $0 < i < j \leq k$, that satisfies one of the following conditions:*

(a) *L_1, L_2, and L_3 are completely disjoint and pairwise connected by bridges. The bridges do not share a vertex with L_1, L_2, and L_3, except their end vertices. See Figure 7.1(a).*

(b) *L_1 and L_2 share an end vertex p and a possibly empty path C starting from p, $L_1 \cap L_3 = L_2 \cap L_3 = \emptyset$. There is a bridge b_1 between L_1 and L_3 and a bridge b_2 between L_2 and L_3, $b_1 \cap L_2 = b_2 \cap L_1 = \emptyset$. See Figure 7.1(b).*

(c) *L_1 and L_2 share an end vertex p and a possibly empty path C_1 starting from p. L_1 and L_3 share an end vertex q ($q \neq p$) and a path C_2 (possibly empty) starting from q, $C_2 \cap C_1 = \emptyset$. L_2 and L_3 are connected by a bridge b, $b \cap L_1 = \emptyset$. See Figure 7.1(c).*



(a) (b) (c)

Figure 7.1. LNP for hierarchies

7.2 Minimum Level Non-Planar Patterns

Healy et al. [82] have presented a characterisation of MLNPs for arbitrary proper level graphs. The properness is no restriction in the topological sense, since every level graph can be made proper by the introduction of dummy vertices. For the rest of this chapter we assume that every level graph is proper. Healy et al. have divided MLNPs for arbitrary level graphs into three categories: trees, level non-planar cycles, and level planar cycles with incident paths. They use some terms that are common to all patterns: The uppermost and lowermost levels that contain vertices of a pattern P are called *extreme levels* of P. They are not necessarily the extreme levels 1 and k of the graph. If a vertex v lies on an extreme level then it is called the *incident extreme level* and the other extreme level the *opposite extreme level* of v. For the following patterns i and j are the respective extreme levels.

7.2.1 Level Non-Planar Trees

Let x denote a root vertex with degree 3 that is located on one of the levels $i, \ldots, j$. From the root vertex 3 subtrees emerge that have the following common properties:

- Each subtree has at least one vertex on both extreme levels.
- A subtree is either a chain or it has two branches which are chains.
- All the leaf vertices of the subtrees are located on the extreme levels, and if there is a leaf vertex v of a subtree S on an extreme level $l \in \{i, j\}$ then v is the only vertex of S on l.
- Those subtrees which are chains have one or more non-leaf vertices on the extreme level opposite to the level of their leaves.

The location of the root vertex distinguishes two characterisations:

MLNP T1 The root vertex x is on an extreme level, i. e., $\phi(x) = i$ or $\phi(x) = j$.

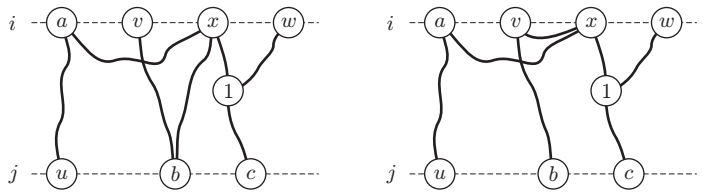


Figure 7.2. MLNP T1

MLNP T2 The root vertex x is on one of the intermediate levels, i. e., $i < \phi(x) < j$. At least one of the subtrees is a chain that starts from x, goes to the level i and finishes on level j. Additionally, at least one of the subtrees is a chain that starts from x, goes to the level j, and finishes on level i.

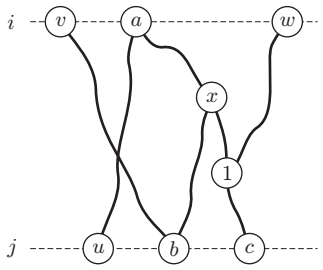
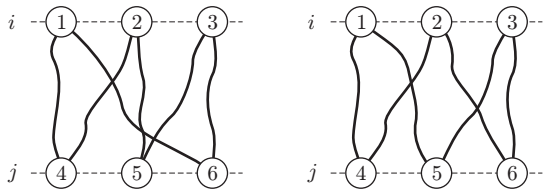


Figure 7.3. MLNP T2

7.2.2 Level Non-Planar Cycles

The next category are cycles that are bounded by the extreme levels i and j of the pattern. A cycle contains at least two distinct paths between the extreme levels having vertices of the extreme level only as their end vertices. These paths are called *pillars*.

MLNP C0 A cycle with more than two distinct pillars.



(a) (b)

Figure 7.4. MLNP C0

7.2.3 Level Planar Cycles with Incident Paths

Level planar cycles can be augmented by a set of chains to obtain minimum level non-planarity. A vertex on a pillar is called an *outer vertex*. The end vertices of pillars are called *corner vertices*. If an extreme level has only one vertex it is called a *single corner vertex*. A bridge in the context of a planar cycle is the shortest path between corner vertices on the same level. A bridge contains two corner vertices as its end vertices and the remainder are inner vertices. A pillar is *monotone* if in a traversal of the cycle the level numbers of subsequent vertices of the pillar are monotonic increasing or decreasing, depending on the direction of traversal. Two paths or chains are *parallel* if they start on the same pillar and end on the same extreme level. If a chain is connected to a cycle by one of its vertices with degree 1 (considering only edges of the chain) then this vertex is called the *start vertex* of the chain and its level, the *start level*. The other vertex of degree 1 of the chain is the *end vertex* and its level, the *end level*.

There are four cases where augmentation of a level planar cycle by paths result in minimum level non-planarity. In all cases the paths start at a vertex on the cycle and end on an extreme level. The bounds of the cycle are the extreme levels i and j.

MLNP C1 A single path p_1 starting from an inner vertex v_{p_1} of a bridge and ending on the opposite extreme level of v_{p_1}; p_1 and the cycle share only the vertex v_{p_1}. The path will have at least one vertex on an extreme level, the end vertex, and at most two, the start and the end vertices.

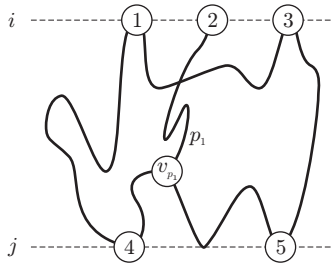


Figure 7.5. MLNP C1

MLNP C2 Two paths p_1 and p_2 starting, respectively, from vertices v_{p_1} and v_{p_2}, $v_{p_1} \neq v_{p_2}$, of the same pillar $L = (v_j, \ldots, v_{p_1}, \ldots, v_{p_2}, \ldots, v_j)$ terminating on extreme levels j and i, respectively. Vertices v_{p_1} and v_{p_2} may be identical to the corner vertices of L ($v_{p_1} = v_i$ or $v_{p_2} = v_j$) only if the corner vertices are not single corner vertices. Paths p_1 and p_2 have no vertices other than their start (if corner) and end vertices on the extreme levels. There are two subcases according to the levels of v_{p_1} and v_{p_2}: $\phi(v_{p_1}) < \phi(v_{p_2})$ or $\phi(v_{p_1}) \geq \phi(v_{p_2})$. The latter means that L is a non-monotonic pillar.

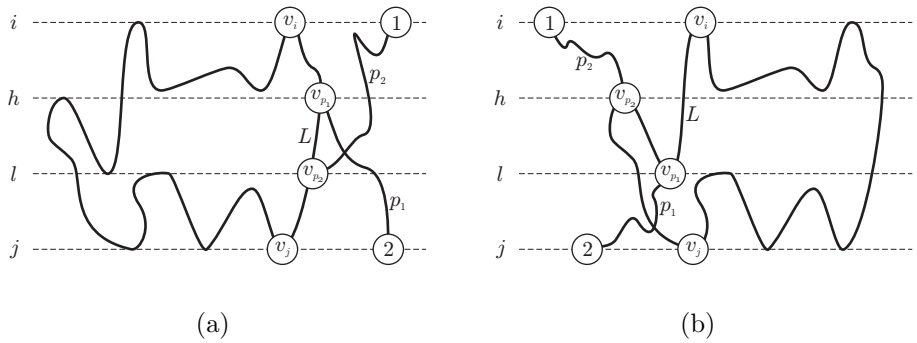


Figure 7.6. MLNP C2

MLNP C3 Three paths, p_1, p_2, and p_3. Path p_1 starts from a single corner vertex c_1 and ends on the opposite extreme level. Paths p_2 and p_3 start from opposite pillars and end on the extreme level of c_1. Neither p_2 nor p_3 can start from a single corner vertex.

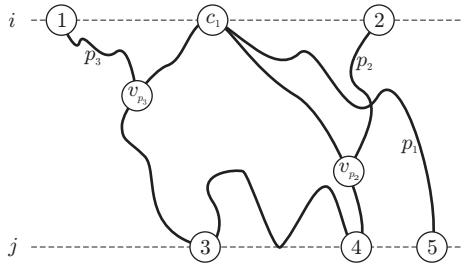


Figure 7.7. MLNP C3

MLNP C4 Four paths, p_1, p_2, p_3, and p_4. The cycle comprises a single corner vertex on each of the extreme levels, c_1 and c_2. Paths p_1 and p_2 start from different corner vertices and end on the opposite extreme level to their start with the paths embedded on either side of the cycle such that they do not intersect. Paths p_3 and p_4 start from distinct non-corner vertices on the same pillar and finish on different extreme levels. The level numbers of the start vertices are such that they do not cause crossing of the last two paths.

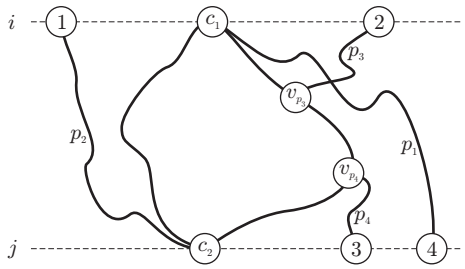


Figure 7.8. MLNP C4

To see why the restriction to proper level graphs is necessary consider for example pattern C1. There the path p_1 needs not to be bounded by the extreme levels of the cycle i and j if long edges were allowed, e. g., vertex 2 in Figure 7.5 could lie on level $i - 1$.

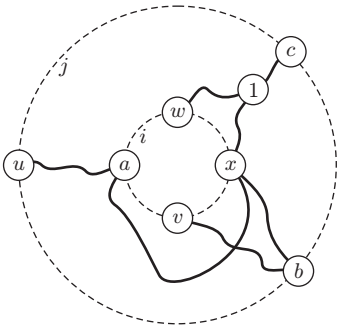
Theorem 7.2 (Healy, Kuusik, and Leipert). *Let $G = (V, E, \phi)$ be a proper level graph with $k > 1$ levels. Then G is not level planar if and only if it contains one of the MLNPs T1, T2, or C0 to C4.*

7.3 Minimum Radial Level Non-Planar Patterns

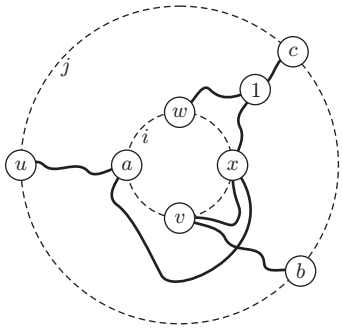
7.3.1 Radial Level Non-Planar Trees

For the radial case we first consider the case that the graph contains no cycle. Hence, here it suffices to consider only connected graphs because a ring can only occur if the graph contains a cycle, cf. Section 5.3.1. Thus there are no rings and analogously

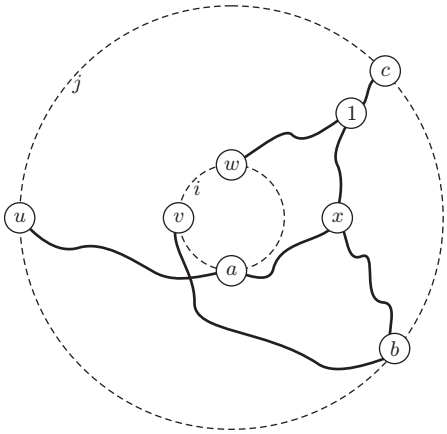
to level planarity the components can be placed side by side. It is easy to see that both MLNP T1 and T2 can not be drawn radial level planar, see Figure 7.9 for an illustration. In the radial context we call them *minimum radial level non-planar subgraph patterns (MRLNPs)* T1 and T2.



(a) A radial drawing of Figure 7.2(a), MRLNP T1



(b) A radial drawing of Figure 7.2(b), MRLNP T1



(c) A radial drawing of Figure 7.3, MRLNP T2

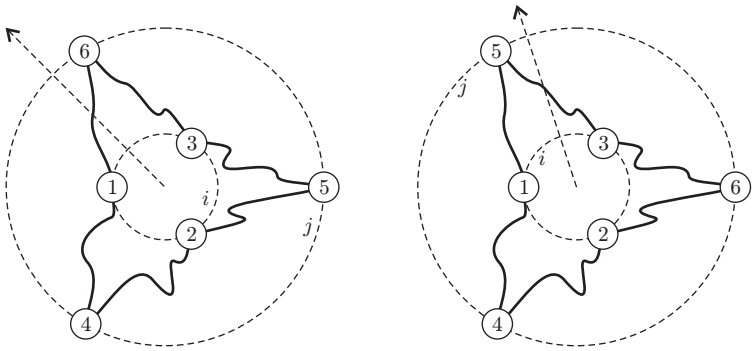
Figure 7.9. Minimum radial level non-planar trees

Lemma 7.1. *Let G be a connected proper level graph without a cycle. Then G is not radial level planar if and only if it contains MRLNP T1 or T2.*

Proof. Theorem 7.2 states that MLNPs T1 and T2 are complete for level planarity without a cycle and that they are minimum patterns. Since radial level planar graphs are a superset of level planar graphs it follows that MRLNPs T1 and T2 are complete for radial level planarity without a cycle. They are minimum patterns because deleting at least one edge results in level planarity, see Theorem 2 of [82], which also means radial level planarity. □

7.3.2 Radial Level Planar Cycles

Next we consider MLNP C0. The cycles are radial level planar because one can take one path between the corner vertices on the same extreme level and route it such that it contains exactly one cut edge.



(a) A radial level planar drawing of Figure 7.10(a)

(b) A radial level planar drawing of Figure 7.10(b)

Figure 7.10. Radial level planar drawings of the pattern C0

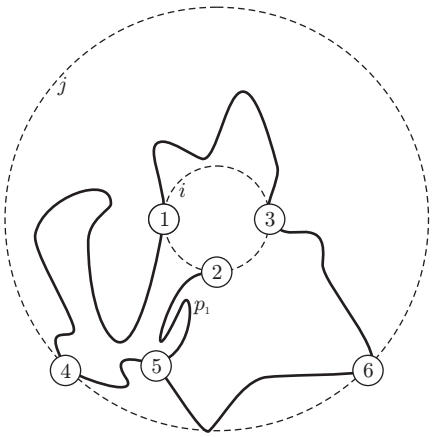
Similar to the pattern C0, the level planar cycles with attached paths, C1 to C4 can also be drawn radial level planar, see Figures 7.11 and 7.12. These five patterns characterise the essential difference between level and radial level planarity. MLNPs C0–C4 can not serve as witnesses to radial level non-planarity.

7.3.3 Radial Level Non-Planar Cycles

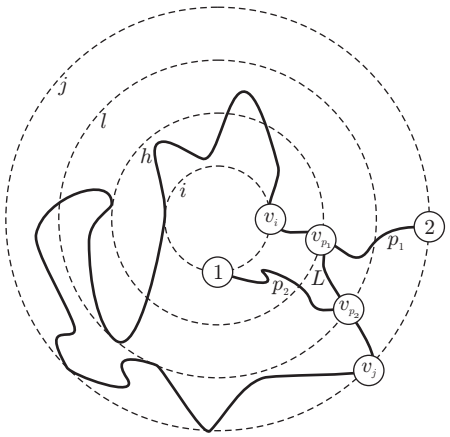
Without loss of generality, we could restrict the input graphs and thus the radial level non-planar subgraph patterns to hierarchies by showing that the patterns for hierarchies and non-hierarchies are the same. This can be done analogously to the proof of Theorem 12 in [82]. The proof must only use the bottom up phase of our radial level embedding algorithm described in Section 5.4, instead of JLM's level embedding algorithm to make a hierarchy out of the input graph. But contrary to the LNPs for level planar hierarchies, there are no known patterns for radial hierarchies. This implies that it is not much easier to treat only hierarchies. Hence, we consider patterns for arbitrary level graphs directly. Again, let i and j be the extreme levels of the patterns with $i < j$.

7.3.3.1 Disjoint Components

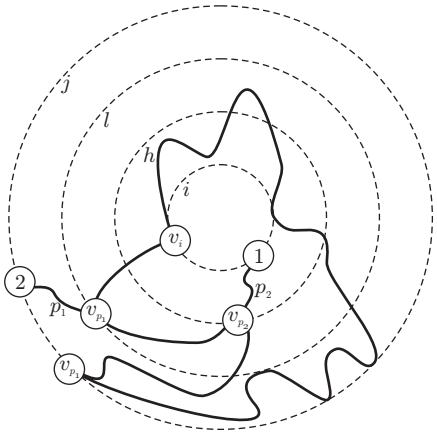
Two non-connected components (or subgraphs of them) between two extreme levels intersect if the nesting of one component in a face of the other component fails. Then at least one component must contain a ring. A ring between the extreme levels i and



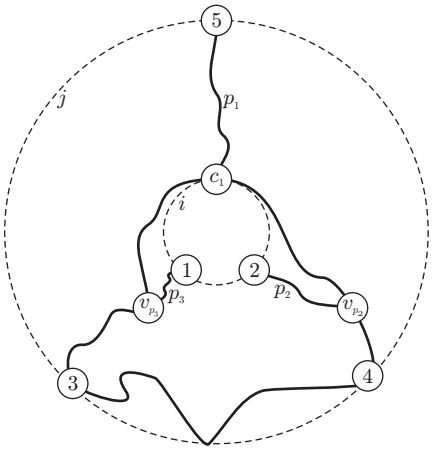
(a) A radial level planar drawing of Figure 7.5



(b) A radial level planar drawing of Figure 7.6(a)



(c) A radial level planar drawing of Figure 7.6(b)



(d) A radial level planar drawing of Figure 7.7

Figure 7.11. Radial level planar drawings of the patterns C1 to C3

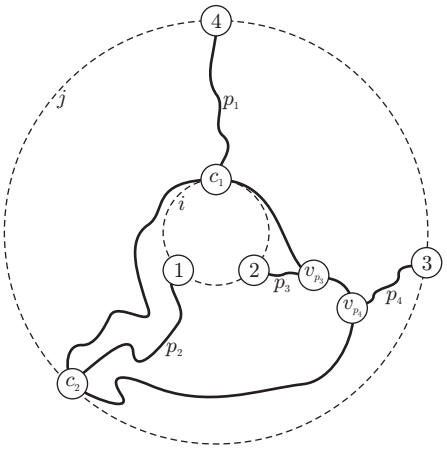


Figure 7.12. A radial level planar drawing of Figure 7.8

j is a subgraph homeomorphic to the $K_{2,2}$ with the extreme vertices $\{c_1, c_2, c_3, c_4\}$ and a levelling $\phi(c_1) = \phi(c_2) = i$ and $\phi(c_3) = \phi(c_4) = j$. Thus it corresponds to a cycle consisting of four paths (pillars) $c_1 \to^* c_3$, $c_1 \to^* c_4$, $c_2 \to^* c_3$, and $c_2 \to^* c_4$. Each of the following RLNPs RC1–RC5 contains such a cycle.

RLNP RC1 A path $p_1 = u \to^* v$ starting on the extreme level in the inner face of the cycle and ending on the opposite extreme level, i. e., $\phi(u) = i$ and $\phi(v) = j$.

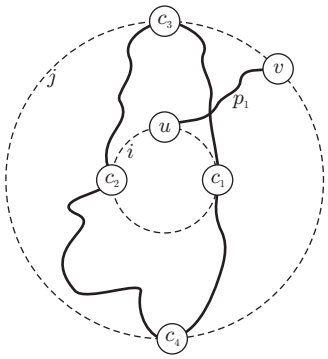


Figure 7.13. RLNP RC1

RLNP RC2 Two paths $p_2 = u \to^* v$ and $p_3 = x \to^* w$ whose vertices are all between or on levels h and j, $i < h < j$; $u \in c_1 \to^* c_3$, $v \in c_1 \to^* c_4$, $w \in c_2 \to^* c_3$, and $x \in c_2 \to^* c_4$. A path $p_1 = y \to^* z$ starting on level $\phi(y) = h$ on the inner face of the cycle $(u, \ldots, v, \ldots, c_1, \ldots, u)$ or of the cycle $(w, \ldots, x, \ldots, c_2, \ldots, w)$ and ending on level $\phi(z) = j$.

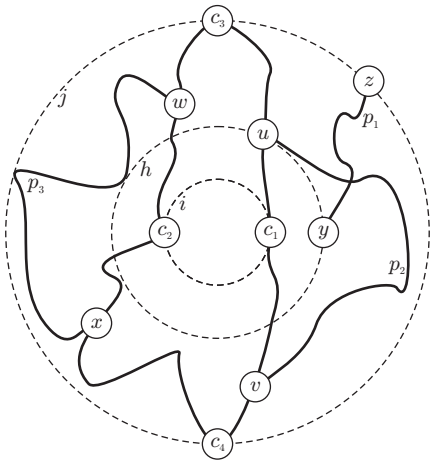


Figure 7.14. RLNP RC2

7.3.3.2 Connected Components

For connected components containing a ring we have identified the following three radial level non-planar patterns:

RLNP RC3 A path $p_1 = u \rightarrow^* v$, where $u \in c_2 \rightarrow^* c_3$ and $v \in c_1 \rightarrow^* c_4$ with $i < \phi(u) < j$ and $i < \phi(v) < j$. If pillar $(c_1, \ldots, w, \ldots, c_3)$ is non-monotonic and $\phi(w) = j$ then u and c_3 may coincide. If pillar $(c_2, \ldots, x, \ldots, c_3)$ is non-monotonic and $\phi(x) = j$ then v and c_4 may coincide.

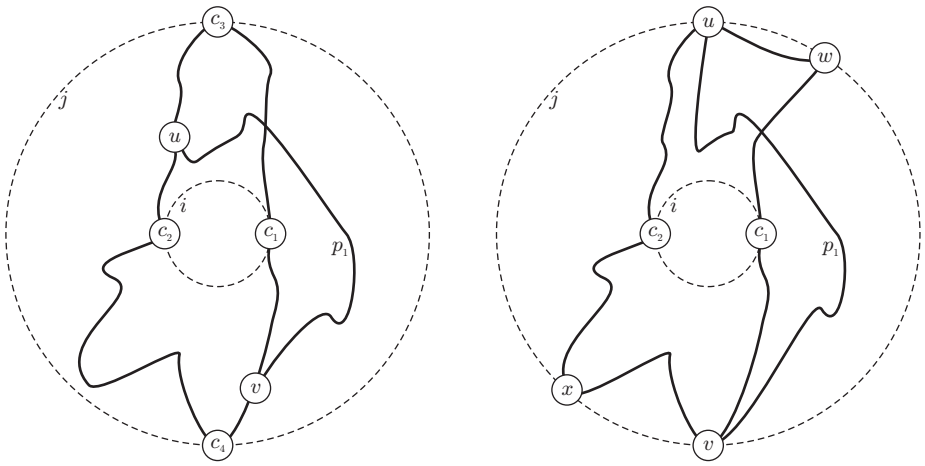


Figure 7.15. RLNP RC3

RLNP RC4 A path $p_1 = (u, \ldots, w, \ldots, v)$, where $u \in c_2 \rightarrow^* c_3$ and $v \in c_1 \rightarrow^* c_3$ with $\phi(w) = j$ and $u, v \neq c_3$.

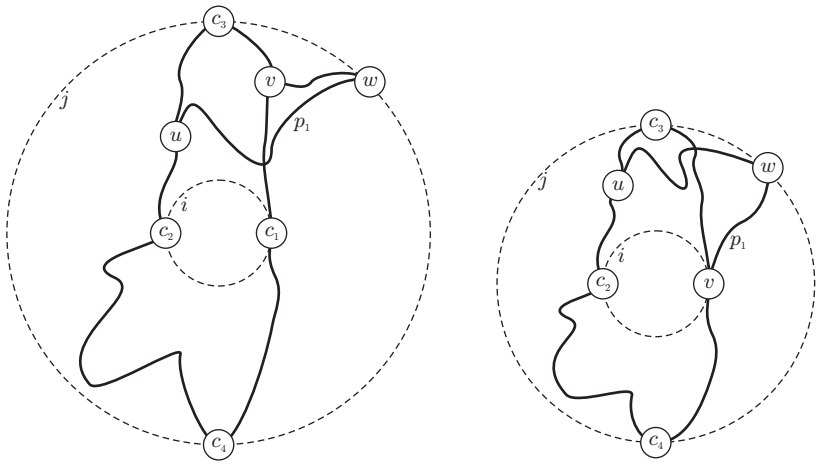


Figure 7.16. RLNP RC4

RLNP RC5 Two disjoint paths $p_1 = w \rightarrow^* x$ and $p_2 = u \rightarrow^* v$, where the path $(u, \ldots, w, \ldots, v)$ is a subpath of $(c_3, \ldots, c_1, \ldots, c_4)$, $u \in c_3 \rightarrow^* c_1$, and $v \in c_1 \rightarrow^* c_4$. The path p_2 is disjoint with $(c_3, \ldots, c_1, \ldots, c_4)$ except its end vertices u and v.

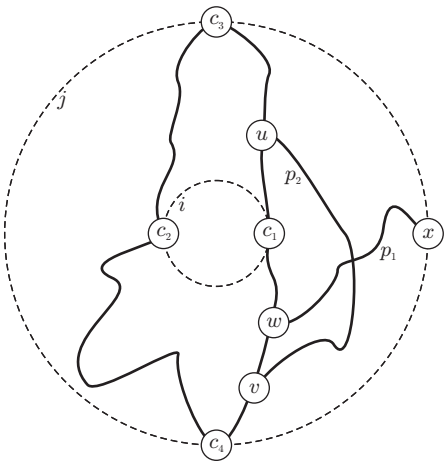


Figure 7.17. RLNP RC5

Theorem 7.3. *Let G be a proper level graph with $k > 1$ levels.*

1. *G is not level planar if it contains a MRLNP T1, T2, or an RLNP RC1 to RC5.*

2. There are MLNPs which are radial level planar.

Theorem 7.3 summarises our observations. Unfortunately, the patterns which are radial level non-planar cycles are not minimum in every case. A complete characterisation for radial level non-planar graphs containing a cycle is left as open.

8 Conclusion

In our opinion there are no good drawing algorithms for planar graphs. For level planar graphs there are some. We have extended the concept of level planarity to obtain a larger class of planar graphs for which good drawing algorithms exist.

8.1 Summary

After an introduction to the well known topics planarity and level planarity we have introduced three new concepts of planarity: track planarity, radial level planarity, and circle planarity. This allows planar drawings of more graphs with vertices partitioned into levels. Figure 8.1 summarises the relationships between the different concepts which directly follow from the definitions and from Remarks 4.1, 5.1, and 6.1.

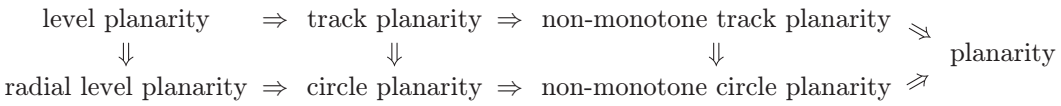


Figure 8.1. Relationships between different concepts

We have presented a linear time reduction from track planarity to level planarity, leading to a linear time algorithm to recognise track planarity. This can be easily integrated into an existing level planarity test algorithm. Clearly, the core algorithms need not be changed because we only extend the input graph with $\mathcal{O}(|V|)$ dummy vertices and $\mathcal{O}(|V|)$ dummy edges in a preprocessing step. Thus the time complexity of the algorithm does not change. Further, we have presented a new algorithm for detecting radial level planarity of k-level graphs in linear time. For this we

have enhanced the PQ-tree data structure of [13] with a new node type, R-nodes, in order to represent ring components of the graph. The obtained data structure is the PQR-tree structure. Our algorithm can also compute a radial level planar embedding within the same linear time bounds as other planar embedding algorithms [26, 95, 97, 99, 112]. Also we have presented a linear time algorithm for the detection of k-circle planar graphs and for computing an embedding for them as a combination of the above.

While working on theoretical subjects we have implemented an object oriented prototype of our algorithm in C++ using the Graph Template Library (GTL) [76] with improved symmetric lists [7], see Appendix B. This algorithm can detect planarity, level planarity, track planarity, radial level planarity, as well as circle planarity. Additionally, it can compute the corresponding embeddings.

8.2 Future Work

An interesting future challenge would be a new method or a extension of the existing methods, e. g., of [20, 22, 23, 51, 53, 131, 132, 134], for x-coordinate assignment which directly works on the radial embedding. Then aesthetic criteria like edges without bends or cut edges with at most two bends like the other edges could be treated more easily. Further, the possibility of rotating a radial level to obtain a better coordinate assignment has not been exploited exhaustively yet. An algorithm like [57] which directly works with long edges without inserting a quadratic number of dummy vertices could speed up this procedure. For the drawing of long edges without, or at least with fewer, dummy vertices it might also be helpful to use the incident edge orderings computed in the embedding phase with Chiba's algorithm.

Further investigations are needed in order to show that all RLNPs of Section 7.3.3 are complete and to find these patterns for track and circle planarity. Additionally, all patterns should be directly characterised for non-proper level graphs. Further, it would be helpful if the patterns can be minimised such that deleting one edge leads to radial, track, or circle planarity, respectively. It is also desirable to efficiently expand the test algorithms for the various kinds of level planarity to detect the forbidden subgraph patterns if the tested graph is not (radial) level, track, or circle planar. This work already has been done for the detection of the Kuratowski graphs $K_{3,3}$ and K_5 in general graph planarity. Both can be efficiently found as [103, 159] or [120, Section 3.2] show. As already mentioned in the conclusion of [112, p. 211] the detection of forbidden subgraph patterns can also be used to verify the results of a (radial) level planarity test. Since such a test is a non-trivial algorithm and thus it is not unlikely that an implementation is faulty, it is desirable to not only prove planarity by an embedding or by a drawing, but also to show non-planarity on the basis of minimum patterns.

Another interesting topic is the generalisation of level graphs to non-monotonic edges while the levels of the vertices remain fixed. How can a graph be tested and embedded efficiently for non-monotone variations of (radial) level planarity? What

has also not been discussed in this thesis and what is unfortunately not addressed very well in literature is planarity or crossing minimisation for recurrent hierarchies as shown in Figure 5.2.

Clearly, level graphs that need to be visualised are not level planar in general. Thus we expect research to continue concentrating on the subject of minimising the number of edge crossings. Since almost all approaches known in literature only attack the problem of 2-level crossing minimisation, studies on more general approaches are desirable in order to obtain a global view on the graph while reducing edge crossings.

A lot of problems arise when using the Sugiyama approach on graphs that do not have a levelling. Then the vertices are assigned to certain levels. Although this is done according to certain requirements, e. g., the level graph should be compact, the levelling has to be proper, or the number of dummy vertices that have to be introduced should be small, this phase is encapsulated and predetermines in some sense the final drawing of the graph. The crossing number is highly dependent on the chosen levelling. Therefore it would be preferable to develop methods that try to combine the phases of the Sugiyama algorithm.

These considerations reveal that there is a large number of open problems related to the topics of this thesis. We hope that the tools and results we have presented will contribute to a deeper understanding in drawing level graphs.

A Improved Symmetric Lists

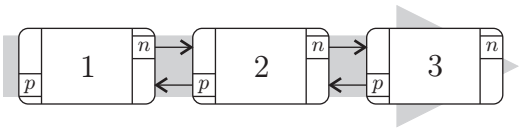
A.1 Motivation

A doubly linked list consists of *cells* each containing a data field storing an *element* of the list and two pointers to its neighbours. In general, the interpretation of these pointers is hard-coded [1, 74, 110] or [120, Section 8.7]. Lists usually rely on the interpretation that the pointer named *prev* refers to the previous cell and the one named *next* to the next cell. Clearly, reversing such a list can either be done in linear time by interchanging the pointers in all cells or in constant time by globally reversing their interpretation. However, then the pointers are interpreted identically for all cells. This prevents inserting a reversed list into another in constant time, since then the pointers of the cells of the inserted list and of the original list have different meanings.

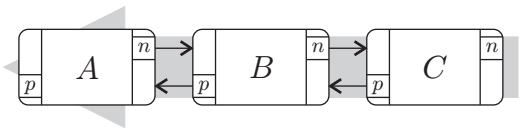
Example A.1. *Consider the two lists in Figure A.1(a) and Figure A.1(b). As the arrows in the background indicate, the list in Figure A.1(b) has been reversed, i. e., the pointers in cells* 1*,* 2*, and* 3 *on the one hand and cells A, B, and C on the other are interpreted differently. If the list in Figure A.1(b) is inserted in the list in Figure A.1(a) between cells* 1 *and* 2 *(in constant time), we obtain Figure A.1(c). Obviously, the adjacency pointers do not have the same meaning in all cells in this list. Of course, this can be fixed by interchanging the pointers in each cell of the inserted list, but this takes linear time.*

A.1.1 Concept

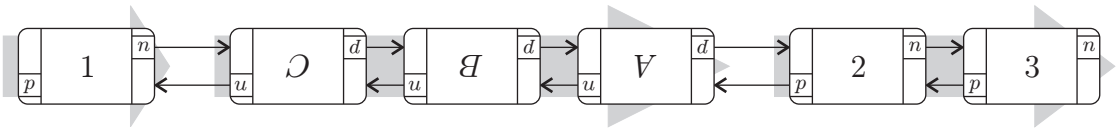
Following an idea of Tarjan [151], our data structure *symlist* solves this problem by ignoring the common direction information of the pointers in a cell. Symlists only



(a) A non-reversed list, i. e., n refers to the next and p to the previous cell



(b) A list which was reversed by switching the interpretation of n and p



(c) The result of inserting the list in Figure A.1(b) into the one in Figure A.1(a). Not all pointers in this list have the same meaning, e. g., n in cell 1 refers to the next cell, but n in cell B refers to the previous one

Figure A.1. Example of the problem of inserting a reversed list into a non-reversed one in constant time with fixed interpretation of the pointers in the cells. In each cell n and p denote pointer *next* and *prev* respectively

preserve the invariant that the two pointers refer to the two adjacent cells without specifying directly or indirectly which one is the previous and which is the next. Although this is not common in data structures, cf. [1, 74, 110, 120], it makes sense from a software engineering perspective because the directional information is not a feature of the cell. It is needed only for traversing a list.

Clearly, a traversal is more complicated for a symlist than for an ordinary doubly linked list. In our implementation we use iterators that store the directional information along with the cell they refer to. Whenever the iterator is moved in either direction both its position and its direction must be updated (see Section A.2 for a detailed description). Although the basic idea of symlists is already described in [151], an anomaly which a straightforward implementation of this idea will have is not mentioned, see Section A.3.1 for details.

A symlist is related to the *quad-edge data structure* [77] that is used for efficiently representing embeddings of graphs in two-dimensional manifolds. It consists of quad-edges and each vertex contains four references to adjacent quad-edges. For iterating through this data structure, two flags are stored along with the current quad-edge. These are used to determine the next quad-edge in the iteration. Depending on the value of these flags, either the graph, its dual, or its mirror image are traversed. If we use two of the four pointers of a quad-edge and one of its flags only, iterating through this data structure is similar to the iteration through a symlist. However, the quad-edge data structure was designed for that particular purpose only and is not meant to be generic. It is used for iterating through the represented graph and does not support reversals and insertions.

Sleator and Tarjan [144] describe the efficient management of *dynamic paths* via *biased trees*. There every internal vertex v of the tree has a flag $reversed(v)$. The *reversal state* of v is an exclusive or of the *reversed flags* from v to the root. This indicates whether the path segment descendant to v is reversed and can be computed within logarithmic time. We touch on this work because a list can be seen as a path containing data information in its vertices. Of course, an insertion of a new element in the path, an operation not mentioned in [144], has to update this tree structure, which is not possible in constant time.

A.1.2 Applications

Symlists can be used for any doubly linked list; in particular, when in addition to the standard operations, both reversing the list and inserting a list into another has to be done in constant time.

Within the PQ-tree data structure, cf. Section 2.2, we have these requirements for the lists storing the children of an internal PQ-node. The most important operation on PQ-trees is REDUCE. This operation may reverse the order of the children of some Q-nodes and insert the children of a Q-node somewhere in between the children of another Q-node. In order to achieve the complexity of REDUCE proved in [13], both reversing a list of children and inserting a list of children into another must be done in constant time. Remember, this complexity of REDUCE is crucial for all linear time planarity testing and embedding algorithms described in this thesis. In some implementations of PQ-trees no separate list data structure is used for storing the children of a Q-node. The references to the adjacent siblings are stored in each child [13, 99, 111]. In these implementations the adjacency pointers are treated as an unordered set. Hence, they have no fixed meaning either and reversing and inserting a reversed list can be done in constant time. However, from a software engineering perspective it is better to have a separate and reusable data structure with a well defined interface that provides appropriate methods for accessing and manipulating the children of a Q-node. There is another linear time planarity test of Boyer and Myrvold which does not use PQ-trees but also needs an efficient data structure for flipping biconnected components, see [16, Section 2]. They call this data structure a *doubly linked cycle with no sense of clockwise*. This is exactly the paradigm of symlists.

Common implementations of doubly linked lists encode the direction of the list into their cell by using pointers with a uniform meaning for the whole list [1, 74, 110] or [120, Section 8.7]. As illustrated by Example A.1, these implementations cannot provide constant time methods for both reversing a list and inserting a reversed list into another. Hence, they cannot be used in a PQ-tree within the desired time bounds. Our data structure symlist fills this gap. As shown in Table A.1, the time complexity of all methods on lists are identical except for reversing a list, which can be done in constant time on a symlist.

Table A.1. Typical operations on lists and iterators and their time complexity

	Operation	symlist	ordinary list
List	`insert`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
	`splice`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
	`erase`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
	`reverse`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
	`size`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	`empty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Iterator	`++`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
	`-`	$\mathcal{O}(1)$	$\mathcal{O}(1)$

A.2 Implementation

Our implementation is based on the interface of the Standard Template Library (STL) class `list` [122, 141] containing methods like `empty`, `insert`, `splice`, `erase`, and `reverse`, see Figure A.2. Furthermore, the interface is enriched with the new methods `attach_sublist`, `detach_sublist`, `blind_insert`, and `blind_erase`, see Section A.3.2.

A symlist consists of cells containing two adjacency pointers, $p\,[0]$ and $p\,[1]$. As suggested in [110] and used in STL, our implementation is a cyclic list with an additional cell between the first and the last cell, see Figure A.3. It is called the *end cell* because this is the cell to which the end iterator `symlist::end()` refers.

The end cell is stored in our list class along with a flag $dir \in \{0, 1\}$ indicating which of the two pointers refers to the first cell of the list. In addition to its cell object *cell*, each iterator stores its current direction as a flag $dir \in \{0, 1\}$ indicating that $p[dir]$ in *cell* leads to the next cell.

Algorithm A.1 explains the effect of the ++-operator (advancing to the next cell) applied to an arbitrary iterator *it* in detail. The −−-operator is defined analogously.

Insertion of a new cell z before an arbitrary iterator *it* referring to cell x works as follows: With $--it$ we determine the previous cell y of x in the sense of the iterator *it*'s local direction. After updating the appropriate pointers of x, y, and z, a new iterator pointing to z and having the same direction as *it* is returned.

For an iteration we need begin and end iterators. As already mentioned, the end iterator points to the end cell and its direction flag is set to the direction flag of the list. In order to get a begin iterator, i. e., `symlist::begin()`, we internally call ++ on the end iterator, which gives an iterator pointing to the first element in the list and heading in the direction of the list.

Now reversing a symlist can be done in constant time by flipping the direction flag of the list. This avoids the problems illustrated in Example A.1 and avoids using linear time for reversing. Furthermore, an iterator can easily be reversed by flipping its direction flag. As in STL, all existing iterators, except the ones referring to erased

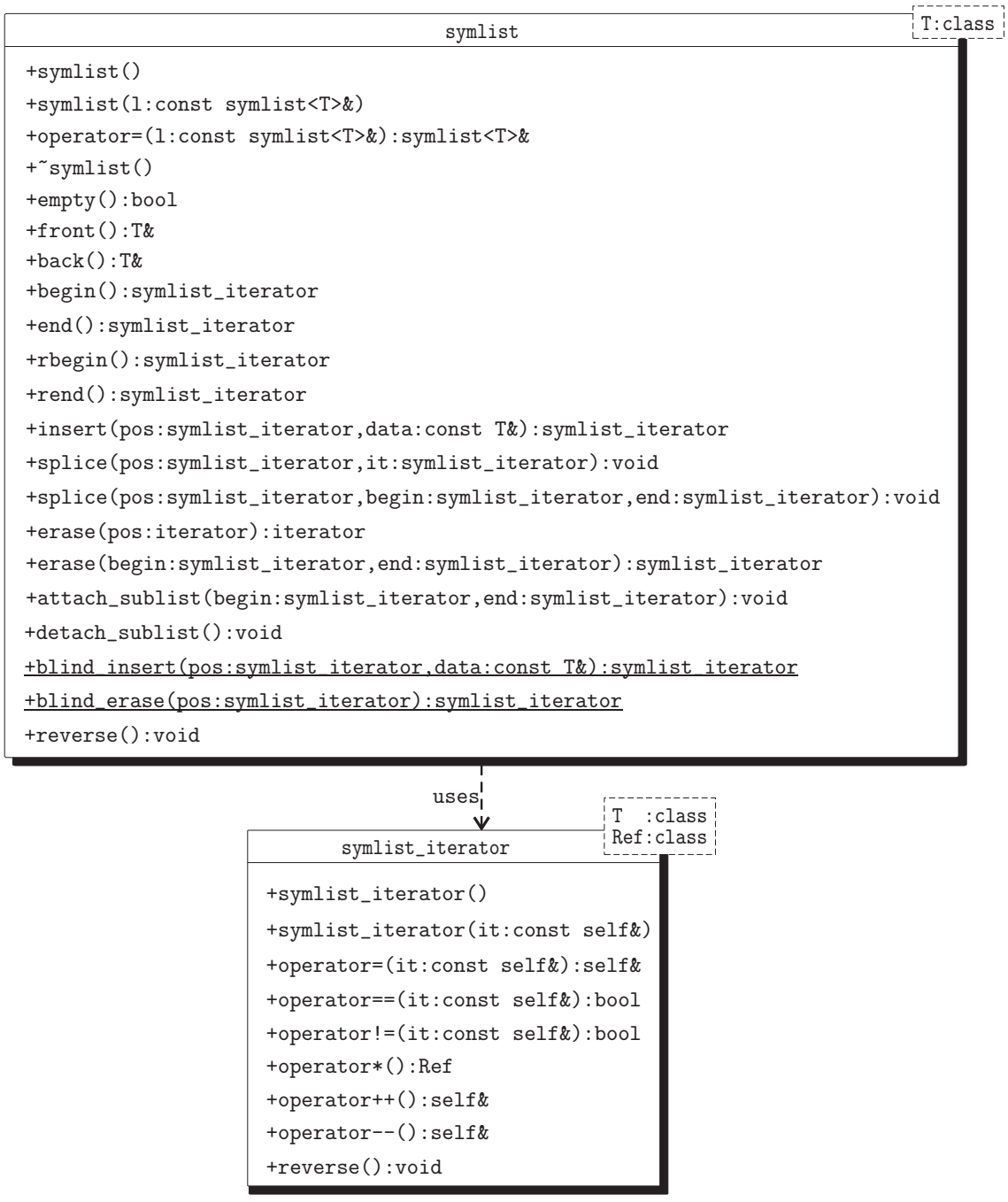


Figure A.2. The interface of our symlist implementation in UML notation

Algorithm A.1. operator++

Input: Nothing
Output: An iterator to the next cell in this iterator's direction

```
tmp ← it.cell                          // make a copy of the current cell
it.cell ← tmp.p[it.dir]                // move to the next cell

if it.cell.p[0] = tmp then
    it.dir ← 1                         // p[0] points back, hence new dir is 1
else
    it.dir ← 0                         // p[1] points back, hence new dir is 0
end

return it
```

cells, remain valid and consistent to their possibly new adjacency throughout the lifetime of a symlist and all update operations on it. This also holds if the blind operations from Section A.3.2 are used.

A.3 Extensions

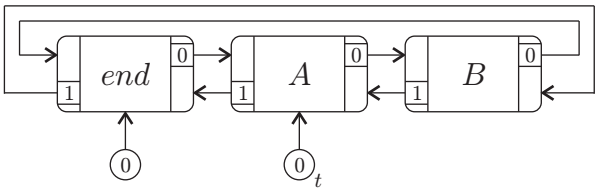
A.3.1 Losing Information

In the basic implementation as outlined above, the information stored in an iterator together with the direction flag of the list may be insufficient. This is clarified by the following example.

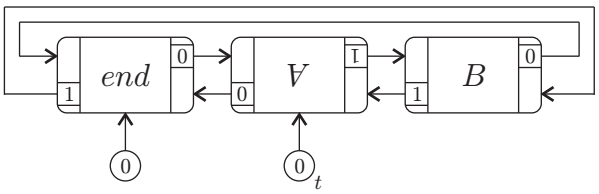
Example A.2. *Consider the symlists in Figure A.3(a) and Figure A.3(b). These are almost identical. The only difference is that* $p[0]$ *and* $p[1]$ *in cell A are switched. Therefore the iterator t in Figure A.3(a) has the same direction as the end iterator, whereas t in Figure A.3(b) has the opposite direction. Hence, inserting a new cell before t in Figure A.3(a) results in a new cell between end and A. On the other hand, in Figure A.3(b) this insertion results in a new cell between A and B.*

If we delete cell B in both Figures A.3(a) and A.3(b), we get the configurations shown by Figures A.3(c) and A.3(d), respectively. Clearly, inserting a new cell before t in Figure A.3(c) should give a list beginning with the new cell, and inserting it in Figure A.3(d) should give a list with the new cell at the end. But the two configurations in Figures A.3(c) and A.3(d) are indistinguishable, i. e., both pointers in each cell refer to the other cell, the direction flag of the list is 0*, and t has direction* 0*. Thus it is impossible to determine what "insertion before t" really means. Both positions could be correct.*

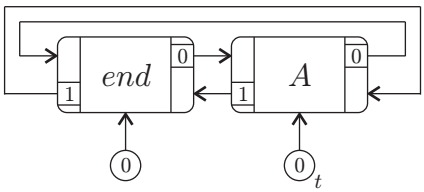
Obviously, this problem occurs only on singleton lists as shown in Figure A.3(c). Through the iterator pointing to the only data cell and its direction flag, it is known which pointer of this cell is affected by an insertion. But it is impossible to determine



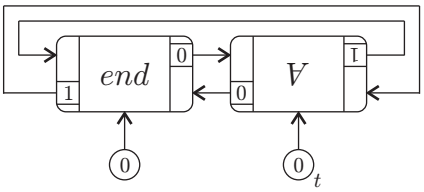
(a) Here inserting before t results in a new cell between *end* and A



(b) In this configuration inserting before t results in a new cell between A and B



(c) Result of deleting cell B in Figure A.3(a). Inserting the new cell should behave as in Figure A.3(a), i. e., should result in a list with the new cell at the beginning



(d) The result of deleting cell B in Figure A.3(b). This is identical to Figure A.3(c), but inserting a new cell should behave as in Figure A.3(b), i. e., with the new cell at the end

Figure A.3. Example where the information stored in an iterator is not sufficient to determine where to insert a new cell. The 0 and 1 in the cells denote pointers $p[0]$ and $p[1]$, respectively. The circles pointing to a cell are iterators. The number in an iterator is its direction flag. The iterator pointing to the end cell is the end iterator, i. e., its direction flag is the direction of the list

which pointer in the end cell needs to be updated, cf. Figures A.3(c) and A.3(d). If the direction is encoded into the cells using pointers *prev* and *next*, this cannot occur because we know that the corresponding pointer for *prev* in the data cell is *next* in the end cell, for instance.

This problem can and must be resolved to prevent errors. As a particular application consider for example level planarity testing, which uses PQ-trees, cf. Section 3. The children of all PQ-nodes are stored in symlists, and in the template matching algorithm it can happen that a Q-node temporarily has only one child.

Since this anomaly occurs only in singleton lists, we can simply keep a *hidden cell* to avoid it. In our implementation this additional cell is always a neighbour of the end cell. This enforces some modifications in the methods of class `symlist` to hide the existence of the additional cell. Beside the insert and erase methods, the ++-operation and the −−-operation on iterators must be changed to ignore the invisible cell. To achieve this efficiently, our extension is to mark each cell according to its type (data, hidden, or end) when it is created.

Exactly the same problem occurs with Tarjan's reversible lists, as described in [151], but on lists of size two and not on singleton lists. This is due to the fact that there is no extra end cell and the global direction of the list is stored within the last data cell, the so-called *tail*. Moreover, Tarjan does not describe what previous or next means to an arbitrary external pointer on a cell. In a symlist every iterator has a local direction, for a simple pointer this is impossible. But for some applications this feature is very useful, e. g., for (level) planarity testing.

A.3.2 Blind Operations

Besides the standard methods on doubly linked lists, we need the more specialised methods `attach_sublist` and `detach_sublist` for maintaining the children of a Q-node. If a planarity testing algorithm uses PQ-trees, sometimes a pseudo Q-node, cf. Section 2.3, must be created which temporarily contains a sequence of children out of the middle of another Q-node. Remember, this parent Q-node may not be known at the moment due to time complexity restrictions. Hence, because only the iterators to the relevant children are known, we need a constant time operation `attach_sublist` which sets the adjacency pointers of that pseudo list's end cell e to the first and the last data cell, w and v, of the sublist. The adjacencies of w and v are adapted accordingly. Moreover, pointers to the previous and next cells from that sequence in the original list are stored in the pseudo list in order to know where the sublist must be inserted in the original list by `detach_sublist` later on in the algorithm.

Another useful feature is the ability to insert or erase an element of a symlist if its position is given by an external iterator, but not the symlist object itself. Therefore, we introduce two static methods `blind_insert` and `blind_erase`, both running in constant time. The use of such methods prohibits maintaining a size counter as a data member of `symlist` which is updated with each operation affecting the size.

B
Implementation

Figure B.1 shows a suitable UML diagram for implementing an algorithm which is capable of performing the various tests shown in this thesis: level planarity, track planarity, radial level planarity, and circle planarity. In each case it can also compute an embedding if the graph has the demanded planarity property. For example, our prototypic implementation is derived from this model, except for the fact that the functionalities of the classes `r_node` and `q_node` are combined into a single class. This prototype is intended to be a feasibility study which is written in the C++ programming language on the basis of the GTL graph data structure [76].

The presented UML model is an 8-circle planar graph which is also 8-track planar if one edge is deleted, e. g., the `uses` arrow from the `level_planarity` class to the `planarity` class.

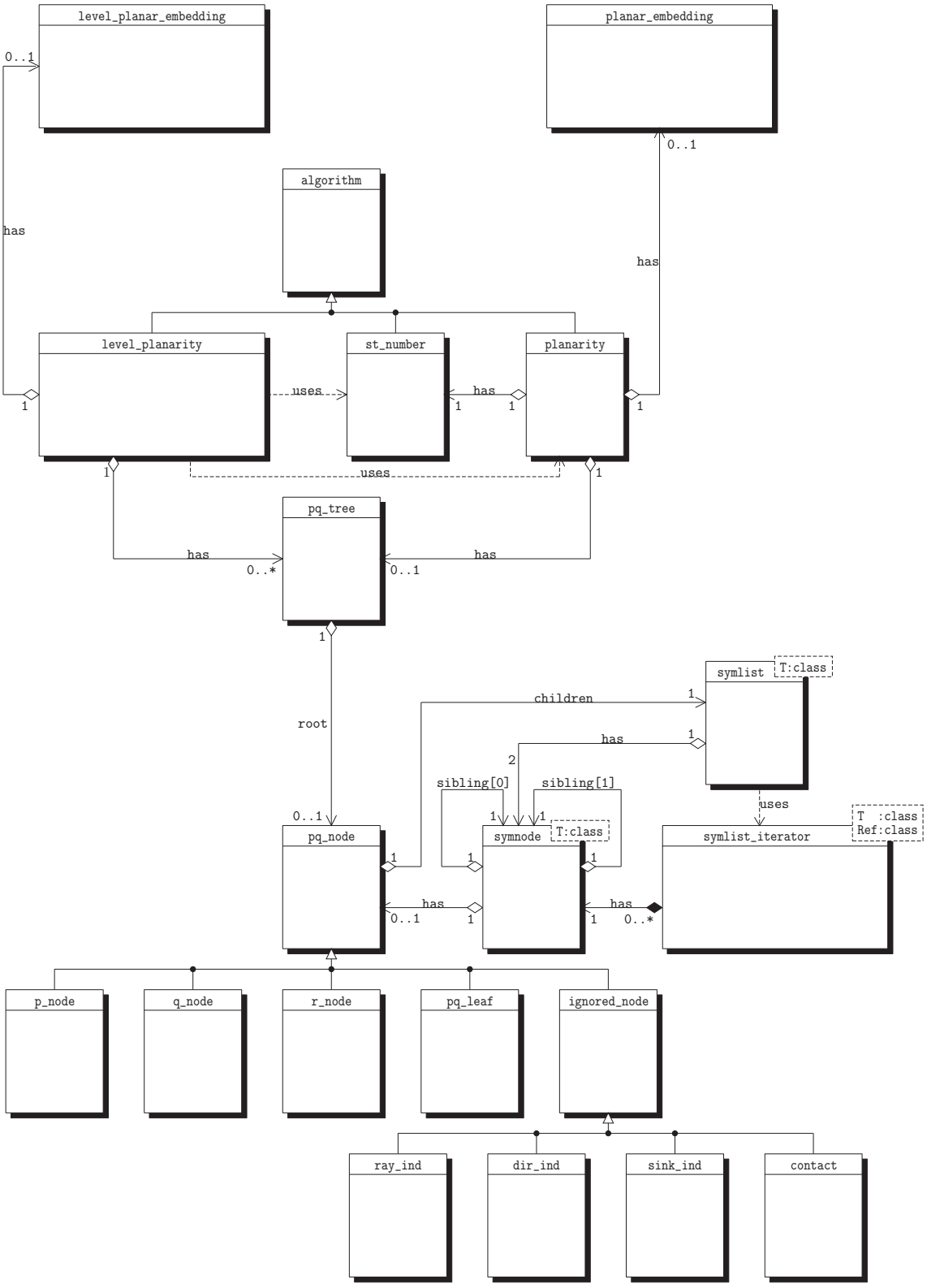


Figure B.1. A suitable UML model

List of Figures

List of Definitions

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*, pages 204–205. Addison Wesley, 1983.

[2] L. Auslander and P. S. V. On imbedding graphs in the plane. *Journal of Mathematics and Mechanics*, 10(3):517–523, 1961.

[3] C. Bachmaier, F. J. Brandenburg, and M. Forster. Radial level planarity testing and embedding in linear time. Technical Report MIP-0303, University of Passau, June 2003.

[4] C. Bachmaier, F. J. Brandenburg, and M. Forster. Radial level planarity testing and embedding in linear time. Submitted for publication, February 2004.

[5] C. Bachmaier, F. J. Brandenburg, and M. Forster. Radial level planarity testing and embedding in linear time (extended abstract). In G. Liotta, editor, *Proc. Graph Drawing, GD 2003*, volume 2912 of *LNCS*, pages 393–405. Springer, 2004.

[6] C. Bachmaier, F. J. Brandenburg, and M. Forster. Track planarity testing and embedding. In P. Van Emde Boas, J. Pokorný, M. Bieliková, and J. Štuller, editors, *Proc. Software Seminar: Theory and Practice of Informatics, SOFSEM 2004*, volume 2, pages 9–17. MatFyzPres, 2004.

[7] C. Bachmaier and M. Raitner. Improved symmetric lists. Submitted for publication. Preprint available at `http://www.infosun.fmi.uni-passau.de/~chris/index.html#publications`, February 2004.

[8] F. Bernhart and P. C. Kainen. The book thickness of a graph. *Journal of Combinatorial Geometry*, page B 27, 1979.

[9] P. Bertolazzi, C. Mannino, G. Di Battista, and R. Tamassia. Optimal upward planarity testing of single-source digraphs. Technical Report CS-94-46, Department of Computer Science, Brown University, 1994.

[10] H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of a small treewidth. Technical Report RUU-CS-92-27, Computer Science Department, Utrecht University, 1992.

[11] N. Bonichon, B. Le Saëc, and M. Mosbah. Optimal area algorithm for planar polyline drawings. In L. Kuera, editor, *Graph-Theoretic Concepts in Computer Science: 28th International Workshop, WG 2002*, volume 2573 of *LNCS*, pages 35–46. Springer, 2002.

[12] G. Booch, J. Rumbaugh, and I. Jacobson. *Das UML Benutzerhandbuch*. Addison-Wesley, first edition, 1999.

[13] 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.

[14] J. M. Boyer, P. F. Cortese, M. Patrignani, and G. Di Battista. Stop minding your P's and Q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. Technical Report RT-DIA-83-2003, Dipartimento di Informatica e Automazione, Università degli studi di Roma Tre, November 2003. http://web.dia.uniroma3.it/ricerca/rapporti/rt/2003-83.pdf.

[15] J. M. Boyer, P. F. Cortese, M. Patrignani, and G. Di Battista. Stop minding your P's and Q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. In G. Liotta, editor, *Proc. Graph Drawing, GD 2003*, volume 2912 of *LNCS*, pages 25–36. Springer, 2003.

[16] J. M. Boyer and W. Myrvold. Stop minding your P's and Q's: A simplified $\mathcal{O}(n)$ planar embedding algorithm. In *Proc. ACM-SIAM Symposium on Discrete Algorithms, SODA 1999*, pages 140–146, 1999.

[17] J. M. Boyer and W. Myrvold. Stop minding your P's and Q's: Simplified planarity by edge addition. Submitted for publication. Preprint available at http://www.pacificcoast.net/~lightning/planarity.pdf, September 2003.

[18] U. Brandes, P. Kenis, and D. Wagner. Centrality in policy network drawings. In J. Kratochvíl, editor, *Proc. Graph Drawing, GD 1999*, volume 1731 of *LNCS*, pages 250–258. Springer, 1999.

[19] U. Brandes, P. Kenis, and D. Wagner. Communicating centrality in policy network drawings. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):241–253, 2003.

[20] U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Proc. Graph Drawing, GD 2001*, volume 2265 of *LNCS*, pages 31–44. Springer, 2001.

[21] R. Brockenauer and S. Cornelsen. *Drawing Clusters and Hierarchies*, chapter 8, pages 193–227. Volume 2025 of *LNCS* [105], 2001.

[22] C. Buchheim, M. Jünger, and S. Leipert. A fast layout algorithm for k-level graphs. Technical Report 99-368, Institut für Informatik Universität zu Köln, 1999.

[23] C. Buchheim, M. Jünger, and S. Leipert. A fast layout algorithm for k-level graphs. In J. Marks, editor, *Proc. Graph Drawing, GD 2000*, volume 1984 of *LNCS*, pages 229–240. Springer, 2001.

[24] M.-J. Carpano. Automatic display of hierarchized graphs for computer aided decision analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 10(11):705–715, 1980.

[25] M. Chandramouli and A. A. Diwan. Upward numbering testing for triconnected graphs. In F. J. Brandenburg, editor, *Proc. Graph Drawing, GD 1995*, volume 1027 of *LNCS*, pages 140–151. Springer, 1996.

[26] 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*, 30:54–76, 1985.

[27] M. Chrobak and G. Kant. Convex grid drawings of 3-connected planar graphs. *International Journal of Computational Geometry and Applications*, 7(3):211–223, 1997.

[28] M. Chrobak and T. H. Payne. A linear-time algorithm for drawing a planar graph on a grid. *Information Processing Letters*, 54(4):241–246, 1995.

[29] F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg. Embedding graphs in books: A layout problem with applications to vlsi-design. *SIAM Journal on Algorithms and Discrete Methods*, 8(1):33–58, 1987.

[30] J. Clark and D. A. Holton. *Graphentheorie, Grundlagen und Anwendungen.* Spektrum, 1994.

[31] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 2000.

[32] S. Cornelsen, T. Schank, and D. Wagner. Drawing graphs on two and three lines. In S. G. Kobourov and M. T. Goodrich, editors, *Proc. Graph Drawing, GD 2002*, volume 2528 of *LNCS*, pages 31–41, 2002.

[33] E. Dahlhaus. A linear time algorithm to recognize clustered planar graphs and its parallelization. In C. L. Lucchesi, editor, *3rd Latin American Symposium on Theoretical Informatics, LATIN '98*, volume 1380 of *LNCS*, pages 239–248. Springer, 1998.

[34] H. de Fraysseix, J. Pach, and R. Pollack. Small sets supporting Fáry embeddings of planar graphs. In *Proc. ACM Symposium on Theory of Computing, STOC 1988*, pages 426–433. ACM Press, 1988.

[35] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10:41–51, 1990.

[36] H. de Fraysseix and P. Rosenstiehl. A depth-first characterisation of planarity. *Annals of Discrete Mathematics*, 13:75–80, 1982.

[37] H. de Fraysseix and P. Rosenstiehl. A characterization of planar graphs by trémaux orders. *Combinatorica*, 5(2):127–135, 1985.

[38] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[39] G. Di Battista and E. Nardelli. Hierarchies and planarity theory. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(6):1035–1046, 1988.

[40] G. Di Battista and R. Tamassia. Algorithms for plane representations of acyclic digraphs. *Theoretical Computer Science*, 61:175–198, 1988.

[41] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25(5):957–997, 1996.

[42] G. Di Battista, R. Tamassia, and I. G. Tollis. Constraint visibility representations of graphs. *Information Processing Letters*, 41:1–7, 1992.

[43] E. Di Giacomo. *Computing Drawings of Graphs with Constraint Vertex Positions*. PhD thesis, Università degli studi di Perugia, 2003.

[44] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, second edition, 2000. electronic version from ftp://ftp.math.uni-hamburg.de/pub/unihh/math/books/diestel/GraphTheoryII.pdf.

[45] M. B. Dillencourt, D. Eppstein, and D. S. Hirschberg. Geometric thickness of complete graphs. *Journal of Graph Algorithms and Applications*, 4(3):5–17, 2000.

[46] C. Dornheim. Planar graphs with topological constraints. *Journal of Graph Algorithms and Applications, JGAA*, 6(1):27–66, 2002.

[47] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[48] V. Dujmović, M. R. Fellows, M. T. Hallett, M. Kitching, G. Liotta, C. McCartin, N. Nishimura, P. Ragde, F. A. Rosamond, M. Suderman, S. H. Whitesides, and D. R. Wood. On the parameterized complexity of layered graph drawing. In F. Meyer auf der Heide, editor, *Proc. European Symposium on Algorithms, ESA 2001*, volume 2161 of *LNCS*, pages 488–499. Springer, 2001.

[49] P. Eades. Drawing free trees. Technical Report IIAS-RR-91-17E, International Institute for Advanced Study of Social Information Science, Fujitsu Limited, Japan, 1991.

[50] P. Eades. Drawing free trees. *Bulletin of the Institute of Combinatorics and its Applications*, 5:10–36, 1992.

[51] P. Eades, Q.-W. Feng, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. Technical Report 96-2, University of Newcastle, 1996.

[52] P. Eades, B. D. McKay, and N. C. Wormald. On an edge crossing problem. In *Proc. Australian Computer Science Conference*, pages 327–334. Australian National University, 1986.

[53] P. Eades, F. Quing-Wen, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. In S. North, editor, *Proc. Graph Drawing, GD 1996*, volume 1190 of *LNCS*, pages 113–128. Springer, 1997.

[54] P. Eades and K. Sugiyama. How to draw a directed graph. *Journal of Information Processing*, 13(4):424–437, 1990.

[55] P. Eades and S. H. Whitesides. Drawing graphs in two layers. *Theoretical Computer Science*, 131:361–374, 1994.

[56] P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(1):379–403, 1994.

[57] M. Eiglsperger, M. Siebenhaller, and M. Kaufmann. An efficient implementation of Sugiyama's algorithm for layered graph drawing. Submitted for publication. Preprint, February 2004.

[58] S. Even. *Graph Algorithms*, chapter 7, pages 148–191. Computer Science Press, 1979.

[59] S. Even and R. E. Tarjan. Computing an *st*-numbering. *Theoretical Computer Science*, 2:339–344, 1976.

[60] I. Fáry. On straight line representing of planar graphs. *Acta Scientiarum Mathematicarum Szeged*, 11:229–233, 1948.

[61] S. Felsner, G. Liotta, and S. K. Wismath. Straight-line drawings on restricted integer grids in two and three dimensions. Technical Report TR-CS-01-01, University of Lethbridge, 2001.

[62] S. Felsner, G. Liotta, and S. K. Wismath. Straight-line drawings on restricted integer grids in two and three dimensions. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Proc. Graph Drawing, GD 2001*, volume 2265 of *LNCS*, pages 328–342. Springer, 2002.

[63] Q.-W. Feng. *Algorithms for Drawing Clustered Graphs*. Thesis for the degree of doctor of philosophy, Department of Computer Science and Software Engineering, University of Newcastle, 1997.

[64] Q.-W. Feng, R. F. Cohen, and P. Eades. Planarity for clustered graphs (extended abstract). In P. Spirakis, editor, *Proc. European Symposium on Algorithms, ESA 1995*, volume 979 of *LNCS*, pages 213–226. Springer, 1995.

[65] F. Fischer. Koordinatenzuweisung beim radialen Zeichnen von gerichteten Graphen (working title). diploma thesis under work, Fakultät für Mathematik und Informatik, Universität Passau, 2004.

[66] M. Forster. Applying crossing reduction strategies to layered compound graphs. In S. G. Kobourov and M. T. Goodrich, editors, *Proc. Graph Drawing, GD 2002*, volume 2528 of *LNCS*, pages 276–284. Springer, 2002.

[67] M. Forster and C. Bachmaier. Clustered level planarity. In P. Van Emde Boas, J. Pokorný, M. Bieliková, and J. Štuller, editors, *Proc. Software Seminar: Theory and Practice of Informatics, SOFSEM 2004*, volume 2932 of *LNCS*, pages 218–228. Springer, 2004.

[68] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.

[69] E. R. Gansner, E. Koutsofios, S. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[70] M. R. Garey and D. S. Johnson. *A Guide to the Theory of NP-Completeness*. W. H. Freemann, 1979.

[71] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[72] A. Garg, M. T. Goodrich, and R. Tamassia. Planar upward tree drawings with optimal area. *International Journal of Computational Geometry and Applications*, 6(3):333–356, 1996.

[73] A. J. Goldstein. An efficient and constructive algorithm for testing whether a graph can be embedded in a plane. Technical Report Contract No. NONR 1858-(21), Department of Mathematics, Princeton University, 1963.

[74] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*, chapter 5.2. John Wiley & Sons, second edition, 2001.

[75] Gravisto. Graph Visualization Toolkit. http://www.gravisto.org/. University of Passau.

[76] GTL. Graph Template Library. http://www.infosun.fmi.uni-passau.de/GTL/. University of Passau.

[77] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[78] C. Gutwenger, M. Jünger, S. Leipert, P. Mutzel, M. Percan, and R. Weiskircher. Advances in c-planarity testing of clustered graphs. In M. T. Goodrich and S. G. Kobourov, editors, *Proc. Graph Drawing, GD 2002*, volume 2528 of *LNCS*, pages 220–235. Springer, 2002.

[79] D. Harel and M. Sardas. An algorithm for straight-line drawings of planar graphs. *Algorithmica*, 20(2):119–135, 1998.

[80] P. Healy and A. Kuusik. Characterisation of level non-planar graphs by minimal patterns. Technical Report UL-CSIS-98-4, Department of Computer Science and Information Systems, University of Limerick, July 1998.

[81] P. Healy and A. Kuusik. The vertex-exchange graph: A new concept for multi-level crossing minimisation. In J. Kratochvíl, editor, *Proc. Graph Drawing, GD 1999*, volume 1731 of *LNCS*, pages 205–216. Springer, 1999.

[82] P. Healy, A. Kuusik, and S. Leipert. Characterization of level non-planar graphs by minimal patterns. In D.-Z. Du, P. Eades, V. Estivill-Castro, X. Lin, and A. Sharma, editors, *Computing and Combinatorics, 6th Annual International Conference, COCOON 2000*, volume 1858 of *LNCS*, pages 74–84. Springer, 2000.

[83] P. Healy, A. Kuusik, and S. Leipert. Characterization of level non-planar graphs by minimal patterns. Technical Report 2000-382, Institut für Informatik, Universität zu Köln, 2000.

[84] P. Healy and N. S. Nikolov. How to layer a directed acyclic graph. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Proc. Graph Drawing, GD 2001*, volume 2265 of *LNCS*, pages 16–30. Springer, 2002.

[85] L. S. Heath and S. V. Pemmaraju. Recognizing leveled-planar dags in linear time. In F. J. Brandenburg, editor, *Proc. Graph Drawing, GD 1995*, volume 1027 of *LNCS*, pages 300–311. Springer, 1996.

[86] L. S. Heath and S. V. Pemmaraju. Stack and queue layouts of directed acyclic graphs: Part II. *SIAM Journal on Computing*, 28(5):1588–1626, 1999.

[87] L. S. Heath, S. V. Pemmaraju, and A. N. Trenk. Stack and queue layouts of directed acyclic graphs: Part I. *SIAM Journal on Computing*, 28(4):1510–1539, 1999.

[88] L. S. Heath and A. L. Rosenberg. Laying out graphs using queues. *SIAM Journal on Computing*, 21(5):927–958, 1992.

[89] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the Association for Computing Machinery, JACM*, 21(4):549–568, 1974.

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

[91] J. P. Hutchinson. On polar visibility representations of graphs. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Proc. Graph Drawing, GD 2001*, volume 2265 of *LNCS*, pages 422–434. Springer, 2001.

[92] M. D. Hutton and A. Lubiw. Upward planar drawing of single source acyclic digraphs. In *Proc. ACM-SIAM Symposium on Discrete Algorithms, SODA 1991*, pages 203–211, 1991.

[93] M. D. Hutton and A. Lubiw. Upward planar drawing of single source acyclic digraphs. *SIAM Journal on Computing*, 25(2):291–311, 1996.

[94] D. S. Johnson. The NP-completeness column: An ongoing guide. *Journal of Algorithms*, 3(1):89–99, 1982.

[95] M. Jünger and S. Leipert. Level planar embedding in linear time. In J. Kratochvíl, editor, *Proc. Graph Drawing, GD 1999*, volume 1731 of *LNCS*, pages 72–81. Springer, 1999.

[96] M. Jünger and S. Leipert. Level planar embedding in linear time. Technical Report 99.374, Institut für Informatik, Universität zu Köln, 1999.

[97] M. Jünger and S. Leipert. Level planar embedding in linear time. *Journal of Graph Algorithms and Applications, JGAA*, 6(1):67–113, 2002.

[98] M. Jünger, S. Leipert, and P. Mutzel. Pitfalls of using PQ-trees in automatic graph drawing. In G. Di Battista, editor, *Proc. Graph Drawing, GD 1997*, volume 1353 of *LNCS*, pages 193–204. Springer, 1997.

[99] M. Jünger, S. Leipert, and P. Mutzel. Level planarity testing in linear time. In S. H. Whitesides, editor, *Proc. Graph Drawing, GD 1998*, volume 1547 of *LNCS*, pages 224–237. Springer, 1998.

[100] M. Jünger, S. Leipert, and P. Mutzel. Level planarity testing in linear time. Technical Report 99.369, Institut für Informatik, Universität zu Köln, 1999.

[101] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[102] G. Kant. *Algorithms for Drawing Planar Graphs*. PhD thesis, Universtät Utrecht, 1993.

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

[104] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[105] M. Kaufmann and D. Wagner. *Drawing Graphs*, volume 2025 of *LNCS*. Springer, 2001.

[106] M. Kaufmann and R. Wiese. Embedding the vertices at points: Few bends suffice for planar graphs. *Journal of Graph Algorithms and Applications, JGAA*, 6(1):115–129, 2002.

[107] M. Kaufmann and R. Wiese. Maintaining the mental map for circular drawings. In S. G. Kobourov and M. T. Goodrich, editors, *Proc. Graph Drawing, GD 2002*, volume 2528 of *LNCS*, pages 12–22. Springer, 2002.

[108] P. N. Klein and J. H. Reif. An efficient parallel algorithm for planarity. In *Proc. IEEE Symposium on Foundations of Computer Science, FOCS 1986*, pages 465–477. IEEE Computer Society Press, 1986.

[109] P. N. Klein and J. H. Reif. An efficient parallel algorithm for planarity. *Journal of Computer and System Sciences*, 37(2):190–246, 1988.

[110] D. S. Knuth. *The Art of Computer Programming*, volume 1, pages 280–281. Addison Wesley Longman, 3 edition, 1997.

[111] S. Leipert. PQ-trees – an implementation as template class in C++. Technical Report 97.259, Institut für Informatik, Universität zu Köln, 1997.

[112] S. Leipert. *Level Planarity Testing and Embedding in Linear Time*. Dissertation, Mathematisch-Naturwissenschaftliche Fakultät der Universität zu Köln, 1998.

[113] 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. Gordon and Breach, 1967.

[114] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982.

[115] X. Lin and P. Eades. Area minimization for grid visibility representation of hierarchically planar graphs. In T. Asano, H. Imai, L. D. T., S.-I. Nakano, and T. Tokuyama, editors, *Computing and Combinatorics, 5th Annual International Conference, COCOON 1999*, volume 1627 of *LNCS*, pages 92–102. Springer, 1999.

[116] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[117] K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. EATCS Monographs on Theoretical Computer Science. Springer, 1984.

[118] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computional Geometry*. EATCS Monographs on Theoretical Computer Science. Springer, 1984.

[119] K. Mehlhorn, P. Mutzel, and S. Näher. An implementation of the Hopcroft and Tarjan planarity test and embedding algorithm. Research Report MPI-I-93-151, Max-Planck-Institut für Informatik, Saarbrücken, October 1993.

[120] K. Mehlhorn and S. Näher. *LEDA, A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[121] J. L. Moreno. *Who Shall Survive: Foundations of Sociometry, Group Psychotherapy, and Sociodrama*. Beacon House, 1953.

[122] D. R. Musser and A. Saini. *The STL Tutorial and Reference Guide*. Addison Wesley, 1996.

[123] T. Nishizeki and N. Chiba. *Planar Graphs: Theory and Algorithms*, volume 32 of *Annals of Discrete Mathematics*. North Holland, 1988.

[124] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum, third edition, 1996.

[125] H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In G. Di Battista, editor, *Proc. Graph Drawing, GD 1997*, volume 1353 of *LNCS*, pages 248–261. Springer, 1997.

[126] H. C. Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing*, 13:501–516, 2002.

[127] M. Raitner. Effiziente Algorithmen zum Test der Planarität von Graphen. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1999.

[128] B. Randerath, E. Speckenmeyer, E. Boros, P. Hammer, A. Kogan, K. Makino, B. Simeone, and O. Cepek. A satisfiability formulation of problems on level graphs. Rutcor Research Report RRR 40-2001, Rutgers Center for Operations Research, Rutgers University, 2001.

[129] M. G. Reggiani and F. E. Marchetti. A proposed method for representing hierarchies. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):2–8, 1988.

[130] P. Rosenstiehl and R. E. Tarjan. Rectilinear planar layouts and bipolar orientations of planar graphs. *Discrete & Computional Geometry*, 1(4):343–353, 1986.

[131] G. Sander. Graph layout through the VCG tool. In R. Tamassia and I. G. Tollis, editors, *Proc. of the DIMACS International Workshop on Graph Drawing (GD 1994)*, volume 894 of *LNCS*, pages 194–205. Springer, 1995.

[132] G. Sander. A fast heuristic for hierarchical manhattan layout. In F. J. Brandenburg, editor, *Proc. Graph Drawing, GD 1995*, volume 1027 of *LNCS*, pages 447–458. Springer, 1996.

[133] G. Sander. Layout of compound directed graphs. Technical Report A/03/96, Universität Saarbrücken, 1996.

[134] G. Sander. *Visualisierungstechniken für den Compilerbau.* PhD thesis, Universität Saarbrücken, 1996.

[135] G. Sander. Graph layout for applications in compiler construction. *Theoretical Computer Science*, 217:175–214, 1999.

[136] T. Schank. Algorithmen zur Visualisierung planarer, partitionierter Graphen. Master's thesis, Universität Konstanz, 2001.

[137] W. Schnyder. Embedding planar graphs on the grid. In *Proc. ACM-SIAM Symposium on Discrete Algorithm, SODA 1990*, pages 138–148, 1990.

[138] F. Schreiber. *Visualisierung biochemischer Reaktionsnetze.* PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2001.

[139] W.-K. Shih and W.-L. Hsu. A simple test for planar graphs. In *Proc. Workshop on Discrete Mathematics and Algorithms*, pages 110–122, 1993.

[140] W.-K. Shih and W.-L. Hsu. A new planarity test. *Theoretical Computer Science*, 223(1–2):179–191, 1999.

[141] Silicon Graphics, Inc. STL. Standard Template Library. http://www.sgi.com/tech/stl/.

[142] J. M. Six and I. G. Tollis. Circular drawings of biconnected graphs. In M. T. Goodrich and C. C. McGeoch, editors, *Proc. International Workshop on Algorithm Engineering and Experimentation, ALENEX 1999*, volume 1619 of *LNCS*, pages 57–73. Springer, 1999.

[143] J. M. Six and I. G. Tollis. A framework for circular drawings of networks. In J. Kratochvíl, editor, *Proc. Graph Drawing, GD 1999*, volume 1731 of *LNCS*. Springer, 1999.

[144] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(1):362–391, 1983.

[145] S. K. Stein. Convex maps. In *Proc. American Mathematical Society*, volume 2, pages 464–466, 1951.

[146] E. Steinitz and H. Rademacher. *Vorlesungen über die Theorie der Polyeder*. Springer, 1934.

[147] K. Sugiyama. *Graph Drawing and Applications for Software and Knowledge Engineers*, volume 11 of *Software Engineering and Knowledge*. World Scientific, 2002.

[148] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[149] R. Tamassia and I. G. Tollis. A unified approach to visibility representations of planar graphs. *Discrete & Computional Geometry*, 1(4):321–341, 1986.

[150] R. Tamassia and I. G. Tollis. Representations of graphs on a cylinder. *SIAM Journal of Discrete Mathematics*, 4(1):139–149, 1991.

[151] R. E. Tarjan. *Data Structures and Network Algorithms*, chapter 1.3, page 11. CBMS-NSF Regional Conferences Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1983.

[152] N. Tomii, Y. Kambayashi, and S. Yajima. On planarization of 2-level graphs. *Papers of Technical Group on Electronic Computers, IECEJ*, EC77-38:1–12, 1977.

[153] W. T. Tutte. Convex representations of graphs. In *Proc. London Mathematical Society, Third Series*, volume 10, pages 304–320, 1960.

[154] W. T. Tutte. How to draw a graph. In *Proc. London Mathematical Society, Third Series*, volume 13, pages 743–768, 1963.

[155] J. D. Ullman. *Computational Aspects of VLSI*, chapter 3.5, pages 111–114. Computer Science Press, 1984.

[156] K. Wagner. Bemerkungen zum Vierfarbenproblem. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 46:26–32, 1936.

[157] J. N. Warfield. Crossing theory and hierarchy mapping. *IEEE Transactions on Systems, Man, and Cybernetics*, 7(7):502–523, 1977.

[158] S. G. Williamson. Embedding graphs in the plane — algorithmic aspects. *Annals of Discrete Mathematics*, 6:349–384, 1980.

[159] S. G. Williamson. Depth-first search and Kuratowski subgraphs. *Journal of the Association for Computing Machinery, JACM*, 31(4):681–693, 1984.

Partial Publications

- C. Bachmaier, F. J. Brandenburg, and M. Forster. Radial level planarity testing and embedding in linear time. Technical Report MIP-0303, University of Passau, June 2003.
- C. Bachmaier, F. J. Brandenburg, and M. Forster. Radial level planarity testing and embedding in linear time. Submitted for publication, February 2004.
- C. Bachmaier, F. J. Brandenburg, and M. Forster. Radial level planarity testing and embedding in linear time (extended abstract). In G. Liotta, editor, *Proc. Graph Drawing, GD 2003*, volume 2912 of *LNCS*, pages 393–405. Springer, 2004.
- C. Bachmaier, F. J. Brandenburg, and M. Forster. Track planarity testing and embedding. In P. Van Emde Boas, J. Pokorný, M. Bieliková, and J. Štuller, editors, *Proc. Software Seminar: Theory and Practice of Informatics, SOFSEM 2004*, volume 2, pages 9–17. MatFyzPres, 2004.
- C. Bachmaier and M. Raitner. Improved symmetric lists. Submitted for publication. Preprint available at http://www.infosun.fmi.uni-passau.de/~chris/index.html#publications, February 2004.

Index