# Level Planarity Testing and Embedding in Linear Time

Inaugural-Dissertation
zur
Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität zu Köln

vorgelegt von

Sebastian Leipert
aus München

Druckerei Sutorius, Köln
1998

| | |
|---|---|
| Berichterstatter: | Prof. Dr. Michael Jünger<br>Prof. Dr. Ewald Speckenmeyer |
| Tag der mündlichen Prüfung: | 22. Dezember 1998 |

Gewidmet
meinen Eltern Neeltje und Klaus-Peter
und meiner Frau Anja

# Preface

It was Petra Mutzel who introduced me in 1993 to the beautiful field of Automatic Graph Drawing. By that time she was working on the maximum planar subgraph problem and her enthusiasm easily dragged me into this field. She showed me that graph drawing is more than simply assigning $x$ and $y$ coordinates to the vertices of a graph. It is a field full of nice and very challenging problems, most of them being very difficult as well. When I was working with Petra Mutzel on the maximal planar subgraph problem, I learned about the $PQ$-tree data structure.

$PQ$-trees kept accompanying my work, when I joined Prof. Dr. Michael Jünger's group in November 1995. He encouraged me to implement this data structure as a reusable and object oriented software framework, when I was working with Peter Störmer on software for finding violated comb inequalities for solving TSP-instances by branch and cut. This reusable software package made all subsequent implementations a lot easier, especially when I started working on the level planarity testing and embedding problem.

I am very grateful to Michael Jünger for supporting my work in many aspects. He provided me with the opportunity to take part in conferences, especially the annual Graph Drawing Conferences that have been very important for me, and he introduced me to many people working in the field of Automatic Graph Drawing. Michael Jünger and Petra Mutzel always took their time to discuss the problems that I encountered in my work, and its due to these discussions that I kept on pursuing for a linear time level planarity test, once I found out how to solve the problem in linear logarithmic time.

I am very grateful to my colleagues Volker Kaibel, Max Böhm, Stefan Thienel, Joachim Kupke, and Martin Wolff. Discussion with Volker used to be very interesting, especially in the later period of my work. He also did careful proofreading of large parts of the thesis. Max took his time to check Chapter 6 and Martin did the proofreading of Chapter 5. It was Stefan's idea to implement a tool for visualizing branch and cut trees for his ABACUS framework. I implemented the *Tree Interface* for him, when I was a master student and extended it to a debugging tool. The tool became very valuable for me during the implementation of the $PQ$-tree algorithms. The *Tree Interface* itself was based Joachim's *Graphical Front End* and his support during the implementation of the *Tree Interface* was very valuable. Besides, Joachim always had an answer whenever I had an "unsolvable" problem with Linux, and thanks to his advise I finally got my printer working.

Of course, Linux was not the only source of problems, and it was Thomas Lange who kept the number of problems very small and provided us with an excellent Unix system. I want to thank Elias Dahlhaus for checking some of my proofs, and Martin Diehl for a lot fruitful discussions on C++ implementation details. I also want to thank Claudia Rötters and Judith Steinmann for providing me with any kind of literature, Michael Belling for his assistance from the library, and our secretary Ursula Neugebauer. I owe many thanks to Christoph Buchheim, Marcus Oswald, and Gørril Vollen for testing earlier versions of my $PQ$-tree implementation.

There have been several other colleagues who supported my work and whom I want to thank: Carsten Gutwenger for his great support on the AGD-Library, Goos Kant for discussions on the maximal planar subgraph problem, Guiseppe Di Battista for discussions on level planarity testing, Hermann Stamm-Wilbrandt for providing his code to generate maximum planar graphs, Manfred Padberg for his advise while he was visiting our group in autumn 1996, Gerhard Reinelt for giving me the opportunity to present my results to a broader audience, Ulrik Brandes for pointing out some interesting papers, Michael Seel for his support when adding my $PQ$-tree implementation as LEDA-Extension Package to LEDA, and David Alberts, Ralf Brockenauer, Matthias Elf, Gunnar Klau, Stefan Näher, René Weiskircher, and Thomas Ziegler for their great cooperation within the DFG-Project "Design, Analysis, Implementation, and Evaluation of Graph Drawing Algorithms".

Finally this work would not have been possible without the encouragement, love, and support of my wife Anja, and my parents Neeltje and Klaus-Peter Leipert.

Köln, October 1998 Sebastian Leipert

# Contents

# Chapter 1

# Introduction

Automatic Graph Drawing was probably born in the early sixties when computer scientists in desperate need of understanding software started visualizing their code in diagrams using graph drawing. A graph is an abstract structure that can be used to represent information that can be modeled as objects and relations between those objects. The objects are represented by the vertices and the relations by the edges of the graph. The computer scientists soon figured out that drawing a graph with more than ten objects by hand is either incredibly difficult or simply not possible. Unfortunately, almost every graph that they wanted to visualize did have a lot more than ten vertices. Therefore, they started thinking about algorithms for drawing graphs for visualization purposes, not being aware of the fact that this way they founded a new area of scientific research, an area full of very difficult problems and overwhelming results.

The reason why Automatic Graph Drawing becomes more and more popular is in fact its broad application in different areas. Chemists need to draw large molecules, and biologists need to draw evolutionary trees. Databases are designed using entity-relationship diagrams, and decision support systems for project management need to visualize PERT-Networks and activity trees. Software engineers want data flow diagrams, subroutine-call graphs and object-oriented class hierarchies to be visualized.

Designers of graph drawing algorithms as well as the readers of the diagrams want certain aesthetics optimized such that the resulting graph drawings help the reader to understand and remember the information embodied in the graph. Examples of these aesthetics include minimizing the number of edge crossings, minimizing the number of edge bends, minimizing the display area of the graph, visualizing a common direction (flow) in the graph, maximizing the angular resolution at the vertices, and maximizing the display of symmetries. Certainly, two aesthetic criteria cannot be simultaneously optimized in general and it depends on the data which criterion should be preferably optimized.

A fundamental issue in Automatic Graph Drawing is to display hierarchical network structures as they appear in software engineering, project management or database design. The network is transformed into a directed acyclic graph $G = (V, E)$ that has to be drawn with

straight line edges that are either all directed upwards or all directed downwards. Most applications imply a partition $V^1 \cup V^2 \cup \cdots \cup V^k$ of the vertices $V$ into $k$ levels that have to be visualized by placing the vertices belonging to the same level $V^i$ on a horizontal line. These graphs are called level graphs. An example of a level graph is shown in Fig. 1.1. It has been produced using the visualization tool VCG Tool by Lemke and Sander (1995), displaying a compiler graph.

A criterion to obtain good readability in the presentation of level graphs is to produce diagrams with a limited number of crossings between the edges. However, the *$k$-level crossing minimization problem* that asks for minimizing the number of crossings in a drawing of a $k$-level graph has shown to be $\mathcal{NP}$-hard by Garey and Johnson (1983), even if $k = 2$. According to Eades, McKay, and Wormald (1986), the problem remains $\mathcal{NP}$-hard if the vertices of one of the two levels are fixed in their position.

Due to the $\mathcal{NP}$-hardness of the problems a lot of effort has been spent to the design of efficient heuristics for reducing the number of crossings in drawings of 2-level graphs. The main idea was to use a "good" heuristic for the 2-level case and to perform a "level by level sweep" on the general $k$-level case, trying to reduce the crossings between consecutive levels. Choosing an appropriate ordering of the first level, the ordering of every level $i$ is kept fix while reordering the level $i+1$ in order to reduce the crossings between level $i$ and $i+1$. The process can be repeated in reverse direction to reduce the crossings further.

The compiler graph in Fig. 1.1 has been produced using the *barycentric heuristic*. This heuristic is the most commonly used one and has been presented by Sugiyama, Tagawa, and Toda (1981). Given a 2-level graph with one level fixed, the barycentric method orders the vertices of the free level according to the barycenter (average) of the $x$-coordinates of their neighbors in the fixed level.

Figure 1.2 shows the upper left part of the drawing of the compiler graph. It reveals one of the problems that come along with the approach of performing a level by level sweep in order to minimize the number of crossings. The figure shows that there is an edge crossing of two edges connecting vertices on the third and the fourth level. It is easy to see that this crossing can be avoided by simply placing both the fourth vertex on level 2 and and the fourth vertex on level 3 to the left side of the first vertex on level 2 and level 3, respectively, and by placing the third vertex on level 4 between the first and the second vertex. The "origin" for this edge crossing is located in the first two levels. Based on a fixed order of the vertices of the first level, the barycentric method has chosen an order for the vertices of the second level, such that no crossings occur between the two levels. Since every vertex on level 2 has at most one outgoing edge this determines the ordering of the vertices on the third level. When the heuristic finally computes the positions of the vertices of the fourth level, it cannot avoid the edge crossing.

The problem that appears here is not specific to the barycentric heuristic. All algorithms based on a "level by level sweep" can produce unnecessary crossings like the one in the example above. This is due to the very local view of these algorithms. More general approaches that try to obtain a global view during the minimization of edge crossings are not

practical yet and demand deeper studies. A first approach has been reported by Jünger, Lee, Mutzel, and Odenthal (1997a) along with preliminary computational results for 2- and 3-level graphs.

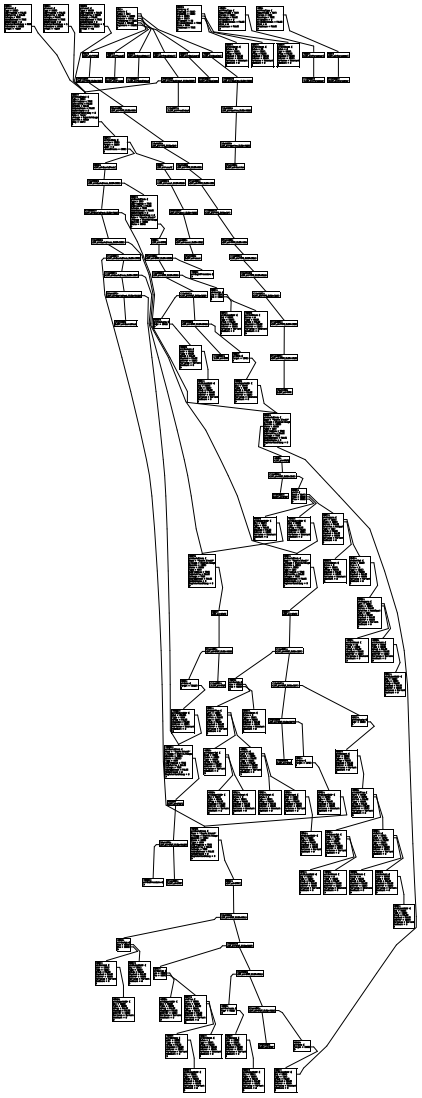


Figure 1.1: Example of a level graph.

However, if a level graph *can* be drawn without any edge crossings, we would like to detect this fact and construct a corresponding drawing. Level graphs that admit such a drawing are called *level planar*. A first approach for testing level planarity has been presented by Di Battista and Nardelli (1988) for the special case of hierarchies. A hierarchy is a level graph such that all sources belong to the same level of the graph. To perform their test, Di Battista and Nardelli (1988) use the $PQ$-tree data structure which will play a central role in this work, too.

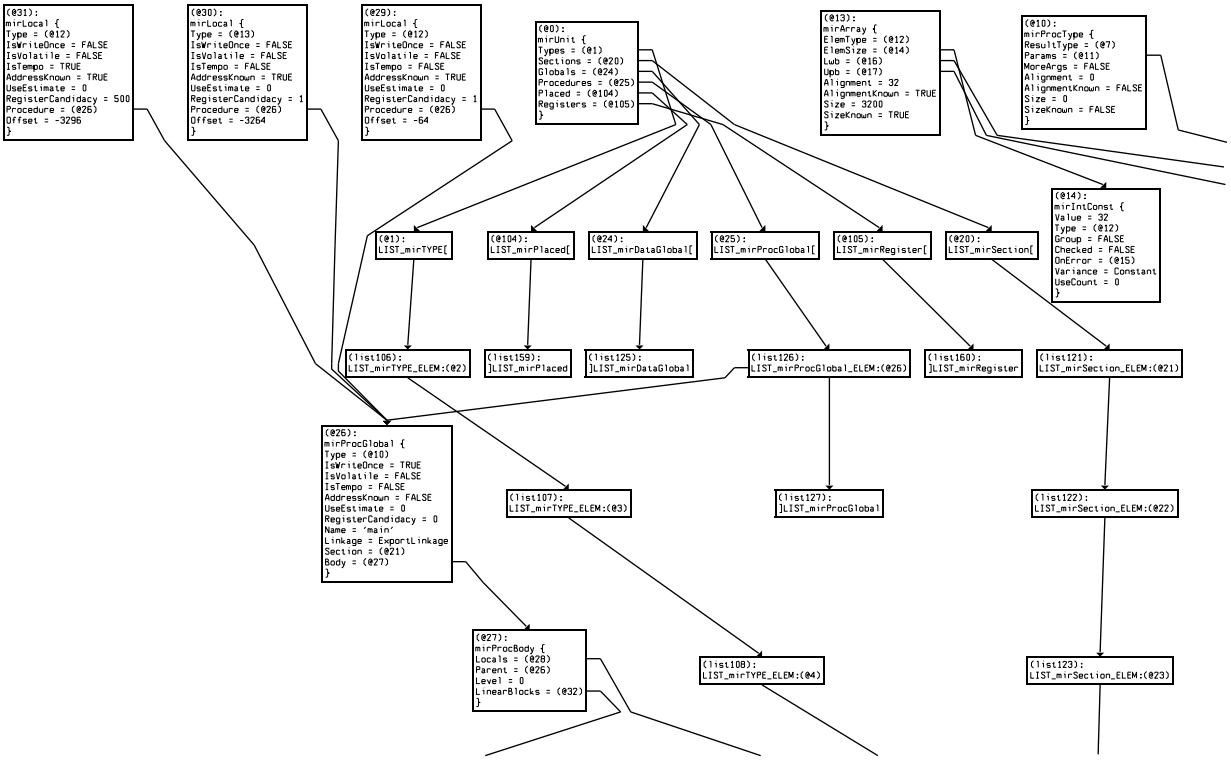



Figure 1.2: Upper leftmost part of the graph shown in Fig.1.1.

A $PQ$-tree is a powerful data structure that represents those permutations of a finite set in which the members of specified subsets occur consecutively. This data structure has been introduced by Booth and Lueker (1976) to solve the problem of testing for the consecutive ones property in matrices, i.e., determine if the rows of a $(0,1)$-matrix can be permuted such that in each column all of the ones are consecutive. The most well-known applications of $PQ$-trees are in Automatic Graph Drawing, i.e., planarity testing (see Lempel, Even, and Cederbaum (1967); Booth and Lueker (1976)) and embedding of planar graphs (see Chiba, Nishizeki, Abe, and Ozawa (1985)).

$PQ$-trees have also been proposed by Heath and Pemmaraju (1995, 1996) to test level planarity of level graphs with several sources and sinks where the sources and sinks may appear on arbitrary levels. However, in Section 4.3.2 we show that their application of $PQ$-trees leads to an incorrect algorithm. Since this algorithm was the only attempt to prove polynomial time complexity in the literature, the complexity status of level planarity testing was open after we had detected the incorrectness of the algorithm of Heath and Pemmaraju. In Chapter 4 we describe one of the main contributions of this thesis: a level planarity test of the general level planarity problem with linear running time.

In order to draw a level planar graph without edge crossings, a level planar embedding of the level graph has to be computed. For this we need to determine an ordering of the vertices that admits such a level planar drawing. In Chapter 5 we present a linear time

algorithm for embedding level planar graphs. This approach is based on the level planarity test and augments a level planar graph $G$ to an $st$-graph $G_{st}$, a graph with a single sink and a single source, without destroying the level planarity. Once the $st$-graph has been constructed, we compute a planar embedding of the $st$-graph. This is done by applying the embedding algorithm of Chiba *et al.* (1985) for general graphs, obeying the topological ordering of the vertices in the $st$-graph. Exploiting the embedding of the $st$-graph $G_{st}$, we are able to determine a level planar embedding of $G$.

This thesis starts by describing some of the basic concepts of graph theory in Chapter 2. Besides fixing the necessary notations we give a short introduction to the theory of planar graphs. More enhanced graph models like cluster graphs and level graphs are described in Sections 2.5 and 2.6.

Chapter 3 introduces the $PQ$-tree data structure and some of its applications like, e.g., the planarity test of Booth and Lueker (1976) and the embedding algorithm of Chiba, Nishizeki, Abe, and Ozawa (1985). We introduce the data structure $PQ$-tree and the so called template matching algorithm that constructs a $PQ$-tree that represents the permutations of a finite set in which the members of specified subsets occur consecutively. After reviewing the planarity test of Lempel, Even, and Cederbaum (1967) and its modifications using $PQ$-trees in Section 3.2, we examine in Section 3.3 the embedding algorithm of Chiba *et al.* (1985). The study of these algorithms is important for this work on testing for level planarity. The embedding of cluster graphs by Feng, Cohen, and Eades (1995) is studied in Section 3.4, since it is a successful adaption of the planarity testing and embedding strategies of Booth and Lueker (1976) and Chiba *et al.* (1985). The chapter closes by reviewing the computation of maximal planar subgraphs as another unsuccessful approach of applying $PQ$-trees in the field of Automatic Graph Drawing in Section 3.5. Several attempts have been tried to solve the problem with the help of $PQ$-trees. The latest approaches have been reported by Jayakumar, Thulasiraman, and Swamy (1989) and Kant (1992). We show that the algorithm presented by Jayakumar *et al.* (1989) is not correct. It does not necessarily compute a maximal planar subgraph, and the same holds for a modified version of the algorithm presented by Kant (1992). Our conclusions suggest not to use $PQ$-trees for this specific problem. As an (intended) side-effect, Section 3.5 should be useful for preparing the first part of the next chapter, were we review the erroneous approach of Heath and Pemmaraju (1995, 1996) for testing level planarity of level graphs.

The subject of Chapter 4 is to examine existing (incorrect) level planarity tests and to develop the first (linear time) level planarity test. After a short introduction to the theoretical background, were we review characterizations of Di Battista and Nardelli (1988) and Healy and Kuusik (1998) of level planarity in terms of forbidden subgraphs, we review the level planarity test of Di Battista and Nardelli (1988). In order to check whether a level graph $G = (V, E)$ is level planar, it is sufficient to find an ordering $\leq_j$ of the vertices of every level $V^j$ such that for every pair of edges $(u_1, v_1), (u_2, v_2) \in E$ with $u_1, u_2 \in V^i$, $v_1, v_2 \in V^{i+1}$ and $u_1 \leq_i u_2$, we have $v_1 \leq_{i+1} v_2$. Let $G^j$ denote the subgraph of $G$ induced by $V^1 \cup V^2 \cup \cdots \cup V^j$. The strategy of Di Battista and Nardelli (1988) for testing the level planarity of hierarchies is to perform a top-down sweep, processing the levels in the order

$V^1, V^2, \ldots, V^k$ and computing for every level $V^j$ the set of permutations of the vertices of $V^j$ that appear in some level planar embedding of $G^j$. Obviously, the graph $G = G^k$ is level planar, if and only if the set of permutations for $G_k$ is not empty. This level planarity test implies that the set of level planar embeddings of a hierarchy can be represented by a $PQ$-tree. Section 4.3 presents the approach of Heath and Pemmaraju (1995, 1996) for general level graphs including our proof of the incorrectness of the algorithm in Section 4.3.2. As in the approach of Di Battista and Nardelli (1988) for hierarchies the basic idea is to perform a top-down sweep. Since a graph $G^j$ is not necessarily connected, separate $PQ$-trees are introduced for every component of $G^j$, and standard $PQ$-tree techniques are applied as long as different components of $G^j$ are not adjacent to a common vertex on level $j+1$. If two components are adjacent to a common vertex $v$ on level $j+1$, they have to be merged somehow and a new $PQ$-tree has to be constructed from the two corresponding $PQ$-trees. The new $PQ$-tree then represents all level planar embeddings of the merged component. Applying a combination of "reduce" operations and "merge" operations for combining $PQ$-trees, Heath and Pemmaraju try to maintain for every level $V^j$ and for every component $F$ of that level the set of permutations of the vertices of $F$ in $V^j$ that appear in some level planar embedding of $G^j$. If the set of permutations for $G^k$ is not empty, the graph $G = G^k$ is obviously level planar. However, the opposite is in general not true for the set of permutations Heath and Pemmaraju end up with. In Sections 4.4 and 4.5 we present a correct level planarity testing algorithm that is based on two main new techniques that replace the incorrect crucial parts of the algorithm of Heath and Pemmaraju (1995, 1996). After having proved the correctness in 4.6, we show that this approach yields an $\mathcal{O}(n \log n)$ time level planarity test. In Section 4.9 we devise strategies that achieve linear running time. For simplicity, we assume in Chapter 4 that all level graphs are proper, having only edges connecting vertices belonging to consecutive levels. In the final section of this chapter we show that our algorithm performs on nonproper graphs with no modification yielding a linear time algorithm for nonproper graphs as well.

Chapter 5 describes a level planar embedding algorithm that is based on our level planarity test. In order to compute a level planar embedding of a level planar graph $G = (V, E)$ the graph $G$ is augmented to a planar $st$-graph $G_{st} = (V_{st}, E_{st})$ with $V_{st} = V \uplus \{s, t\}$ and $E \subset E_{st}$. An $st$-graph is a directed acyclic graph with a single source $s$, a single sink $t$, and an edge $(s, t)$. The graph $G_{st}$ is planar embedded with the edge $(s, t)$ on the boundary of the outer face using the algorithm of Chiba *et al.* (1985). The level planar embedding is then constructed from the planar embedding. After having proved in Section 5.1 that every level graph is level planar if and only if it is a subgraph of a planar $st$-graph, we show how to obtain a level planar embedding of $G$ from the planar embedding of $G_{st}$. Section 5.2 considers the augmentation of a level graph $G$ to an $st$-graph $G_{st}$. This augmentation step is divided into two phases. In the first phase an outgoing edge is added to every sink of $G$. Using the same algorithmic concept as in the first phase, an incoming edge is added to every source of $G$ in the second phase. We finish Chapter 5 with proofs on the correctness and the linear running time of the level planar embedding algorithm.

While working on the theoretical subjects we have implemented an object oriented prototype of our level planarity testing and embedding algorithm using the C++ language. Chapter 6 reviews the concepts of the implementation including our implementation of the $PQ$-tree data structure as a class template in C++, and pays attention to some details. We refrained from including the complete software and concentrated on a few details of the implementation. The last three sections of Chapter 6 contain such details.

The discussions of Chapter 7 are intended to summarize the results of this work and to suggest directions of further investigations.

# Chapter 2

# Basic Concepts in Graph Theory

This chapter introduces basic concepts in graph theory and its intention is mainly to provide some notational conventions. The first section gives some general notation for (undirected) graphs that is also valid for more enhanced models as directed or clustered graphs, and level graphs. The second section deals with directed graphs. Since the notions of graph planarity and graph embedding are important for this work, two individual sections are devoted to these subjects. Finally, clustered graphs and level graphs are treated in the last two sections. The notations of the first four sections is mainly motivated by Harary (1969), Cormen, Leiserson, and Rivest (1990), Chiba and Nishizeki (1988), and Di Battista, Eades, Tamassia, and Tollis (1998). The notation of the last two sections is based on recent papers.

## 2.1 Graphs

A *graph* $G$ is a pair $(V, E)$, where $V$ is a finite set of *vertices* and $E$ is a finite set of *edges*, where each edge $e \in E$ consists of an unordered pair of distinct vertices $u, v \in V$. The set $V$ is called the *vertex set* of $G$ and $E$ is called the *edge set* of $G$. Vertices of a graph are denoted throughout this work by small letters such as $u, v, w, x, y$, or $z$.

Let $e \in E$ be an edge and $u \in V$ be one of its vertices, then $e$ and $u$ are called *incident*. If $e \in E$ is an edge with incident vertices $u, v \in V$, the edge $e$ *connects* the vertices $u$ and $v$, and we say that $u$ and $v$ are *adjacent*. Two edges $e_1, e_2 \in E$ that are incident to the same vertex $v \in V$ are called *adjacent*.

If $e \in E$ is an edge incident to vertices $u, v \in V$, we will use the notation $e = (u, v)$. This notation is not unique in general. However, multiple edges are not relevant to the problems discussed in this work. We therefore restrict our studies to *simple* graphs, having no multiple edges. The notation $e = (u, v)$ is unique for simple graphs.

The *degree* of a vertex $v$ is the number of its incident edges. A vertex $v$ is called *isolated* if no edge is incident to $v$. Throughout this work let $n = |V|$ denote the number of vertices and $m = |E|$ denote the number of edges of a graph.

A graph $G' = (V', E')$ is said to be a *subgraph* of $G = (V, E)$ or *contained* in $G$ if $V' \subseteq V$ and $E' \subseteq E$. If $V' = V$, $G'$ is called a *spanning subgraph* of $G$. If $E'$ contains exactly those edges of $G$ that connect two vertices in $V'$ then $G'$ is said to be *induced* by $V'$. If $G'$ is a subgraph of $G$, then $G - G' = (V, E - E')$ denotes the *difference* of $G$ and $G'$. Thus $G - G'$ is a subgraph of $G$ induced by removing all edges that are in $G'$. If $V' \subseteq V$, then $G - V'$ is the subgraph of $G$ induced by $V - V'$. For a single vertex $v \in V$, the graph $G - \{v\}$ denotes the subgraph of $G$ induced by $V - \{v\}$.

A *walk* $W$ in a graph $G$ is an alternating sequence of vertices and edges $v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$, beginning and ending with vertices $v_0$ and $v_k$ and $e_i = (v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. This walk *connects* $v_0$ and $v_k$ and may also be denoted by $W = (v_0, v_1, \ldots, v_k)$, with the edges being evident by context. A walk $W$ is called a *path* if all vertices are distinct. The *length* of a path is the number of edges on the path. A walk is called a *cycle* if all vertices are distinct except for $v_0 = v_k$ and $k \geq 3$. A graph that does not have any cycles as subgraphs is called a *forest*.

A graph $G$ is *connected* if every pair of vertices is connected by a path. A *component* of $G$ is a maximal connected subgraph of $G$. Thus a disconnected graph has at least two components. A *cut vertex* is a vertex, whose removal increments the number of components. Thus if $G$ is connected, at least one vertex has to be removed from $G$ in order to disconnect it. If no such cut vertex in $G$ exists, $G$ is called *biconnected*. A pair of vertices $u, v \in V$ is called a *split pair* if its removal disconnects the graph. The components that remain after the removal of a split pair $u$, $v$ are called *split components* with respect to the vertices $u$ and $v$. If no split pair in $G$ exists, $G$ is called *triconnected*.

A connected forest $G$ is called a *free tree*. The adjective "free" is generally omitted and we say that a graph is a tree. A subgraph $G'$ of a tree $G$ is a *subtree* if $G'$ itself is a tree. A *rooted tree* is a free tree in which one of the vertices is distinguished from the others. The distinguished vertex is called the *root* of the tree. We shall refer to a vertex of a rooted tree as a *node* of the tree. The term "node" is often used in the graph theory literature as a synonym of a "vertex". We shall reserve the term "node" to mean a vertex of a rooted tree.

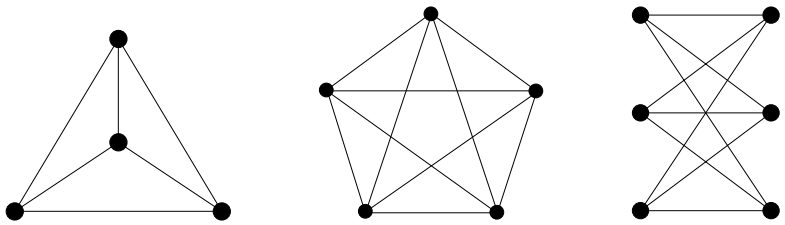
Let $Z$ be the root of a rooted tree $T$ and let $X$ be a node in $T$. Any node $Y$ on the unique path from $Z$ to $X$ is called an *ancestor* of $X$. If $Y$ is an ancestor of $X$, then $X$ is a *descendant* of $Y$. If $Y$ is an ancestor of $X$ and $X \neq Y$, then $Y$ is a *proper ancestor* of $X$ and $X$ is a *proper descendant* of $Y$. If $Y$ is a proper ancestor of $X$ and $(Y, X)$ is an edge in $T$, then $Y$ is called *parent* of $X$ and $X$ is a *child* of $Y$. The *subtree rooted at* $X$ is the tree induced by the descendants of $X$, rooted at $X$. The length of the path from the root to a node $X$ is the *depth* of $X$. The largest depth of any node in $T$ is the *height* of $T$.

In a rooted tree, every node has exactly one parent except for the root. If two nodes have the same parent, they are *siblings*. A node with no children is an *external node* or a *leaf*. A node that is not a leaf is called an *internal node*. Throughout this work we make use of capitalized letters such as $X, Y$, and $Z$ to denote nodes of a tree.

An *ordered tree* is a rooted tree in which the children of each node are ordered. Two ordered trees $T_1$ and $T_2$ are equal if their underlying rooted trees are equal and the children of every internal node $X$ of $T_1$ appear in the same order as the children of $X$ in $T_2$. Two nodes $X$, $Y$ of an ordered tree are called *adjacent* if they are siblings and appear consecutively in the order of children of their parent.

A graph $G'$ is said to be a *subdivision* of a graph $G$, if $G'$ is obtained from $G$ by replacing every edge $e = (v, w) \in E$ by a path $p = (v = u_0, u_1, \ldots, u_\mu = w)$, $\mu \geq 1$, and $u_1, u_2, \ldots, u_{\mu-1} \notin V$. Two graphs $G$ and $G'$ are said to be *isomorphic* if there exists a one to one correspondence of the vertices that induces a one to one correspondence of the edges. Usually, graphs that are isomorphic are identified.

A graph $G = (V, E)$ is said to be *complete* if every vertex $v \in V$ is adjacent to every vertex $w \in V - \{v\}$. A complete graph with $n$ vertices is denoted by $K_n$. Figure 2.1(a) and (b) show the complete graphs $K_4$ and $K_5$. A *bipartite* graph $G$ is a graph whose vertex set $V$ can be partitioned into two subsets $V_1$ and $V_2$ such that every edge connects a vertex in $V_1$ and a vertex in $V_2$. If every vertex in $V_1$ is adjacent to every vertex in $V_2$, $G$ is a *complete bipartite graph*. If $|V_1| = n_1$ and $|V_2| = n_2$, the complete bipartite graph is denoted by $K_{n_1,n_2}$. Figure 2.1(c) shows the complete bipartite graph $K_{3,3}$.



(a) $K_4$ (b) $K_5$ (c) $K_{3,3}$

Figure 2.1: Some examples of graphs.

An operation that is used very often throughout this work is the identification of vertices. Two vertices $u$ and $v$ of a graph $G$ are *identified* by introducing a new vertex $w$, inserting for every edge $(\tilde{u}, u)$ and $(\tilde{v}, v)$ an edge $(\tilde{u}, w)$ and $(\tilde{v}, w)$, respectively, removing $u$ and $v$ and all edges incident on $u$ and $v$.

## 2.2 Directed Graphs

A *directed graph* or *digraph* $G$ is a pair $(V, E)$, where $V$ is a finite set of *vertices* and $E$ is a finite set of *edges*, where each edge $e \in E$ consists of an ordered pair of vertices $u, v \in V$. Ignoring for every edge the order of its vertices, we get an undirected graph that is called the *underlying graph* of $G$. If $e = (u, v)$ is an edge in $G$, we say that $e$ *leaves* vertex $u$ and

*enters* vertex $v$. The vertex $u$ is the *tail* of $e$ and the vertex $v$ is the *head* of $e$, with $e$ being the *outgoing* edge of $u$ and the *incoming* edge of $v$. For an edge $e = (u, v)$ we say that $u$ *dominates* $v$. A *source* is a vertex with no incoming edges and a *sink* is a vertex with no outgoing edges.

A directed graph $G' = (V', E')$ is said to be a *subgraph* of a directed graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. If $V' = V$, $G'$ is called a *spanning subgraph* of $G$. A *directed walk* $W$ in a digraph $G$ is a walk $W = (v_0, v_1, \ldots, v_k)$ in the underlying graph of $G$ with $e_i = (v_{i-1}, v_i) \in E$, for $i = 1, 2, \ldots, k$. An *undirected walk* $W$ in a digraph $G$ is a walk in the underlying graph such that either $e_i = (v_{i-1}, v_i) \in E$ or $e_i = (v_i, v_{i-1}) \in E$, for $i = 1, 2, \ldots, k$. A directed or undirected walk $W$ is called a *directed* or *undirected path* if all vertices are distinct. A directed walk is called a *cycle* if all vertices are distinct except for $v_0 = v_k$. A digraph that does not have any cycles is called *acyclic*. A digraph is *connected* or *biconnected* if its underlying graph is connected or biconnected, respectively.

An acyclic digraph with exactly one source is called a *single source* graph. Consequently, an acyclic digraph with exactly one sink is called a *single sink* graph. A directed acyclic graph with exactly one source $s$ and exactly one sink $t$ and an edge $(s, t)$ is called an *st-graph*.

A *topological numbering* of $G$ is an assignment of numbers to the vertices of $G$ such that for every edge $(u, v)$ of $G$ the number assigned to $v$ is greater than the one assigned to $u$ (i.e., $number(v) > number(u)$). A *topological sorting* of $G$ is a topological numbering of $G$ such that every vertex is assigned a distinct integer between 1 and $n$. It is easy to see that $G$ admits a topological numbering or sorting if and only if $G$ is acyclic.

## 2.3 Embedded Graphs

A graph $G = (V, E)$ is generally visualized by a *drawing* in the plane with the vertices drawn as points and the edges drawn as closed Jordan curves, connecting their incident vertices. An intersection of two edges in a drawing is called a *crossing*.

An *embedding* $\mathcal{E}$ of $G$ consists of the clockwise orderings of the incident edges for every vertex with respect to a drawing. Usually, a graph $G = (V, E)$ is represented as a collection of *adjacency lists*. For each vertex $v \in V$, the adjacency list $adj(v)$ is an ordering of all vertices $u \in V$ that are adjacent to $v$.

Let $\mathcal{E}$ be an embedding of an undirected graph $G = (V, E)$ and let $e = (v, w) \in E$ be an edge in $G$. Then the vertex $v$ appears in the adjacency list $adj(w)$ and the vertex $w$ appears in the list $adj(v)$. The adjacency lists are said to *mirror* an embedding $\mathcal{E}$ of $G$, if the neighbors of a vertex $v$ appear in $adj(v)$ in the same order as in $\mathcal{E}$.

An *upward drawing* of a directed acyclic graph $G$ is a drawing of $G$ such that each edge is drawn as a curve monotonically increasing in vertical direction. An *upward embedding* $\mathcal{U}$ is a representation of $G$ that consists of the clockwise orderings of the incoming edges

for every vertex with respect to an upward drawing. Usually, any drawing of a directed acyclic graph $G$ is called upward if the edges follow a common direction. Thus if all edges are drawn as curves monotonically decreasing in vertical direction, we call this an upward drawing by convention.

Directed graphs are represented in adjacency lists as well. For the special case of upward embeddings of directed acyclic graphs, one more convention for the representation in adjacency lists is made. Let $\mathcal{U}$ be an upward embedding of a directed acyclic graph $G = (V, E)$ and let $e = (v, w) \in E$ be an edge in $G$. Then the tail $v$ of the edge $e$ appears in the adjacency list $adj(w)$ while the head $w$ does not appear in the adjacency list $adj(v)$.

## 2.4 Planar Graphs

A graph $G = (V, E)$ is called *planar* if it can be drawn in the plane such that no two edges cross each other except at common endpoints. A *planar embedding* of a planar graph $G$ is an embedding with respect to a planar drawing. A graph with a given fixed planar embedding is also called a *plane graph*. Given any drawing with respect to a planar embedding of a graph $G$, a *face* of $G$ is any topologically connected region in the drawing surrounded by the edges of $G$. A face of a plane graph is uniquely described by its surrounding edges. The one unbounded face of a plane graph is called the *outer face* or *exterior face*. All other faces are called *interior faces*. By a stereographic projection, every face of a planar embedding can be made the outer face. The *boundary* of a face $F$ is the set of edges in the closure of the face. Thus the boundary is a walk in general and is a cycle if $G$ is a biconnected graph with at least three vertices.

In general, a planar graph has many planar embeddings in the plane, and two embeddings are said to be *equivalent* if the boundary of a face in one planar embedding always corresponds to the boundary of a face in the other planar embedding. If $G$ is a disconnected plane graph, a new nonequivalent embedding can be obtained simply by replacing a connected component within another face. A plane embedding of a graph is said to be *unique* if the planar embeddings are all equivalent. Whitney (1933) proved that the planar embedding of a triconnected graph is unique.

There is a simple formula relating the number of vertices, edges and faces in a connected plane graph discovered by Euler.

**Theorem 2.1 (Euler 1750).** *Let $G = (V, E)$ be a planar, connected graph with $n = |V|$ and $m = |E|$. Let $f$ be the number of faces in a planar embedding of $G$. Then $n$, $m$ and $f$ are related by*

$$n - m + f \quad = \quad 2 \ .$$

An important corollary of Euler's formula that will be needed throughout this work is the following.

**Corollary 2.2.** *A planar graph $G = (V, E)$ satisfies $m \leq 3n - 6$. If $G$ is bipartite, then the inequality can be strengthened to $m \leq 2n - 4$.*

Corollary 2.2 immediately yields the nonplanarity of the graphs $K_5$ and $K_{3,3}$, shown in Fig. 2.1. It follows that every graph that contains a subdivision of $K_5$ and $K_{3,3}$ as subgraph is not planar. Surprisingly, the converse is also true as has been shown by Kuratowski (1930). The result yields a first and very nice characterization of planar graphs in terms of forbidden subgraphs.

**Theorem 2.3 (Kuratowski (1930)).** *A graph is planar if and only if it does not contain a subdivision of $K_5$ and $K_{3,3}$.*

A directed acyclic graph $G = (V, E)$ is called *upward planar* if it has an upward drawing that is planar with respect to the underlying graph. An *upward planar embedding* is an upward embedding with respect to an upward planar drawing. The following theorem gives a simple characterization of upward planarity.

**Theorem 2.4 (Kelly (1987),Di Battista and Tamassia (1988)).** *Let $G$ be a directed acyclic graph. Then $G$ is upward planar if and only if $G$ is a spanning subgraph of a planar st-graph.*

We need the result of this theorem later in Chapter 5 when finding a simple characterization of the special class of level planar graphs. Planar *st*-graphs were introduced in conjunction with an early planarity testing algorithm by Lempel, Even, and Cederbaum (1967) and their properties are further explored in Rosenstiehl and Tarjan (1986), Tamassia and Preparata (1990), and Tamassia and Tollis (1986). As described in Chapter 3, planarity of a graph can be tested in $\mathcal{O}(n)$ time by the approach of Lempel *et al.* (1967) using the special data structure $PQ$-tree. Despite of its simple characterization, upward planarity testing of directed acyclic graphs is $\mathcal{NP}$-complete as has been shown by Garg and Tamassia (1994). Only directed acyclic graphs having a single source can be tested for upward planarity. Hutton and Lubiw (1996) presented an upward planarity test for this class of graphs using $\mathcal{O}(n^2)$ time. This upward planarity test was later improved by Bertolazzi, Di Battista, Mannino, and Tamassia (1998) to a linear running time algorithm using a special data structure called $SPQR$-tree.

## 2.5 Cluster Graphs

Cluster graphs are graphs with recursive clustering structures over the vertices. A *cluster graph* $C = (G, T)$ consists of an undirected graph $G$ and a rooted tree $T$ such that the leaves of $T$ are exactly the vertices of $G$. Each node $\nu$ of $T$ represents a *cluster* $V(\nu)$ of the vertices of $G$ that are leaves of the subtree rooted at $\nu$. The tree $T$ describes an inclusion relation between clusters. The tree $T$ is called the *inclusion tree* of $C$. The graph $G$ is

called the *underlying graph* of $C$. The tree $T(\nu)$ represents the subtree of $T$ rooted at the node $\nu$, and $G(\nu)$ denotes the subgraph of $G$ induced by the cluster associated with node $\nu$. We define $C(\nu) = (G(\nu), T(\nu))$ to be the *subcluster graph* associated with node $\nu$. An edge $(v, w)$ with $v$ being a vertex in $G(\nu)$ and $w$ being a vertex in $G - G(\nu)$ is said to be *incident* to cluster $\nu$.

In a *drawing* of a cluster graph $C = (G, T)$, the graph $G$ is drawn as points and curves as usual. For each node $\nu$ of $T$, the cluster is drawn as simple closed region $R$ (i.e., a region without holes) that contains the drawing of $G(\nu)$, such that the following three conditions hold.

(i) The regions for all subclusters of $\nu$ are completely contained in the interior of $R$.

(ii) The regions for all other clusters are completely contained in the exterior of $R$.

(iii) If there is an edge $e$ between two vertices of $V(\nu)$ then the drawing of $e$ is completely contained in $R$.

A cluster graph $C = (G, T)$ is a *connected cluster graph* if each cluster induces a connected subgraph of $G$. Let $C_1 = (G_1, T_1)$ and $C_2 = (G_2, T_2)$ be two cluster graphs, such that $T_1$ is a subtree of $T_2$ and for each node $\nu$ of $T_1$, $G_1(\nu)$ is a subgraph of $G_2(\nu)$. Then $C_1$ is said to be a *subcluster graph* of $C_2$.

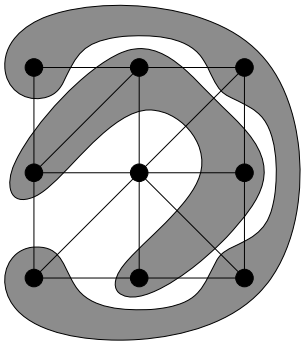


Figure 2.2: A cluster graph that is not $c$-planar.

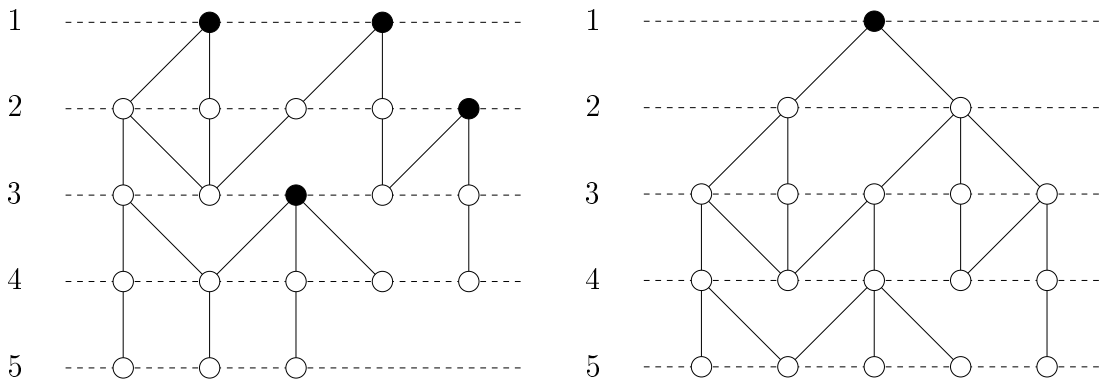
The drawing of an edge $e$ and a region $R$ have an *edge-region crossing* if the drawing of $e$ crosses the boundary more than once. A drawing of a cluster graph is *c-planar* if there are no edge crossings or edge-region crossings. A graph having a $c$-planar drawing is called *c-planar*. Notice that the planarity of the underlying graph does not imply the existence of a $c$-planar drawing of a cluster graph. Figure 2.2 taken from Feng, Cohen, and Eades (1995) shows a cluster graph whose underlying graph is planar. However, the cluster graph is not $c$-planar. This can be easily observed since the underlying graph is triconnected. The figure shows a drawing with two edge-region crossings.

## 2.6 Level Graphs

Let $G = (V, E)$ be a directed acyclic graph. A *leveling* of $G$ is a topological numbering of $G$ $\text{lev} : V \to \mathbb{Z}$ mapping the vertices of $G$ to integers such that $\text{lev}(v) \geq \text{lev}(u) + 1$ for all $(u, v) \in E$. $G$ is called a *level graph* if it has a leveling. If $\text{lev}(v) = j$, then $v$ is a *level-$j$ vertex*. Let $V^j = \text{lev}^{-1}(j)$ denote the set of level-$j$ vertices. Each $V^j$ is a *level* of $G$. If $G = (V, E)$ has a leveling with $k$ being the largest integer such that $V^k$ is not empty, $G$ is said to be a *$k$-level graph*. For a $k$-level graph $G$, we sometimes write $G = (V^1, V^2, \ldots, V^k; E)$.

An drawing of $G$ in the plane is a *level drawing* if the vertices of every $V^j$, $1 \leq j \leq k$, are placed on a horizontal line $l_j = \{(x, k - j) \mid x \in \mathbb{R}\}$, and every edge $(u, v) \in E$, $u \in V^i$, $v \in V^j$, $1 \leq i < j \leq k$, is drawn as a monotone decreasing curve between the lines $l_i$ and $l_j$. A level drawing of $G$ is called *level planar* if no two edges cross except at common endpoints. A level graph is *level planar* if it has a level planar drawing. A level graph $G$ obviously is level planar if and only if all its components are level planar. A graph that is not level planar is usually called *nonlevel planar*.

A level drawing of $G$ determines for every $V^j$, $1 \leq j \leq k$, a total order $\leq_j$ of the vertices of $V^j$, given by the left to right order of the vertices on $l_j$. A *level embedding* consists of a permutation of vertices of $V^j$ for every $j \in \{1, 2, \ldots, k\}$ with respect to a level drawing. Thus a level embedding is not given by the clockwise order of edges incident to every vertex, but by a permutation of the level-$j$ vertices for each level $j \in \{1, 2, \ldots, k\}$. A level embedding with respect to a level planar drawing is called *level planar*.




(a) A level graph. (b) A hierarchy.

Figure 2.3: Examples of proper level graphs. Sources are drawn black.

A level graph $G = (V, E)$ is said to be *proper* if every edge $e \in E$ connects only vertices belonging to consecutive levels. Figure 2.3 shows two proper level graphs. If a level graph is not proper, it must have an edge $e = (v, w) \in E$ such that $v \in V^i$ and $w \in V^j$ with $1 \leq i < j - 1 \leq k - 1$. Such an edge is called a *long edge* and it is said to be *traversing* the

levels $l$ with $i < l < j$. Any nonproper level graph can be transformed into a proper level graph by replacing every long edge by a path having a dummy vertex for every traversed level.

A $k$-level graph $G$ may have sinks and sources placed on various levels of the graph. A *hierarchy* is a level graph such that all sources belong to the first level $V^1$ of the graph. If $G$ is a hierarchy having more than one vertex in $V^1$ it is always possible to add a new subset $V^0$ with exactly one vertex connected to every vertex of $V^1$. Such a transformation does not modify the planarity properties of the given hierarchy. As a consequence, we consider only hierarchies with $|V^1| = 1$. Figure 2.3(b) shows a hierarchy.

# Chapter 3

# $PQ$-trees and Applications

The $PQ$-tree data structure has been developed by Booth and Lueker (1976) for solving the problem of finding permissible permutations of a set $U$. The permissible permutations are those in which certain subsets $S \subseteq U$ occur as consecutive subsequences. A $PQ$-tree represents a class of permissible permutations. As the elements of each new subset $S$ are constrained to appear together, the number of permissible permutations is reduced. The corresponding $PQ$-tree operation is called reduction with respect to $S$. Booth and Lueker (1976) presented an efficient algorithm for computing the reduction of a $PQ$-tree, and showed that its running time is linear in the size of the input.

Booth and Lueker (1976) mentioned that typical applications for $PQ$-trees are the consecutive ones property in matrices and the recognition of interval graphs. A $(0,1)$-matrix has the consecutive ones property if and only if its rows can be permuted such that in each column all of the ones are consecutive (see Fulkerson and Gross (1965)). A graph $G = (V, E)$ is an interval graph if and only if there is a one to one correspondence between its vertices and a set of intervals on the real line such that two vertices are adjacent if and only if the corresponding intervals have a nonempty intersection (see, e.g., Fulkerson and Gross (1965)). Booth and Lueker (1976) also noticed that the application of $PQ$-trees in the planarity test of Lempel, Even, and Cederbaum (1967) improves the efficiency of the original algorithm from quadratic to linear running time. The application of $PQ$-trees in the planarity test is probably the most well known one. Other applications in the field of Automatic Graph Drawing are for instance the computation of an embedding of a planar graph by Chiba, Nishizeki, Abe, and Ozawa (1985), and planarity testing and embedding of clustered graphs by Feng, Cohen, and Eades (1995).

This chapter introduces the data structure $PQ$-tree and the so called template matching algorithm that reduces a $PQ$-tree with respect to a subset $S \subseteq U$. After reviewing the planarity test of Lempel *et al.* (1967) and its modification using the $PQ$-trees, we examine the embedding algorithm of Chiba *et al.* (1985). The study of these algorithms is important for this work on testing for level planarity. The embedding of clustered graphs by Feng *et al.* (1995) is studied since it is a successful adaption of the planarity testing and embedding strategies of Booth and Lueker (1976) and Chiba *et al.* (1985).

The chapter closes by reviewing the computation of maximal planar subgraphs as an unsuccessful approach of applying $PQ$-trees in the field of Automatic Graph Drawing. A subgraph $G'$ of a simple graph $G = (V, E)$ is a maximal planar subgraph if for all edges $e \in G - G'$ the addition of $e$ to $G'$ destroys planarity. In 1981 Ozawa and Takahashi presented an algorithm for computing a maximal planar subgraph using $PQ$-trees. This algorithm was a modified version of the planarity test of Booth and Lueker (1976), and has been proven to be incorrect by Jayakumar, Thulasiraman, and Swamy (1986). In 1989 Jayakumar, Thulasiraman, and Swamy presented a new algorithm for computing a maximal planar subgraph, also using $PQ$-trees. However, this algorithm was still not correct, as has been shown by Kant (1992), who suggested profound modifications of the approach of Jayakumar *et al.* (1989). Leipert (1995) showed that Kant's approach fails as well. Finally, Jünger, Leipert, and Mutzel (1998a) discovered the existence of a substantial flaw in the main strategy that has been used in the approaches of Ozawa and Takahashi (1981), Jayakumar *et al.* (1989), and Kant (1992). This flaw suggests not to use $PQ$-trees at all for computing maximal planar subgraphs. Most parts of the last section in this chapter have been published by Jünger, Leipert, and Mutzel (1998a). However, it is very useful to examine the mistakes closely in preparation of the next chapter, were we review an erroneous approach of Heath and Pemmaraju (1995, 1996) for testing level planarity of level graphs.

$PQ$-trees have not only successfully been applied in Automatic Graph Drawing but also in various other fields such as physical mapping with end-probes in computational biology (see Christof, Jünger, Kecegioglu, Mutzel, and Reinelt (1997) and Christof (1997)) and finding violated comb inequalities when solving TSP instances by branch and cut (see Applegate, Bixby, Chvàtal, and Cook (1994)).

$PQ$-trees are rather difficult to implement, thus attempts have been made to simplify the data structure, e.g., Novick (1989) introduces generalized $PQ$-trees including a processor parallel algorithm for computing the generalized $PQ$-trees. Meidanis and Munuera (1996) and Meidanis, Porto, and Telles (1997) give a simple extension of the $PQ$-tree called the $PQR$-tree. However, their reduction algorithms for computing the set of permissible permutations need quadratic time, while the reduction algorithm of the original $PQ$-trees uses only linear time in the size of the input. The only simplification that preserves the linear time bound has been developed by Korte and Möhring (1988). However, these simplified $PQ$-trees have been constructed to deal specifically only with the recognition of interval graphs.

There are several algorithms restricted to special problems that do not use $PQ$-trees and that perform also in linear time. Hsu (1992) presented a linear time algorithm for testing for the consecutive ones property. Corneil, Kim, Natarajan, Olariu, and Sprague (1995), and de Figueiredo, Meidanis, and Mello (1995) showed how to solve various problems for interval graph recognition in linear time.

## 3.1 $PQ$-trees

In this section, an introduction to the data structure $PQ$-tree is given. The functionality of the data structure is very important for the following chapters. The $PQ$-tree and the template matching algorithm are therefore introduced in detail.

Let $U = \{a_1, a_2, \ldots, a_m\}$, $m \geq 1$, denote some finite set called the *universal set*. Let $\mathcal{S} = \{S_i \subseteq U \mid i = 1, 2, \ldots, n\}$ be a family of $n \geq 1$ subsets of $U$. Then all permutations of the elements of $U$ have to be found, where the elements of each subset $S_i$, $i = 1, 2, \ldots, n$, occur as a consecutive subsequence. These permutations of $U$ are called the *permissible permutations with respect to* $\mathcal{S}$ and we refer to them in general as the *permissible permutations*. Let

- $\Pi := \{\pi \mid \pi \text{ is a permutation of } U\}$ and
- $\Pi_{S_i} := \{\pi \in \Pi \mid \text{all elements of } S_i \text{ are consecutive within } \pi\}$ .

Computing the permissible permutations $\Pi_{\mathcal{S}}$ with respect to $\mathcal{S}$ will be performed by a function REDUCTION that starts on $\Pi$ as the permissible permutations and removes successively for every $S_i$, $i = 1, 2, \ldots, n$, those permutations, in which elements of $S_i$ are separated by elements of $U - S_i$.

$\Pi_{\mathcal{S}}$ **REDUCTION**$(U, \mathcal{S}, n)$
  begin
    $\Pi_{\mathcal{S}} := \Pi$;
    for $i = 1$ to $n$ do
      $\Pi_{\mathcal{S}} := \Pi_{\mathcal{S}} \cap \Pi_{S_i}$;
    return $\Pi_{\mathcal{S}}$;
  end.

The operation $\Pi_{\mathcal{S}} := \Pi_{\mathcal{S}} \cap \Pi_{S_i}$ is called a *reduction with respect to* $S_i$, and is performed by a function REDUCE. We now give the definition of a $PQ$-tree.

**Definition 3.1.** *The class of* $PQ$-trees *over a universal set* $U$ *is defined to be all rooted, ordered trees, whose leaves are elements of* $U$ *and whose internal nodes are distinguished as being either* $P$-nodes *or* $Q$-nodes. *A* $PQ$*-tree over* $U$ *is said to be* proper *if*

*(i) every element* $a_i \in U$ *appears precisely once as a leaf,*

*(ii) every* $P$*-node has at least two children, and*

*(iii) every* $Q$*-node has at least three children.*

From now on, only proper $PQ$-trees are considered. In illustrations, a $P$-node is drawn as a circle, a $Q$-node is drawn as a rectangle, and a leaf $a_i \in U$ is drawn as the element itself. The $PQ$-tree shown Fig. 3.1 is a tree over a set $U = \{A, B, C, D, E, F, G, H, I, J, K\}$.

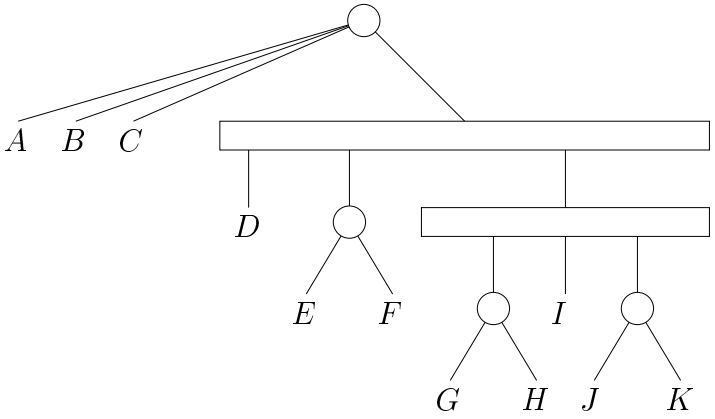



Figure 3.1: A $PQ$-tree over a set $U = \{A, B, C, D, E, F, G, H, I, J, K\}$.

The *frontier* of a $PQ$-tree $T$ is the sequence of all leaves of $T$ read from left to right, and the frontier of a node $X$, denoted by frontier($X$), is the sequence of its descendant leaves read from left to right. The frontier of the $PQ$-tree in Fig. 3.1 is $ABCDEFGHIJK$. The frontier of a $PQ$-tree is a permutation of the set $U$. We use the notion frontier($T$) and frontier($X$) also to denote the set of elements in frontier($T$) and frontier($X$), respectively, its meaning being evident by context. An *equivalence transformation* specifies a legal reordering of the nodes within a $PQ$-tree. The only legal equivalence transformations are

(i) any permutation of the children of a $P$-node, and

(ii) the reverse permutation of the children of a $Q$-node.

Two $PQ$-trees $T$ and $T'$ are *equivalent* if and only if their underlying trees are equal and $T$ can be transformed into $T'$ by a sequence of equivalence transformations. The equivalence of two trees is denoted $T \equiv T'$. The set of *consistent permutations* of a $PQ$-tree is the set of all frontiers that can be obtained by a sequence of equivalence transformations and is denoted by

$$\text{PERM}(T) \quad = \quad \{\text{frontier}(T') \mid T' \equiv T\} \quad .$$

A permutation $\pi$ is said to be *consistent* with a $PQ$-tree $T$, if $\pi \in \text{PERM}(T)$.

Figure 3.2 shows a $PQ$-tree that is equivalent to the $PQ$-tree of Fig. 3.1 yielding the frontier $BGHIKJFEDCA$. There are 768 different $PQ$-trees in the equivalence class of the tree of Fig. 3.1, thus the tree represents 768 different permutations of $U = \{A, B, C, D, E, F, G, H, I, J, K\}$ (see Booth and Lueker (1976)). With the following theorem, Booth and Lueker (1976) showed that for every finite set $U$ and every family of subsets of $U$, there exists an equivalence class of $PQ$-trees representing all permissible permutations of $U$ with respect to $\mathcal{S}$.

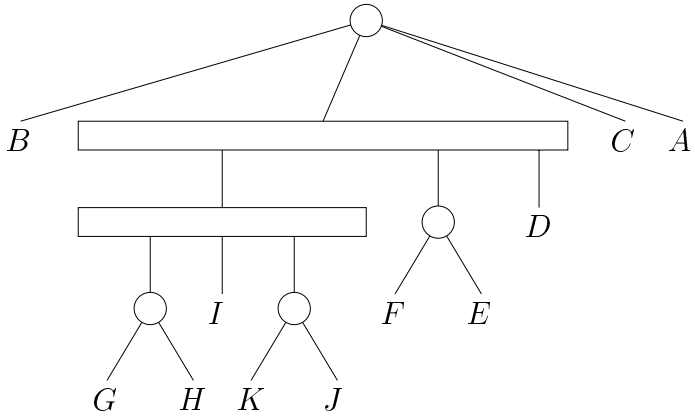



Figure 3.2: A $PQ$-tree equivalent to the one shown in Fig. 3.1.

**Theorem 3.2 (Booth and Lueker (1976)).** *Let $U$ be a finite set and let $S_i \subseteq U$, $1 \leq i \leq n$, be subsets of $U$. There exists a $PQ$-tree $T$ such that*

$$\mathrm{PERM}(T) \;=\; \bigcap_{i=1}^{n} \Pi_{S_i} \;.$$

**Remark 3.3.** *If a $PQ$-tree $T$ over a set $U$ is given, there exist $n \geq 1$ subsets $S_i \subseteq U$ such that $\mathrm{PERM}(T) = \bigcap_{i=1}^{n} \Pi_{S_i}$. However, the subsets $S_i$ are not unique.*

A *universal tree* $T$ over a set $U$ is a $PQ$-tree over $U$ consisting exactly of a $P$-node $X$ as root and the set $U$ as children of $X$, if $|U| \geq 2$. If $|U| = 1$, then $T$ consist of a single leaf. Given any $PQ$-tree $T$ over a finite set $U$ and a subset $S \subseteq U$, the function $\mathrm{REDUCE}(T, S)$ computes a $PQ$-tree $T'$ such that $\mathrm{PERM}(T') = \mathrm{PERM}(T) \cap \Pi_S$. Using the function REDUCE, the function REDUCTION can be reformulated as follows.

$PQ$-tree **REDUCTION**$(U, \mathcal{S}, n)$
  begin
    construct the universal $PQ$-tree $T$ of $U$;
    for $i = 1$ to $n$ do
      $T := REDUCE(T, S_i)$;
    return $T$;
  end.

Let $T$ be a $PQ$-tree over $U$ and $S \subseteq U$. A node $X$ in $T$ is said to be *full* if $\mathrm{frontier}(X) \subseteq S$. A node $X$ is said to be *empty* if $\mathrm{frontier}(X) \cap S = \emptyset$. A node $X$ is *partial* if it is neither empty nor full. Nodes are said to be *pertinent* if they are either full or partial. The *pertinent subtree of $T$ with respect to $S$* is the subtree of minimum height whose frontier contains all elements of $S$. The pertinent subtree and its root are unique.

The function REDUCE applies a sequence of *templates* to the nodes of a $PQ$-tree starting at the leaves, and proceeding upwards until the root of the pertinent subtree is reached. Each

template has a *pattern* and a *replacement*. If a node matches the pattern of a template, the pattern is replaced within the tree by the replacement of the template. The return value of REDUCE is a new $PQ$-tree. It is the *null tree*, a tree with no nodes at all, if the original tree could not be reduced for the specified set $S$. If a null tree is returned, the set of permissible permutations on the set $U$ is empty and the null tree represents an empty set of permutations. Therefore it is convenient to denote the null tree by $\emptyset$.

Each template specifies a local change within the tree. Only the node $X$ that has to be matched and its children are altered. The patterns to which nodes are matched depend upon the set $S$ and the frontier of the subtree rooted at the particular node $X$. The matched pattern is selected by examining the node $X$ and its children after the children themselves have been matched. Depending on the situation in the frontier of $X$ the node is labeled indicating whether $X$ is empty, full, or partial. This bottom-up strategy ensures that all information on the situation in the frontier of the children of $X$ is available when processing $X$.

Figures 3.3 – 3.10 illustrate the template matchings. A pattern at the left hand side is to be transformed into a pattern at the right hand side. A full node or a full subtree is hatched, and a partial $Q$-node that roots a pertinent subtree is hatched partially. We use a triangle for symbolizing a subtree. A subtree is either full or empty, so its precise form has no effect on the templates.

Let $X$ be the node in a $PQ$-tree $T$ that has to be matched to a template. If $X$ is a $P$-node, and all its children are empty, template P0 (see Fig. 3.3) has to be applied, labeling $X$ as empty. If $X$ is a $P$-node and all its children are full, template P1 (see also Fig. 3.3) has to be applied, labeling $X$ as full.

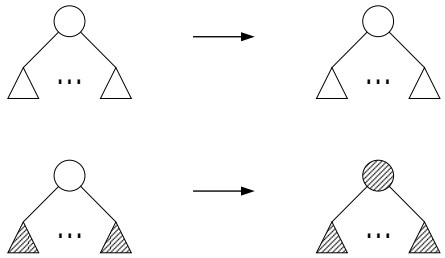


Figure 3.3: Template P0 and template P1.

If $X$ is a $P$-node, at least one child of $X$ is empty, at least one child of $X$ is full, and $X$ is the root of the pertinent subtree, template P2 shown in Fig. 3.4 has to be applied. The node $X$ is left unlabeled since the reduction of $T$ is complete.

If $X$ is a $P$-node, at least one child of $X$ is empty, at least one child of $X$ is full, and $X$ is not the root of the pertinent subtree, template P3 shown in Fig. 3.5 has to be applied. The $Q$-node that replaces $X$ is labeled *singly partial*.

If $X$ is a $P$-node, at least one child of $X$ is full, exactly one child of $X$ is partial, and $X$ is the root of the pertinent subtree, template P4 shown in Fig. 3.6 has to be applied. The

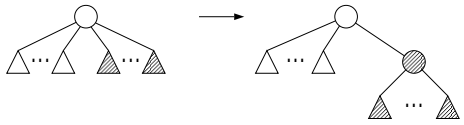


Figure 3.4: Template P2.

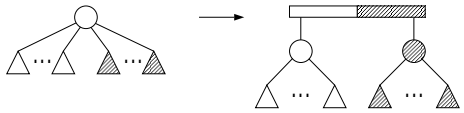


Figure 3.5: Template P3.

bottom up strategy in the application of the template matchings ensures that the partial child is a $Q$-node labeled singly partial. The node $X$ is left unlabeled since the reduction of $T$ is complete.

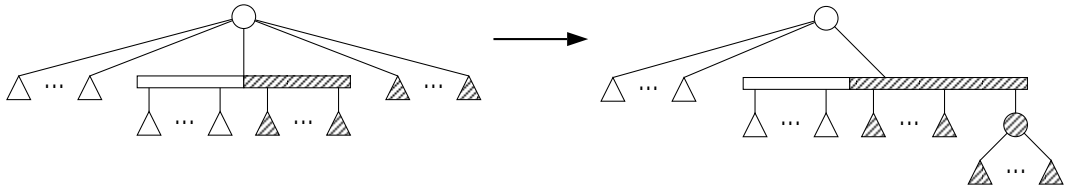


Figure 3.6: Template P4.

If $X$ is a $P$-node, at least one child of $X$ is full or empty, exactly one child of $X$ is partial, and $X$ is not the root of the pertinent subtree, template P5 shown in Fig. 3.7 has to be applied. The $Q$-node that replaces $X$ has been labeled singly partial before.

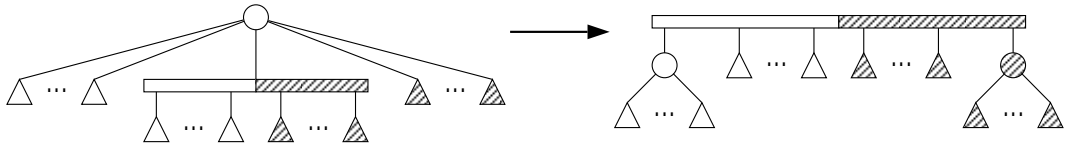


Figure 3.7: Template P5.

If a $P$-node $X$ has at least two children, and exactly two of them are singly partial, template P6 shown in Fig. 3.8 has to be applied. If this case applies, $X$ has to be the root of the pertinent subtree. Otherwise, $T$ cannot be reduced with respect to $S$. Thus $X$ is left unlabeled. The partial child of $X$ is said to be *doubly partial*.

The shown template matchings leave out a number of cases that can be adapted straightforward to existing templates. Consider for instance a $P$-node $X$ that is not the root of

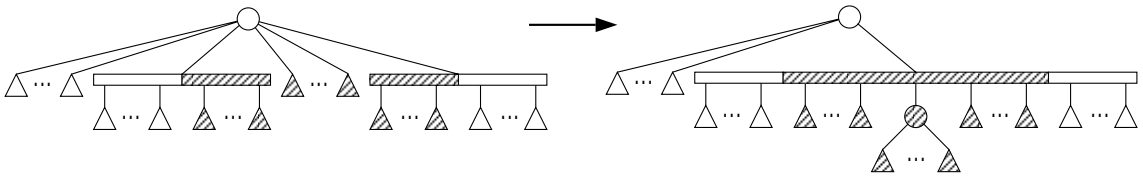


Figure 3.8: Template P6.

the pertinent subtree, having only full children, and exactly one singly partial child. Then an apropriate modification of template P5 handles this case.

$Q$-nodes also have a number of different templates. The cases when all children are labeled identically full or empty are taken care of by the templates Q0 and Q1 that are analogous to templates P0 and P1.

A $Q$-node $X$ is singly partial, if it has at most one singly partial child, and if the left to right order of the children is as shown in Fig. 3.9. After the application of template Q2, the node $X$ is labeled singly partial.

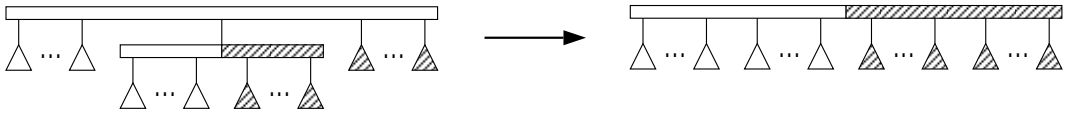


Figure 3.9: Template Q2.

A $Q$-node $X$ is *doubly partial*, if it has at most two singly partial children, and if the left to right order of the children is as shown in Fig. 3.10. After the application of template Q3, the node $X$ is labeled doubly partial. As for the case of tempalte P6, $X$ has to be the root of the pertinent subtree. Otherwise $T$ is not reducible with respect to $S$.

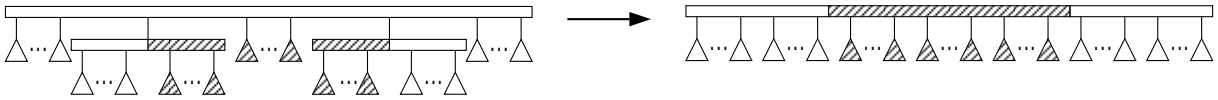


Figure 3.10: Template Q3.

Booth and Lueker (1976) achieve an efficient implementation of the template matching algorithm by combining two strategies,

(i) scanning only the pertinent nodes in the pertinent subtree, and

(ii) keeping parent pointers only for $P$-nodes and endmost children of $Q$-nodes.

Reducing a $PQ$-tree $T$ over $U$ with respect to $S \subseteq U$ is then performed by a two phase algorithm. The first phase, called BUBBLE, ensures that every pertinent node in the pertinent subtree has a valid parent pointer. If the first phase reveals pertinent nodes that do not have a valid parent pointer, we either have that template Q3 applies to the root of the pertinent subtree (none of the pertinent children of the pertinent root $X$ is an endmost child of $X$, and therefore none of these children has a valid parent pointer), or the tree is not reducible with respect to $S$. The second phase applies the templates to the pertinent nodes, starting at the pertinent leaves, proceeding the tree upwards to the root, processing each node after all its children have been processed. Using this strategy, it is possible to show the following nontrivial theorem.

**Theorem 3.4 (Booth and Lueker (1976)).** *The data structure $PQ$-tree and the template matchings can be implemented such that the class of permutations in which the elements of each set $S_i$ of a family $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ of subsets of $U$ occur as a consecutive sequence can be computed in $\mathcal{O}(|U| + n + \sum_{i=1}^{n} |S_i|)$ time.*

One result that is achieved in the proof of Theorem 3.4 is the following corollary that is needed later in Chapter 5 when proving the correctness of a level planar embedding algorithm.

**Corollary 3.5 (Booth and Lueker (1976)).** *Let $X$ be a child of a $Q$-node $Y$. Throughout the template matching algorithm $X$ remains a child of a $Q$-node.*

## 3.2 Planarity Testing

A simple planarity testing algorithm based on a divide and conquer strategy has been presented by Auslander and Parter (1961). Goldstein (1963) corrected this algorithm and Shirey (1969) showed how it can be implemented such that it needs $\mathcal{O}(n^3)$ steps. Using a depth first search approach, Hopcroft and Tarjan (1971) developed an $\mathcal{O}(n \log n)$ time planarity test based on the ideas of Auslander and Parter (1961). Tarjan (1971) and Hopcroft and Tarjan (1974) later improved their own algorithm to require only linear time. The algorithm of Hopcroft and Tarjan (1974) searches for a cycle $C$ in a given graph $G$ such that the removal of $C$ disconnects the graph. This is done recursively for all components of $G - C$. If every component of $G - C$ is planar, the algorithm checks if the components of $G - C$ can be arranged around $C$ such that $G$ is planar. Due to its concept, the algorithm of Hopcroft and Tarjan (1974) is called a "path addition" algorithm.

A "vertex addition" approach has been applied in the planarity test of Lempel, Even, and Cederbaum (1967). They suggested to start with a subgraph of $G$ that is induced by a single vertex. This subgraph is trivially planar. Lempel *et al.* then construct a sequence of induced subgraphs of $G$ by adding successively all vertices of $G$, and testing in every step if the induced subgraph is still planar. Tarjan (1969) pointed out that this approach leads to an $\mathcal{O}(n^2)$ algorithm. Booth and Lueker (1976) showed that the usage of $PQ$-trees improves

the algorithm of Lempel *et al.* (1967) to a linear time algorithm. Another approach for testing planarity has been presented by de Fraysseix and Rosenstiehl (1982).

Since the "vertex addition" algorithm of Lempel *et al.* (1967) using $PQ$-trees is essential for this work, a brief description of it is presented in this section. Let $G = (V, E)$ be a simple graph with $n$ vertices and $m$ edges. A graph is obviously planar if and only if its biconnected components are planar (see, e.g., Harary (1969)). We therefore assume that $G$ is biconnected. The biconnected components can be computed in $\mathcal{O}(n + m)$ time according to Aho, Hopcroft, and Ullman (1974). Furthermore, we may assume by the result of Corollary 2.2 that $m \in \mathcal{O}(n)$.

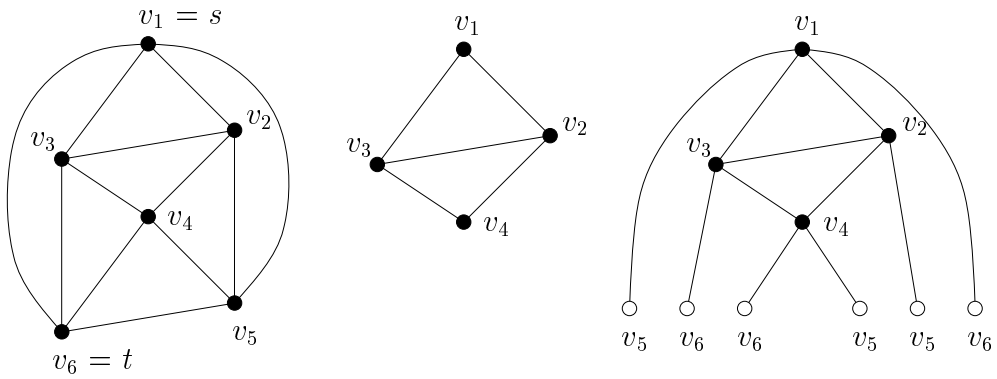
The planarity testing algorithm of Lempel, Even, and Cederbaum (1967) first labels the vertices of $G$ as $v_1, v_2, \ldots, v_n$ using an $st$-numbering. A numbering of the vertices of $G$ by $1, 2, \ldots, n$ is an *$st$-numbering* if the vertices $v_1$ and $v_n$ are adjacent and each other vertex $v_j$ is adjacent to two vertices $v_i$ and $v_l$ such that $i < j < l$. The vertex $v_1$ is denoted by $s$ and the vertex $v_n$ is denoted by $t$. An $st$-numbering can be computed in $\mathcal{O}(n)$ time as has been shown by Even and Tarjan (1976). The $st$-numbering induces an orientation of the graph, in which every edge is directed from the incident vertex with the lower $st$-number towards the incident vertex with the higher $st$-number. Thus it is convenient to say that an edge $(v_i, v_j)$ with $i < j$ is an incoming edge of $v_j$ and an outgoing edge of $v_i$.

For the following, let us suppose that we have a fixed $st$-numbering of $G$. For $1 \leq i \leq n$, let $G_i$ denote the subgraph of $G$ induced by the vertex set $V_i := \{v_1, v_2, \ldots, v_i\}$. The strategy of Lempel *et al.* (1967) is to start with $G_1$ induced by $\{s\} \subset V$ and to proceed by adding vertices to the subgraph, constructing the sequence of subgraphs $G_i$, for $i = 1, 2, \ldots, n$. When adding a vertex $v_{i+1}$ to $G_i$ in order to construct $G_{i+1}$, it is tested if this operation preserves planarity of the induced subgraph. If the test fails, the graph $G$ is obviously not planar.

For $1 \leq i \leq n$, let $G_i'$ be the graph arising from $G_i$ as follows: For each edge $e = (u, v)$, where $u \in V_i$ and $v \in V - V_i$, we introduce a *virtual vertex* $v_e$ with label $v$ and a *virtual edge* $(u, v_e)$. Let $B_i$ be a planar embedding of $G_i'$ such that all virtual vertices are placed on the outer face. Such a $B_i$ is called a *bush form* of $G_i'$. It has been shown by Lempel *et al.* (1967) that $G$ is planar if and only if for every $1 \leq i \leq n-1$ and for every bush form $B_i$ of $G_i'$ there exists a bush form $B_i'$ equivalent to $B_i$ such that all virtual vertices labeled $v_{i+1}$ appear consecutively in $B_i'$. Figure 3.11 shows an example taken from Chiba and Nishizeki (1988) showing a graph $G$ with an $st$-numbering, a graph $G_i$ and a bush form $B_i$ for $i = 4$. The following lemma implies that every planar $st$-numbered graph $G$ has a bush form $B_i$ for $1 \leq i \leq n - 1$.

**Lemma 3.6 (Even (1979)).** *Let $G = (V, E)$ be a planar graph with an $st$-numbering and let $1 \leq i \leq n$. If the edge $(s, t)$ is drawn on the boundary of the outer face in a planar drawing of $G$, then all vertices and edges of $G - G_i$ are drawn in the outer face of $G_i$.*

Given a bush form $B_i$, we construct a new bush form $B_i'$ (that is, a new planar embedding of $G_i'$ with all virtual vertices drawn on a line in the outer face of $G_i$) by permuting for every




(a) $st$-graph (b) $G_4$ (c) bush form $B_4$

Figure 3.11: An example of an $st$-numbered graph $G$, a $G_i$ with $i = 4$ and its bush form. The virtual vertices in (c) are not filled.

cut vertex $v \in V_i$ the components of $G'_i - \{v\}$ around $v$, and by reversing (flipping over) the biconnected components. The key idea of the vertex addition algorithm is to reduce the planarity testing of $G_{i+1}$ to a problem that asks for permutations and reversions to make all virtual vertices labeled $v_{i+1}$ occupy consecutive positions. If such permutations and reversions can be found, the vertex $v_{i+1}$ obviously can be introduced to $G_i$ without destroying planarity. Figure 3.12 shows a bush form $B_{11}$ and an equivalent bush form $B'_{11}$. The virtual vertices labeled $v_{12}$ occupy consecutive positions in $B'_{11}$. The following nontrivial lemma guarantees that the transformation is possible.

**Lemma 3.7 (Lempel *et al.* (1967)).** *Let $B_i$ be any bush form of a subgraph $G_i$ of a planar st-numbered graph $G$. Then there exists a sequence of permutations of components of $G_i - \{v\}$ around cut vertices $v \in V_i$, and reversions of biconnected components to make all the virtual vertices labeled $v_{i+1}$ appear consecutively on a horizontal line.*

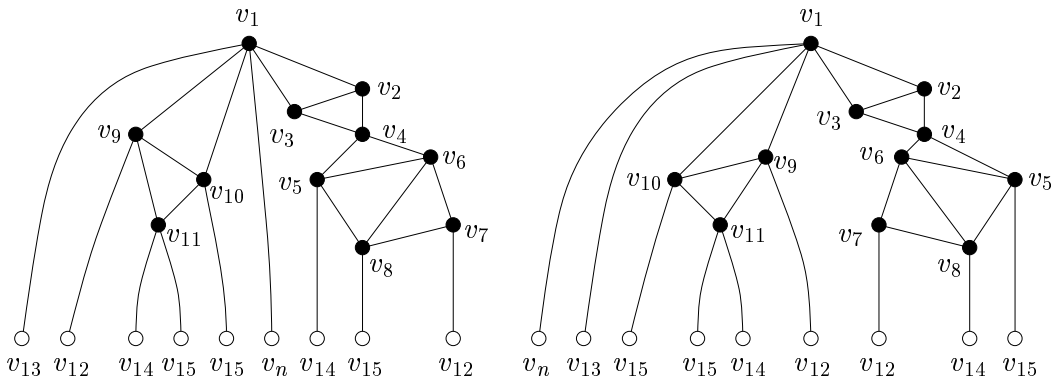
Computing the permutations and reversions of $B_i$ such that all virtual vertices labeled $v_{i+1}$ appear consecutively can be done efficiently using the $PQ$-tree data structure. While the original algorithm of Lempel *et al.* (1967) needs $\mathcal{O}(n^2)$ steps (see Tarjan (1969)), the usage of $PQ$-trees yields an $\mathcal{O}(n)$ time algorithm. For every bush form $B_i$, a $PQ$-tree $T_i$ that represents all possible (with respect to a plane embedding) permutations of the virtual vertices on the horizontal line is conceived by introducing

(i) a leaf for every virtual edge of $B_i$,

(ii) a $P$-node for every cut vertex in $B_i$,

(iii) a $Q$-node for every biconnected component in $B_i$,

(iv) rooting $T_i$ at the node corresponding to the vertex $s$.




(a) $B_{11}$ (b) $B'_{11}$

Figure 3.12: Two equivalent bush forms $B_{11}$ and $B'_{11}$.

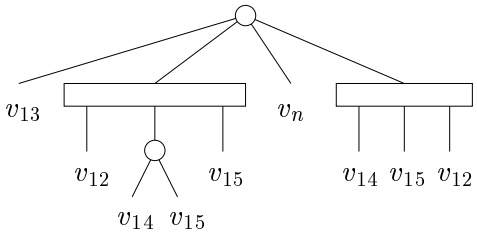


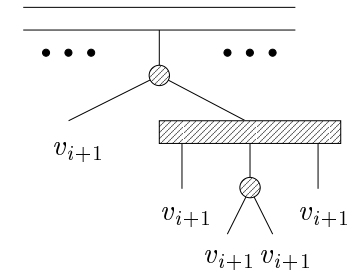

Figure 3.13: A $PQ$-tree corresponding to bush form $B_{11}$ of Fig. 3.12.

Figure 3.13 shows a $PQ$-tree corresponding to the bush form $B_{11}$ of Fig. 3.12.

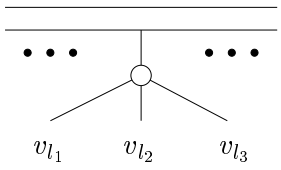
Rather than constructing a $PQ$-tree for every bush form $B_i$, the modified planarity test starts with a universal $PQ$-tree $T_1$ for the bush form $B_1$. For every $i = 1, 2, \ldots, n-1$ the tree $T_i$ is then reduced with respect to the leaves that correspond to the incoming edges of the vertex $v_{i+1}$. If the reduction was successful, the subgraph $G_{i+1}$ is planar. The leaves in $T_i$ corresponding to virtual edges incident to a vertex $v_j$, $i < j \leq n$, are called leaves *labeled* $v_j$. A reduction of $T_i$ with respect to the leaves labeled $v_{i+1}$ is called a *reduction of* $T_i$ *with respect to* $v_{i+1}$.

After a successful reduction with respect to the leaves labeled $v_{i+1}$, the $PQ$-tree $T_{i+1}$ is constructed from $T_i$, using a function REPLACE. The full nodes in the pertinent subtree with respect to $v_{i+1}$ are replaced by a $P$-node, whose children are the leaves corresponding to the outgoing edges of the vertex $v_{i+1}$ in $G$. In case that the vertex $v_{i+1}$ has just one outgoing edge $e$, we do not introduce a new $P$-node but replace the full nodes by a single leaf corresponding to $e$. The parameter $S_{\text{red}}$ in the function REPLACE denotes the set of incoming edges of $v_{i+1}$ and the parameter $S_{\text{new}}$ denotes the set of outgoing edges of a vertex $v_{i+1}$. Since the vertices of the graph $G$ are reduced according to an $st$-numbering, the set $S_{\text{new}}$ is not empty.

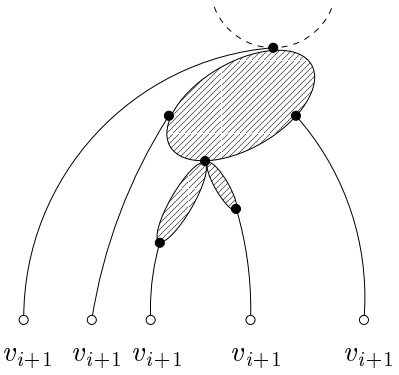
void **REPLACE**($S_{\text{red}}$,$S_{\text{new}}$)
  begin
    if $|S_{\text{new}}| \geq 2$ then
      let $X$ be a new $P$-node;
      add the elements of $S_{\text{new}}$ as leaves to $X$;
    else if $|S_{\text{new}}| = 1$ then
      let $X$ be a leaf corresponding to the only element of $S_{\text{new}}$;
    if the root $R$ of the pertinent subtree is full then
      replace the pertinent subtree with respect to $S_{\text{red}}$ by $X$;
    else
      replace the (consecutive) sequence of full children of $R$ by $X$;
  end.



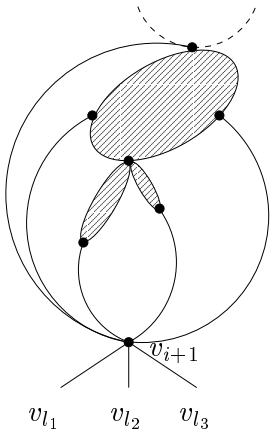

(a) Pertinent subtree with respect to $v_{i+1}$.




(b) Replacement of the pertinent subtree.




(c) Corresponding part of $B_i$.




(d) Construction of $B_{i+1}$.

Figure 3.14: Illustration of function REPLACE.

Figure 3.14 illustrates the replacement step. In Fig. 3.14(a) a pertinent subtree with respect to leaves labeled $v_{i+1}$ is shown. The root of the pertinent subtree is full. The corresponding

part of the bush form is shown in Fig. 3.14(c). By identifying all virtual vertices labeled $v_{i+1}$, and adding virtual edges $S_{\text{new}} = \{(v_{i+1}, v_{l_1}), (v_{i+1}, v_{l_2}), (v_{i+1}, v_{l_3})\}$ the bush form $B_{i+1}$ as shown in Fig. 3.14(d) is constructed. The corresponding new $PQ$-tree of the bush form $B_{i+1}$ is shown in Fig. 3.14(b).

The function REPLACE is used in subsequent algorithms presented in this work. For a correct application of REPLACE it is sufficient that the $PQ$-tree has been reduced with respect to the set $S_{\text{red}}$ before calling the function REPLACE.

The following theorem is based on the results of Lemma 3.7, Theorem 3.4, and Corollary 2.2.

**Theorem 3.8 (Booth and Lueker (1976)).** *Given a graph $G = (V, E)$, the described planarity test requires $\mathcal{O}(n)$ time.*

Let $T_i$ be a $PQ$-tree corresponding to a bush form $B_i$, and let $X$ be a $Q$-node in $T_i$. As mentioned before, the $Q$-node corresponds to a biconnected component $b$ in $B_i$. The children of $X$ each correspond to a cut vertex on the border of the outer face of $b$. If $X$ is not the root (which is always the case for the planarity test, but not for some other algorithms that are discussed later in this work), then there exists an extra cut vertex on the border of the outer face of $b$ that separates the subgraph $G'$ induced by the subtree rooted at $X$ from $G - G'$. This cut vertex is called the *connective cut vertex* of $b$. Figure 3.15 gives an example of a $Q$-node having four children. Every child of the $Q$-node corresponds to one of the four cut vertices $c_1, c_2, c_3, c_4$. Since the $Q$-node is not the root of the $PQ$-tree, there exists a cut vertex $c_0$ associated with the parent of the $Q$-node. The vertex $c_0$ is the connective cut vertex. The connective cut vertex of a $P$-node $Y$ is the cut vertex corresponding to $Y$.

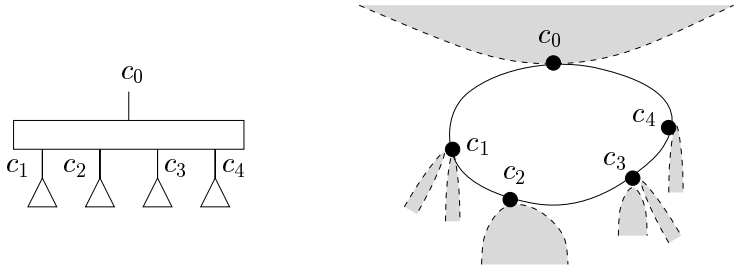



Figure 3.15: The biconnected component corresponding to the $Q$-node has five cut vertices on the border of its outer face. The vertex $c_0$ is its connective cut vertex.

## 3.3 Embedding Planar Graphs

Most applications require not only testing the planarity of a graph but also embedding (or drawing) a planar graph in the plane. For both, the "path addition" and the "vertex

addition" planarity test, modified versions have been developed to compute a planar embedding of a planar graph. An embedding algorithm based on the Hopcroft and Tarjan (1974) algorithm has been presented by Mutzel (1992), and Mehlhorn and Mutzel (1996).

A modification of the Booth and Lueker planarity testing algorithm to obtain a planar embedding algorithm for planar graphs has been given by Chiba, Nishizeki, Abe, and Ozawa (1985). A brief description of this algorithm is given in this section. We may assume for the rest of this section that the graph $G$ that has to be embedded is planar.

The embedding algorithm is based on the observation that a naive embedding algorithm can be obtained easily as follows.

1. Write down the partial embedding of the bush form $B_1$.
2. With each reduction of the $PQ$-tree, rewrite the adjacency lists of the bush form.

Clearly, the final bush from is an embedding of the graph. However, the algorithm needs $\mathcal{O}(n^2)$ time since it takes $\mathcal{O}(n)$ time per reduction of the $PQ$-tree to update the adjacency lists of the bush form. As mentioned in the previous section, an $st$-numbering induces an orientation of $G$. Let $G_{st}$ be the graph of $G$ such that every edge is oriented from the lower numbered vertex to the higher numbered vertex. Since $G$ is planar, the graph $G_{st}$ is an upward planar $st$-graph. Chiba *et al.* (1985) (and also independently Tamassia and Tollis (1986), and Rosenstiehl and Tarjan (1986)) observed the following interesting characteristics of planar $st$-graphs.

**Lemma 3.9 (Chiba *et al.* (1985)).** *Consider a planar embedding of a graph $G$ ($st$-numbered by $v_1, v_2, \ldots, v_n$) by the naive embedding algorithm. For every $1 \leq j \leq n$, all neighbors $v_i$ of $v_j$ with $j < i$ appear consecutively around $v_j$ as do all neighbors $v_i$ with $j > i$.*

The embedding algorithm consists of two stages:

(i) Construct an upward planar embedding $\mathcal{E}_{st}$ of $G_{st}$.

(ii) Construct the entire planar embedding $\mathcal{E}$ of $G$ from the upward planar embedding $\mathcal{E}_{st}$ of $G_{st}$.

First, we consider how to compute an upward planar embedding. Let the adjacency lists $adj_{st}(v_i)$ represent the graph $G_{st}$. During the algorithm, a $PQ$-tree $T_i$ is reduced with respect to the leaves labeled $v_{i+1}$ constructing a $PQ$-tree $T_i'$. Let $X_{i+1}$ be the root of the pertinent subtree of $T_i'$ with respect to $v_{i+1}$. The adjacency list $adj_{st}(v_{i+1})$ then is obtained by scanning the leaves labeled $v_{i+1}$ in $T_i'$ from left to right or vice versa. If $adj_{st}(v_{i+1})$ is correctly determined in step $i$, then by counting the number of subsequent reversions of the node $X_{i+1}$ the direction of $adj_{st}(v_{i+1})$ is corrected if the number of reversion is odd.

However, a straightforward method for determining the direction of $adj_{st}(v_{i+1})$ in $T_i'$ would require $\mathcal{O}(n)$ time. Rather than determining the direction, the algorithm of Chiba *et al.*

(1985) adds a new special node to the tree as a child to $X_{i+1}$. The new node is called *direction indicator* and is labeled $v_{i+1}$. The indicator covers two tasks.

(i) A direction indicator labeled $v_{i+1}$ traces subsequent reversions of the incoming edges of $v_{i+1}$.

(ii) A direction indicator labeled $v_{i+1}$ transfers the relative direction of the incoming edges of the node $v_{i+1}$ to its siblings.

In subsequent reductions of $PQ$-trees $T_j$, $i+1 \leq j < n$, the presence of the direction indicator is ignored. In the vertex addition step of $v_{j+1}$ the sequence of pertinent leaves labeled $v_{j+1}$ and all direction indicators contained within the pertinent sequence are scanned. The incoming edges of $v_{j+1}$ corresponding to the pertinent leaves and the direction indicators within the pertinent sequence are stored in $adj_{st}(v_{j+1})$ in the order they have been detected. When the reduction of the $PQ$-tree $T_{n-1}$ with respect to the vertex $v_n$ is complete, the adjacency lists are scanned in reverse order. When detecting a direction indicator of a vertex $v_i$ in the adjacency list of a vertex $v_j$ with $i < j$, and the indicator indicates an odd number of reversions of the $v_i$, the list $adj_{st}(v_i)$ is reversed.

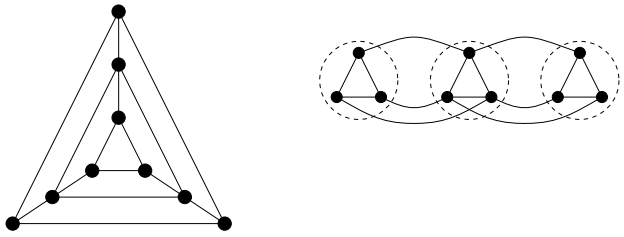
Chiba *et al.* (1985) showed how to implement this strategy, using only $\mathcal{O}(n)$ time to compute the upward embedding $\mathcal{E}_{st}$ of $G_{st}$. Finally the upward embedding is extended to a planar embedding by applying a simple depth-first-search starting at $v_n$ and adding a vertex $v$ to the top of the adjacency list $adj_{st}(w)$ when the directed edge $(w, v)$ is visited in $adj_{st}(v)$. Due to the characteristics of the $st$-numbering, Chiba *et al.* (1985) proved that this leads to a correct planar embedding, yielding the following theorem.

**Theorem 3.10 (Chiba *et al.* (1985)).** *There is a linear time algorithm to test whether a graph $G = (V, E)$ is planar, and if so, it outputs a planar embedding.*

## 3.4 Planarity Testing and Embedding of Cluster Graphs

It appears that $c$-planarity testing is not a trivial extension of planarity testing of classical graphs. Consider for example the cluster graph in Fig. 3.16 taken from Feng, Cohen, and Eades (1995). Suppose that the vertices on the three triangles belong to three separate clusters. It is obvious that the graph is planar, and the graph obtained by collapsing any cluster into a vertex is also planar. However, this cluster graph is not $c$-planar.

A first algorithm for testing a connected cluster graph for $c$-planarity in quadratic time has been presented by Lengauer (1989) using a sophisticated approach that decomposes the graph. Feng *et al.* (1995) obviously were not aware of the algorithm presented by Lengauer (1989), possibly because Lengauer used the term "hierarchy" for cluster graphs which may cause confusion in the graph drawing community. Feng *et al.* describe an algorithm



(a) (b)

Figure 3.16: A planar but not $c$-planar clustered graph.

for testing $c$-planarity and finding a $c$-planar embedding of connected cluster graphs in $\mathcal{O}(n^2)$ time using the $PQ$-tree data structure. Very recently, Dahlhaus (1998a) described an algorithm that tests a connected cluster graph in $\mathcal{O}(n)$ time for $c$-planarity, and also allows to compute a $c$-planar embedding in $\mathcal{O}(n)$ time. The approach of Dahlhaus (1998a) is based on the strategy of Lengauer (1989), also decomposing the graph. We mention that the decomposition of the graph is quite similar to the decomposition of graphs performed by $SPQR$-trees of Di Battista and Tamassia (1996). Up to now, no implementation of the linear time $c$-planarity test exists as Dahlhaus (1998b) confirms, while implementations of the $c$-planarity test using $PQ$-trees are available (see, e.g., Liebel (1998)).

The algorithm of Feng *et al.* (1995) is based on the following theorem giving a necessary and sufficient condition for the $c$-planarity of connected graphs.

**Theorem 3.11 (Feng *et al.* (1995)).** *A connected cluster graph $C = (G, T)$ is $c$-planar if and only if $G$ is planar, and there exists a planar drawing $\mathcal{D}$ of $G$, such that for each node $\nu$ of $T$, all vertices and edges of $G - G(\nu)$ are in the outer face of the drawing of $G(\nu)$.*

Feng *et al.* (1995) gave also a characterization of $c$-planarity for general cluster graphs.

**Theorem 3.12 (Feng *et al.* (1995)).** *A cluster graph $C = (G, T)$ is $c$-planar if and only if it is a subcluster graph of a connected and $c$-planar cluster graph.*

Unfortunately, it is not known how to extend a nonconnected clustered graph to a connected one without destroying $c$-planarity in polynomial time. In fact, while testing for $c$-planarity has been solved for connected cluster graphs, the problem whether an unconnected cluster graph can be tested for $c$-planarity in polynomial time is still unsolved.

The algorithm of Feng *et al.* (1995) is based on Theorem 3.11. Given a connected cluster graph $C = (G, T)$, it is tested whether there is a planar embedding of $G$ such that for each node $\nu$ of $T$ the subgraph $G - G(\nu)$ is embedded in the outer face of $G(\nu)$. The subgraphs induced by the clusters are embedded one by one, following a traversal of $T$

from bottom to top. For each node $\nu$ of $T$ the subcluster graph $G(\nu)$ is tested whether it has any planar embedding that satisfies the conditions of Theorem 3.11 for $C(\nu)$ (and if so all these embeddings are stored implicitly). If the algorithm proceeds successfully to the root cluster, and such an embedding exists for the root of $T$, then the cluster graph is $c$-planar. In order to construct a $c$-planar embedding of $G$ a traversal of $T$ from top to bottom is performed, embedding recursively for each cluster the corresponding subgraph using the information gathered while testing the graph for $c$-planarity.

We now give a brief description of the $c$-planarity test. For a node $\nu$ of $T$ with children $\mu_1, \mu_2, \ldots, \mu_k$ we try to find an embedding of the subgraph $G(\nu)$ satisfying the conditions of Theorem 3.11 by combining the possible embeddings of each child cluster $\mu_i$, $i = 1, 2, \ldots, k$, that are found recursively. If the construction of such an embedding terminates successfully, the subgraph $G(\nu)$ is replaced in $G$ by a *representative graph*. The objective of a representative graph is to allow only those embeddings that mirror all possible orderings of edges that are incident to cluster $\nu$ in a $c$-planar embedding. The replacement of $G(\nu)$ by its representative graph is done recursively for every node $\nu$ of the tree $T$. Thus the graph $G$ is changed at every node $\nu$ of the inclusion tree.

At cluster $\nu$, the subgraph $G(\nu)$ is not only tested for planarity but it is also tested if the edges that are incident to cluster $\nu$ can be drawn in the outer face of $G(\nu)$. A graph $G'(\nu)$ is constructed by adding virtual edges to $G(\nu)$, each virtual edge corresponding to an edge incident to cluster $\nu$. Every virtual edge is equipped with a virtual vertex and $G'(\nu)$ is the graph that results from connecting all virtual edges to a single vertex $t_\nu$.

Feng *et al.* (1995) showed that all virtual edges are contained in the same biconnected component $B$ of $G'(\nu)$. An $st$-numbering is computed for the biconnected component $B$ by choosing the single virtual vertex $t_\nu$ as sink and any vertex $s_\nu$ of $G'(\nu)$ that is connected to $t_\nu$ as source. Using this $st$-numbering, the planarity test of Booth and Lueker (1976) is applied to $G'(\nu)$. If the planarity test returns true, then $G'(\nu)$ is planar, and by Lemma 3.6 all edges incident to cluster $\nu$ can be drawn in the outer face of $G(\nu)$. After the planarity test has been performed successfully at $G'(\nu)$, the subgraph $G(\nu)$ is replaced in $G$ by a representative graph. The representative graph of $G(\nu)$ is constructed such that the only possible planar embeddings are those, where

(i) the edges incident to the cluster $\nu$ (and therefore incident to a vertex in $G(\nu)$) are embedded in the outer face of the representative graph,

(ii) the edges incident to the cluster $\nu$ appear exactly in those orderings as in all planar embeddings of $G'(\nu)$ with the edge $(s_\nu, t_\nu)$ drawn on the outer face of $G'(\nu)$.

The representative graph can be constructed in $\mathcal{O}(n)$ time by help of the $PQ$-tree as it exists before the final reduction of the leaves corresponding to the virtual vertex $t_\nu$ of $\nu$. Using a representative graph for each child of a cluster $\nu$, Feng *et al.* (1995) showed that the cluster $\nu$ itself can be tested for $c$-planarity.

In order to construct a $c$-planar embedding of a cluster graph $C = (G, T)$, the $PQ$-tree techniques of Chiba *et al.* (1985) are applied. Feng *et al.* (1995) find a circular ordering of

the edges incident to each cluster recursively, following a traversal of the tree from top to bottom. Storing for each cluster its corresponding $PQ$-tree during the $c$-planarity testing, an arbitrary embedding is chosen for the cluster $G(\nu)$, if $\nu$ is the root of $T$. Using the $c$-planar embedding of $G(\nu)$ of any cluster $\nu$, the $c$-planar embeddings of all its children can be then recursively computed.

Since the $c$-planarity testing algorithm has to replace for every cluster $\nu$ the subgraph $G(\nu)$ by its representative graph, the algorithm needs $\mathcal{O}(n^2)$ time. The results of the algorithm of Feng *et al.* (1995) are stated in the following theorem.

**Theorem 3.13 (Feng *et al.* (1995)).** *There is an $\mathcal{O}(n^2)$ time algorithm using the $PQ$-tree data structure to test whether a clustered graph $C = (G, T)$ is $c$-planar, and if so, it outputs a $c$-planar embedding.*

## 3.5 The Maximal Planar Subgraph Problem

In Automatic Graph Drawing, a widely-used method for drawing nonplanar graphs is to transform the graph into a planar graph, and then use planar graph drawing methods. A popular method for this transformation is to delete edges in order to get a planar subgraph, and to reinsert the removed edges after embedding the planar subgraph such that the number of edge crossings is small. Similarly, in VLSI-design the thickness problem is approximated by successively subtracting large planar subgraphs from a given nonplanar graph. However, Liu and Geldmacher (1977) showed that the problem of finding the minimum number of edges that have to be removed from a given graph in order to obtain a planar subgraph is $\mathcal{NP}$-hard. The first algorithm for solving this problem was a branch and bound algorithm suggested by Foulds and Robinson (1976). However this approach was only useful for very small and dense graphs. Mutzel (1994) and Jünger and Mutzel (1996) attacked the *maximum planar subgraph* problem by a branch and cut technique that gave good and in many cases provably optimum solutions for sparse as well as for very dense graphs.

Due to the $\mathcal{NP}$-hardness of the problem the attention has focused on computing maximal planar subgraphs. Let $G = (V, E)$ be a simple graph with $n$ vertices and $m$ edges. Then a planar subgraph $G'$ of $G$ is a *maximal planar* subgraph if for all edges $e \in G - G'$ the addition of $e$ to $G'$ destroys planarity. Besides a trivial $\mathcal{O}(nm)$ algorithm that can be constructed using any $\mathcal{O}(n)$ planarity test, three different approaches are known for solving this problem.

Chiba, Nishioka, and Shirakawa (1979) presented an algorithm based on the path addition algorithm that computes a maximal planar subgraph in $\mathcal{O}(nm)$ time. Cai, Han, and Tarjan (1993) presented later an $\mathcal{O}(m \log n)$ algorithm that is based on the path addition algorithm as well.

Based on the strategy of incremental planarity testing, Di Battista and Tamassia (1996) described an algorithm that checks in $\mathcal{O}(\log n)$ amortized time per edge insertion, whether

an edge can be added to $G$ without destroying planarity, obtaining an $\mathcal{O}(m \log n)$ time algorithm as well. Using an approach similar to the approach of Di Battista and Tamassia (1989), Westbrook (1992) describes an algorithm that works in $\mathcal{O}(n \log n + m\alpha(m, n))$ worst case time plus an additional $\mathcal{O}(n)$ expected time. La Poutré (1994) gave an incremental planarity test that takes $\mathcal{O}(\alpha(m, n))$ amortized time per edge insertion yielding an $\mathcal{O}(n + m\alpha(m, n))$ time algorithm. Djidjev (1995) gave an $\mathcal{O}(n + m)$ time algorithm using $SPQR$-trees and special dynamic data structures that allow set union and set split operations in constant amortized time. So far, this algorithm provides the best known running time. However, it is extraordinary difficult to implement, and as Djidjev (1998) confirms, no implementation is known.

Ozawa and Takahashi (1981) have presented an $\mathcal{O}(nm)$ algorithm using the vertex addition algorithm. Jayakumar, Thulasiraman, and Swamy (1986) showed that in general this algorithm does not determine a maximal planar subgraph. Moreover, the resulting planar subgraph may not even contain all vertices. Jayakumar, Thulasiraman, and Swamy (1989) presented an algorithm called PLANARIZE that computes a spanning planar subgraph $G_p$ of $G$ in $\mathcal{O}(n^2)$ time. Furthermore, they present an algorithm called MAX-PLANARIZE that augments $G_p$ to a subgraph $G'$ of $G$ by adding additional edges in $\mathcal{O}(n^2)$ time. They claim that $G'$ is a maximal planar subgraph of $G$ if $G_p$ (the result of phase 1 of the two phase algorithm) turns out to be biconnected. Kant (1992) shows that this algorithm is incorrect, and suggests a modification of the second phase of the algorithm that augments $G_p$ to a maximal planar subgraph of $G$, even if $G_p$ is not biconnected, maintaining $\mathcal{O}(n^2)$ time requirement.

In this section, we describe a substantial flaw in both the original and the modified two phase algorithm that was not detected previously as well as new mistakes introduced by Kant. First, the principle of the planarization algorithm using the $PQ$-trees is described. Then, we show that the algorithm of Jayakumar *et al.* is incorrect. We give a detailed description of the major mistake. We finish by discussing the attempt of Kant and make some concluding remarks in the last section.

### 3.5.1 A Principle of an Approach for Planarization

The basic idea of a planarization algorithm using $PQ$-trees presented by Jayakumar *et al.* (1989) is to construct (following the algorithm of Booth and Lueker (1976)) a sequence of $PQ$-trees $T_1, T_2, \ldots, T_{n-1}$ by deleting an appropriate number of pertinent leaves every time the reduction fails such that the resulting $PQ$-tree becomes reducible. In every step of the algorithm PLANARIZE, a maximal consecutive sequence of pertinent leaves is computed by using a $[w, h, a]$-*numbering* (see Jayakumar *et al.* (1989)). All pertinent leaves that are not adjacent to a maximal pertinent sequence are removed from the $PQ$-tree in order to make it reducible. Hence, the edges corresponding to the leaves are removed from $G$ and the resulting graph $G_p$ is planar.

It has been shown by Jayakumar *et al.* (1989) that the graph $G_p$ computed by PLANARIZE is not necessarily maximal planar. The authors therefore suggest to apply a second phase

called MAX-PLANARIZE, also based on $PQ$-trees. Knowing which edges have been removed from $G$ to construct $G_p$, edges from $G - G_p$ are added back to $G_p$ in the second phase without destroying planarity.

During the reduction of a vertex $v_i$, $1 < i < n$, there may exist nonpertinent leaves that are in all permissible permutations of the $PQ$-tree $T_{i-1}$ between a pertinent leaf $l$, and a maximal pertinent sequence of pertinent leaves. The maximal pertinent sequence has been determined by the help of the $[w, h, a]$-numbering. In order to make the tree $T_{i-1}$ reducible, the leaf $l$ is removed from the tree and the corresponding edge is removed from the graph $G$, guaranteeing that the subgraph $G_p$ will be planar. However, it may occur that the nonpertinent leaves that are positioned between $l$ and the maximal pertinent sequence in $T_{i-1}$, are removed as well from a tree $T_j$, $i \leq j < n$, in order to obtain reducibility. Therefore, there is no need to remove the edge corresponding to $l$ from the graph $G$.

In order to find leaves such as $l$, Jayakumar *et al.* (1989) use the algorithm MAX-PLANARIZE. In step $i$, both PLANARIZE as well as MAX-PLANARIZE reduce the same vertex $v_i$. The difference between the $PQ$-trees in the two algorithms is according to the authors that all leaves that have been deleted in PLANARIZE are ignored in MAX-PLANARIZE from the moment they are introduced into the tree until they get pertinent. This causes the nonpertinent leaves between the pertinent leaf $l$ and its maximal pertinent sequence to be ignored. Hence $l$ is adjacent to its maximal pertinent sequence and the corresponding edge can be added back to $G_p$, while the leaves between $l$ and the maximal pertinent sequence are removed from the $PQ$-tree.

### 3.5.2 On the Incorrectness of the Algorithm

While some incorrect details of the approach of Jayakumar *et al.* have been described in a technical report by Kant (1992), who attempted to correct the algorithm, a major problem has not been detected.

Jayakumar *et al.* assume that the maximal planar subgraph $G_p$ is biconnected for the correct application of the Lempel-Even-Cederbaum algorithm. Furthermore, as they have stated correctly, this is necessary in order to have an $st$-numbering. Nevertheless, the $PQ$-trees in MAX-PLANARIZE are constructed according to the $st$-numbering that was computed for the graph $G$.

As a matter of fact, the $st$-numbering of $G$ in general does not induce an $st$-numbering of a subgraph $G_p$ even if the subgraph $G_p$ is biconnected. This results in two problems, of which one is crucial and cannot be dealt with even by the ideas described by Kant (1992).

Both problems are based on the fact that during the application of PLANARIZE for some vertices of $V$ all outgoing edges may be deleted from the graph while the resulting graph $G_p$ stays biconnected.

Let $v_i \in V$, $1 < i < n$, be such a node with no outgoing edges in $G_p$. Since $G_p$ is biconnected, $v_i$ must have at least two incoming edges $(v_\nu, v_i)$ and $(v_\mu, v_i)$, with $\nu, \mu < i$. Let $v_j \in V$

be a vertex in $G$ such that $j < i$, hence the leaves corresponding to the incoming edges of $v_j$ are reduced before the leaves of $v_i$. Let $T_{j-1}$ be the $PQ$-tree during the application of MAX-PLANARIZE in which the leaves corresponding to the incoming edges of $v_j$ have to be reduced. Assume that the leaves of both vertices $v_j$ and $v_i$ are on the outer face of the same biconnected component of the bush form that corresponds to the $PQ$-tree $T_{j-1}$. Assume further that one designated leaf $v_{j_{k+1}}$ of the vertex $v_j$ is separated by the leaves $v_{i_\nu}$ and $v_{i_\mu}$ corresponding to $(v_\nu, v_i)$ and $(v_\mu, v_i)$ from the leaves $v_{j_1}, v_{j_2}, \ldots, v_{j_k}$, where the latter form the maximal pertinent sequence (see Fig. 3.17 for an illustration).

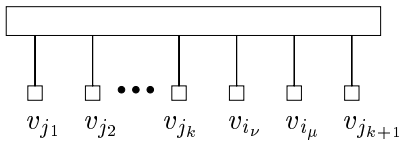



Figure 3.17: Leaf $v_{j_{k+1}}$ is separated by $v_{i_\nu}$ and $v_{i_\mu}$ from its maximal pertinent sequence $v_{j_1}, v_{j_2}, \ldots, v_{j_k}$.

If $(v_\nu, v_i)$ and $(v_\mu, v_i)$ are the only incoming edges of $v_i$ in $G_p$, then the leaves $v_{i_\nu}$ and $v_{i_\mu}$ will be replaced after the reduction of the $PQ$-tree $T_{i-1}$ by a $P$-node with leaves corresponding to edges in $E - E_p$. Hence, if the vertex $v_i$ had been reduced before the vertex $v_j$, then MAX-PLANARIZE would have considered the leaf $v_{j_{k+1}}$ as being adjacent to the maximal pertinent sequence $v_{j_1}, v_{j_2}, \ldots, v_{j_k}$. The edge corresponding to the leaf $v_{j_{k+1}}$ could have been added to the graph $G_p$ without destroying planarity. In case that none of the outgoing edges of $v_i$ is added to $G_p$ in a $PQ$-tree $T_l$, $i < l \leq n$, the resulting graph $G_p$ is not a maximal planar subgraph.

We now consider the second problem. The planarization algorithm of Jayakumar *et al.* (1989) does not obey an important invariant implied by Lemma 3.6. This result allowed Lempel, Even, and Cederbaum (1967) to transform the problem of planarity testing to the construction of a sequence of bush forms $B_k$. For a planar graph $G$, edges and vertices that have not been introduced into the current subgraph $G_k$ are always embedded into the outer face of $G_k$.

The approach of Jayakumar *et al.* (1989) does not obey this invariant in the second phase. There exist edges that have to be embedded into an inner face of some $G_k$, even if $(s, t)$ is drawn on the outer face. Due to the above lemma, the correction step MAX-PLANARIZE only considers edges for reintroduction into the planar subgraph $G_p$ that are on the outer face of the current graph $G_k$. Since the numbering that is used to determine the order in which the vertices are reduced does not correspond to an $st$-numbering of $G_p$ in general, the algorithm of Jayakumar *et al.* (1989) ignores edges that have to be added into an inner face of the embedding of a current graph $G_k$. This fact is fatal, as we are about to show now.

In Fig. 3.18, a part of a bush form $B_{k-1}$ of a graph $G$ is shown. The virtual vertices corresponding to the vertex $v_k$ are labeled $k_1, k_2, \ldots, k_5$ and all other virtual vertices are left unlabeled. The corresponding part of the $PQ$-tree is shown in Fig. 3.19. Obviously,

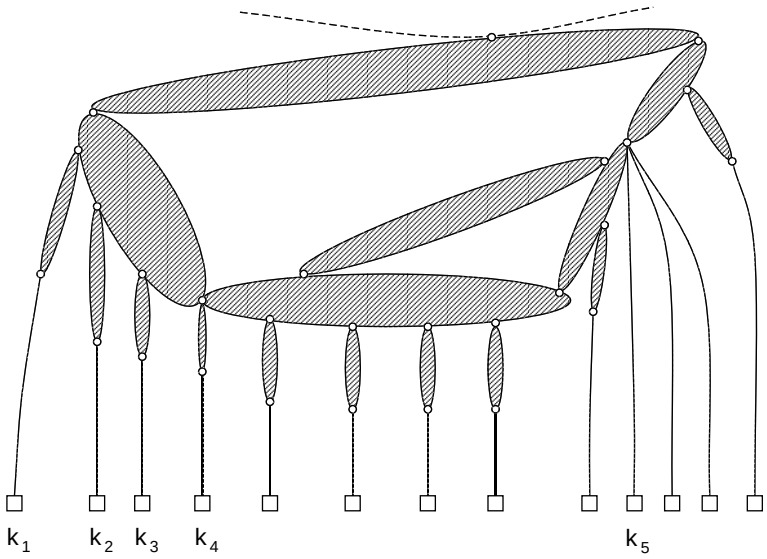



Figure 3.18: Part of a bush form $B_{k-1}$.

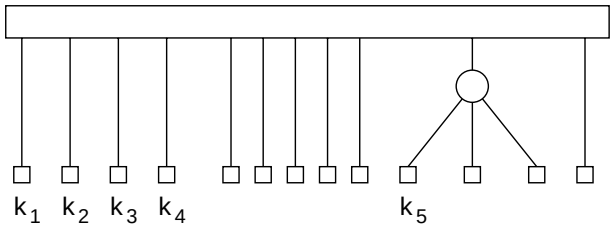



Figure 3.19: Part of a $PQ$-tree corresponding to bush form $B_{k-1}$.

there do not exist any reversions or permutations such that the virtual vertices of $v_k$ occupy consecutive positions. Hence, the graph $G$ is not planar. Applying the $[w, h, a]$-numbering of Jayakumar *et al.* (1989) allows us to delete the virtual vertex labeled $k_5$ and to reduce the other four vertices labeled $k_1, k_2, k_3, k_4$. The resulting graph $G_k$ is planar, and the relevant part of a bush form $B_k$ is shown in Fig. 3.20. Figure 3.21 shows the corresponding part of the $PQ$-tree. Assume now that all leaves corresponding to the outgoing edges of $v_k$ have to be removed from the $PQ$-tree in a later step. Hence all outgoing edges incident on $v_k$ are removed from the graph. Now assume further that there exists a path $v_{i_1}, v_{i_2}, \ldots, v_{i_l}$ in $G_p$ such that

- for all $\nu, \mu$, $1 \leq \nu < \mu \leq l$, the inequality $i_\nu < i_\mu$ holds,
- the edge $(v_{i_1}, v_{i_2})$ corresponds to one of the leaves that are between the leaf labeled $k_5$ and the maximal pertinent sequence of leaves labeled as $k_1, k_2, k_3, k_4$ in all $PQ$-trees equivalent to $T_{k-1}$,
- $v_{i_l} = t$.

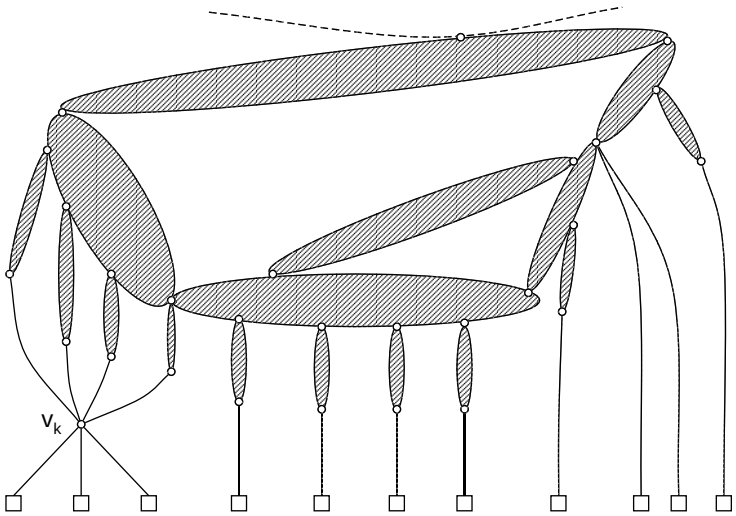



Figure 3.20: Part of a bush form $B_k$.

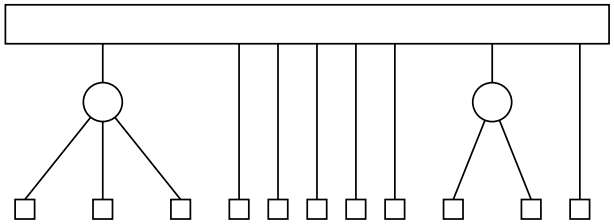


Figure 3.21: Part of a $PQ$-tree corresponding to bush form $B_k$.

This path guarantees that all incoming edges of the vertex $v_k$ cannot be embedded into the outer face of the embedding of $B_{k-1}$ without crossing an edge on this path. Hence the edge $e_{k_5}$ corresponding to the leaf labeled $k_5$ is not considered by the algorithm MAX-PLANARIZE as being an edge that does not destroy planarity. Therefore, $e_{k_5}$ is not added back to the planar subgraph $G_p$.

Nevertheless adding the edge $e_{k_5}$ to $G_p$ may not destroy planarity of $G_p$ as is shown in our example in Fig. 3.22. Since all outgoing edges of the vertex $v_k$ have been deleted by PLANARIZE and are not added back by MAX-PLANARIZE, it may be possible to swap the vertex $v_k$ into an inner face of the embedding of $B_k$ such that the virtual vertex labeled $k_5$ can be identified with $v_k$ and the edge $e_{k_5}$ is embedded into the bush form $B_k$ without destroying planarity.

Therefore, the strategy of using $PQ$-trees presented by Jayakumar *et al.* (1989) does not compute a maximal planar subgraph in general. Furthermore, we point out that the same problem holds for the modified version of this algorithm, presented by Kant (1992). This version follows a similar strategy of computing a spanning planar subgraph $G_p$ using PLA-

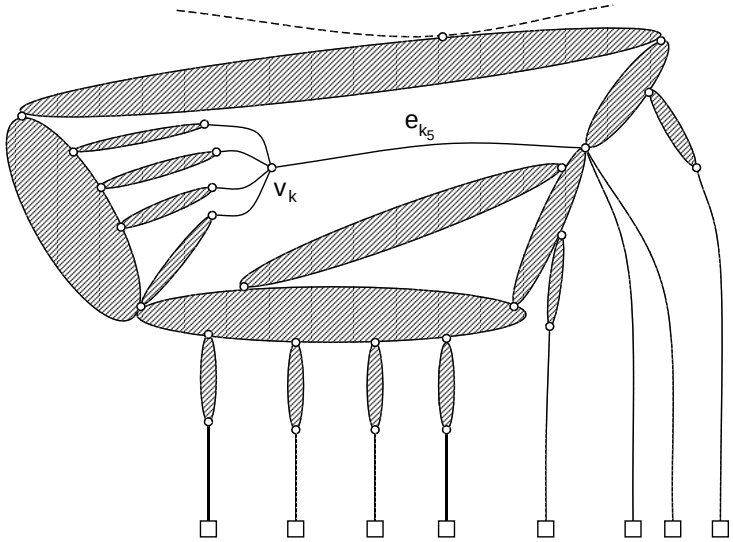



Figure 3.22: Part of a bush form $B_k$ with $e_{k_5}$ embedded.

NARIZE and then adding edges that do not destroy planarity in a second phase. The order of vertex additions to the planar subgraph is the same as the one implied by the *st*-numbering on $G$. Hence this approach is not able to compute a maximal planar subgraph for the same reason.

Summarizing, we state the following lemma that has been shown in the discussion above.

**Lemma 3.14.** *Let $G = (V, E)$ be a nonplanar graph. Let $G_p = (V, E_p)$, $E_p \subseteq E$, be a planar subgraph of $G$, such that $G_p$ was obtained from $G$ by*

1. *computing an st-numbering for all vertices and*
2. *applying the algorithm of Lempel, Even, and Cederbaum (1967) constructing a sequence of bush forms $B_{k-1}$, $k = 2, 3, \ldots, n$, by embedding a maximal number of incoming edges of a vertex $v_k$ in the outer face of $B_{k-1}$ without crossings, deleting all other incoming edges of $v_k$.*

*Let $G'_p = (V, E'_p)$, be a planar subgraph of $G$ such that*

1. *$E_p \subseteq E'_p \subseteq E$,*
2. *the graph $G'_p$ is computed by constructing a sequence of bush forms $B'_{k-1}$, $k = 2, 3, \ldots, n$, based on the st-numbering used for determining $G_p$, and possibly embedding incoming edges $e \in E - E_p$ of every vertex $v_k$, without crossings in the outer face of $B_{k-1}$.*

*Then the subgraph $G'_p$ is not necessarily maximal planar.*

Considering a computation of an $st$-numbering for the planar subgraph $G_p$ in order to augment $G_p$ to a maximal planar subgraph of $G$ and then construct a sequence of bush forms $B'_{k-1}$, $k = 2, 3, \ldots, n$, is aggravated by the fact that the graph $G_p$ is not biconnected in general. Furthermore, the difference between the bush forms of the first phase and the second phase may result in the deletion of the edges of $G_p$ as soon as edges of $E - E_p$ are added to $G_p$. Adding an edge $e \in E - E_p$ to $G_p$ is able to change the corresponding bush form in such a way that the pertinent leaves corresponding to the incoming edges of some node $v$ in $E_p$ cannot form a consecutive sequence in any permissible permutation.

### 3.5.3 Further Problems

We show now that even if the $st$-numbering of $G$ is as well an $st$-numbering of $G_p$, and even if we consider the suggested modifications of Kant (1992), the algorithm presented by Jayakumar *et al.* (1989) still does not work correctly.

Kant (1992) suggested a correction of the second phase by introducing sequence indicators and by delaying the decision, whether a deleted leaf can be added back to $G_p$, until enough information is available. In his version of MAX-PLANARIZE, a leaf $l$ that was deleted in PLANARIZE will be a normal nonpertinent leaf which is not ignored until it becomes pertinent. Again, the maximal pertinent sequence of a vertex $v_i$ is reduced. This maximal pertinent sequence is the same as in PLANARIZE. The pertinent leaves that are not adjacent to the maximal pertinent sequence stay in the $PQ$-tree and their presence will be ignored in the template matching algorithm from then on. They are called *potential leaves*. If potential leaves of a vertex $v_i$ remain in the tree after the reduction of $v_i$, a *sequence indicator* $\langle v_i \rangle$ is added to the tree in order to indicate the position of the reduced pertinent sequence. The presence of the sequence indicator will be ignored as well. If a node $X$ contains only nodes in its frontier that are ignored, $X$ is itself an ignored node. Applying this idea, a sequence of $PQ$-trees $T_1, T_2, \ldots, T_{n-1}$ is constructed during the augmentation phase that is equivalent to the sequence of $PQ$-trees constructed during the first phase. This ensures that the corresponding bush forms of both phases are equivalent. In order to augment $G_p$ to the maximal planar subgraph $G'_p$, edges are added to $G_p$, when their corresponding leaf and its sequence indicator can be reduced by deleting only other potential leaves and sequence indicators. Doing this, it is not allowed to bind empty nodes to new places since this would change the equivalence class of the actual $PQ$-tree.

Let $l$ be a potential leaf and denote the corresponding virtual vertex by $v(l)$. A potential leaf $l$ is *near* its sequence indicator $\langle v(l) \rangle$ if and only if the $PQ$-tree $T_i$ can be reduced with respect to $l$ and $\langle v(l) \rangle$, by deleting only ignored nodes, and not binding empty nodes to new places. If a potential leaf $l$ is near its sequence indicator $\langle v(l) \rangle$, then $l$ and $\langle v(l) \rangle$ are called a *near pair*. Not every near pair, formed by a potential leaf and its sequence indicator, can be reduced. In general, we have to choose between different near pairs, since the reduction of one near pair might cause the deletion of the other. This is, in particular, typical for near pairs that intersect. Two near pairs $l$, $\langle v(l) \rangle$ and $l'$, $\langle v(l') \rangle$ are called *intersecting* in

$T_i$, if either $l'$ or $\langle v(l') \rangle$ is between $l$ and $\langle v(l) \rangle$ in all equivalent $PQ$-trees of $T_i$. In this case only one edge corresponding to $l$ or $l'$ can be embedded into the outer face of the current bush form without causing a crossing.

Kant (1992) suggests to test for near pairs just within the maximal pertinent sequence, since by definition there will be no empty nodes inside the pertinent sequence. A near pair is found while applying the pattern matching algorithm of Booth and Lueker (1976) to the maximal pertinent sequence. Every time a pertinent node $X$ is matched, we search for near pairs $l$ and $\langle v(l) \rangle$ in the frontier of $X$, where $X$ is the first common ancestor of $l$ and $\langle v(l) \rangle$.

This search can be done efficiently using two arrays $PL_X$ and $SI_X$ that are introduced for every internal node $X$ in the maximal pertinent sequence. In $PL_X[i]$ those children of $X$ are stored that are ancestors of some potential leaves corresponding to edges $(v_j, v_i)$, $j < i$, while $SI_X[i]$ contains the children that are ancestors of some sequence indicator $\langle v_i \rangle$. It can be shown that by using the $PL_X$ and $SI_X$ arrays all near pairs will be found (see Kant (1992); Leipert (1995)). After finding a near pair $l$, $\langle v(l) \rangle$ we make sure that before the near pair is reduced, the first common ancestor $X$ of $l$, and $\langle v(l) \rangle$ is a $Q$-node. This can obviously be done without changing the equivalence class of the $PQ$-tree.

Kant (1992) now suggests a reduction process which is encapsulated within a procedure REDUCE-NEAR and that begins with the first common ancestor $X$ of the near pair $l$, $\langle v(l) \rangle$ and goes down the $PQ$-tree to $l$ and $\langle v(l) \rangle$. An ignored $P$-node is said to be of *type U* if all children except one child, yet unknown, must be removed with their descendants from the $PQ$-tree in a later step. An ignored $Q$-node is said to be of type U if it has one special marked child $Y$, and all children of the $Q$-node between $Y$ and one of the endmost children, yet unknown, must be removed with their descendants from the $PQ$-tree in a later step.

We now give a brief description of the reduction process. After a near pair $l$, $\langle v(l) \rangle$ is found, the following situation occurs: A $Q$-node $X$ is the first common ancestor of $l$ and $\langle v(l) \rangle$, and the leaf $l$ corresponds to some incoming edge of a vertex $v_i$. There exists a sequence $Y_1, Y_2, \ldots, Y_k$, $k \geq 2$, of children of $X$, such that, say, $Y_1$ is an ancestor of $l$ and $Y_k$ is an ancestor of $\langle v(l) \rangle$ and the children $Y_2, Y_3, \ldots, Y_{k-1}$ between $Y_1$ and $Y_k$ are ignored.

First, all children $Y_2, Y_3, \ldots, Y_{k-1}$ and their descendants are removed from the tree by REDUCE-NEAR. If a deleted leaf $l'$ corresponds to an incoming edge of the vertex $v_i$, it forms a near pair with $\langle v(l) \rangle$, so REDUCE-NEAR adds it to $G_p$. Then REDUCE-NEAR goes along the paths from $Y_1$ to the potential leaves of vertex $v_i$ applying a top-down reduction. There might be more than one potential leaf forming a near pair with $\langle v(l) \rangle$. The same is done with the path from $Y_k$ to the sequence indicator $\langle v(l) \rangle$. Since there may be other near pairs in the frontier of $Y_1$ and $Y_k$, they are reduced correspondingly. Afterwards, all ignored nodes between the outermost leaf that corresponds to an incoming edge of $v_i$, and $\langle v(l) \rangle$ are removed from the $PQ$-tree and the arrays $PL_X$ and $SI_X$ are updated. If a deleted leaf corresponds to an incoming edge of $v_i$, then it is added to $G_p$.

However, the algorithm REDUCE-NEAR is not correct for three reasons:

**Problem 1:** Adding every incoming edge of a vertex $v_i$ that corresponds to some deleted potential leaf might result in a nonplanar subgraph. This is due to the fact that different potential leaves may be descendants of the same type U node, or descendants of different pertinent nodes.

**Problem 2:** Reducing the near pairs top-down does not restrict the permissible permutations in such a way that in all permutations $l$ and $\langle v(l)\rangle$ form a consecutive sequence.

**Problem 3:** Let $l$, $\langle v(l)\rangle$ be the first detected near pair in the frontier of $Y_1, Y_2, \ldots, Y_k$. Let $l'$, $\langle v(l')\rangle$ be some other near pair in the frontier of the same nodes. The near pairs are reduced in the order they have been detected. This implies reducing $l$, $\langle v(l)\rangle$ before reducing $l'$, $\langle v(l')\rangle$. This is not correct since reducing $l$, $\langle v(l)\rangle$ first might cause the deletion of $l'$, $\langle v(l')\rangle$ while the reduction of $l'$, $\langle v(l')\rangle$ might not cause the deletion of $l$, $\langle v(l)\rangle$. Hence $G'$ is not necessarily maximal planar.

So in order to correct REDUCE-NEAR, we are confronted with solving the following three problems:

1. If there are several potential leaves that form a near pair with $\langle v(l)\rangle$, a maximal subset of leaves has to be found, which guarantees that all leaves of the subset can be reduced together.

2. A near pair $l$, $\langle v(l)\rangle$ has to be reduced, such that $l$ and $\langle v(l)\rangle$ form a consecutive sequence in all permissible permutations.

3. If there are several near pairs, an ordering of the near pairs has to be found in such a way that the reduction of one near pair does not hinder the reduction of the near pairs which still have to be reduced.

The first two problems have been shown to be solvable by Leipert (1995), but the last one still remains unsolved.

### 3.5.4 Remarks

We showed that the attempt of Jayakumar *et al.* (1989) to solve the maximal planar subgraph problem with $PQ$-trees is not correct. The problem is due to the fact that an important invariant for planarity testing is ignored. We have further shown that even a "corrected" version of the algorithm applied in the best possible case, where the $st$-numbering of a graph $G$ is as well an $st$-numbering of the planar subgraph $G_p$, is not correct.

Since this best case is a very rare case and since the modifications for the solved problems (see Leipert (1995)) are far beyond any reasonable implementation, we doubt that a useful algorithm based on the strategy presented by Jayakumar *et al.* (1989) can be found.

# Chapter 4

# Level Planarity Testing

A criterion to obtain a good readability in the presentation of hierarchies and level graphs is to produce diagrams with a limited number of crossings between the edges (see Batini, Furlani, and Nardelli (1985), Purchase, Cohen, and James (1995) and Purchase (1997)). However, the *k-level crossing minimization problem*, that is minimizing the number of crossings for $k$-level graphs, has shown to be $\mathcal{NP}$-hard by Garey and Johnson (1983), even if $k = 2$. According to Eades, McKay, and Wormald (1986) the problem remains $\mathcal{NP}$-hard if the vertices of one of the two levels are fixed in their positions. Moreover, Tomii, Kambayashi, and Shuzo (1977) and Eades and Whitesides (1994) proved that the problem of deleting the minimum number of edges to obtain a 2-level planar graph is $\mathcal{NP}$-hard.

Due to the $\mathcal{NP}$-completeness of the problems a lot of effort was spent to the design of efficient heuristics for reducing the number of crossings in drawings of 2-level graphs. The main idea of this approach was to use a "good" heuristic for the 2-level case and to perform a "level by level sweep" on the general $k$-level case, trying to reduce the crossings between consecutive levels. Choosing an appropriate ordering of the first level, the ordering of every level $i$ is kept fix while reordering the level $i + 1$ in order to reduce crossings between level $i$ and $i + 1$. The process can be repeated in reverse direction to reduce crossings further.

Among others, Warfield (1977) developed various heuristics for the general case. The most common heuristic has been presented by Sugiyama, Tagawa, and Toda (1981). It is called the *barycentric* method and works on a 2-level graph with one level fixed. The barycentric method orders the vertices of the free level according to the barycenter (average) of the $x$-coordinates of their neighbors in the fixed level. The *median* heuristic presented by Eades and Wormald (1994) orders the vertices of the free level using the median of the $x$-coordinates of the neighbors in the fixed level. Various greedy heuristics have been presented by Eades and Kelly (1986). The *greedy switch* heuristic passes over all consecutive pairs of vertices and switches them if this decreases the number of crossings. The *greedy insert* heuristic proceeds by successively choosing the next vertex $v$ to be the one that minimizes the number of crossings that edges adjacent to $v$ make with edges adjacent to vertices inserted before $v$ if $v$ is put to the right of all yet placed vertices. The *split* heuristic

chooses a pivot vertex $v$, and places every other vertex to the left or right of $v$ depending on the smaller number of crossings that a placement produces. More recent heuristics are the *stochastic* heuristic by Dresbach (1994) and the *assignment* heuristic by Catarci (1995). As reported by Jünger and Mutzel (1997) the barycentric heuristic yields the best results in terms of number of crossings and solution time.

Other approaches involve formulating the problem as an integer program and was first employed by Jünger and Mutzel (1997). They show that the 2-level crossing minimization problem with one level fixed may be transformed into a linear ordering problem which can then be solved by a branch-and-cut algorithm, yielding exact solutions within reasonable time. The computational results of Jünger and Mutzel (1997) lead to the conclusion that there is no need for heuristics if one level is fixed.

For the general case of two variable levels, a tabu search approach was employed by Valls, Marti, and Lino (1996a), and branch-and-bound approaches have been applied by Valls, Marti, and Lino (1996b), and Jünger and Mutzel (1997). However, as has been stated by Jünger and Mutzel (1997), the true optimum usually can be computed only for sparse instances with less than 15 vertices on every level using the branch-and-bound approach. A first approach to the multi-level crossing minimization problem based on branch-and-cut has been reported by Jünger, Lee, Mutzel, and Odenthal (1997a), along with preliminary computational results for 2- and 3-level graphs.

An alternative method for crossing minimization on hierarchical graphs has been presented by Mutzel (1996). Her method removes a minimum number of edges such that the resulting graph is $k$-level planar. For the final diagram the removed edges are reinserted into a $k$-level planar drawing. Hence, instead of considering the $k$-level crossing minimization problem, the *$k$-level planarization problem* is considered. In order to apply this strategy, it is necessary to have an algorithm that computes a level planar embedding of a level planar graph. However, as Carpano (1980) noticed, none of the planarity testing and embedding algorithms can be used to construct a level planar embedding of a level graph, even if the level graph is a hierarchy.

Algorithms for testing planarity of 2-level graphs are given by Harary and Schwenk (1972), Tomii, Kambayashi, and Shuzo (1977) and Eades, McKay, and Wormald (1986). However, it is not possible in general to reduce the problem of level planarity testing of a $k$-level graph $G$ to the one of testing the level planarity of the $(k-1)$ 2-level graphs that compose $G$. Di Battista and Nardelli (1988) developed a $k$-level planarity test for hierarchies using the $PQ$-tree data structure of Booth and Lueker (1976). Di Battista and Nardelli (1988) moreover gave a characterization of level planar hierarchies in terms of forbidden subgraphs. Chandramouli and Diwan (1995) gave an algorithm that can easily be transformed into a level planarity testing and embedding algorithm for triconnected level graphs.

A level planarity test for the general case of level graphs has been presented by Heath and Pemmaraju (1995, 1996), who apply the $PQ$-tree data structure, too. However, it has been shown by Jünger, Leipert, and Mutzel (1997b) that this algorithm does not state correctly level planarity of every level planar graph.

This chapter presents a linear time algorithm for level planarity testing that is based on three main new techniques that replace the incorrect crucial parts of the algorithm of Heath and Pemmaraju (1995, 1996). The first section of this chapter gives a characterization of level planarity in terms of forbidden subgraphs, including recent results by Healy and Kuusik (1998). The second section presents the level planarity test of Di Battista and Nardelli (1988) for hierarchies. The correctness of this algorithm is needed in the correctness proof of our level planarity test. The next section then presents the approach of Heath and Pemmaraju (1995, 1996) including the proof of the incorrectness of the algorithm. In the fourth section, a level planarity test is developed, employing two new strategies that replace the incorrect parts of the algorithm of Heath and Pemmaraju (1995, 1996). However, this approach does not yield a linear time algorithm. The Section 4.9 shows how to obtain an $\mathcal{O}(n)$ time algorithm by applying a third new strategy to the level planarity test (see also Jünger, Leipert, and Mutzel (1998b)).

For simplicity, we assume for the rest of this chapter that all level graphs are proper. In the last section we show that our algorithm performs on nonproper graphs with no modification yielding a linear time algorithm for nonproper graphs as well.

## 4.1 Characterization of Level Planar Graphs

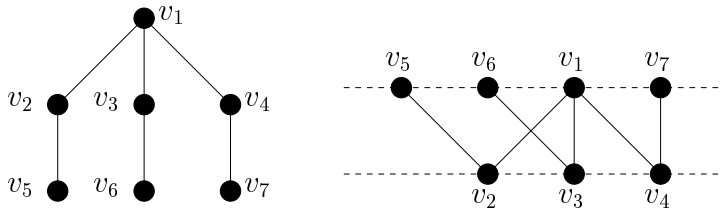
Due to the commonly used strategy of traversing level graphs top to bottom trying to reduce crossings in consecutive levels, research has focused early on the examination of 2-level graphs. First characterizations of level planar graphs have been given by Harary and Schwenk (1972), Tomii, Kambayashi, and Shuzo (1977) and Eades, McKay, and Wormald (1986) for the special case of 2-level graphs. The difference between a planar bipartite graph and a 2-level planar graph should be obvious. Consider the example shown in Fig. 4.1 that was taken from Mutzel (1996). It shows a planar bipartite graph that is not 2-level planar. The graph of Fig. 4.2 is called a *double claw*.

(a) Planar bipartite graph

(b) Nonlevel planar graph

Figure 4.1: A planar bipartite graph not being 2-level planar.

**Theorem 4.1 (Harary and Schwenk (1972)).** *A 2 level graph is 2-level planar if and only if it contains no cycle and no double-claw.*




(a) Not level embedded double claw.

(b) Level embedded double claw.

Figure 4.2: A double claw.

In 1988, Di Battista and Nardelli gave a characterization of level planar hierarchies. They identified three patterns of nonlevel planar subgraphs. To describe these patterns, we introduce some more terminology, as described in Di Battista and Nardelli (1988). Let $G = (V, E)$ be a $k$-level hierarchy. Then $\text{LACE}(i, j)$, $0 < i < j \leq k$, denotes the set of paths $P$ connecting any two vertices $u \in V^i$ and $v \in V^j$ such that $P$ traverses only vertices in $V^i \cup V^{i+1} \cup \cdots \cup V^j$. If $P_1$ and $P_2$ are vertex disjoint paths belonging to $\text{LACE}(i, j)$ then a *bridge* is a path connecting vertices $u \in P_1$ and $v \in P_2$ traversing only vertices in $V^i \cup V^{i+1} \cup \cdots \cup V^j$. The following theorem gives a characterization of level planar hierarchies.

**Theorem 4.2 (Di Battista and Nardelli (1988)).** *Let $G = (V, E)$ be a $k$-level hierarchy. $G$ is level planar if and only if there is no triple $P_1, P_2, P_3 \in \text{LACE}(i, j)$, $0 < i < j \leq k$, that satisfies one of the following conditions:*

*(i) The paths $P_1$, $P_2$, and $P_3$ are pairwise vertex disjoint and pairwise connected by bridges. The bridges do not share vertices with $P_1$, $P_2$, and $P_3$ except for the endpoints. See Fig. 4.3(a) for an illustration.*

*(ii) The paths $P_1$ and $P_2$ precisely share an endpoint $u$ and a path $P_u$ (possibly empty) starting from $u$, and $P_1 \cap P_3 = P_2 \cap P_3 = \emptyset$. Furthermore, there exist a bridge $b_1$ between $P_1$ and $P_3$ and a bridge $b_2$ between $P_2$ and $P_3$ with $b_1 \cap P_2 = b_2 \cap P_1 = \emptyset$. See Fig. 4.3(b) for an illustration.*

*(iii) The paths $P_1$ and $P_2$ precisely share an endpoint $u_1$ and a path $P_{u_1}$ (possibly empty) starting from $u_1$. Furthermore, the paths $P_2$ and $P_3$ precisely share an endpoint $u_2$, with $u_1 \neq u_2$, and a path $P_{u_2}$ (possibly empty) starting from $u_2$, and $P_1$ and $P_3$ are connected by a bridge $b$, with $b \cap P_2 = \emptyset$. See Fig. 4.3(c) for an illustration.*

Very recently Healy and Kuusik (1998) have shown that the forbidden patterns as described in Theorem 4.2 yield a characterization of level graphs in general.

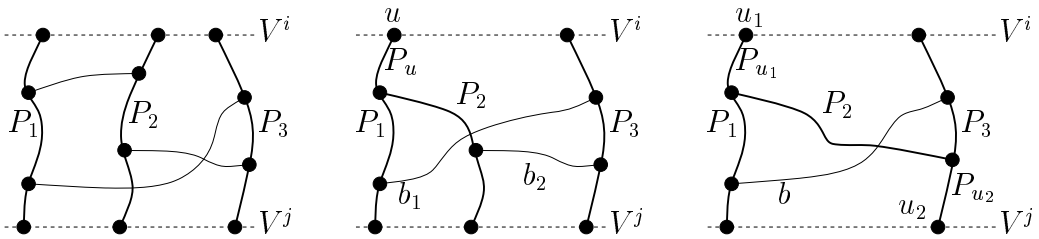



Figure 4.3: Illustration for Theorem 4.2, taken from Di Battista and Nardelli (1988).

**Theorem 4.3 (Healy and Kuusik (1998)).** *Let $G = (V, E)$ be a $k$-level graph. Then $G$ is level planar if and only if there is no triple of paths that satisfies the conditions from Theorem 4.2.*

Moreover, Healy and Kuusik (1998) give a characterization of level planar graphs in terms of *minimal nonlevel planar subgraph patterns* (MNLP-patterns). Such a MNLP-pattern is defined to have the property that the removal of any edge in the pattern makes the pattern level embeddable without edge crossings. Healy and Kuusik characterize MNLP-pattern as trees, nonlevel planar cycles, or as level planar cycles with certain path augmentations. Since the results are not needed in this work and are rather complex, we refrain from citing them.

## 4.2 Level Planarity Testing of Hierarchies

This section gives a brief introduction to the level planarity test of Di Battista and Nardelli (1988) for hierarchies. In order to test whether a level graph $G = (V, E)$ is level planar, it is sufficient to find an ordering $\leq_j$ of the vertices of every set $V^j$, $1 \leq j < k$, such that for every pair of edges $(u_1, v_1), (u_2, v_2) \in E$ with $u_1, u_2 \in V^j$ and $u_1 \leq_j u_2$, we have $v_1 \leq_{j+1} v_2$. Let $G^j$ denote the subgraph of $G$ induced by $V^1 \cup V^2 \cup \cdots \cup V^j$. The strategy of Di Battista and Nardelli (1988) for testing the level planarity of hierarchies is to perform a top-down sweep, processing the levels in the order $V^1, V^2, \ldots, V^k$ and computing for every level $V^j$, $1 \leq j \leq k$, the set of permutations of the vertices of $V^j$ that appear in some level planar embedding of $G^j$. Obviously, the graph $G = G_k$ is level planar if and only if the set of permutations for $G_k$ is not empty.

Since a hierarchy has only one single source, testing for level planarity can be done efficiently using the $PQ$-tree data structure, similar to the approach of Booth and Lueker (1976) for testing planarity of general (undirected) graphs. Let $s \in V$ denote the only source of $G$.

The following algorithm LPTH (*Level Planar Test for Hierarchies*) basically is the one given by Di Battista and Nardelli (1988).

Boolean **LPTH**$(G = (V^1, V^2, \ldots, V^k; E))$

```
begin
   construct the universal PQ-tree T for s and its outgoing edges;
   for j = 2 to k do
      for every vertex v ∈ V^j do
         S^v := {(w, v) | (w, v) ∈ E, w ∈ V^{j-1}};
         if REDUCE(T, S^v) = ∅ then
            return "false";
         S̃^v := {(v, w) | (v, w) ∈ E, w ∈ V^{j+1}};
         if |S̃^v| ≥ 1 then
            REPLACE(S^v,S̃^v);
         else
            remove pertinent subtree with respect to S^v from T;
   return "true";
end.
```

The algorithm as it was originally presented in Di Battista and Nardelli (1988) does not perform a reduction with respect to all leaves in $S^v$ in one step. The authors partition the set of incoming edges of a vertex $v$ into three subsets and reduce these subsets rather than reducing the complete set of incoming edges. This is more complicated, but it allows Di Battista and Nardelli to prove Theorem 4.2 taking advantage of the correctness of the algorithm. However, a close examination of their method reveals that reducing all leaves of $S^v$ at once constructs the same $PQ$-tree.

An embedding of the hierarchy is computed by first choosing any spanning tree $T_G$ of $G$ rooted at the source $s$ of $G$, and keeping an "ignored" leaf for every leaf of $T_G$ in the $PQ$-tree. The ignored leaves are ignored in subsequent applications of the template matching algorithm and are needed to determine the relative position of the leaves of $T_G$ in the corresponding levels. Using an arbitrary permutation of the final $PQ$-tree, including all ignored leaves, a depth first search is performed starting at the root $s$ of the spanning tree $T_G$ in order to compute the final embedding. The following theorem states the main result of Di Battista and Nardelli (1988).

**Theorem 4.4 (Di Battista and Nardelli (1988)).** *There exists a linear time algorithm to test whether a proper hierarchy is level planar, and if so, it outputs a level planar embedding.*

Let $G$ be a level planar graph, and let $H^j$, $1 \leq j < k$, be the graph constructed from $G^j$ by adding for every outgoing edge of $V^j$ a virtual edge and a virtual vertex, where every

virtual vertex is labeled by its counterpart in the original graph $G$. Clearly, we are able to identify every level planar embedding of $H^j$ with a $PQ$-tree $T$ by

- replacing every cut vertex by a $P$-node,
- replacing every biconnected component by a $Q$-node,
- replacing every virtual vertex by a leaf, and
- rooting $T$ at the node corresponding to the biconnected component or cut vertex that contains the source $s$.

According to the algorithm presented Di Battista and Nardelli (1988) the following lemma can be obtained for hierarchies.

**Lemma 4.5 (Di Battista and Nardelli (1988)).** *Let $G = (V, E)$ be a level planar hierarchy with $k$ levels. Let $G^j$ and $H^j$, $1 \leq j < k$, be defined as above. Let $B^j$ be some level planar embedding of $H^j$. Then there exists a sequence of permutations of components around cut vertices and reversions of biconnected components such that for every $v \in V^{j+1}$, the virtual vertices labeled $v$ occupy consecutive positions on the horizontal line $j+1$. Moreover, for every level planar embedding of $H^j$ the set of permutations of virtual vertices in which for all $v \in V^{j+1}$ the virtual vertices corresponding to $v$ occupy consecutive positions can be represented by a $PQ$-tree.*

We will need this lemma later in the inductive proof on the correctness of our level planarity test.

## 4.3 Level Planarity Testing by Heath and Pemmaraju

This section deals with the approach of Heath and Pemmaraju (1995, 1996) for testing general level graphs for level planarity. As in the approach of Di Battista and Nardelli (1988) for hierarchies the basic idea is to perform a top-down sweep, processing the levels in the order $V^1, V^2, \ldots, V^k$. Since the graph $G^j$, $1 \leq j < k$, is not necessarily connected, a separate $PQ$-tree is introduced for every component of $G^j$ and standard $PQ$-tree techniques are applied, as long as different components of $G^j$ are not adjacent to a common vertex on level $j+1$. If two components are adjacent to a common vertex $v$ on level $j+1$, they have to be merged and a new $PQ$-tree has to be constructed from the two corresponding $PQ$-trees. The new $PQ$-tree then represents all level planar embeddings of the merged component. Applying a combination of reduce operations and merge operations for combining $PQ$-trees, Heath and Pemmaraju try to maintain for every level $V^j$ and for every component $F$ of $G^j$ the set of permutations of the vertices of $F$ in $V^j$ that appear in some level planar embedding of $G^j$. If the set of permutations for $G^k$ is not empty, the graph $G = G^k$ is obviously level planar. This section gives a complete introduction to the approach of Heath

and Pemmaraju (1995, 1996), including a complete description of the merge operations. The section is finished by showing the incorrectness of the approach of Heath and Pemmaraju (1995, 1996).

Since a level graph is level planar if all its components are level planar, we assume without loss of generality that $G$ is connected. As long as the graph $G^j$ is connected for some $j \in \{1, 2, \ldots, k\}$ standard $PQ$-tree techniques similar to the ones used in the level planarity test for hierarchies in Section 4.2 can be applied for determining the required set of permutations. Unlike $G$, $G^j$ is not necessarily connected. A permutation induced by an ordering $\leq_j$ on the vertices of $V^j$ is called a *witness* of $G^j$ if the permutation appears in some level planar embedding of $G^j$. In case that $G^j$, $1 \leq j < k$, consists of more than one connected component, Heath and Pemmaraju suggest to use a $PQ$-tree for every component and formulate a set of rules for merging $PQ$-trees $T_1$ and $T_2$ corresponding to components $F_1$ and $F_2$ if $F_1$ and $F_2$ both are adjacent to some vertex $v \in V^{j+1}$.

The authors first reduce the pertinent leaves of $T_1$ and $T_2$ with respect to the vertex $v$. After successfully performing these reductions, the consecutive sequences of pertinent leaves labeled $v$ are replaced by single pertinent representatives in both $T_1$ and $T_2$. Going up one of the trees, say $T_1$, from its pertinent representative, an appropriate position is searched, allowing the tree $T_2$ to be placed into $T_1$. After successfully performing this step, the resulting tree $T'$ has two pertinent leaves labeled $v$ that again are reduced. If any of the steps fails, Heath and Pemmaraju state that the graph $G$ is not level planar. The following code fragment shows a function HP-TEST by Heath and Pemmaraju (1995, 1996) that transforms the subgraph $G^j$ into $G^{j+1}$, thereby testing for level planarity.

The function needs as input the set of $PQ$-trees $\mathcal{T}(G^j)$, where each $PQ$-Tree $T \in \mathcal{T}(G^j)$ is associated with a component $F$ of $G^j$, and represents all possible permutations of the level-$j$ vertices of $F$ in level planar embeddings of $F$. The function returns after successful termination the set $\mathcal{T}(G^{j+1})$ representing all possible permutations of $V^{j+1}$ in level planar embeddings of the components of $G^{j+1}$. By a (justified) abuse of notations, we will usually talk about "all level planar embeddings of the components of $G^{j+1}$" rather than "all possible permutations of $V^{j+1}$ in level planar embeddings of the components of $G^{j+1}$". The function MERGE used by HP-TEST will be described in 4.3.1.

$\mathcal{T}(G^{j+1})$ **HP-TEST**$(\mathcal{T}(G^j))$

```
begin
  for every vertex v ∈ V^j do
    S̃^v := {(v,w) | (v,w) ∈ E, w ∈ V^{j+1}};
    if |S̃^v| ≥ 1 then
      REPLACE(S^v, S̃^v);
    else
      remove S^v from the corresponding PQ-tree T;
```

**First Merge Phase**
for every vertex $v \in V^{j+1}$ do
  for every $PQ$-tree $T$ do
    $S_T^v :=$ leaves in $T$ labeled $v$;
    if REDUCE$(T, S_T^v) = \emptyset$ then
      return $\emptyset$;
    else
      let $v_T$ be a single representative of $S_T^v$;
      REPLACE$(S_T^v, \{v_T\})$;
**Second Merge Phase**
for every pair $T_1$, $T_2$ of $PQ$-trees in $\mathcal{T}(G^j)$ do
  let $U := \{v \mid v \in \text{frontier}(T_1) \cap \text{frontier}(T_2)\}$;
  if $U \neq \emptyset$ then
    let $v \in U$ be an arbitrary vertex;
    let $T' :=$ MERGE($T_1$,$T_2$,$v$);
    let $S_{T'}^v$ be the set of leaves labeled $v$ in $T'$;
    if REDUCE$(T', S_{T'}^v) = \emptyset$ then
      return $\emptyset$;
    else
      let $v'_T$ be a single representative of $S_{T'}^v$;
      REPLACE$(S_{T'}^v, \{v'_T\})$;
    $U := U - \{v\}$;
    while $U \neq \emptyset$ do
      let $v \in U$ be an arbitrary vertex;
      let $S_{T'}^v$ be the set of leaves labeled $v$ in $T'$;
      if REDUCE$(T', S_{T'}^v) = \emptyset$ then
        return $\emptyset$;
      else
        let $v'_T$ a single representative of $S_{T'}^v$;
        REPLACE$(S_{T'}^v, \{v'_T\})$;
      $U := U - \{v\}$;
end.

There are some interesting details to be noticed on the approach of Heath and Pemmaraju (1995, 1996). In the code fragment of HP-TEST we make use of the function REDUCE as it has been presented in Section 3.1. Heath and Pemmaraju do not explicitly use the template matching algorithm of Booth and Lueker (1976). They simply "reinvent" the template matching algorithm by developing their own templates and renaming the function REDUCE to ISOLATE. The templates that are used in the function ISOLATE only perform on sets of size two. A close examination immediately reveals that the templates developed by Heath and Pemmaraju are exactly the templates of Booth and Lueker for pertinent sets of size two. However, if three or more pertinent leaves exist in a $PQ$-tree $T$,

it is not possible in general to reduce all leaves by applying an arbitrary sequence of pairwise reductions even if the complete set of all pertinent leaves is reducible. In Heath and Pemmaraju (1996) this error is noticed. Nevertheless, the authors still suggest to reduce the pertinent leaves pairwise, but to apply a certain order by choosing pairs of pertinent leaves. Given a permutation $\pi \in \text{PERM}(T)$ where the pertinent leaves occupy consecutive positions, the leaves are reduced pairwise according to this given order. However, Heath and Pemmaraju (1996) do not describe how the permutation $\pi$ is obtained. The only known strategy to get a valid permutation is to reduce the $PQ$-tree with respect to all pertinent leaves and to read the frontier of the pertinent subtree from left to right. But once all pertinent leaves have been reduced it is unnecessary to reduce the pertinent leaves pairwise in the same order.

During the second merge phase, the $PQ$-trees are merged pairwise in an arbitrary order. It will be shown later in this section that this reveals a source of errors. Furthermore, Heath and Pemmaraju claim linear running time for their level planarity test. Considering the presented strategy in the second merge phase it is not clear how linear running time is obtained. Obviously, leaves need to know to which $PQ$-tree they belong to. Thus the merge operations involve update operations and Heath and Pemmaraju do not discuss how to obtain linear running time with respect to these operations.

### 4.3.1 Merge Operations

Merging two $PQ$-trees $T_1$ and $T_2$ at leaves labeled $v$ as required by the function MERGE corresponds to merging two components $F_1$ and $F_2$. Usually, it is not sufficient to merge $F_1$ and $F_2$ by simply placing them next to each other, identifying the virtual vertices with the same label. In general one component, say $F_2$, needs to be nested within the other component. Fig. 4.4 shows a simple example where the smaller component $F_2$, drawn with shaded vertices, has to be placed into $F_1$ for merging both components at their vertices labeled $v$.

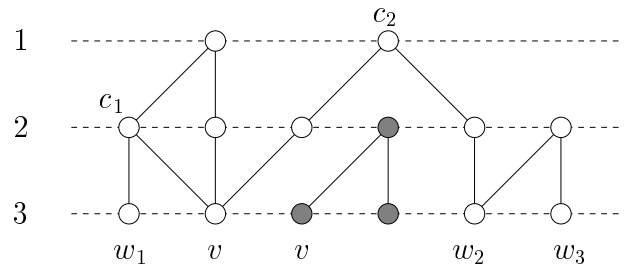



Figure 4.4: Nesting the smaller component $F_2$ with shaded vertices into the larger component $F_1$.

However, for merging $F_1$ and $F_2$, it is necessary to know how much "space" in $F_1$ is available between the vertex $v$ and its neighbors in every level planar embedding of $F_1$. In

the example 4.4 possible neighbors of $v$ in $F_1$ are $w_1$ on the left side and $w_2$ or $w_3$ on the right side ($w_3$ can be easily placed next to $v$ without causing an edge crossing). The space between $w_1$ and $v$ is obviously too small to nest $F_2$ between $w_1$ and $v$. There exists a cut vertex $c_1$ on level 2 and a path $p$ from $w_1$ to $v$ using $c_1$ such that for every vertex $u$ on the path we have $\text{lev}(u) \geq 2$. Since $F_2$ has one vertex on level 2, a placement of $F_2$ between $w_1$ and $v$ creates crossings. If $F_2$ is placed between $v$ and $\{w_2, w_3\}$, no two edges cross each other since any path connecting $v$ and a vertex of $\{w_2, w_3\}$ must traverse the cut vertex $c_2$ on level 1.

Heath and Pemmaraju (1995, 1996) formalize this observation as follows. For any subset $S$ of the set of vertices in $V^j$ that belong to a component $F$ of $G^j$, we define $\text{ML}(S)$ to be the greatest $d \leq j$ such that $V^d \cup V^{d+1} \cup \cdots \cup V^j$ induces a subgraph of $G$ in which all nodes of $S$ occur in the same connected component. The level $\text{ML}(S)$ is said to be the *meet level* of $S$. If a subset $S \subseteq V^j$ is not contained in a component of $G^j$, set $\text{ML}(S) = 0$. Obviously we have that $\text{ML}(S) \geq 1$, if $S$ induces a connected subgraph. Figure 4.5 shows an example of a graph $G^5$ having two components. We have listed different subsets of the vertices $V^5 = \{v_1, v_2, \ldots, v_8\}$ and their ML-values.

- $\text{ML}(\{v_1, v_2, v_3, v_4\}) = \text{ML}(\{v_1, v_4\}) = 3$.
- $\text{ML}(\{v_6, v_7\}) = 4$.
- $\text{ML}(\{v_6, v_7, v_8\}) = 3$.
- $\text{ML}(\{v_5, v_6\}) = 0$.

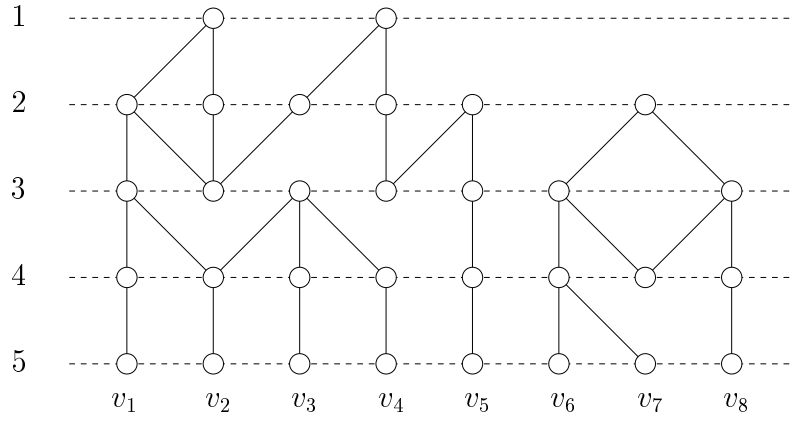



Figure 4.5: A $G^5$ with two components.

Since the level planarity test performs on $PQ$-trees instead on the components of $G^j$, we need to show how to maintain ML-values in a $PQ$-tree. Let $Y$ be a $Q$-node in a $PQ$-tree $T_F$ corresponding to a component $F$ of $G^j$ and let $Y_1, Y_2, \ldots, Y_t$ be the sequence of children of $Y$. At node $Y$ we maintain integers denoted by $\text{ML}(Y_i, Y_{i+1})$, where $1 \leq i < t$, satisfying

$\mathrm{ML}(Y_i, Y_{i+1}) = \mathrm{ML}(\mathrm{frontier}(Y_i) \cup \mathrm{frontier}(Y_{i+1}))$. At a $P$-node $X$ in $T(F)$ we maintain a single integer $\mathrm{ML}(X)$ that satisfies $\mathrm{ML}(X) = \mathrm{ML}(\mathrm{frontier}(X))$. Figure 4.6 shows the $PQ$-trees corresponding to the graph $G^5$ of Fig. 4.5 together with the ML-values that are stored at the nodes.

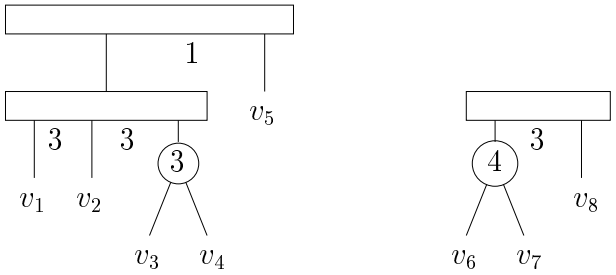



Figure 4.6: $PQ$-trees corresponding to $G^5$ shown in 4.5.

Furthermore, define $\mathrm{LL}(F)$, the *low indexed level*, to be the smallest $d$ such that $F$ contains a vertex in $V^d$ and maintain this integer at the root of the corresponding $PQ$-tree $T$, denoting it by $\mathrm{LL}(T)$ (thus, $\mathrm{LL}(F)$ equals $\mathrm{LL}(T)$). The *height* of $F$ is $j - \mathrm{LL}(F)$. The LL-value merely describes the size of the component. The LL-value of the left component in example 4.5 is 1 and the LL-value of the right component is 2. The maintenance of the ML-values during the template matching algorithm is straightforward.

Using these LL- and ML-values, Heath and Pemmaraju (1995, 1996) describe a set of rules how to connect two $PQ$-trees, claiming that the leaves of the new tree $T'$ having the same label are reducible if and only if the "merged" component $F'$ is level planar.

**Proposition 4.6 (Heath and Pemmaraju (1995, 1996)).** *Suppose that $X$ is the least common ancestor of a pair of leaves $v$ and $w$ in a $PQ$-tree in $\mathcal{T}(G^j)$. If $X$ is a $P$-node, then*

$$\mathrm{ML}(\{v, w\}) = \mathrm{ML}(X) \ .$$

**Proposition 4.7 (Heath and Pemmaraju (1995, 1996)).** *Suppose that $X$ is the least common ancestor of a pair of leaves $v$ and $w$ in a $PQ$-tree in $\mathcal{T}(G^j)$. Suppose further that $X$ is a $Q$-node with ordered children $X_1, X_2, \ldots, X_t$ such that $v \in \mathrm{frontier}(X_p)$ and $w \in \mathrm{frontier}(X_q)$, where $1 \leq p < q \leq t$. Then*

$$\mathrm{ML}(\{v, w\}) = \min\{\mathrm{ML}(X_i, X_{i+1}) \mid p \leq i < q\} \ .$$

The next proposition of Heath and Pemmaraju (1996) formally states the fact that as we follow a path in a $PQ$-tree from a leaf to the root, the encountered ML-values are nonincreasing.

**Proposition 4.8 (Heath and Pemmaraju (1995, 1996)).** *Suppose that node $X$ is the parent of an internal node $Y$ in a $PQ$-tree. Define $x$ as follows:*

$$x = \begin{cases} \mathrm{ML}(X) & \text{if } X \text{ is a } P\text{-node;} \\ \max\{\mathrm{ML}(Y, Z) \mid Z \text{ is a child of } X \text{ adjacent to } Y\} & \text{if } X \text{ is a } Q\text{-node.} \end{cases}$$

*Define $y$ as follows:*

$$y = \begin{cases} \mathrm{ML}(Y) & \text{if } Y \text{ is a } P\text{-node;} \\ \min\{\mathrm{ML}(Y_i, Y_{i+1}) \mid 1 \leq i < t\} & \text{if } Y \text{ is a } Q\text{-node with ordered children } Y_1, Y_2, \ldots, Y_t. \end{cases}$$

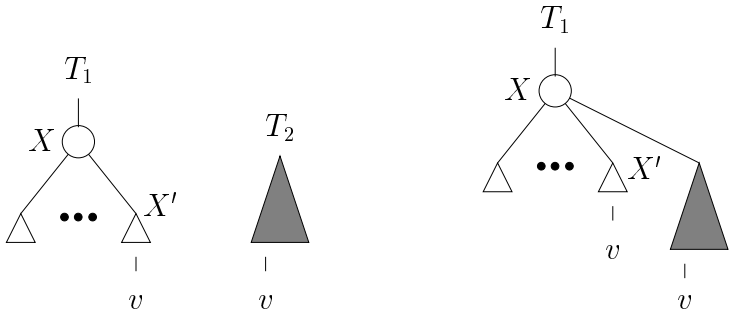
*Then $x \leq y$ holds.*

A detailed description of the merge operations of Heath and Pemmaraju (1995, 1996) is now given. Let $G = (V, E)$ be a $k$-level graph and $F_1$ and $F_2$ be two components of $G^j$, $1 \leq j < k$, both being adjacent to the same vertex $v \in V^{j+1}$. Let $T_1$ and $T_2$ be the $PQ$-trees of $F_1$ and $F_2$, both representing all level planar embeddings of their corresponding components after the application of the first merge phase for the level $j+1$. Identifying the vertices labeled $v$ of the components $F_1$ and $F_2$ constructs a new component $F$. For this new component $F$ a new $PQ$-tree $T$ is needed that represents all level planar embeddings of $F$. Heath and Pemmaraju now formulate a set of rules of how to construct the $PQ$-tree $T$ using the two existing ones $T_1$ and $T_2$.

Without loss of generality, we may assume that $\mathrm{LL}(T_1) \leq \mathrm{LL}(T_2)$. Thus component $F_2$ is the smaller component and an embedding of $F_1$ has to be found such that $F_2$ can be nested within the embedding of $F_1$. This corresponds to adding the root of $T_2$ as a child to a node of the $PQ$-tree $T_1$ constructing a new $PQ$-tree $T'$. In order to find an appropriate location to insert $T_2$ into $T_1$, we start with the leaf labeled $v$ in $T_1$ and proceed upwards in $T_1$ until a node $X'$ and its parent $X$ are encountered satisfying one of the following five conditions.

### Merge Condition A

The node $X$ is a $P$-node with $\mathrm{ML}(X) < \mathrm{LL}(T_2)$. Attach $T_2$ as child of $X$ in $T_1$. Figure 4.7 illustrates the merge operation A.

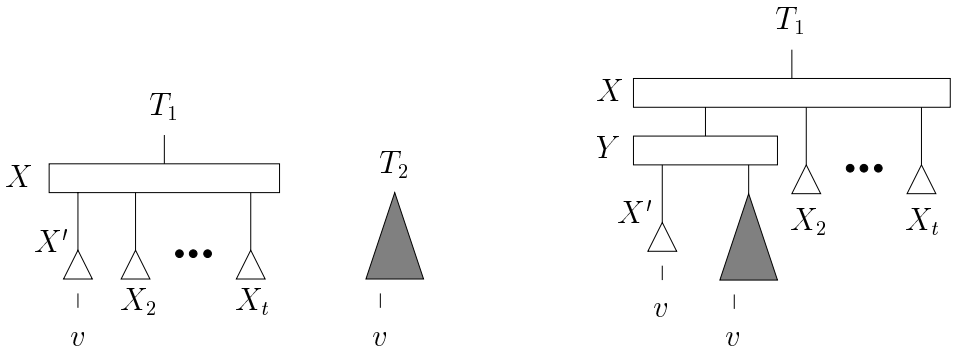



(a) $\mathrm{ML}(X) < \mathrm{LL}(T_2)$. (b)

Figure 4.7: Illustration for merge operation A.

### Merge Condition B

The node $X$ is a $Q$-node with ordered children $X_1, X_2, \ldots, X_t$, $X' = X_1$, and $\mathrm{ML}(X_1, X_2) < \mathrm{LL}(T_2)$. Replace $X'$ in $T_1$ by a $Q$-node $Y$ having two children, $X'$ and the root of $T_2$. The case where $X' = X_t$ and $\mathrm{ML}(X_{t-1}, X_t) < \mathrm{LL}(T_2)$ is symmetric. Figure 4.8 illustrates the merge operation B for the case of $X' = X_1$.




(a) $\mathrm{ML}(X', X_2) < \mathrm{LL}(T_2)$ . (b)

Figure 4.8: Illustration for merge operation B.

Although the ML-value of $X'$ and $T_2$ is 0 (since they are not connected), we set for technical purposes $\mathrm{ML}(X', T_2) := \mathrm{ML}(X', X_2)$. This is legal, since the two leaves labeled $v$ in the frontier of $X'$ and $T_2$ are reduced after applying the merge operation and replaced by a single representative which removes the value $\mathrm{ML}(X', T_2)$ from the tree. We will see later, how we benefit from this value.

### Merge Condition C

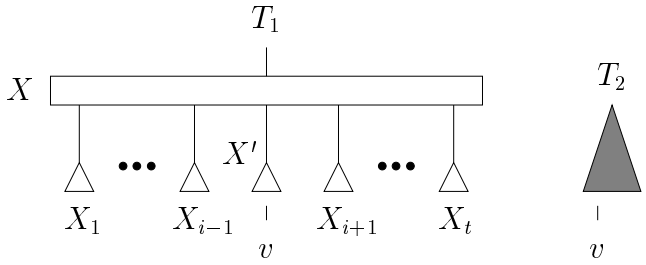
The node $X$ is a $Q$-node with ordered children $X_1, X_2, \ldots, X_t$, $X' = X_i$, $1 < i < t$, and $\mathrm{ML}(X_{i-1}, X_i) < \mathrm{LL}(T_2)$ and $\mathrm{ML}(X_i, X_{i+1}) < \mathrm{LL}(T_2)$. Replace $X'$ by a $Q$-node $Y$ with two children, $X'$ and the root of $T_2$, and set (for technical purposes again) $\mathrm{ML}(X', T_2) := \max\{\mathrm{ML}(X_{i-1}, X'), \mathrm{ML}(X', X_{i+1})\}$. Figure 4.9 illustrates the merge operation C.

### Merge Condition D

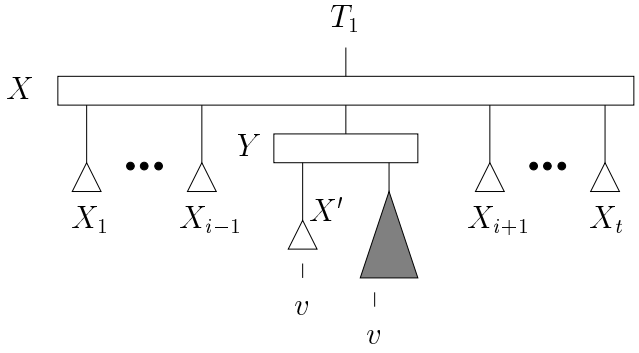
The node $X$ is a $Q$-node with ordered children $X_1, X_2, \ldots, X_t$, $X' = X_i$, $1 < i < t$, and

$$\mathrm{ML}(X_{i-1}, X_i) < \mathrm{LL}(T_2) \leq \mathrm{ML}(X_i, X_{i+1}) \ .$$

Attach the root of $T_2$ as child of $X$ between $X_{i-1}$ and $X'$, and set (for technical purposes) $\mathrm{ML}(X_{i-1}, T_2) := \mathrm{ML}(X_{i-1}, X')$ and $\mathrm{ML}(T_2, X') := \mathrm{ML}(X_{i-1}, X')$.




(a) $\mathrm{ML}(X_{i-1}, X') < \mathrm{LL}(T_2)$ and $\mathrm{ML}(X', X_{i+1}) < \mathrm{LL}(T_2)$.




(b)

Figure 4.9: Illustration for merge operation C.

In case that

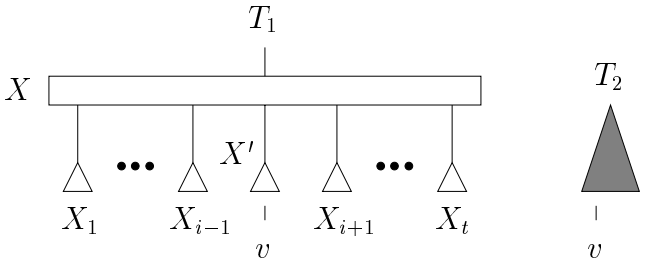
$$\mathrm{ML}(X_i, X_{i+1}) < \mathrm{LL}(T_2) \leq \mathrm{ML}(X_{i-1}, X_i) \ ,$$

attach the root of $T_2$ as child of $X$ between $X'$ and $X_{i+1}$ and set the ML-values correspondingly. Figure 4.10 illustrates the merge operation D for the case of $\mathrm{ML}(X_{i-1}, X_i) < \mathrm{LL}(T_2) \leq \mathrm{ML}(X_i, X_{i+1})$.

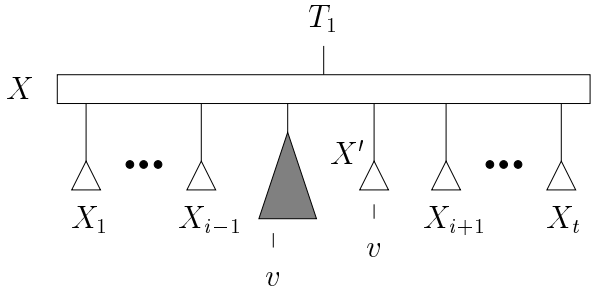
### Merge Condition E

The node $X'$ is the root of $T_1$. Reconstruct $T_1$ by inserting a $Q$-node $Y$ as new root of $T_1$ with two children $X'$ and the root of $T_2$ and set $\mathrm{ML}(X', T_2) := 0$. Figure 4.11 illustrates the merge operation E.

The merge operations for the conditions B and C both take care of the fact that the subgraph corresponding to $T_2$ can be embedded on either side of the subgraph corresponding to $X'$ with respect to the subgraph $X$. By construction, Heath and Pemmaraju (1996) make the following observations on the merge rules A, B, ... , E.




(a) $\mathrm{ML}(X_{i-1}, X') < \mathrm{LL}(T_2) \leq \mathrm{ML}(X', X_{i+1})$ .




(b)

Figure 4.10: Illustration for merge operation D.

**Observation 4.9 (Heath and Pemmaraju (1996)).** *Let* $\pi_1 \in \mathrm{PERM}(T_1)$ *be a permutation of the leaves of* $T_1$*, such that* $\mathrm{frontier}(X')$ *is adjacent to a leaf labeled* $w$*, and* $\mathrm{ML}(\mathrm{frontier}(X') \cup \{w\}) < \mathrm{LL}(T_2)$*. For any* $\pi_2 \in \mathrm{PERM}(T_2)$*, there exists a permutation* $\pi \in \mathrm{PERM}(T')$ *with* $T'$ *being the new PQ-tree that is consistent with* $\pi_1$ *and* $\pi_2$ *and in which the* $\mathrm{frontier}(T_2)$ *occurs immediately after* $\mathrm{frontier}(X')$ *and immediately before* $w$*.*

**Observation 4.10 (Heath and Pemmaraju (1996)).** *Let* $\pi_1 \in \mathrm{PERM}(T_1)$ *be a permutation such that the leaves of* $\mathrm{frontier}(X')$ *appear at the end of* $\pi_1$*. Let* $\pi_2 \in \mathrm{PERM}(T_2)$ *be an arbitrary permutation of the leaves of* $T_2$*. Then there exists a permutation* $\pi \in \mathrm{PERM}(T')$ *that is consistent with* $\pi_1$ *and* $\pi_2$ *and in which the leaves in* $\mathrm{frontier}(T_2)$ *occur immediately after* $\mathrm{frontier}(X')$*.*

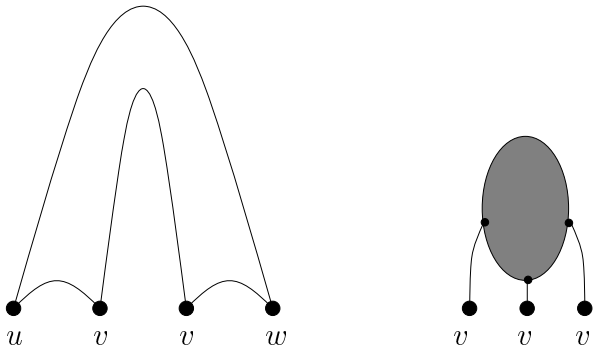
### 4.3.2 On the Incorrectness of the Algorithm

Heath and Pemmaraju apply a two phase algorithm in order to construct a set of $PQ$-trees that represents the set of all level planar embeddings of the components of the subgraph $G^{j+1}$. During the first merge phase, their algorithm reduces for every $PQ$-tree all sets of leaves corresponding to the same vertex. Suppose, $T_1$ and $T_2$ correspond to components $F_1$ and $F_2$ of $G^j$ and both components are adjacent to some vertex $v \in V^{j+1}$. Then the


(a) $X'$ is root of $T_1$. (b)

Figure 4.11: Illustration for merge operation E.

set of all leaves labeled $v$ in $T_1$ as well as in $T_2$ is reduced. Furthermore, these sets are replaced in $T_1$ as well as in $T_2$ by a single leaf labeled $v$. In the second merge phase, the algorithm merges different $PQ$-trees corresponding to components that are adjacent to the same vertex. Thus, $T_1$ and $T_2$ are merged, each of the trees having only one pertinent leaf labeled $v$.




(a) $F_1$ (b) $F_2$

Figure 4.12: Two components both adjacent to a vertex $v$.

Now let $F_1 = (V_1, E_1)$ be a component, such that the number of vertices in $V_1$ being adjacent to some vertex $v \in V^{j+1}$ is at least 2 and $F_1$ is also adjacent to at least two other vertices of $V^{j+1}$. Let $F_2 = (V_2, E_2)$ be a component such that $F_2$ is adjacent to $v \in V^{j+1}$ and not adjacent to any other vertex in $V^{j+1}$ and let the following inequality hold:

$$\mathrm{LL}(F_1) < \mathrm{LL}(F_2) \ .$$

Assume further that in all possible level planar embeddings of $F_1$, there is a vertex $u$ on the left and a vertex $w$ and on the right side of the virtual vertices labeled $v$ such that the

following inequality holds:

$$\mathrm{ML}(u, v) > \mathrm{LL}(T_2) \text{ and } \mathrm{ML}(v, w) > \mathrm{LL}(T_2) \ .$$

As an example, see Fig. 4.12, where the height of the two components in the drawing indicates their LL-values. Assume further, that there is enough space between two vertices labeled $v$ of $F_1$ such that $F_2$ can be nested inside $F_1$ constructing a level planar embedding of $F_1$ and $F_2$ with the vertices labeled $v$ identified as indicated in Fig. 4.13.

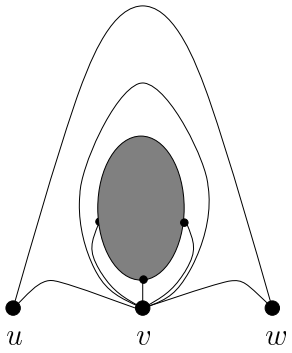



Figure 4.13: Only possible level planar embedding of the two components.

According to Heath and Pemmaraju (1995, 1996), the leaves corresponding to $v$ are reduced in both $PQ$-trees $T_1$ and $T_2$ and the pertinent subtree is replaced by a single leaf. But this replacement corresponds to the creation of interior faces in both $F_1$ and $F_2$. By constructing interior faces, the information about the space, where $F_2$ can be nested into $F_1$ gets lost as is indicated in Fig. 4.14.

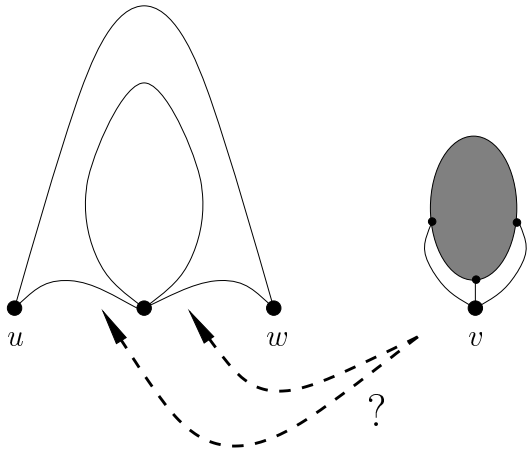



Figure 4.14: The situation caused by the first merge phase.

So the algorithm will fail when trying to place the tree $T_2$ next to the single leaf $v$ in $T_1$, constructing a $PQ$-tree $T'$ where in all possible permutations one of the leaves labeled $u$ or $w$ is between the leaf labeled $v$ of $T_1$ and the leaf labeled $v$ of $T_2$. The reduction $T'$ with respect to the vertices labeled $v$ then fails, and the algorithm states incorrectly that the level planar graph is not level planar.

Within the second merge phase, components are merged in an arbitrary order. We show that choosing an arbitrary order may result in $PQ$-trees that are not reducible with respect to a vertex $v$ (although the graph is level planar). Consider four different components $F_1, F_2, F_3, F_4$ and their corresponding $PQ$-trees $T_1, T_2, T_3, T_4$ each having a pertinent leaf corresponding to some level-$(j+1)$ vertex $v$. For simplicity we assume that for every component the leaf labeled $v$ appears in all permutations at one end of the $PQ$-tree. Assume further that the smallest common ancestor of a leaf labeled $v$ and any other leaf adjacent to it is a $Q$-node. Figure 4.15 shows such a component $F_i$ and its corresponding $PQ$-tree $T_i$. The number $l_i = \mathrm{ML}(\{w^i_{p_i}, v\})$ is the ML-value between the pertinent leaf labeled $v$ and the frontier of its left neighbor.

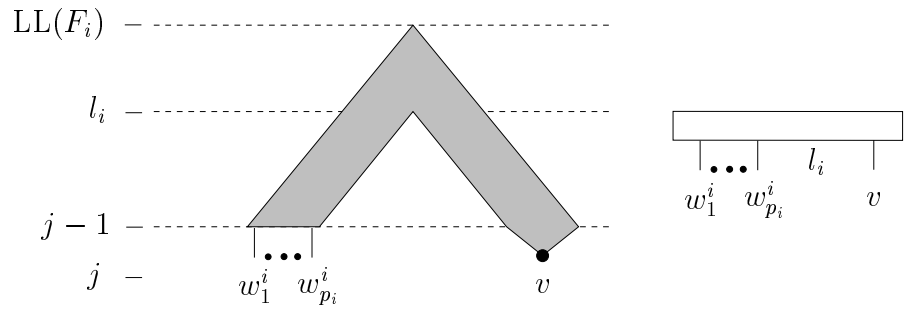



Figure 4.15: Component $F_i$ and its corresponding $PQ$-tree $T_i$.

Assuming that the condition

$$\mathrm{LL}(F_1) \le l_1 < \mathrm{LL}(F_2) \le l_2 < \mathrm{LL}(F_3) \le l_3 < \mathrm{LL}(F_4) \le l_4$$

on the ML- and LL-values of the components holds, it is possible to merge all four components into one component such that the pertinent leaves form a consecutive sequence. Figure 4.16 shows the four components, indicating how the components can be merged constructing a level planar embedding.

Consider the following merge operations on the components $F_1, F_2, F_3, F_4$ and their corresponding $PQ$-trees:

1. merge $F_1$ and $F_4$ into component $F'$,
2. merge $F'$ and $F_3$ into component $F''$,
3. merge $F''$ and $F_2$ into component $F'''$.

The resulting $PQ$-tree $T'''$ corresponding to $F'''$ is shown in Figure 4.17. The pertinent leaves labeled $v$ do not form a consecutive sequence in any permissible permutation of the $PQ$-tree $T'''$. Hence the algorithm presented by Heath and Pemmaraju (1995, 1996) states nonlevel planarity for certain level planar graphs.

Obviously, the order of merging the components is important for testing a level graph for level planarity. Moreover, it is easy to see, that using different orderings while merging three or more components results in different equivalence classes of $PQ$-trees. So even if every order of merging $PQ$-trees with pertinent leaves results in a reducible $PQ$-tree, a $PQ$-tree

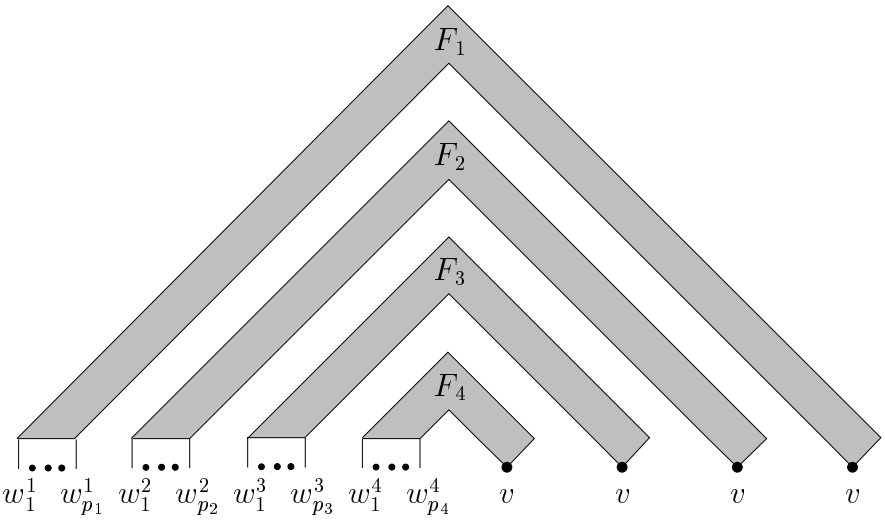



Figure 4.16: Possible level planar arrangement of the components $F_1, F_2, F_3, F_4$.

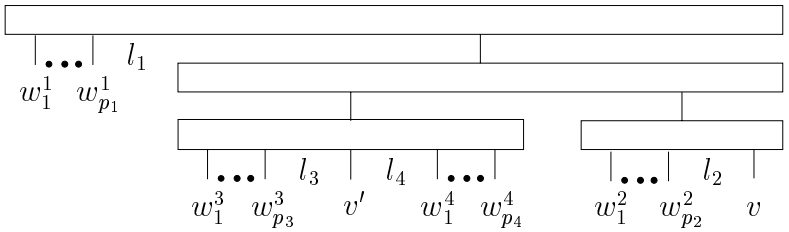



Figure 4.17: $PQ$-tree $T'''$ whose pertinent leaves labeled $v$ are not reducible. The vertex $v'$ denotes the single representative of the the leaves labeled $v$ corresponding to the components $F_1$, $F_3$ and $F_4$.

may be constructed such that the leaves corresponding to some vertex $w$, $\text{lev}(w) > j + 1$, are not reducible, although the graph $G$ is level planar. Again, the algorithm presented by Heath and Pemmaraju (1995, 1996) may state incorrectly the nonlevel planarity of a level planar graph.

## 4.4 Introduction to a Correct Level Planarity Test

In this section we introduce new strategies for obtaining a correct algorithm that tests a level graph $G = (V^1, V^2, \ldots, V^k; E)$ for level planarity. Before we describe our algorithm, called LEVEL-PLANARITY-TEST, let us recall some old and introduce some new terminology. Since $G^j$ is not necessarily connected, let $m_j$ denote the number of components of $G^j$ and let $F_i^j$, $i = 1, 2, \ldots, m_j$, denote the components of $G^j$. Figure 4.18 shows a $G^4$ with $m_4 = 2$ components $F_1^4$ and $F_2^4$. The set of vertices in $F_i^j$ is denoted by $V(F_i^j)$.

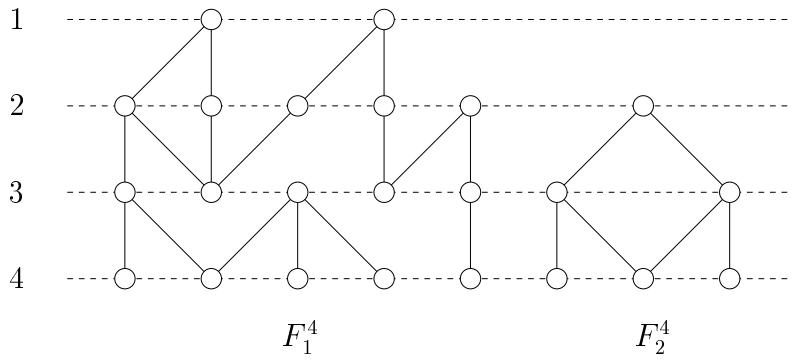



Figure 4.18: A $G^4$ with $m_2 = 2$ components $F_1^4$ and $F_2^4$.

Let $H_i^j$ be the graph arising from $F_i^j$ as follows: For each edge $e = (u, v)$, where $u$ is a vertex in $F_i^j$ and $v \in V^{j+1}$ we introduce a *virtual vertex* with label $v$ and a *virtual edge* that connects $u$ and this virtual vertex. Thus there may be several virtual vertices with the same label, adjacent to different components of $G^j$ and each with exactly one entering edge. The form $H_i^j$ is called the *extended form* of $F_i^j$ and the set of virtual vertices of $H_i^j$ is denoted by frontier($H_i^j$). Figure 4.19 shows possible extended forms $H_1^4$ and $H_2^4$ of the example in Fig. 4.18. The virtual vertices on level 5 are denoted by their labels. The frontier or $H_1^4$ consists of one virtual vertex labeled $u$, two vertices labeled $v$, and two vertices labeled $w$.

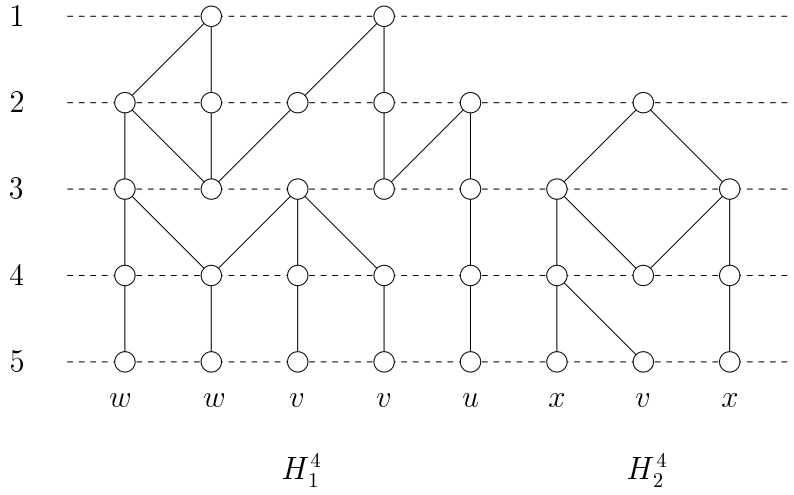



Figure 4.19: Two possible extended forms $H_1^4$ and $H_2^4$ of 4.18.

Let $B_i^j$ be a level planar embedding of $H_i^j$. Obviously, all virtual vertices of $H_i^j$ are placed on the same horizontal line on the outer face. As in the planarity test of Lempel *et al.* (1967), $B_i^j$ is called a *bush form* of $H_i^j$. The set of virtual vertices of $H_i^j$ that are labeled $v \in V^{j+1}$ is denoted by $S_i^v$. Figure 4.20 shows the sets $S_1^w$, $S_1^v$, and $S_1^u$, the set of virtual vertices labeled $w$,$v$ and $u$ of $H_1^4$, respectively. The set $S_1^x$ is empty.

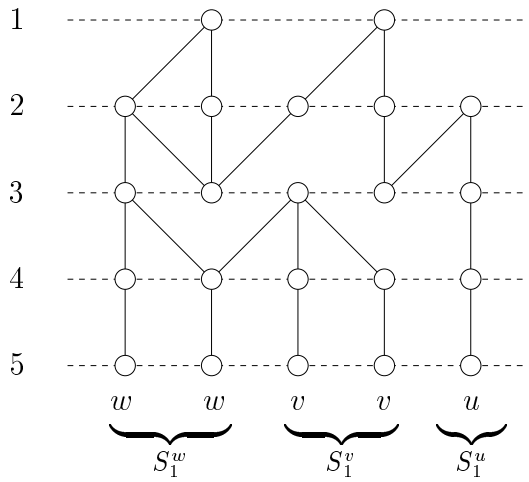



Figure 4.20: Sets $S_1^w$, $S_1^v$, and $S_1^u$ of $H_1^4$.

For two different nonempty sets $S_i^v$ and $S_l^v$ we denote the graph resulting by identifying the vertices $S_i^v \cup S_l^v$ by $H_i^j \cup_v H_l^j$.

The graph that is created from an extended form $H_i^j$ by identifying all virtual vertices with the same label to a single vertex is called *reduced extended form* and denoted by $R_i^j$. Figure 4.21 shows the reduced extended forms $R_1^4$ and $R_2^4$ of $H_1^4$ and $H_2^4$. In $R_1^4$ the vertices labeled $w$ have been identified and the vertices labeled $v$ have been identified. In order to identify the two vertices labeled $x$ in $R_2^4$, it was necessary to permute the left most vertex labeled $x$ and $v$. Both forms $R_1^4$ and $R_2^4$ now have exactly one vertex labeled $v$.

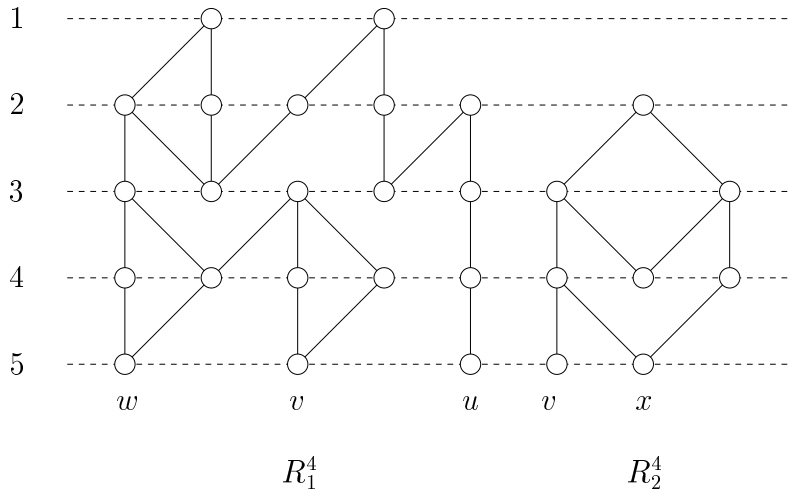



Figure 4.21: Extended forms $R_1^4$ and $R_2^4$ of $H_1^4$ and $H_2^4$.

The set of virtual vertices of $R_i^j$ is denoted by frontier($R_i^j$). If $S_i^v$ of $H_i^j$ is not empty, we denote the vertex with label $v$ of $R_i^j$ (i.e., the vertex that arose from identifying all virtual

vertices of $S_i^v$) by $v_i$ and update $S_i^v = \{v_i\}$. The graph arising from the identification of two virtual vertices $v_i$ and $v_l$ (labeled $v$) of two reduced extended forms $R_i^j$ and $R_l^j$ is denoted $R_i^j \cup_v R_l^j$. We call $R_i^j \cup_v R_l^j$ a *merged reduced form*. The vertex arising from the identification of $v_i$ and $v_l$ is denoted by $v_{\{i,l\}}$ (and labeled by $v$ of course). If $\mathrm{LL}(R_i^j) \leq \mathrm{LL}(R_l^j)$ we say $R_l^j$ is *$v$-merged* into $R_i^j$. The form that is created by $v$-merging $R_l^j$ into $R_i^j$ and identifying all virtual vertices with the same label $w \neq v$ is again a reduced extended form and denoted by $R_i^j$ (thus renaming $R_i^j \cup_v R_l^j$ with the name of the "higher" form). Figure 4.22 shows the resulting merged reduced extended form $R_1^4 \cup_v R_2^4$ after $R_2^4$ (the smaller form) has been $v$-merged into $R_1^4$ (the higher form). Since $R_1^4$ is the higher form, $R_1^4 \cup_v R_2^4$ is renamed into $R_1^4$.

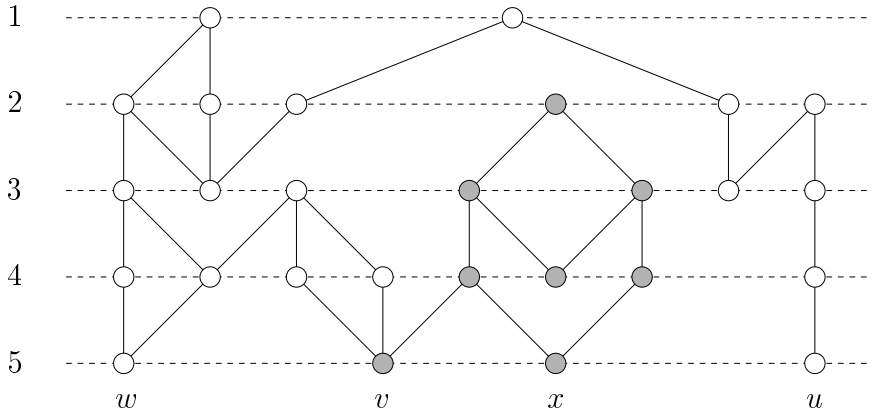



Figure 4.22: Merged reduced extended form $R_1^4 \cup_v R_2^4$ after $R_2^4$ has been $v$-merged into $R_1^4$. The former vertices of $R_2^4$ are drawn shaded.

If some reduced extended form has been $v$-merged into $R_i^j$, the form $R_i^j$ is called *$v$-connected*, otherwise $R_i^j$ is called *$v$-unconnected*. Thus, $R_1^4$ shown in Fig. 4.21 is $v$-unconnected, $R_1^4$ shown in Fig. 4.22 is $v$-connected.

A reduced extended form $R_i^j$ that is $v$-unconnected for all $v \in V^{j+1}$ is called *primary*. A reduced extended form $R_i^j$ that is $v$-connected for at least one $v \in V^{j+1}$ is called *secondary*. Again, $R_1^4$ shown in Fig. 4.21 is primary, $R_1^4$ shown in Fig. 4.22 is secondary.

Let $R_i^j$ be a reduced extended form such that $S_i^v \neq \emptyset$ for some $v \in V^{j+1}$ and $S_i^w = \emptyset$ for all $w \in V^{j+1} - \{v\}$, then $R_i^j$ is called *$v$-singular*. Figure 4.23 shows a $v$-singular form.

In case that $H_i^j$ is a hierarchy, we know from Lemma 4.5 that the set of level planar embeddings of $H_i^j$ can be represented by the equivalence class of a $PQ$-tree. Figure 4.24 shows an example of an extended form $H_i^j$ and its corresponding $PQ$-tree, representing all permutations of the virtual vertices that appear in some level planar embedding of $H_i^j$. The form $H_i^j$ has two virtual vertices labeled $v$. Figure 4.25 shows an example of a reduced extended form $R_i^j$ and its corresponding $PQ$-tree. The form $R_i^j$ has been constructed from the extended form $H_i^j$ shown in Fig. 4.24 by identifying the two virtual vertices labeled

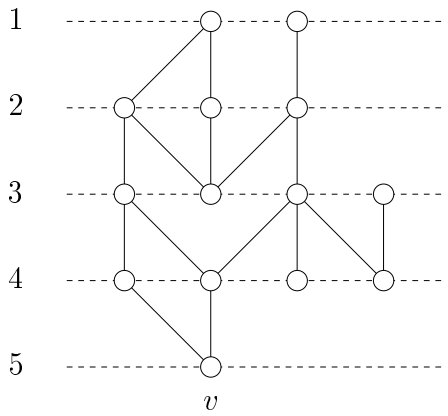



Figure 4.23: A $v$-singular form.

$v$. The corresponding $PQ$-tree has been constructed by reducing the two leaves labeled $v$ applying the pattern matching algorithm of Booth and Lueker (1976), and replacing the pertinent subtree by a single representative.

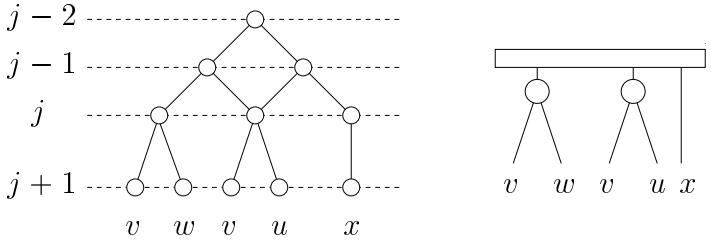



Figure 4.24: An extended form $H_i^j$ and its $PQ$-tree.

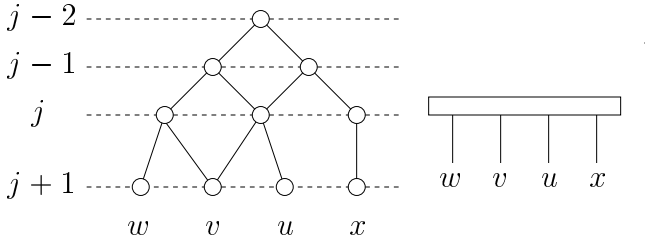


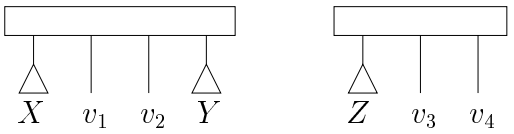

Figure 4.25: The corresponding reduced extended form $R_i^j$ and its $PQ$-tree.

Let $\mathcal{T}(G^j)$, be the set of level planar embeddings of all components of $G^j$. We will show that in case that $G^j$ is level planar, the set of permutations of level-$j$ vertices in level planar embeddings of each component $F_i^j$ of $G^j$ can be described by a $PQ$-tree $T_i$. Clearly this is true, if every $F_i^j$ is a hierarchy. Hence it remains to be shown that it is also possible to maintain a $PQ$-tree for every component $F_i^j$ that is not a hierarchy. Two problems have to be solved in order to get this result.

(i) Singular forms need to be treated correctly.

(ii) An ordering has to be found that merges several forms correctly at the same vertex.

Once a solution has been found for the problem of finding a correct ordering, it seems to be straightforward to handle singular forms correctly. It looks like a good approach not to replace the set of pertinent leaves in every $PQ$-tree by a single representative. This keeps the pertinent leaves and all the information stored in the ML-values of the pertinent nodes in the $PQ$-tree, and seems to avoid the problem of not being able to place singular forms into interior faces.




(a) $T_i^j$ (b) $T_l^j$

Figure 4.26: $PQ$-trees $T_i^j$ and $T_l^j$.

Unfortunately, this does not work. Consider for instance two different $PQ$-trees $T_i^j$, and $T_l^j$ corresponding to two extended forms $H_i^j$, and $H_l^j$ both having some virtual vertices labeled $v$. Assume that $v_1$ and $v_2$ are the virtual vertices labeled $v$ of $H_i^j$, and $v_3$ and $v_4$ are the virtual vertices labeled $v$ of $H_l^j$. Assume further that $\mathrm{LL}(T_i^j) < \mathrm{LL}(T_l^j)$ and the trees have shapes as indicated in Fig. 4.26.

Obviously, the leaves $v_1$ and $v_2$ are between the subtrees $X$ and $Y$ in all permissible permutations of tree $T_i^j$. The leaves $v_3$ and $v_4$ are on one side of the subtree $Z$ in all permutations of $T_l^j$. Assume now that the following inequalities hold:

$$\mathrm{ML}(X, v_1) \;<\; \mathrm{LL}(T_l^j)$$

and

$$\mathrm{ML}(Y, v_2) \;<\; \mathrm{LL}(T_l^j)\ .$$

These inequalities imply that there is enough space to nest the smaller tree $T_l^j$ on either side of the pertinent sequence $v_1, v_2$ within the larger tree $T_i^j$. This implies that the subtree $Z$ must be either between $X$ and the pertinent sequence with respect to $v$ or between $Y$ and the pertinent sequence with respect to $v$ in any level planar embedding of $G$.

As described in Section 4.3, Heath and Pemmaraju proceed as follows:

1. Replace the pertinent sequence $v_1, v_2$ by a single representative $v_i$ and $v_3, v_4$ by a single representative $v_l$.

2. Replace $v_i$ by a new $Q$-node with children $v_i$ and the root of tree $T_l^j$.
3. Reduce the tree $T_i^j$ with respect to $v_i$ and $v_l$.
4. If the reduction was successful, replace $v_i$ and $v_l$ by a new representative $v_{\{i,l\}}$.

After the reduction, $Z$ may appear either between $X$ and $v_{\{i,l\}}$, or between $Y$ and $v_{\{i,l\}}$. Hence the subgraph corresponding to $Z$ is embedded either between the subgraph corresponding to $X$ and the vertex $v$ or between the subgraph corresponding to $Y$ and $v$.

As shown in Section 4.3.2 this approach implies the loss of information stored between the pertinent leaves, and the ML-values $\mathrm{ML}(v_1, v_2)$, and $\mathrm{ML}(v_3, v_4)$ get lost.

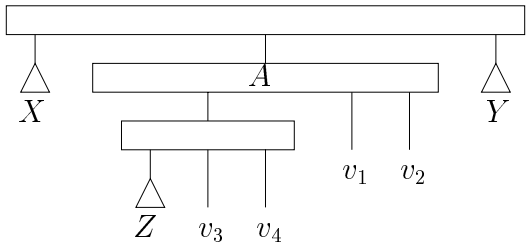



Figure 4.27: Resulting $PQ$-tree after attaching $T_l^j$ to $T_i^j$.

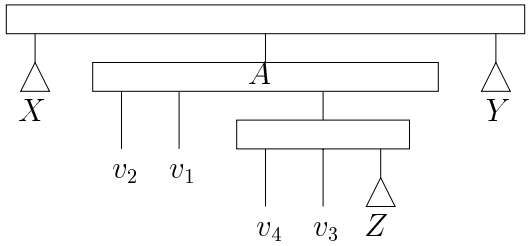



Figure 4.28: Permutation that is not consistent.

However, keeping the pertinent leaves together with all ML-information in the tree, not replacing them by representatives yields problems. Somehow swapping the subtree $Z$ around the pertinent sequence has to be allowed in order to mirror the possible embeddings. This can only be achieved by introducing a new $P$- or $Q$-node $A$ and by making the root of $T_l^j$ and the pertinent subtree of $T_i^j$ children of $A$. However, the set of permutations represented in the new $PQ$-tree is not consistent with the set of permutations of $T_i^j$. Consider for instance the replacement shown in Fig. 4.27. This shows a $PQ$-tree after replacing the pertinent sequence $v_1, v_2$ by new $Q$-node $A$, and adding the sequence $v_1, v_2$ as children to $A$. The permutation shown in Fig. 4.27 is consistent with $\mathrm{PERM}(T_i^j)$ and $\mathrm{PERM}(T_l^j)$. Figure 4.28 shows a permissible permutation of the new $PQ$-tree that is not consistent with $\mathrm{PERM}(T_i^j)$. The permutation $[\mathrm{frontier}(X), v_2, v_1, \mathrm{frontier}(Y)]$ is not in $\mathrm{PERM}(T_i^j)$. Hence, we cannot keep the pertinent sequences of the two $PQ$-trees. Rather we replace the pertinent sequence of every $PQ$-tree by a single representative and the following lemmas justify this approach.

**Lemma 4.11.** *Let $G = (V, E)$ be a $k$-level graph. Let $H_1^j, H_2^j, \ldots, H_l^j$ be $l \geq 2$ extended forms of $G^j$ and $v \in V^{j+1}$ with $S_i^v \neq \emptyset$ for all $i = 1, 2, \ldots, l$. Let* $\text{frontier}(H_i^j) - S_i^v \neq \emptyset$ *for all $i$. Let $G' = (V', E')$ be a $(k+1)$-level graph constructed from $G$ as follows:*

1. *Set $G' := G$.*
2. *Insert for every set $S_i^v$ a new vertex $v_i$.*
3. *For every $i$ and every $\tilde{v} \in S_i^v$ let $(w, \tilde{v})$ be the only incoming edge of $\tilde{v}$ in $H_i^j$, and*
   (a) *remove the edge $(w, v)$ corresponding to $(w, \tilde{v})$ from $G'$, and*
   (b) *add a new edge $(w, v_i)$.*
4. *Add for all $i$ the edge $(v_i, v)$ to $G'$.*
5. *For every level $i = j+1, j+2, \ldots, k$, and every $w \in V^i$ set* $\text{lev}(w) = i + 1$.
6. *Place all new vertices $v_i$ on level $j + 1$.*
7. *Introduce new dummy vertices for new long edges.*

*Then $G$ is level planar if and only if $G'$ is level planar.*

Figure 4.29 shows an example of four extended forms merged at a vertex $v$, where Fig. 4.30 shows an example of the graph $G'$.

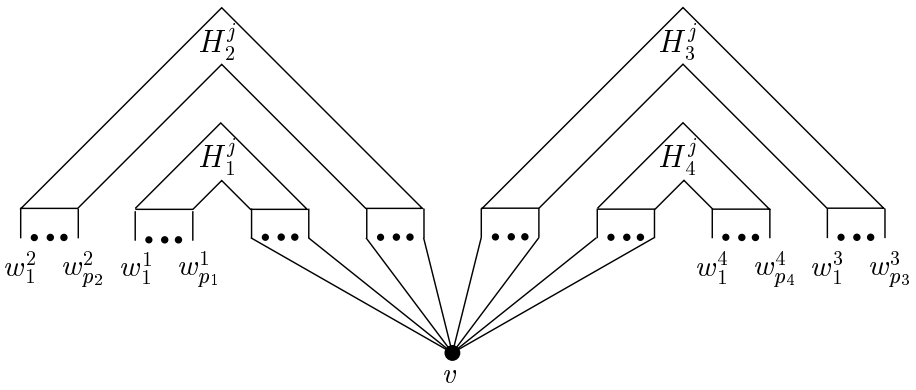



Figure 4.29: Four nonsingular extended forms merged at vertex $v$.

*Proof.* Consider a level planar embedding of $G'$. By removing the dummy vertices and identifying all vertices $v_i$ and the vertex $v$, a level planar embedding of $G$ is constructed. Consider now a level planar embedding of $G$. The assumption that $\text{frontier}(H_i^j) - S_i^v \neq \emptyset$ implies that the edges corresponding to $S_i^v \neq \emptyset$ form a consecutive sequence in the clockwise order of the incoming edges of $v$. Replacing every set of edges corresponding to $S_i^v$ as described in the lemma constructs a level planar embedding of $G'$. □

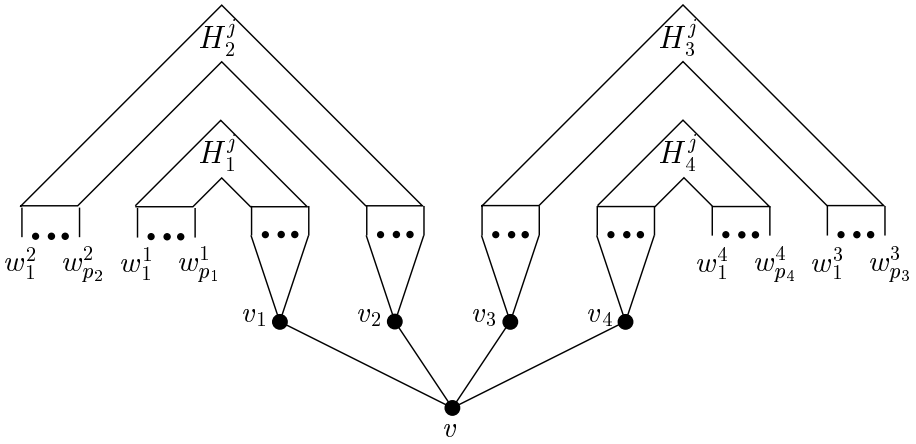



Figure 4.30: A graph $G'$ with vertices $v_i$ inserted.

Lemma 4.11 leaves open in which order pairwise merge operations have to be performed. The lemma only allows to replace every pertinent sequence in a $PQ$-tree by a single representative, as long as no two $PQ$-trees have been merged at that vertex before. If more than two extended forms are adjacent to the same vertex, we are confronted with the fact that after merging and reducing the two corresponding $PQ$-trees two pertinent representatives exist in the new $PQ$-tree that again have to be replaced by a single representative. The following lemma handles this in a suitable way.

**Lemma 4.12.** *Let $G = (V, E)$ be a $k$-level graph. Let $H_1^j, H_2^j, \ldots, H_l^j$ be $l \geq 2$ extended forms of $G^j$ and $v \in V^{j+1}$ with $S_i^v \neq \emptyset$ for all $i = 1, 2, \ldots, l$. Let frontier$(H_i^j) - S_i^v \neq \emptyset$ for all $i$. Let $\pi_l$ be any permutation of $\{1, 2, \ldots, l\}$. Let $G'_{\pi_l} = (V'_{\pi_l}, E'_{\pi_l})$ be a $(k + l - 1)$-level graph constructed from $G$ by performing steps 1 to 3 of Lemma 4.11 and the following steps.*

4. *For every level $i = j + 1, j + 2, \ldots, k$, and every $w \in V^i$ set* $\mathrm{lev}(w) = i + l - 1$.
5. *Place all new vertices $v_i$ on level $j + 1$.*
6. *For every $i = j + 2, j + 3, \ldots, j + l - 1$ insert a single vertex $x_i$ on level $i$.*
7. *Add the edges $(v_{\pi_l(1)}, x_{j+2})$, $(v_{\pi_l(l)}, v)$ and $(x_{j+l-1}, v)$.*
8. *For every $i = 2, 3, \ldots, l - 1$ add the edge $(v_{\pi_l(i)}, x_{j+i})$.*
9. *For every $i = 3, 4, \ldots, l - 1$ add the edge $(x_{j+i-1}, x_{j+i})$.*
10. *Introduce new dummy vertices for long edges.*

*Then $G$ is level planar if and only if there exists a permutation $\pi_l$ such that $G'_{\pi_l}$ is level planar.*

*Proof.* Consider a level planar embedding of $G'_{\pi_l}$. By removing the dummy vertices and identifying the vertices $v, x_{j+l-1}, x_{j+l-2}, \ldots, x_{j+2}$ a level planar embedding of the graph $G'$ is constructed. According to Lemma 4.11 $G$ is level planar.

Consider now a level planar embedding of $G$. As described in Lemma 4.11, a level planar embedding of $G'$ is constructed. Taking the order from left to right of the vertices $v_1, v_2, \ldots, v_l$ as they appear on level $j+1$ in the embedding, a permutation $\pi_l$ is obtained. Introducing the $l-2$ extra vertices on the $l-2$ extra levels, removing the edges $(v_i, v)$, and inserting edges according to the induced permutation $\pi_l$, a level planar embedding of a graph $G'_{\pi_l}$ is constructed. □

Figure 4.31 shows a graph $G'_{\pi_l}$ constructed from our example shown in Fig. 4.30. The permutation is $\pi_l = [2,3,1,4]$. The embedding of $G'$ shown in Fig. 4.30 indicates that it is also possible to choose $\tilde{\pi}_l = [1,2,3,4]$ as permutation for constructing a graph $G'_{\tilde{\pi}_l}$. The graph $G'_{\tilde{\pi}_l}$ is shown in Fig 4.32. Observe the difference between the two graphs. Let $(G'_{\pi_l})^{j+l}$, and $(G'_{\tilde{\pi}_l})^{j+l}$, be the subgraph of $G'_{\pi_l}$ and $G'_{\tilde{\pi}_l}$, respectively, induced by the first $j+l$ levels. Then $(G'_{\pi_l})^{j+l}$ is able to exchange $H_1^j$ and $H_4^j$ without destroying level planarity, while the level planar embedding of $(G'_{\tilde{\pi}_l})^{j+l}$ is fixed (up to reversion).

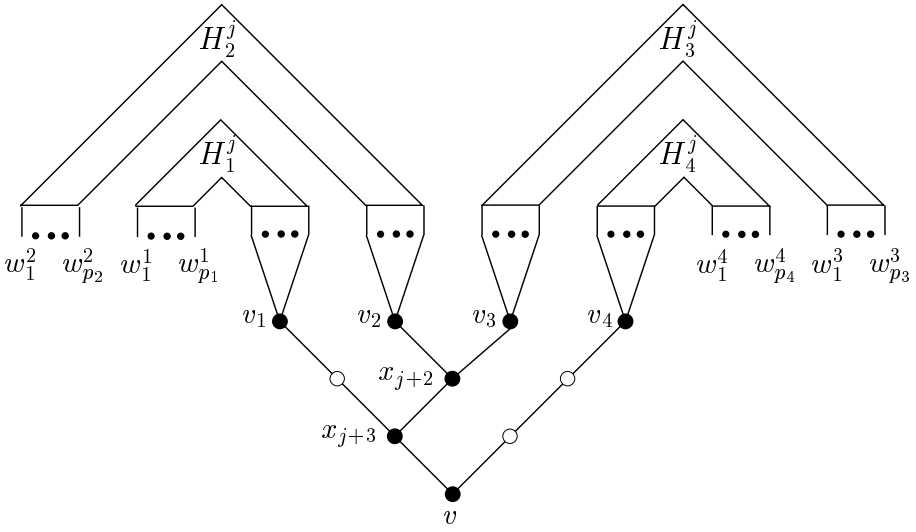



Figure 4.31: The graph $G'_{\pi_l}$ constructed from $G'$ with $\pi_l = [2,3,1,4]$. The black vertices have been introduced by lemma 4.12, the white vertices are new dummy vertices.

The $l-2$ extra vertices $x_{j+2}, x_{j+3}, \ldots, x_{j+l-1}$ each correspond to a pairwise merge operation of two (remaining) forms that have a virtual vertex labeled $v$ in their frontier.

Although according to Lemma 4.12 there exists a permutation such that merging according to this permutation constructs a level planar graph, we are not allowed to chose an arbitrary permutation that seems to fit. Consider the example shown in Fig. 4.32. If a sequence of

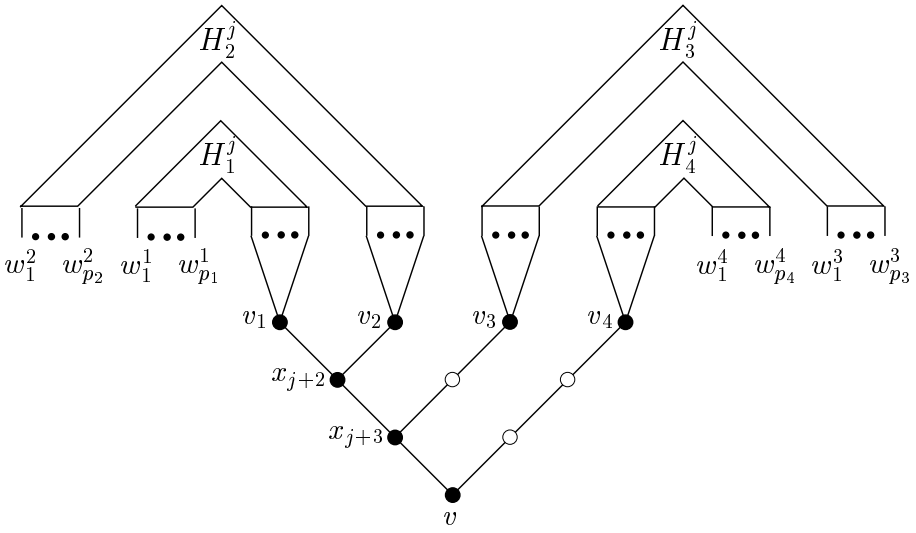



Figure 4.32: The graph $G'_{\tilde{\pi}_l}$ constructed from $G'$ with $\tilde{\pi}_l = [1, 2, 3, 4]$.

merge operations is applied as indicated by the drawing, subsequent reductions in level $l > \text{lev}(v)$ fail, if they require $H_4^j$ to be nested inside of $H_2^j$ and $H_1^j$ to be nested inside of $H_3^j$.

In general, forms are adjacent to more than one vertex $v \in V^{j+1}$. In case that forms are also adjacent to vertices $u_1, u_2, \ldots, u_\mu \in V^{j+1}$, $\mu \geq 1$, and $v \neq u_i$, for all $i = 1, 2, \ldots, \mu$, we not only have to merge the forms at the vertex $v$ but also have to identify all other virtual vertices with the same label $u_i$. We will see in Sections 4.5 and 4.6 how this is implemented.

The following lemma takes $v$-singular forms of $G^j$ into account. Since we need to replace the pertinent sequence by a single representative, the information of the space that is available to place forms within the pertinent sequence gets lost. Hence another strategy has to be applied that is prepared by this lemma.

**Lemma 4.13.** *Let $G = (V, E)$ be a $k$-level graph. Let $H_1^j, H_2^j, \ldots H_l^j$ be $l \geq 2$ extended forms of $G^j$ and $v \in V^{j+1}$ with $S_i^v \neq \emptyset$ for all $i = 1, 2, \ldots, l$. Assume further that for some $1 \leq q < l$*

$$\text{frontier}(H_i^j) - S_i^v \neq \emptyset \qquad \textit{for all } i \in \{1, 2, \ldots, q\}$$

*and*

$$\text{frontier}(H_i^j) = S_i^v \qquad \textit{for all } i \in \{q+1, q+2, \ldots, l\} \ .$$

*Let $\pi_q$ be any permutation of $\{1, 2, \ldots, q\}$. Let $p \in \{1, 2, \ldots, q\}$ be an arbitrary number. Let $G'_{\pi_q}$ be the $(k + q - 1)$-level graph constructed from $H_1^j, H_2^j, \ldots, H_q^j$ according to the rules of Lemma 4.12 if $q \geq 2$. Otherwise let $G'_{\pi_q} = G$. Let $G_{\pi_q}^{v_p} = (V_{\pi_q}^{v_p}, E_{\pi_q}^{v_p})$ be a $(k+q+1)$-level graph constructed from $G'_{\pi_q}$ as follows:*

1. *Set* $G^{v_p}_{\pi_q} := G'_{\pi_q}$.
2. *For every level* $i = j+1, j+2, \ldots, k+q-1$, *and every* $w \in V_i$ *set* $\mathrm{lev}(w) = i+2$.
3. *Insert for every* $H^j_i$, $i = q+1, q+2, \ldots, l$, *a new vertex* $v_i$.
4. *For every* $\tilde{v} \in S^v_i \neq \emptyset$, $i = q+1, q+2, \ldots, l$, *let* $(w, \tilde{v})$ *be the only incoming edge of* $\tilde{v}$ *in* $H^j_i$, *and*
   (a) *remove the edge* $(w, v)$ *corresponding* $(w, \tilde{v})$ *from* $G^{v_p}_{\pi_q}$, *and*
   (b) *add a new edge* $(w, v_i)$.
5. *Place all new vertices* $v_i$, $i = q+1, q+2, \ldots, l$, *on level* $j+1$.
6. *Insert a new vertex* $v_s$ *on level* $j+2$.
7. *For every* $i = q+1, q+2, \ldots, l$, *add the edge* $(v_i, v_s)$.
8. *Add the edge* $(v_s, v_p)$.
9. *Introduce new dummy vertices for long edges.*

*Then* $G$ *is level planar if and only if there exists a permutation* $\pi_q$ *and a vertex* $v_p$, $p \in \{1, 2, \ldots, q\}$, *such that* $G^{v_p}_{\pi_q}$ *is level planar.*

Figure 4.33 shows an example of three nonsingular and two $v$-singular extended forms, merged at a vertex $v$. Figure 4.34 shows a graph $G^{v_1}_{\pi_3}$ constructed from $G_{\pi_3}$.

*Proof.* Consider a level planar embedding of $G^{v_p}_{\pi_q}$. By removing the dummy vertices and identifying the vertices $v$, $v_s$, $v_{q+1}, v_{q+2}, \ldots, v_l$, $x_{j+l-1}, x_{j+l-2}, \ldots, x_{j+2}$, and $v_1, v_2, \ldots, v_q$ a level planar embedding of $G$ is constructed.

Consider now a level planar embedding $\mathcal{E}$ of $G$. Let $r \in \{q+1, q+2, \ldots, l\}$ such that $\mathrm{LL}(H^j_r) \leq \mathrm{LL}(H^j_i)$ for all $i \in \{q+1, q+2, \ldots, l\}$. Let $\tilde{\mathcal{E}}$ be the level planar embedding arising from $\mathcal{E}$ by placing all $H^j_i$ for all $i \in \{q+1, q+2, \ldots, l\} - \{r\}$ next to $H^j_r$ in a sequence. (This embedding is obviously level planar since all $v$-singular forms are "smaller" than $H^j_r$.) The following two cases may occur.

1. The incoming edges corresponding to $S^v_r$ appear within the incoming edges corresponding to $S^v_i$ of some $H^j_i$, $i \in \{1, 2, \ldots, q\}$ (Hence, the incoming edges corresponding to $S^v_i$ do not form a consecutive sequence in the clockwise order of incoming edges of $v$.): Set $p := i$.
2. The incoming edges corresponding to $S^v_r$ appear next to the incoming edges corresponding to $S^v_i$ of some $H^j_i$, $i \in \{1, 2, \ldots, q\}$: Set $p := i$.

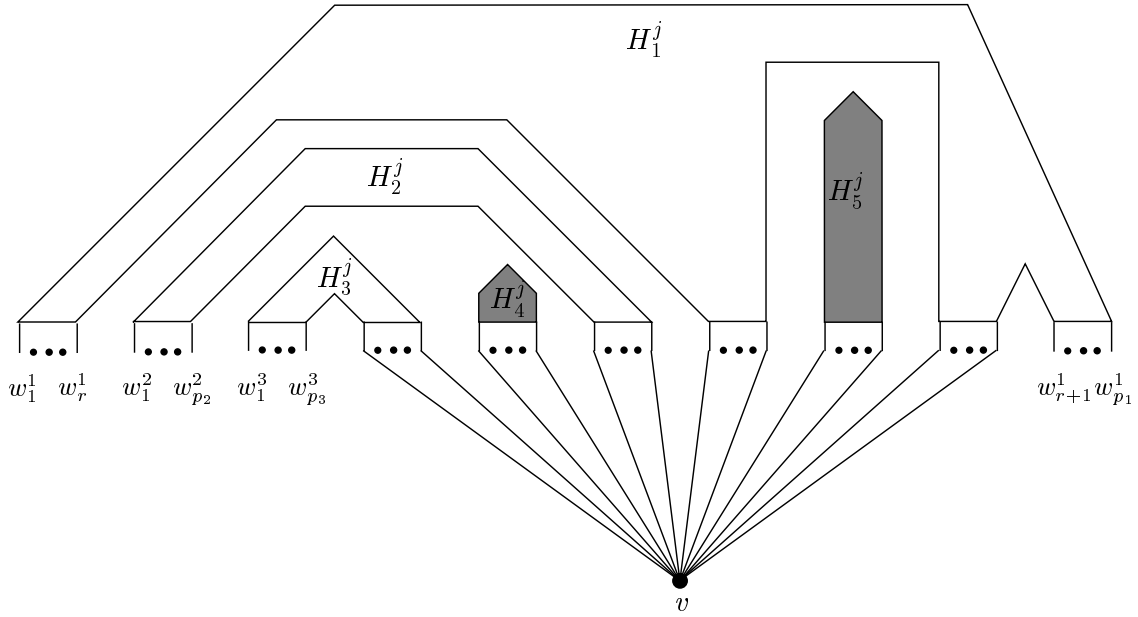



Figure 4.33: Three nonsingular and two $v$-singular extended forms merged at vertex $v$.

We now add vertices $v_1, v_2, \ldots, v_q$ for the nonsingular forms as described in Lemma 4.11, constructing a graph $\tilde{G}'$, transforming the embedding $\tilde{\mathcal{E}}$ correspondingly into $\tilde{\mathcal{E}}'$. Taking the order of the vertices $v_1, v_2, \ldots, v_q$ as they appear on level $j+1$ in the embedding $\tilde{\mathcal{E}}'$, we get a permutation $\pi_q$. Introducing the extra vertices on the extra levels and inserting edges according to the induced permutation $\pi_q$ a level planar embedding of a graph $G'_{\pi_q}$ is constructed. We then introduce the extra vertices $v_i$, $i = q+1, q+2, \ldots, l$ and $v_s$ at the position of $H_i^j$, $i = q+1, q+2, \ldots, l$ (all appearing consecutively in $\tilde{G}$) and expanding $G'_{\pi_q}$ by two levels we get a level planar embedding of $G_{\pi_q}^{v_p}$. □

If the highest $v$-singular form can be embedded level planar into $G'_{\pi_q}$, all other $v$-singular forms can be embedded as well, taking advantage of the fact that these forms all can be embedded next to the highest $v$-singular form. The position, where the highest $v$-singular form is embedded, is defined by either two consecutive incoming edges of $v$, belonging to nonsingular forms, or by one of the endmost incoming edges also belonging to nonsingular forms. Thus we need to detect if there are incoming edges of $v$ such that the $v$-singular forms can be embedded between them.

Using $PQ$-trees, an interpretation for placing a $v$-singular form $R_i^j$ between two consecutive incoming edges of $v$ corresponding to some other form $R_l^j$, can be given. The $PQ$-tree $T_i$ corresponding to the reduced extended form $R_i^j$ is added to the pertinent subtree of the $PQ$-tree $T_l$ corresponding to $R_l^j$. More precisely, the $PQ$-tree $T_i$ is added as a child to a node of the pertinent subtree of $T_l$. The following lemma handles this in a suitable way.

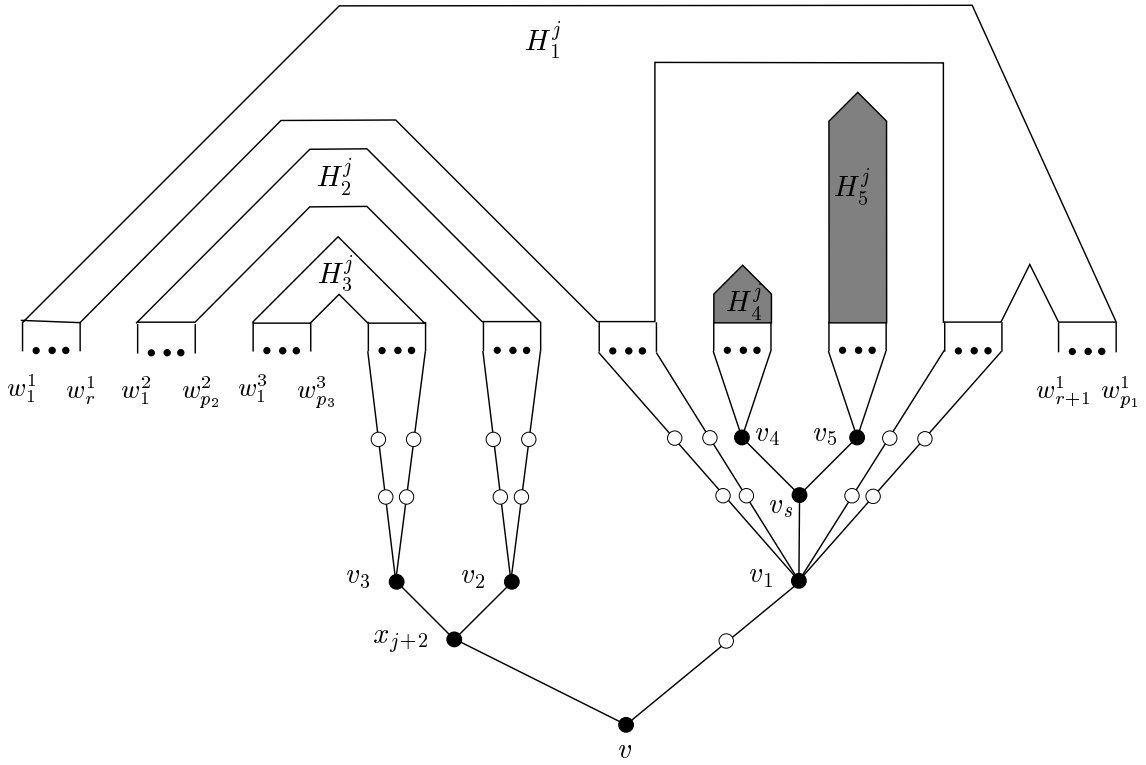



Figure 4.34: The graph $G_{\pi_3}^{v_1}$ constructed from $G'_{\pi_3}$.

**Lemma 4.14.** *Let $H_1^j$ and $H_2^j$ be two forms such that $H_1^j$ and $H_2^j$ both have at least one virtual vertex labeled $v$. Let $T_1$ and $T_2$ be the corresponding PQ-trees of $H_1^j$ and $H_2^j$ such that the trees $T_1$ and $T_2$ both have been reduced with respect to $v$ and the sets of leaves labeled $v$ have not been replaced by single representatives. Assume further that $\mathrm{LL}(T_1) \leq \mathrm{LL}(T_2)$, and that $S_2^v = \mathrm{frontier}(T_2)$. Let $X$ be the pertinent root of $S_1^v$. Then there exists a level planar embedding of $H_1^j \cup_v H_2^j$ such that $H_2^j$ is embedded into an interior face of $H_1^j$, if one of the following two conditions for $X$ holds.*

1. *$X$ is a $P$-node and the inequality $\mathrm{ML}(X) < \mathrm{LL}(T_2)$ holds.*
2. *$X$ is a $Q$-node with ordered children $X_1, X_2, \ldots, X_t$ where $X_\mu, X_{\mu+1}, \ldots, X_\nu$, $1 \leq \mu < \nu \leq t$, is the pertinent sequence of children of $X$, and for some $a \in \{\mu+1, \mu+2, \ldots, \nu\}$ the inequality $\mathrm{ML}(X_{a-1}, X_a) < \mathrm{LL}(T_2)$ holds.*

*Proof.* Assume that $X$ is a $P$-node and let $Y$ and $Z$ be two children of $X$. It follows from $\mathrm{ML}(X) < \mathrm{LL}(T_2)$ that $\mathrm{ML}(\mathrm{frontier}(Y) \cup \mathrm{frontier}(Z)) < \mathrm{LL}(T_2)$. Since $Y$ and $Z$ are children of a $P$-node there exists a leaf $y \in \mathrm{frontier}(Y)$ and a leaf $z \in \mathrm{frontier}(Z)$ and a permutation $\pi \in \mathrm{PERM}(T_1)$ such that $y$ and $z$ appear consecutively. The permutation $\pi$ witnesses a level planar embedding $\mathcal{E}$ of $H_1^j$. Identifying the virtual vertices $S_1^v$ in $\mathcal{E}$ constructs a level planar embedding $\mathcal{E}'$ with an interior face large enough to embedded $H_2^j$ level planar in it. The proof for $X$ being a $Q$-node is analogous. □

The Lemma 4.14 reveals a strategy to handle the singular forms. Since we want to replace the pertinent sequences of $T_1$ and $T_2$ by single representatives (in order to permit the construction of $PQ$-trees that are not consistent with their origins) before the tree $T_2$ is attached to $T_1$, we have to install a system to keep the information, where actually to place $T_2$. Hence, for every representative $v'$ two numbers $\mathrm{PML}(v')$ and $\mathrm{QML}(v')$ are introduced. In case that the root $X$ of the pertinent subtree of a $PQ$-tree is a $P$-node, set

$$\mathrm{PML}(v') = \mathrm{ML}(X) \ .$$

In case that $X$ is a $Q$-node, we set

$$\mathrm{QML}(v') = \min \left\{ \mathrm{ML}(Y,Z) \;\middle|\; \begin{array}{l} Y, Z \text{ consecutive children of } X, \\ Y \text{ and } Z \text{ pertinent} \end{array} \right\} \ .$$

Using Lemma 4.14 the information necessary for attaching $T_2$ to $T_1$ is now available at the single representative. We need to consider how the information of the single representatives is maintained, when representatives of different $PQ$-trees are reduced and replaced by a new single representative.

When merging two nonsingular forms while reducing and replacing their single representatives $v_1$ and $v_2$ by a new single representative $v_{\{1,2\}}$, the numbers $\mathrm{PML}(v_{\{1,2\}})$ and $\mathrm{QML}(v_{\{1,2\}})$ have to be computed by taking the minimum of the PML- and QML-values of $v_1$ and $v_2$. Here we need to take into account that merging two forms may yield something that we call a *cavity*. Considering the intersection $\mathcal{C}$ of the half space $\{x \in \mathbb{R}^2 \mid x_2 \geq k - j - 1\}$ and the outer face of a level planar drawing of the current extended forms, a $v$-cavity $\mathcal{C}_v$ is defined to be a region of $\mathcal{C}$ such that $v$ is adjacent to the region. Obviously $v$ can be adjacent to several such regions. Moreover, these regions are not unique, since they depend on the current embedding. This is no drawback, since we only need to maintain a lower bound on the size of the largest $v$-cavity which can be easily implemented using the $PQ$-trees and the LL- and ML-values of Heath and Pemmaraju (1995, 1996). Figure 4.35 shows such a $v$-cavity. The arrow on the right side of the figure determines the height of the cavity. A $v$-singular form can only be level planar embedded within this cavity, if it is smaller than the height of the cavity. We define $\mathrm{LL}(C_v)$ to be the *low indexed level* of a $v$-cavity $\mathcal{C}_v$ as $\mathrm{ML}(\{w \in V^{j+1} \mid w \text{ is on the boundary of } \mathcal{C}_v\})$. The height of a $v$-cavity $C_v$ is $j + 1 - \mathrm{LL}(C_v)$. The following lemma reveals how to obtain a lower bound on the height of the largest $v$-cavity in every level planar embedding of two forms that have been $v$-merged. If two $PQ$-trees have been merged at their leaves labeled $v$, the LL-value of a $v$-cavity is obviously smaller or equal to the ML-value stored at the root of the pertinent subtree. Notice that we here make use of the ML-values that have been "technically" set during the merge operations.

**Lemma 4.15.** *Let $R_1^j$ and $R_2^j$ be two nonsingular reduced extended forms of $G^j$ of a level planar graph $G$ with $S_1^v \neq \emptyset$ and $S_2^v \neq \emptyset$. Let $T_1$ and $T_2$ be their corresponding $PQ$-trees with $\mathrm{LL}(T_1) \leq \mathrm{LL}(T_2)$, representing all level planar embeddings of $R_1^j$, and $R_2^j$, respectively. Let $T$ be the $PQ$-tree constructed by $v$-merging $T_2$ into $T_1$. Let $X$ be the root of the pertinent*

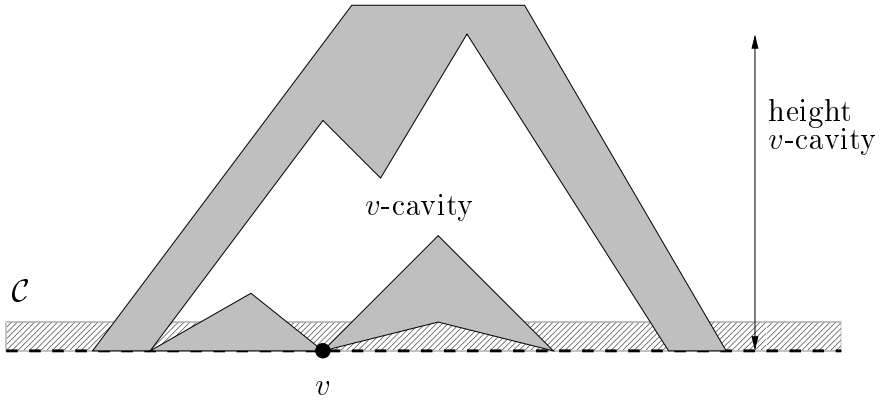



Figure 4.35: A $v$-cavity.

*subtree in $T$. If $X$ is a $P$-node let $h = \mathrm{ML}(X)$, and if $X$ is a $Q$-node let $Y$ and $Z$ be the (only) pertinent children of $X$ and let $h = \mathrm{ML}(Y, Z)$. Let $\mathcal{C}_v$ be the largest $v$-cavity in an arbitrary level planar embedding of $R_1^j \cup_v R_2^j$. Then the following holds.*

$$\mathrm{LL}(C_v) \leq h.$$

*Proof.* By construction of the merge operation we have that $h < \mathrm{LL}(T_2)$ holds. Since $R_2^j$ can be embedded level planar in $R_1^j$, there must exist in every level planar embedding at least one $v$-cavity $\mathcal{C}_v$ such that $\mathrm{LL}(C_v) \leq h$. □

If the $PQ$-tree that has been constructed by a merge operation indeed represents all level planar embeddings of the corresponding merged forms, the result of lemma 4.15 is easily expanded to iterated $v$-merge operations. This allows us to apply a sequence of reduce and merge operations of single representatives $v_1, v_2, \ldots, v_l$. Replacing them by a new single representative $v'$, the PML- and QML-values are chosen by taking the minimum of the values $\mathrm{PML}(v_i)$ and $\mathrm{QML}(v_i)$ for all $i = 1, 2, \ldots, l$ and the ML-values stored at the root of the pertinent subtree.

The second task was to find an ordering that merges the forms correctly at the same vertex. In the next two chapters, we present results that justify an ordering of merging the forms according to their height. The results are more technical.

## 4.5 Correct Level Planarity Testing

In this section, a detailed description of the level planarity test is given. Using a function CHECK-LEVEL that computes for every level $j = 2, 3, \ldots, k$, the set $\mathcal{T}(G^j)$ of $PQ$-trees representing the possible permutations of $V^j$ in level planar embeddings of the components of $G^j$, the algorithm LEVEL-PLANARITY-TEST can be formulated as follows.

Bool **LEVEL-PLANARITY-TEST**($G = (V^1, V^2, \ldots, V^k; E)$)

```
begin
   Initialize 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";
   return "true";
end.
```

The procedure CHECK-LEVEL computes all possible permutations of $V^{j+1}$ in level planar embeddings of the components of $G^{j+1}$ returning a set of $PQ$-trees $\mathcal{T}(G^{j+1})$. The procedure CHECK-LEVEL is diveded into two phases. The *first reduction phase* constructs the $PQ$-trees corresponding to the reduced extended forms of $G^j$. Every $PQ$-tree $T(F_i^j)$ that represents all level planar embeddings of some component $F_i^j$ is transformed into a $PQ$-tree representing all level planar embeddings of the extended form $H_i^j$. This is easily done by using the function REPLACE as described in Section 3.2.

The first reduction phase then reduces in every $PQ$-tree $T(H_i^j)$, $i = 1, 2, \ldots, m_j$, all leaves with the same label, thereby constructing a new $PQ$-tree, representing all level planar embeddings of $H_i^j$, where leaves with the same label occupy consecutive positions. If one of the reductions fails, then $G$ cannot be level planar. Leaves with the same label $v$ are replaced by a single representative $v_i$. Such a single representative $v_i$ gets the same label $v$, storing either $\mathrm{PML}(v_i) = \mathrm{ML}(X)$ if the root $X$ of the pertinent subtree was a $P$-node or $\mathrm{QML}(v_i) = \min\{\mathrm{ML}(Y, Z) \mid Y, Z \text{ consecutive, pertinent children of } X\}$, if the root was a $Q$-node. The value of the undefined variable of $\mathrm{QML}(v_i)$ and $\mathrm{PML}(v_i)$ is set to $k + 1$. The representative corresponds to the newly introduced node $v_i$ of Lemma 4.11

$PQ$-trees of several reduced extended forms are merged in the *second reduction phase* using a function INSERT if the forms are adjacent to the same vertex $v$ on level $j + 1$. The $PQ$-trees corresponding to the reduced extended forms are merged pairwise and according to their height. Initially, the $PQ$-trees of the two highest forms are merged to construct a $PQ$-tree $T$. We then merge $PQ$-trees of the smaller forms into $T$, always considering the $PQ$-tree of the highest remaining form first. It is shown in the next section that using this ordering a $PQ$-tree $T$ is constructed, representing all possible level planar embeddings of the merged form. If there are more than one $v$-singular reduced extended forms for some $v \in V^{j+1}$, we only need to consider the highest one of these forms acccording to Lemma 4.13.

Situations may occur, where several forms are not only adjacent to a common vertex $v \in V^{j+1}$ but also to other common vertices $w \in V^{j+1}$, $w \neq v$. Thus, the corresponding $PQ$-trees that have to be $v$-merged contain several leaves with common label $w \neq v$. After the $PQ$-trees have been $v$-merged, these leaves have to be reduced as well. If one of the

reductions applied in this phase fails, the graph $G$ is not level planar. The PML- and QML-values are updated after every reduction using a function UPDATE.

Finally, a $PQ$-tree is added for every source of $V^{j+1}$, and the set of $PQ$-trees constructed by the function CHECK-LEVEL represents all level planar embeddings of the components $G^{j+1}$.

The following code fragment contains operations that perform on the graph $G$. They are kept in the code for documentational purposes. Any implementation would of course rely only on the manipulation of $PQ$-trees.

$\mathcal{T}(G^{j+1})$ **CHECK-LEVEL**$(\mathcal{T}(G^j), V^{j+1})$

begin
**First Reduction Phase**
for every component $F_i^j$ in $G^j$ and its corresponding $PQ$-tree in $T(F_i^j)$ do
construct $H_i^j$;
construct $T(H_i^j)$ (from the $PQ$-trees obtained in the previous iteration);
for every $v \in V^{j+1}$ do
for every extended form $H_i^j$ do
if $S_i^v \neq \emptyset$ then
if REDUCE$(T(H_i^j), S_i^v) = \emptyset$ then return $\emptyset$;
else
let $v_i$ be a single representative of $S_i^v$;
REPLACE($S_i^v$,$v_i$);
determine PML($v_i$) and QML($v_i$);
for every extended form $H_i^j$ do
$T(R_i^j) := T(H_i^j)$;
**Second Reduction Phase**
for every $v \in V^{j+1}$ do
reorder indices such that $S_1^v, S_2^v, \ldots, S_p^v \neq \emptyset$, and $S_{p+1}^v, S_{p+2}^v, \ldots, S_{m_j}^v = \emptyset$;
$W := \{w \in V^{j+1} \mid \exists i, l \in \{1, 2, \ldots, p\}, i \neq l, S_i^w \neq \emptyset \text{ and } S_l^w \neq \emptyset\}$;
let $q$ be the number of $v$-singular reduced extended forms;
eliminate all $v$-singular $R_i^j$ except for the one with the lowest LL-value;
renumber the remaining $R_i^j$ from 1 to $p - q + 1$;
$p := p - q + 1$;
sort the $R_i^j$ such that $\text{LL}(R_1^j) \leq \text{LL}(R_2^j) \leq \text{LL}(R_3^j) \leq \cdots \leq \text{LL}(R_p^j)$;
for $i = 2$ to $p$ do
$T(R_1^j) :=$ INSERT$(T(R_1^j), T(R_i^j), v)$;
$R_1^j := R_1^j \cup_v R_i^j$;
if REDUCE$(T(R_1^j), S_1^v) = \emptyset$ then return $\emptyset$;
else

```
            let v' be a new single representative of S_1^v;
            UPDATE(S_1^v,v');
            REPLACE(S_1^v,v');
        for every w ∈ W do
            if REDUCE(T(R_1^j), S_1^w) = ∅ then return ∅;
            else
                let w' be a new single representative of S_1^w;
                UPDATE(S_1^w,w');
                REPLACE(S_1^w,w');
    add for every source a corresponding PQ-tree to T(G^j);
    return T(G^j);
end.
```

We now give a method INSERT that merges two $PQ$-trees. INSERT itself uses the function MERGE as it has been introduced in Section 4.3.1 and guarantees correct treatment of singular forms. Let $T_{large}$ and $T_{small}$ be two $PQ$-trees such that $S^v_{large} \neq \emptyset$ and $S^v_{small} \neq \emptyset$, and $\mathrm{LL}(T_{large}) \leq \mathrm{LL}(T_{small})$. Assume further that $S^v_{large}$ and $S^v_{small}$ have been reduced and replaced by single representatives $v_{large}$ and $v_{small}$, respectively, and that $S^v_{large} = \{v_{large}\}$, and $S^v_{small} = \{v_{small}\}$, respectively. INSERT returns a new $PQ$-tree $T_{merge}$. The method does not reduce the pertinent sequence, nor does it replace pertinent leaves by a single leaf. Observe that in case of $\mathrm{frontier}(T_{small}) = S^v_{small}$, we do not really add $T_{small}$ to $T_{large}$, if $T_{small}$ can be added to the former pertinent subtree of $T_{large}$. This merge operation leaves $T_{large}$ unchanged.

$T_{merge}$ **INSERT**$(T_{large}, T_{small}, v)$

```
begin
    if frontier(T_small) ≠ S^v_small then
        T_large := MERGE(T_large,T_small,v);
    else if PML(v_large) ≠ k + 1 then
        if PML(v_large) < LL(T_small) then
            do nothing;
        else
            T_large := MERGE(T_large,T_small,v);
    else if QML(v_large) ≠ k + 1 then
        if QML(v_large) < LL(T_small) then
            do nothing;
        else
            T_large := MERGE(T_large,T_small,v);
    return the new PQ-tree T_large;
end.
```

The method UPDATE is a straightforward implementation of finding a lower bound on the height of a cavity that could possibly embed singular components.

void **UPDATE**($S_i^v$,$v'$)

begin
  $\mathrm{PML}_{\min} := \min\{\mathrm{PML}(\tilde{v}) \mid \tilde{v} \in S_i^v\}$;
  $\mathrm{QML}_{\min} := \min\{\mathrm{QML}(\tilde{v}) \mid \tilde{v} \in S_i^v\}$;
  let $X$ be the root of the pertinent subtree.
  if $X$ is a $P$-node then
    $\mathrm{PML}_X := \mathrm{ML}(X)$;
  else
    $\mathrm{QML}_X := \min\left\{\mathrm{ML}(Y,Z) \;\middle|\; \begin{array}{l} Y,Z \text{ consecutive children of } X, \\ Y \text{ and } Z \text{ pertinent} \end{array}\right\}$;
  if $\min\{\mathrm{PML}_{\min}, \mathrm{PML}_X\} < \min\{\mathrm{QML}_{min}, \mathrm{QML}_X\}$ then
    $\mathrm{PML}(v') := \min\{\mathrm{PML}_{\min}, \mathrm{PML}_X\}$;
    $\mathrm{QML}(v') := k+1$;
  else
    $\mathrm{QML}(v') := \min\{\mathrm{QML}_{\min}, \mathrm{QML}_X\}$;
    $\mathrm{PML}(v') := k+1$;
end.

## 4.6 Proving the Correctness

In this section we prove the correctness of the level planarity test. The strategy is to apply an inductive argument. Since a subgraph of $G$ that is induced by a source and its outgoing edges is a trivial hierarchy, we know by Lemma 4.5 that for such a subgraph there exists a $PQ$-tree that represents the set of level planar embeddings. We need to show that throughout every iteration the $PQ$-trees are correctly maintained and the set of permissible permutations always represents exactly the set of level planar embeddings of the corresponding form.

In Lemma 4.16, the first reduction phase is proven to be correct. Proving the correctness of the second reduction phase is more involved. We show in Lemmas 4.18 and 4.19 that merging a set of $PQ$-trees at their leaves labeled $v$ is performed correctly if the functions INSERT and REDUCE are applied as described in the Section 4.5. If several reduced extended forms have been $v$-merged, the new form may contain several vertices with a same label $w \neq v$. Lemma 4.20 shows that the reduction of these leaves labeled $w$ in the corresponding $PQ$-tree is performed correctly.

We start with a lemma on the correctness of the first reduction phase. Let us assume that we are given a $k$-level planar graph $G$, an extended form $H_i^j$, $1 \leq j < k$, of $G$, and a $PQ$-tree $T_i$ that represents all level planar embeddings of $H_i^j$. To prove the correctness of the first phase we show that there exists a $PQ$-tree $\tilde{T}_i$ equivalent to $T_i$, such that all leaves with a common label appear consecutively in the frontier of $\tilde{T}_i$. If such a $PQ$-tree exists, we

are obviously able to reduce for every $v \in V^{j+1}$ the $PQ$-tree $T_i$ with respect to the leaves labeled $v$ and to replace these leaves by a single representative. It is easy to see that this new $PQ$-tree represents level planar embeddings of the reduced extended form $R_i^j$ and it remains to show that the $PQ$-tree represents exactly all level planar embeddings of $R_i^j$.

**Lemma 4.16.** *Let $G = (V, E)$ be a level planar graph with $k > 1$ levels. Let $F_i^j$, $i \in \{1, 2, \ldots, m_j\}$, be an arbitrary component of $G^j$, $1 \leq j < k$, and let $H_i^j$ be its extended form and $R_i^j$ be its reduced extended form. If $T_i$ is a $PQ$-tree representing all level-planar embeddings of $H_i^j$, the $PQ$-tree $T_i'$ constructed from $T_i$ by reducing every set $S_i^v$ and replacing it by a single representative $v_i$ witnesses all level planar embeddings of $R_i^j$.*

*Proof.* Although the results of the lemma should be clear by the previous discussions, we give the proof in full detail. We first show that there exists a $PQ$-tree $\tilde{T}_i$ that is equivalent to $T_i$, such that for all $v \in V^{j+1}$, the leaves corresponding to $S_i^v$ occupy consecutive positions in the frontier of $\tilde{T}_i$.

Consider an arbitrary level planar embedding $\mathcal{E}(R_i^j)$ of the reduced extended form $R_i^j$ and let $\pi$ be the witness of $\mathcal{E}(R_i^j)$. The level-$j$ neighbors $w \in V(F_i^j)^j$ of $v \in V^{j+1}$ in $F_i^j$ form a consecutive sequence on level $j$ in $\mathcal{E}(R_i^j)$ (except for possible sinks). Every edge $e = (w, v)$, $w \in V(F_i^j)^j$, $v \in V^{j+1}$ corresponds to a virtual vertex of $S_i^v$. Therefore, we get a level planar embedding of $\mathcal{E}(H_i^j)$ of $H_i^j$ by replacing every edge $e$ by a virtual edge with an incident virtual vertex labeled $v$ in $\mathcal{E}(R_i^j)$. See Fig. 4.36 for an illustration. Let $\pi'$ be a witness to $\mathcal{E}(H_i^j)$. By construction, all virtual vertices labeled $v$ form a consecutive sequence in $\pi'$ for every $v \in V^{j+1}$. Since $\mathcal{E}(H_i^j)$ is a level planar embedding of $H_i^j$, its witness $\pi'$ must be in $\mathrm{PERM}(T_i)$. Thus there exists a $PQ$-tree $\tilde{T}_i$ equivalent to $T_i$ with $\mathrm{frontier}(\tilde{T}_i) = \pi'$.

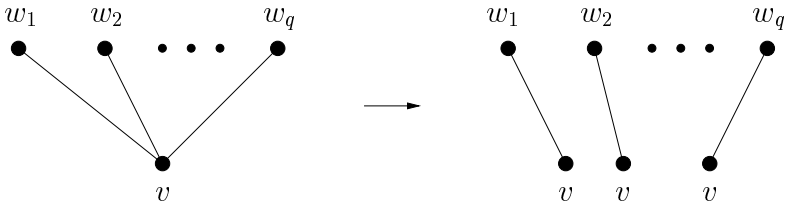



Figure 4.36: Illustration of the proof of Lemma 4.16. For every edge $(w_i, v)$ $w_i \in V(F_i^j)^j$, $i = 1, 2, \ldots, q$, a virtual edge with a virtual vertex labeled $v$ is introduced.

The existence of the $PQ$-tree $\tilde{T}_i$ that is equivalent to $T_i$ guarantees that the reduction of $T_i$ with respect to $S_i^v$ for every $v \in V^{j+1}$ is successful. These reductions construct a $PQ$-tree $\tilde{T}_i'$ with $\mathrm{PERM}(\tilde{T}_i') \subseteq \mathrm{PERM}(T_i)$. Furthermore we have $\pi' \in \mathrm{PERM}(\tilde{T}_i')$ and we may assume that $\pi' = \mathrm{frontier}(\tilde{T}_i')$.

Replacing in $\tilde{T}_i'$ all leaves with a common label by a single representative, we obtain a $PQ$-tree $T_i'$, where we have by construction for the witness $\pi$ of $\mathcal{E}(R_i^j)$ that $\pi \in \mathrm{PERM}(T_i')$. Thus $T_i'$ represents all level planar embeddings of $R_i^j$. □

The next lemma shows a more technical result that is needed for proving the correctness of the second reduction phase. The lemma is motivated by Observations 4.9 and 4.10. Let $R_i^j$ and $R_l^j$ be two reduced extended forms and let $T_i$ and $T_l$ be their corresponding $PQ$-trees where $\mathrm{LL}(T_i) \leq \mathrm{LL}(T_l)$. When merging the $PQ$-trees $T_i$ and $T_l$ at leaves labeled $v$, we insert the $PQ$-tree $T_l$ into the $PQ$-tree $T_i$. After applying any of the merge operations presented in 4.3.1, the tree $T_l$ is completely contained as a subtree of $T_i$. While the frontier of $T_i$ has changed (by inserting $T_l$ as a subtree) the frontier of $T_l$ has not changed at all. Hence, all leaves in frontier($T_l$), including the leaf labeled $v$, form a consecutive sequence in the new $PQ$-tree $T_i$.

This implies that if we want to use these merge operations for $PQ$-trees, the level-(j+1) vertices of $R_l^j$ must form a consecutive sequence on level $j+1$ in every level planar embedding of $R_i^j \cup_v R_l^j$. However, this is not the case in general. Consider the example shown in Fig. 4.37 showing four reduced extended forms $R_1^j, R_2^j, R_3^j, R_4^j$ that have been $v$-merged. The forms are constructed similarly to the components $F_1, F_2, F_3, F_4$ that are shown in the counterexample of Fig. 4.16 on page 66. If we first $v$-merge $R_4^j$ into $R_1^j$ and then $v$-merge $R_3^j$ into $R_1^j$ and then $v$-merge $R_2^j$ into $R_1^j$ we know already from Section 4.3.2 that the $PQ$-tree constructed by this sequence of merge operations is not reducible (see Fig. 4.17 on Page 66). In fact, there exist level planar embeddings of $R_1^j \cup_v R_2^j \cup_v R_3^j \cup_v R_4^j$ such that the virtual vertices of $R_2^j$ do not form a consecutive sequence on level $j+1$. Such an embedding is shown in Fig. 4.37 where the virtual vertices $w_1^2, w_2^2, \ldots, w_{q_2}^2$ of $R_2^j$ and the vertex $v$ are separated by the virtual vertices $w_1^3, w_2^3, \ldots, w_{q_3}^3$ of $R_3^j$.

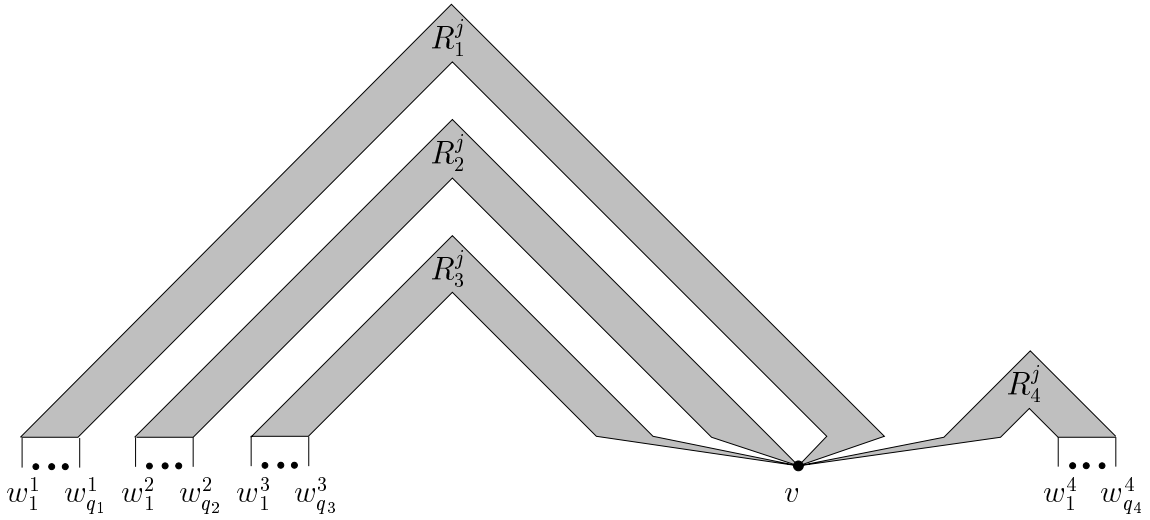



Figure 4.37: Level planar embedding of the components $R_1^j \cup_v R_2^j \cup_v R_3^j \cup_v R_4^j$ where the virtual vertices $w_1^2, w_2^2, \ldots, w_{q_2}^2$ are separated from $v$ by $w_1^3, w_2^3, \ldots, w_{q_3}^3$.

If we want to use the merge operations, we have to guarantee that in all level planar embeddings of two $v$-merged forms, the virtual vertices of the smaller form appear consecutively

on level $j+1$. As the counterexample shows, this does not necessarily hold for every merge operation.

The following two lemmas show that if there are two or more reduced extended forms that have to be $v$-merged, there exists an ordering such that pairwise $v$-merging the forms according to this ordering guarantees the following. When $v$-merging two forms, the virtual vertices of the smaller form always form a consecutive sequence in all level planar embeddings of the merged form. The ordering is obtained by sorting the forms according to their LL-values. We merge the two reduced extended forms with lowest LL-value (that is, we merge the two largest forms). This constructs a new form, say $F$, and we then start merging the largest remaining form into $F$ until all forms are merged into $F$.

Since the order of merging the forms is very important, we expand our terminology. Let $H_1^j, H_2^j, \ldots, H_p^j$, $p \geq 2$, be extended forms of $G^j$ such that $S_i^v \neq \emptyset$ for all $i \in \{1, 2, \ldots, p\}$. Assume without loss of generality that

$$\mathrm{LL}(H_1^j) \leq \mathrm{LL}(H_2^j) \leq \mathrm{LL}(H_3^j) \leq \cdots \leq \mathrm{LL}(H_p^j) \ .$$

Let $F$ be the subgraph constructed by $v$-merging $H_1^j, H_2^j, \ldots, H_p^j$. Thus, $F$ equals $H_1^j \cup_v H_2^j \cup_v \cdots \cup_v H_p^j$. If for some vertex $w \neq v$ the sets $S_i^w$ and $S_l^w$ of two extended forms $H_i^j$ and $H_l^j$, $i \neq l$, are not empty, the virtual vertices in these sets are not identified in $F$. Thus all virtual vertices with common label are kept separate except for the virtual vertices labeled $v$.

Let $H_{\{1,2,\ldots,i\}}^j = H_1^j \cup_v H_2^j \cup_v \cdots \cup_v H_i^j$ denote the form that is constructed by $v$-merging $H_1^j, H_2^j, \ldots, H_i^j$ in this order. (In our previous terminology, which is more useful to describe the algorithm, $H_{\{1,2,\ldots,i\}}^j$ is renamed into $H_1^j$.) Obviously, we have that $H_{\{1,2,\ldots,p\}}^j = F$. Furthermore, let $R_1^j, R_2^j, \ldots, R_p^j$ be the reduced extended forms of $H_1^j, H_2^j, \ldots, H_p^j$, and define $R_{\{1,2,\ldots,i\}}^j$ analogously to $H_{\{1,2,\ldots,i\}}^j$.

For an extended form $H_i^j$ or its reduced extended form $R_i^j$ let (depending on the context) $\overline{S}_i^v$ denote the set of virtual vertices of $H_i^j$ or $R_i^j$ except for the vertices labeled $v$. Let $\overline{S}_{\{1,2,\ldots,i\}}^v$ denote the set of virtual vertices except for the vertices labeled $v$ of $H_{\{1,2,\ldots,i\}}^j$ or $R_{\{1,2,\ldots,i\}}^j$, depending on the context. Let $\pi_{\{1,2,\ldots,i\}}$, $i \leq p$, denote a witness to a level planar embedding of $H_{\{1,2,\ldots,i\}}^j$ or $R_{\{1,2,\ldots,i\}}^j$, respectively. In the example of Fig. 4.37 we have $\overline{S}_4^v = \{w_1^4, w_2^4, \ldots, w_{q_4}^4\}$, and $\overline{S}_{\{1,2,3\}}^v = \{w_1^1, w_2^1, \ldots, w_{q_1}^1\} \cup \{w_1^2, w_2^2, \ldots, w_{q_2}^2\} \cup \{w_1^3, w_2^3, \ldots, w_{q_3}^3\}$. The witness of the shown level planar embedding is $\pi_{\{1,2,3,4\}} = [w_1^1, w_2^1, \ldots, w_{q_1}^1, w_1^2, w_2^2, \ldots, w_{q_2}^2, w_1^3, w_2^3, \ldots, w_{q_3}^3, v, w_1^4, w_2^4, \ldots, w_{q_4}^4]$.

In order to prove that the virtual vertices of the smaller form $H_i^j$ (that is merged into the larger form $H_{\{1,2,\ldots,i-1\}}^j$) appear consecutively in any level planar embedding of the new form $H_{\{1,2,\ldots,i\}}^j$, we need to show that $\overline{S}_i^v$ and the vertex $v$ are consecutive. The concept of the proof is to assume the opposite and then to find a path in $H_i^j$ and a path in $H_{\{1,2,\ldots,i-1\}}^j$ that cross each other in $H_{\{1,2,\ldots,i\}}^j$.

**Lemma 4.17.** *Let $G = (V, E)$ be a level planar graph with $k > 1$ levels, and let $v \in V^{j+1}$ be an arbitrary vertex, where $j < k$. Let $H_1^j, H_2^j, \ldots, H_p^j$, $p \geq 2$, be extended forms such*

*(i)* $S_i^v \neq \emptyset$ *for all* $i \in \{1, 2, \ldots, p\}$, *and*

*(ii)* $\mathrm{LL}(H_1^j) \leq \mathrm{LL}(H_2^j) \leq \mathrm{LL}(H_3^j) \leq \cdots \leq \mathrm{LL}(H_p^j)$.

*Then the following holds. If $\pi_{\{1,2,\ldots,i\}}$, $i \leq p$, is a witness to a level planar embedding of $H^j_{\{1,2,\ldots,i\}}$, then the vertices of $\overline{S}_i^v$ form a consecutive sequence in $\pi_{\{1,2,\ldots,i\}}$ and the vertex $v$ appears next to $\overline{S}_i^v$ in $\pi_{\{1,2,\ldots,i\}}$.*

*Proof.* Throughout the proof, we will consider $H_1^j, H_2^j, \ldots, H_p^j$ as well as $H_{\{1,2,\ldots,i-1\}}$ as subgraphs of $H_{\{1,2,\ldots,i\}}$. Let $\pi_{\{1,2,\ldots,i\}}$, $2 \leq i \leq p$, be a witness of a level planar embedding $\mathcal{E}_{\{1,2,\ldots,i\}}$ of $H_{\{1,2,\ldots,i\}}$. The lemma holds trivially, if $\overline{S}^v_{\{1,2,\ldots,i-1\}} = \emptyset$ or $\overline{S}_i^v = \emptyset$. Thus assume, there exists an $x \in \overline{S}^v_{\{1,2,\ldots,i-1\}} \cup \{v\}$, such that $x$ appears between two vertices $y_1$ and $y_2$ of $\overline{S}_i^v$ in $\pi_{\{1,2,\ldots,i\}}$. By definition, $H_i^j$ is connected. Furthermore, $v$ is not a cut vertex in $H_i^j$ (otherwise $H_i^j$ would be $v$-connected). Hence, there exists a path $P$ in $H_i^j$ connecting $y_1$ and $y_2$ not containing $v$. Since $\mathrm{LL}(H^j_{\{1,2,\ldots,i-1\}}) \leq \mathrm{LL}(H_i^j)$ and $H^j_{\{1,2,\ldots,i-1\}}$ is connected, there exist a vertex $z \in H^j_{\{1,2,\ldots,i-1\}}$ such that $lev(z) \leq lev(w)$ for all $w \in P$ and a path $\tilde{P}$ in $H^j_{\{1,2,\ldots,i-1\}}$ connecting $x$ and $z$ (see Fig. 4.38). By construction the paths $P$ and $\tilde{P}$ are disjoint (since $H_{\{1,2,\ldots,i-1\}}$ and $H_i^j$ are identified only in $v$), but cross each other,. Thus, $\pi_{\{1,2,\ldots,i\}}$ cannot be a witness of a level planar embedding of $H^j_{\{1,2,\ldots,i\}}$, which is a contradiction.

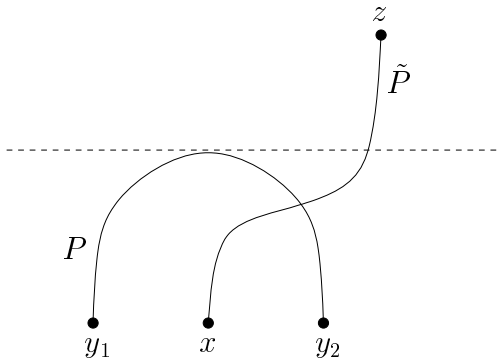



Figure 4.38: Illustration to the proof of Lemma 4.17. Path $P$ connecting $y_1$ and $y_2$ in $H_i^j$ and path $\tilde{P}$ connecting $x$ and $z$ in $H^j_{\{1,2,\ldots,i-1\}}$ cross each other in a level embedding of $H^j_{\{1,2,\ldots,i\}}$ if $x \in \overline{S}^v_{\{1,2,\ldots,i-1\}}$ appears between $y_1, y_2 \in \overline{S}_i^v$.

Assume now that there exists an $x \in \overline{S}^v_{\{1,2,\ldots,i-1\}}$, such that $x$ appears between the vertices of $\overline{S}_i^v$ and $v$ in $\pi_{\{1,2,\ldots,i\}}$, and such that there is a vertex $y \in \overline{S}_i^v$ that appears next to $x$. Since

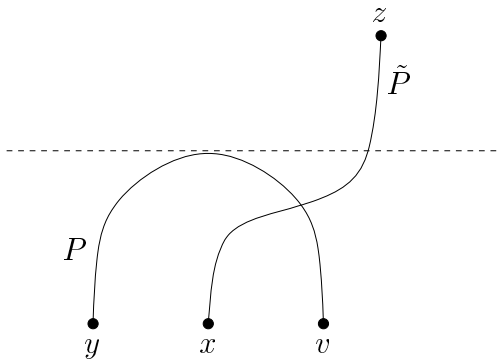



Figure 4.39: Illustration to the proof of Lemma 4.17. Path $P$ connecting $y$ and $v$ in $H_i^j$ and path $\tilde{P}$ connecting $x$ and $z$ in $H_l^j$, $l \in \{1, 2, \ldots, i-1\}$, cross each other in a level embedding of $H^j_{\{1,2,\ldots,i\}}$ if $x \in \overline{S}^v_{\{1,2,\ldots,i-1\}}$ appears between $y \in \overline{S}_i^v$ and $v$.

$H_i^j$ is connected, there exists a path $P$ in $H_i^j$ connecting $y$ and $v$. By construction $x \in \overline{S}_l^v$ for some $l \in \{1, 2, \ldots, i-1\}$. (Reconsider that $v$ is a cut vertex in $H^j_{\{1,2,\ldots,i-1\}}$ and the cut components are exactly $H_1^j, H_2^j, \ldots, H_{i-1}^j$.) But $\mathrm{LL}(H_l^j) \leq \mathrm{LL}(H_i^j)$ implies that there exist $z \in H_l^j$ such that $lev(z) \leq lev(w)$ for all $w \in P$. Since $v$ is not a cut vertex in $H_l^j$, there exists a path $\tilde{P}$ in $H_l^j$ connecting $x$ and $z$ that does not contain $v$ (see Fig. 4.39). Again, since $H^j_{\{1,2,\ldots,i-1\}}$ and $H_i^j$ have only $v$ in common, the paths $P$ and $\tilde{P}$ are disjoint but cross each other, which is a contradiction. □

**Lemma 4.18.** *Let $G = (V, E)$ be a level planar graph with $k > 1$ levels, and let $v \in V^{j+1}$ be an arbitrary vertex, where $j < k$. Let $R_1^j, R_2^j, \ldots, R_p^j$, $p \geq 2$, be reduced extended forms such*

*(i) $S_i^v \neq \emptyset$ for all $i \in \{1, 2, \ldots, p\}$, and*

*(ii) $\mathrm{LL}(R_1^j) \leq \mathrm{LL}(R_2^j) \leq \mathrm{LL}(R_3^j) \leq \cdots \leq \mathrm{LL}(R_p^j)$.*

*Then the following holds. If $\pi_{\{1,2,\ldots,i\}}$, $i \leq p$, is a witness to a level planar embedding of $R^j_{\{1,2,\ldots,i\}}$, then the vertices of $\overline{S}_i^v$ form a consecutive sequence in $\pi_{\{1,2,\ldots,i\}}$ and the vertex $v$ appears next to $\overline{S}_i^v$ in $\pi_{\{1,2,\ldots,i\}}$.*

*Proof.* Analogously to the proof of Lemma 4.17. □

We need to mention again that the results of Lemma 4.17 and Lemma 4.18 do not hold if components are not merged according to the specified order. Consider as another example, the forms shown in Fig. 4.29 on page 73. Assume that the extended forms $H_1^j$ and $H_4^j$ are merged first. If $H_2^j$ and $H^j_{\{1,4\}}$ are $v$-merged, the virtual vertices of $\overline{S}_2^v$ and $v$ are not adjacent in any level planar embedding of $H^j_{\{1,4,2\}}$. (Unlike our first example of Fig. 4.37,

where the virtual vertices $\overline{S}_2^v$ are separated from $v$ only in some level planar embeddings of $R^j_{\{1,4,3,2\}}$.)

Using Lemma 4.18, we are able to show Lemma 4.19 which proves the correctness of the merge operations during the second reduction phase. The lemma states that every $PQ$-tree constructed by $v$-merging all reduced extended forms with a virtual vertex labeled $v$ according to their size represents exactly all level planar embeddings of the new $v$-connected form.

**Lemma 4.19.** *Let $G$, $R^j_1, R^j_2, \ldots, R^j_p$, and the vertex $v$ be defined as in Lemma 4.18 except that $G$ is not necessary level planar. Let the reduced forms $R^j_1, R^j_2, \ldots, R^j_p$ be level planar and suppose that the $PQ$-trees $T(R^j_1), T(R^j_2), \ldots, T(R^j_p)$ represent all level planar embeddings of $R^j_1, R^j_2, \ldots, R^j_p$.*

*Let $T(R^j_{\{1,2,\ldots,p\}})$ be the $PQ$-tree constructed as described in the second merge phase of CHECK-LEVEL. Then $\mathrm{PERM}(T(R^j_{\{1,2,\ldots,p\}}))$ is exactly the set of permutations of level-$(j+1)$ vertices that appear in level planar embeddings of $R^j_{\{1,2,\ldots,p\}}$.*

*Proof.* We first show that if $\pi \in \mathrm{PERM}(T(R^j_{\{1,2,\ldots,p\}}))$ is a permutation represented by the $PQ$-tree $T(R^j_{\{1,2,\ldots,p\}})$, then $\pi$ is a witness to some level planar embedding of $R^j_{\{1,2,\ldots,p\}}$. This can be shown following an idea of Heath and Pemmaraju (1996). The authors have shown in one of their lemmas the special case of two components $R^j_{\{1,2\}} = R^j_1 \cup_v R^j_2$. We adapt that proof to the more general case and consider $v$-singular forms.

For all $2 \leq i \leq p$ let $T(R^j_{\{1,2,\ldots,i\}})$ be the $PQ$-tree constructed in the $i$-th iteration of the for-loop in the second merge phase. Now, let $2 \leq i \leq p$ be fixed and assume (by induction) that $T(R^j_{\{1,2,\ldots,i-1\}})$ represents (all) level planar embeddings of $R^j_{\{1,2,\ldots,i-1\}}$. We show that if $\pi_{\{1,2,\ldots,i\}} \in \mathrm{PERM}(T(R^j_{\{1,2,\ldots,i\}}))$, then $\pi_{\{1,2,\ldots,i\}}$ is a witness to a level planar embedding of $R^j_{\{1,2,\ldots,i\}}$.

Let $v_{\{1,2,\ldots,i-1\}}$ be the virtual vertex labeled $v$ in $R^j_{\{1,2,\ldots,i-1\}}$, and let $v_i$ be the virtual vertex labeled $v$ in $R^j_i$. Two cases may occur, depending on whether $R^j_i$ is $v$-singular ($\overline{S}^v_i = \emptyset$) or not ($\overline{S}^v_i \neq \emptyset$). We start with the nonsingular case.

1. $\overline{S}^v_i \neq \emptyset$.

   The $PQ$-tree $T(R^j_{\{1,2,\ldots,i\}})$ has been constructed by reducing the leaves corresponding to $v_{\{1,2,\ldots,i-1\}}$ and $v_i$ in the $PQ$-tree $\tilde{T}(R^j_{\{1,2,\ldots,i\}})$, and replacing them by the single representative $v_{\{1,2,\ldots,i\}}$ afterwards, where $\tilde{T}(R^j_{\{1,2,\ldots,i\}})$ was the result of the INSERT operation performed on $T(R^j_{\{1,2,\ldots,i-1\}})$ and $T(R^j_i)$. Thus, there exists a $\pi'_{\{1,2,\ldots,i\}} \in \mathrm{PERM}(\tilde{T}(R^j_{\{1,2,\ldots,i\}}))$ such that $\pi_{\{1,2,\ldots,i\}}$ arises from $\pi'_{\{1,2,\ldots,i\}}$ by identifying the two elements $v_{\{1,2,\ldots,i-1\}}$ and $v_i$ that appear next to each other in $\pi'_{\{1,2,\ldots,i\}}$. Since $\overline{S}^v_i \neq \emptyset$, the function INSERT has called the function MERGE. The function MERGE has

added the root of $T(R_i^j)$ as a sibling to a node $X'$ in $T(R^j_{\{1,2,\ldots,i-1\}})$. The node $X'$ and its parent $X$ (in case $X'$ was not the root of $T(R^j_{\{1,2,\ldots,i-1\}})$) have been subject to the merge operation in $T(R^j_{\{1,2,\ldots,i-1\}})$. As a result of the merge operation, the leaves of frontier$(T(R_i^j))$ occur consecutively in $\pi'_{\{1,2,\ldots,i\}}$, as do the leaves of frontier$(X')$. Without loss of generality, we assume that the leaves of frontier$(X')$ are immediately followed by the leaves of frontier$(T(R_i^j))$ in $\pi'_{\{1,2,\ldots,i\}}$. Hence, the permutation $\pi'_{\{1,2,\ldots,i\}}$ can be written as $\pi^a_{\{1,2,\ldots,i\}}\pi^b_{\{1,2,\ldots,i\}}\pi^c_{\{1,2,\ldots,i\}}$ with

$$\pi^b_{\{1,2,\ldots,i\}} \in \text{PERM}(T(R_i^j))$$

and

$$\pi^a_{\{1,2,\ldots,i\}}\pi^c_{\{1,2,\ldots,i\}} \in \text{PERM}(T(R^j_{\{1,2,\ldots,i-1\}}))$$

with $v_{\{1,2,\ldots,i-1\}}$ in $\pi^a_{\{1,2,\ldots,i\}}$ and $v_i$ in $\pi^b_{\{1,2,\ldots,i\}}$ appearing consecutively in $\pi'_{\{1,2,\ldots,i\}}$. By assumption, $\pi^a_{\{1,2,\ldots,i\}}\pi^c_{\{1,2,\ldots,i\}}$ is a witness to a level planar embedding $\mathcal{E}^j_{\{1,2,\ldots,i-1\}}$ of $R^j_{\{1,2,\ldots,i-1\}}$ and $\pi^b_{\{1,2,\ldots,i\}}$ is a witness to a level planar embedding $\mathcal{E}^j_i$ of $R^j_i$. There are two cases that apply depending on whether $\pi^c_{\{1,2,\ldots,i\}}$ is empty or not

(a) $\pi^c_{\{1,2,\ldots,i\}} = \emptyset$. A level planar embedding of $R^j_{\{1,2,\ldots,i\}}$ can be constructed by simply placing $R^j_i$ next to $R^j_{\{1,2,\ldots,i-1\}}$ and then identifying the vertices $v_{\{1,2,\ldots,i-1\}}$ and $v_i$ to a vertex $v_{\{1,2,\ldots,i\}}$. Hence, $\pi_{\{1,2,\ldots,i\}} \in \text{PERM}(T(R^j_{\{1,2,\ldots,i\}}))$ is a witness to a level planar embedding of $R^j_{\{1,2,\ldots,i\}}$.

(b) $\pi^c_{\{1,2,\ldots,i\}} \neq \emptyset$. Let $w$ be the first vertex in $\pi^c_{\{1,2,\ldots,i\}}$ and let $Y$ be the smallest common ancestor of $w$ and $v_{\{1,2,\ldots,i-1\}}$. Clearly, $w \notin$ frontier$(X')$. Thus, $Y$ is an ancestor (not necessarily proper) of $X$ ($X$ being the parent of $X'$ before the merge operation). By construction of the merge operation and by Observations 4.6, 4.7, and 4.8 we have

$$\text{ML}(\{v_{\{1,2,\ldots,i-1\}}, w\}) < \text{LL}(T(R_i^j)) \ .$$

Hence, the level planar embedding $\mathcal{E}^j_i$ of $R^j_i$ can be nested inside the level planar embedding $\mathcal{E}^j_{\{1,2,\ldots,i-1\}}$ of $R^j_{\{1,2,\ldots,i-1\}}$. Merging the virtual vertices $v_{\{1,2,\ldots,i-1\}}$ and $v_i$ to a vertex $v_{\{1,2,\ldots,i\}}$, a level planar embedding $\mathcal{E}^j_{\{1,2,\ldots,i\}}$ of $R^j_{\{1,2,\ldots,i\}}$ is constructed in which the virtual vertices appear according to $\pi_{\{1,2,\ldots,i\}}$. Hence, $\pi_{\{1,2,\ldots,i\}} \in \text{PERM}(T(R^j_{\{1,2,\ldots,i\}}))$ is a witness to a level planar embedding of $R^j_{\{1,2,\ldots,i\}}$.

2. $\overline{S}^v_i = \emptyset$.

There are two possible cases.

(a) The function MERGE was called by INSERT. This case is proven analogously to the case $\overline{S}_i^v \neq \emptyset$.

(b) The function INSERT did not call the function MERGE. Thus either one of the following inequalities holds:

$$\mathrm{PML}(v_{\{1,2,\ldots,i-1\}}) \;<\; \mathrm{LL}(T(R_i^j)) \;,$$

or

$$\mathrm{QML}(v_{\{1,2,\ldots,i-1\}}) \;<\; \mathrm{LL}(T(R_i^j)) \;.$$

It follows from Lemma 4.14 and by construction of the function UPDATE that there exists an interior face or a cavity in some embedding of $R^j_{\{1,2,\ldots,i-1\}}$ that is large enough to level planar embed $R_i^j$ into it. Hence, $\pi_{\{1,2,\ldots,i\}}$ is a witness to a level planar embedding of $R^j_{\{1,2,\ldots,i\}}$.

Thus, one direction of the equivalence stated in the lemma is proved.

To prove the reverse direction, we show that the $PQ$-tree $T(R^j_{\{1,2,\ldots,i\}})$ represents all level planar embeddings of $R^j_{\{1,2,\ldots,i\}}$. We show (by induction) that for any witness $\pi_{\{1,2,\ldots,i\}}$, $2 \leq i \leq p$, of a level planar embedding of $\mathcal{E}^j_{\{1,2,\ldots,i\}}$ of $R^j_{\{1,2,\ldots,i\}}$ the following holds:

$$\pi_{\{1,2,\ldots,i\}} \in \mathrm{PERM}(T(R^j_{\{1,2,\ldots,i\}})) \;.$$

1. $\overline{S}_i^v \neq \emptyset$.

   The level-$(j+1)$ vertices in $R^j_{\{1,2,\ldots,i\}}$ can be partitioned into three sets: $\overline{S}^v_{\{1,2,\ldots,i-1\}}$, the set of all level-$j+1$ vertices of $R^j_{\{1,2,\ldots,i-1\}}$ except the vertex $v$, $\overline{S}_i^v$, the set of all level-$(j+1)$ vertices of $R_i^j$ except the vertex $v$, and the level-$(j+1)$ vertex $v$. According to Lemma 4.18, the vertices of $\overline{S}_i^v$ appear consecutively in $\pi_{\{1,2,\ldots,i\}}$, either immediately followed by or immediately preceded by $v$. We may assume that the latter case applies. Let $\tilde{R}^j_{\{1,2,\ldots,i\}}$ be the graph that consists of $R^j_{\{1,2,\ldots,i-1\}}$ and $R_i^j$, where the level-$(j+1)$ vertices labeled $v$ of the two components are not identified and kept separate. Let $S^v_{\{1,2,\ldots,i-1\}} := \{v_{\{1,2,\ldots,i-1\}}\}$ where $v_{\{1,2,\ldots,i-1\}}$ is the single representative of $v$ in $R^j_{\{1,2,\ldots,i-1\}}$ and $S_i^v := \{v_i\}$ where $v_i$ is the single representative of $v$ in $R_i^j$. "Splitting" in $\pi_{\{1,2,\ldots,i\}}$ the vertex $v$ into $v_{\{1,2,\ldots,i-1\}}$ and $v_i$, we get a permutation $\tilde{\pi}_{\{1,2,\ldots,i\}}$ that witnesses a level planar embedding $\tilde{\mathcal{E}}^j_{\{1,2,\ldots,i\}}$ of $\tilde{R}^j_{\{1,2,\ldots,i\}}$.

   The witness $\tilde{\pi}_{\{1,2,\ldots,i\}}$ can be written as $\tilde{\pi}^a_{\{1,2,\ldots,i\}}\tilde{\pi}^b_{\{1,2,\ldots,i\}}\tilde{\pi}^c_{\{1,2,\ldots,i\}}$ such that $\tilde{\pi}^a_{\{1,2,\ldots,i\}}\tilde{\pi}^c_{\{1,2,\ldots,i\}}$ is a witness of a level planar embedding $\mathcal{E}^j_{\{1,2,\ldots,i-1\}}$ of $R^j_{\{1,2,\ldots,i-1\}}$ and $\tilde{\pi}^b_{\{1,2,\ldots,i\}}$ is a witness of a level planar embedding $\mathcal{E}_i^j$ of $R_i^j$, and such that (without loss of generality) $\tilde{\pi}^a_{\{1,2,\ldots,i\}}$ ends with $S^v_{\{1,2,\ldots,i-1\}}$, and $\tilde{\pi}^b_{\{1,2,\ldots,i\}}$ starts with $S_i^v$.

Since $T(R^j_{\{1,2,\ldots,i-1\}})$ and $T(R^j_i)$ correspond to $R^j_{\{1,2,\ldots,i-1\}}$ and $R^j_i$, respectively, it follows by induction that $\tilde{\pi}^a_{\{1,2,\ldots,i\}}\tilde{\pi}^c_{\{1,2,\ldots,i\}} \in \mathrm{PERM}(T(R_{\{1,2,\ldots,i-1\}}))$, and $\tilde{\pi}^b_{\{1,2,\ldots,i\}} \in \mathrm{PERM}(T(R^j_i))$. We show that $\tilde{\pi}_{\{1,2,\ldots,i\}} \in \mathrm{PERM}(\tilde{T}(R^j_{\{1,2,\ldots,i\}}))$, where $\tilde{T}(R^j_{\{1,2,\ldots,i\}})$ is the $PQ$-tree that is constructed by the function INSERT without reducing the $PQ$-tree with respect to $S^v_{\{1,2,\ldots,i-1\}} \cup S^v_i$. There are two cases depending on whether $\tilde{\pi}^c_{\{1,2,\ldots,i\}}$ is empty or not.

(a) $\tilde{\pi}^c_{\{1,2,\ldots,i\}} \neq \emptyset$. Suppose that the first vertex in $\tilde{\pi}^c_{\{1,2,\ldots,i\}}$ is $w$. Since according to Lemma 4.18 the vertices of $\overline{S}^v_i$ occur consecutively preceded by $v$ and since the embedding $\tilde{\mathcal{E}}^j_{\{1,2,\ldots,i\}}$ of $\tilde{R}^j_{\{1,2,\ldots,i\}}$ is level planar the following must hold:

$$\mathrm{ML}(\{S^v_{\{1,2,\ldots,i-1\}}, w\}) < \mathrm{LL}(T(R^j_i)) \ .$$

Let $Y$ be the node in $\tilde{T}(R^j_{\{1,2,\ldots,i-1\}})$ that is the least common ancestor of $S^v_{\{1,2,\ldots,i-1\}}$ and $w$. Then there exists a child $Y'$ of $Y$ such that $S^v_{\{1,2,\ldots,i-1\}} \subseteq \mathrm{frontier}(Y')$. Since $\mathrm{ML}(\mathrm{frontier}(Y') \cup \{w\}) \leq \mathrm{ML}(S^v_{\{1,2,\ldots,i-1\}} \cup \{w\})$, we have according to Observation 4.9 that $\tilde{\pi}_{\{1,2,\ldots,i\}} \in \mathrm{PERM}(\tilde{T}(R^j_{\{1,2,\ldots,i\}}))$.

(b) $\tilde{\pi}^c_{\{1,2,\ldots,i\}} = \emptyset$. According to Observation 4.10 $\tilde{\pi}_{\{1,2,\ldots,i\}} \in \mathrm{PERM}(\tilde{T}(R^j_{\{1,2,\ldots,i\}}))$ holds.

It follows that $\tilde{\pi}_{\{1,2,\ldots,i\}} \in \mathrm{PERM}(\tilde{T}(R^j_{\{1,2,\ldots,i\}}))$ with $S^v_{\{1,2,\ldots,i-1\}} \cup S^v_i$ appearing consecutively in $\tilde{\pi}_{\{1,2,\ldots,i\}}$. This implies that the $PQ$-tree $\tilde{T}(R^j_{\{1,2,\ldots,i\}})$ can be reduced with respect to $S^v_{\{1,2,\ldots,i-1\}} \cup S^v_i$ and therefore $\pi_{\{1,2,\ldots,i\}}$ is contained in $\mathrm{PERM}(T(R^j_{\{1,2,\ldots,i\}}))$.

2. $\overline{S}^v_i = \emptyset$.

There are two cases that may appear.

(a) The set of incoming edges of $v$ in $R^j_i$ (corresponding to $S^v_i$) separates within the clockwise order of incoming edges of $v$ in $\mathcal{E}^j_{\{1,2,\ldots,i\}}$ the set of incoming edges corresponding to $S^v_{\{1,2,\ldots,i-1\}}$ into two nonempty subsets.

The level-$(j+1)$ vertices in $R^j_{\{1,2,\ldots,i\}}$ can be partitioned into two sets: $\overline{S}^v_{\{1,2,\ldots,i-1\}}$ the set of all level-$(j+1)$ vertices of $R^j_{\{1,2,\ldots,i-1\}}$ except the vertex $v$, and the level-$(j+1)$ vertex $v$. Let $\tilde{R}^j_{\{1,2,\ldots,i\}}$ be the form that contains the components $R^j_{\{1,2,\ldots,i-1\}}$ and $R^j_i$ where the incoming edges of $v$ corresponding to $R^j_{\{1,2,\ldots,i-1\}}$ are not identified to $v$ but kept separate.

Obviously, $\tilde{R}^j_{\{1,2,\ldots,i\}}$ is level planar. Let $\tilde{\mathcal{E}}^j_{\{1,2,\ldots,i\}}$ be the level planar embedding of $\tilde{R}^j_{\{1,2,\ldots,i\}}$ that is induced by $\mathcal{E}^j_{\{1,2,\ldots,i\}}$. Let $S^{left}_{\{1,2,\ldots,i-1\}}$ be the set of virtual vertices corresponding to the incoming edges of $v$ in $R^j_{\{1,2,\ldots,i-1\}}$ on the left side of $R^j_i$ in $\tilde{\mathcal{E}}^j_{\{1,2,\ldots,i\}}$. Let $S^{right}_{\{1,2,\ldots,i-1\}}$ be the set of virtual vertices corresponding to the incoming edges of $v$ in $R^j_{\{1,2,\ldots,i-1\}}$ on the right side of $R^j_i$ in $\tilde{\mathcal{E}}^j_{\{1,2,\ldots,i\}}$. See

Fig. 4.40 for an illustration. Let $S_i^v := \{v_i\}$, where $v_i$ is the single representative of $v$ in $R_i^j$. Replacing in $\pi_{\{1,2,\ldots,i\}}$ the vertex $v$ by the set of vertices $S^{left}_{\{1,2,\ldots,i-1\}} \cup S_i^v \cup S^{right}_{\{1,2,\ldots,i-1\}}$ we get a permutation $\tilde{\pi}_{\{1,2,\ldots,i\}}$ that witnesses a level planar embedding $\tilde{\mathcal{E}}^j_{\{1,2,\ldots,i\}}$ of $\tilde{R}^j_{\{1,2,\ldots,i\}}$.

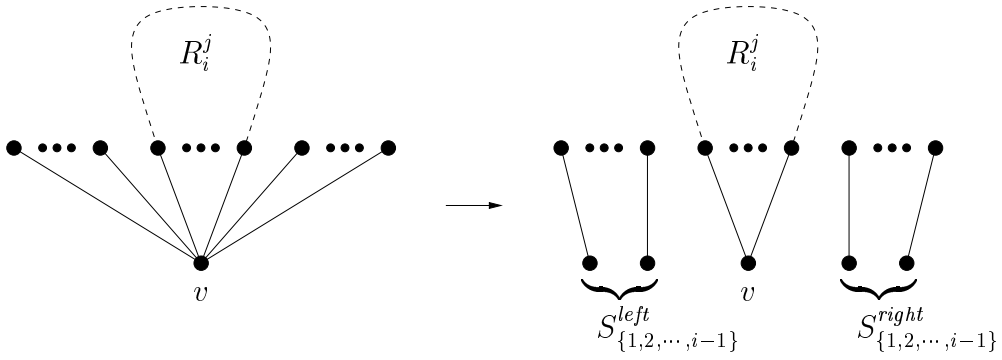



Figure 4.40: Illustration of the proof of Lemma 4.19. $R_i^j$ is a singular form. The incoming edges are partitioned into sets $S^{left}_{\{1,2,\ldots,i-1\}}$, $S^{right}_{\{1,2,\ldots,i-1\}}$ and the edges belonging to $R_i^j$.

The witness $\tilde{\pi}_{\{1,2,\ldots,i\}}$ can be written as $\tilde{\pi}^a_{\{1,2,\ldots,i\}}\tilde{\pi}^b_{\{1,2,\ldots,i\}}\tilde{\pi}^c_{\{1,2,\ldots,i\}}$ where $\tilde{\pi}^a_{\{1,2,\ldots,i\}}$ ends with $S^{left}_{\{1,2,\ldots,i-1\}}$, $\tilde{\pi}^b_{\{1,2,\ldots,i\}} = \{v_i\}$ and, $\tilde{\pi}^c_{\{1,2,\ldots,i\}}$ starts with $S^{right}_{\{1,2,\ldots,i-1\}}$. Merging the set of vertices $S^{left}_{\{1,2,\ldots,i-1\}}$ and $S^{right}_{\{1,2,\ldots,i-1\}}$ the form $R^j_{\{1,2,\ldots,i-1\}}$ is constructed. By induction, a permutation $\pi^a_{\{1,2,\ldots,i\}}\pi^c_{\{1,2,\ldots,i\}} \in \mathrm{PERM}(T(R^j_{\{1,2,\ldots,i-1\}}))$ is obtained by replacing in $\tilde{\pi}^a_{\{1,2,\ldots,i\}}\tilde{\pi}^c_{\{1,2,\ldots,i\}}$ the sets $S^{left}_{\{1,2,\ldots,i-1\}}$ and $S^{right}_{\{1,2,\ldots,i-1\}}$ by the single representative $v_{\{1,2,\ldots,i-1\}}$. Since after replacing $v_{\{1,2,\ldots,i-1\}}$ by $v_{\{1,2,\ldots,i\}}$ we have $\pi_{\{1,2,\ldots,i\}} = \pi^a_{\{1,2,\ldots,i\}}\pi^c_{\{1,2,\ldots,i\}}$ this implies that we need to show $T(R^j_{\{1,2,\ldots,i\}}) = T(R^j_{\{1,2,\ldots,i-1\}})$.

Let $v'_{\{1,2,\ldots,i-1\}}$ be the rightmost virtual vertex of $S^{left}_{\{1,2,\ldots,i-1\}}$, and let $v''_{\{1,2,\ldots,i-1\}}$ be the leftmost virtual vertex of $S^{right}_{\{1,2,\ldots,i-1\}}$. Since the embedding of $\tilde{R}^j_{\{1,2,\ldots,i\}}$ is level planar, the following inequality must hold.

$$\mathrm{ML}(v'_{\{1,2,\ldots,i-1\}}, v''_{\{1,2,\ldots,i-1\}}) < \mathrm{LL}(T(R_i^j)) \ .$$

Two possible subcases apply.

i. $R_i^j$ is embedded into an interior face of $R^j_{\{1,2,\ldots,i-1\}}$. Since $v_{\{1,2,\ldots,i-1\}}$ is a cut vertex in $R^j_{\{1,2,\ldots,i-1\}}$, with the cut components being $R_1^j, R_2^j, \ldots, R_{i-1}^j$, the form $R_i^j$ is embedded into an interior face of a form $R_l^j$, $l \in \{1, 2, \ldots, i-1\}$. Thus the virtual vertices $v'_{\{1,2,\ldots,i-1\}}$, and $v''_{\{1,2,\ldots,i-1\}}$ correspond to edges of $R_l^j$. Let $X$ be the smallest common ancestor of $v'_{\{1,2,\ldots,i-1\}}$ and $v''_{\{1,2,\ldots,i-1\}}$ in

$T(R_l^j)$. If $X$ is a $P$-node we have by proposition 4.8 for the PML-value of the single representative $v_l$ of $R_l^j$

$$\mathrm{PML}(v_l) \leq \mathrm{ML}(X) \leq \mathrm{ML}(v'_{\{1,2,\ldots,i-1\}}, v''_{\{1,2,\ldots,i-1\}}) \ .$$

If $X$ is a $Q$-node, and $X'$ and $X''$ are the children of $X$ with $v'_{\{1,2,\ldots,i-1\}}$ and $v''_{\{1,2,\ldots,i-1\}}$ in their frontier, respectively, we have by proposition 4.8 for the QML-value of the single representative $v_l$ of $R_l^j$

$$\mathrm{QML}(v_l) \leq \mathrm{ML}(X', X'') \leq \mathrm{ML}(v'_{\{1,2,\ldots,i-1\}}, v''_{\{1,2,\ldots,i-1\}}) \ .$$

By construction of the function UPDATE it follows that

$$\min\{\mathrm{QML}(v_{\{1,2,\ldots,i-1\}}), \mathrm{PML}(v_{\{1,2,\ldots,i-1\}})\} \leq \mathrm{PML}(v_l)$$

or

$$\min\{\mathrm{QML}(v_{\{1,2,\ldots,i-1\}}), \mathrm{PML}(v_{\{1,2,\ldots,i-1\}})\} \leq \mathrm{QML}(v_l) \ ,$$

and thus INSERT "does nothing".

ii. $R_i^j$ is embedded into a cavity of $R^j_{\{1,2,\ldots,i-1\}}$. Thus $i$ must be at least 3, otherwise no $v$-cavity exists in $R^j_{\{1,2,\ldots,i-1\}}$. By assumption, we have

$$\mathrm{LL}(R_{i-2}^j) \leq \mathrm{LL}(R_{i-1}^j) \leq \mathrm{LL}(R_i^j) \ .$$

Let $X$ be the root of the pertinent subtree when $v$-merging $R_{i-1}^j$ into $R^j_{\{1,2,\ldots,i-2\}}$. If $X$ is a $P$-node, we have by construction $\mathrm{ML}(X) < \mathrm{LL}(T(R_{i-1}^j))$. If $X$ is a $Q$-node with pertinent adjacent children $Y$ and $Z$, we have by construction that $\mathrm{ML}(Y,Z) < \mathrm{LL}(T(R_{i-1}^j))$. Let $h$ denote $\mathrm{ML}(X)$ or $\mathrm{ML}(Y,Z)$, respectively. Then we have by construction of the function UPDATE that

$$\min\{\mathrm{QML}(v_{\{1,2,\ldots,i-1\}}), \mathrm{PML}(v_{\{1,2,\ldots,i-1\}})\} \leq h < \mathrm{LL}(R_{i-1}^j) \leq \mathrm{LL}(R_i^j) \ .$$

Thus, again, INSERT "does nothing", and the tree $T(R^j_{\{1,2,\ldots,i-1\}})$ is left unchanged.

(b) Both sets of incoming edges of $v$ corresponding to $S^v_{\{1,2,\ldots,i-1\}}$ and $S_i^v$ form a consecutive sequence within the clockwise order of incoming edges of $v$ in $\mathcal{E}^j_{\{1,2,\ldots,i\}}$. The result follows analogously to the proof of the case $\overline{S}_i^v \neq \emptyset$, with $\tilde{\pi}^b_{\{1,2,\ldots,i\}} = S_i^v$.

□

If the $PQ$-trees of several reduced extended forms $R_i^j$, $i = 1, 2, \ldots, p$, have been $v$-merged, the new $PQ$-tree $T$ may contain several leaves labeled by the same $w \neq v$. The following lemma shows that the reduction of these leaves constructs a $PQ$-tree that represents all level planar embeddings of the subgraph induced by the vertices $\bigcup_{i=1}^p V(R_i^j)$.

**Lemma 4.20.** *Let $G$, $R_1^j, R_2^j, \ldots, R_p^j$, and the vertex $v$ be defined as in Lemma 4.19. Let the reduced forms $R_1^j, R_2^j, \ldots, R_p^j$, and the $v$-merged form $R^j_{\{1,2,\ldots,p\}}$ be level planar. Let $T(R^j_{\{1,2,\ldots,p\}})$ be the PQ-tree constructed as described in the second merge phase of CHECK-LEVEL, representing all level planar embeddings of $R^j_{\{1,2,\ldots,p\}}$.*

*Let $F$ be the component constructed by identifying for all $w \in V^{j+1}$ all virtual vertices with the label $w$ to a single vertex $w$. Let $T(F)$ be the PQ-tree constructed as described in the second merge phase of CHECK-LEVEL, by reducing in $T(R^j_{\{1,2,\ldots,p\}})$ all leaves with a common label $w$. Then* $\mathrm{PERM}(T(F))$ *is exactly the set of permutations of level-$(j+1)$ vertices that appear in level planar embeddings of $F$.*

*Proof.* We first show that $T(F)$ represents level planar embeddings of $F$. According to Lemma 4.19, $T(R^j_{\{1,2,\ldots,i\}})$ represents all level planar embeddings of $R^j_{\{1,2,\ldots,i\}}$. Applying the function REDUCE with respect to the leaves labeled $w \in V^{j+1}$, $w \neq v$, creates either a $PQ$-tree $\tilde{T}(R^j_{\{1,2,\ldots,i\}})$ such that for every $w$ the vertices of $\bigcup_{i\in\{1,2,\ldots,p\}} S_i^w$ occupy consecutive positions or an empty $PQ$-tree. If $\tilde{T}(R^j_{\{1,2,\ldots,i\}})$ is not the empty $PQ$-tree, we have that $\mathrm{PERM}(\tilde{T}(R^j_{\{1,2,\ldots,i\}})) \subseteq \mathrm{PERM}(T(R^j_{\{1,2,\ldots,i\}}))$ represents all level planar embeddings of $R^j_{\{1,2,\ldots,i\}}$ such that all leaves with the same label form a consecutive subsequence. Identifying all leaves with the same label $w$ to one leaf, a $PQ$-tree $T(F)$ is constructed that represents level planar embeddings of $F$.

We now get to the "only if" part, showing that for any level planar embedding $\mathcal{E}_F^j$, the witness $\pi$ of $\mathcal{E}_F^j$ is in $\mathrm{PERM}(T(F))$. The idea is to transform the level planar embedding $\mathcal{E}_F^j$ into a level planar embedding of $R^j_{\{1,2,\ldots,i\}}$. The transformation replaces in $\mathcal{E}_F^j$ every vertex $w \in V^{j+1}$, $w \neq v$, by a sequence of virtual vertices. We associate every virtual vertex of $w$ with one of the reduced extended forms $R_i^j$ that have been $v$-merged into $R^j_{\{1,2,\ldots,i\}}$ if this reduced extended form was adjacent to $w$. However, in order to perform this transformation, we first need to show that the incoming edges of $w$ that are associated with $R_i^j$ appear consecutively around $w$ in $\mathcal{E}_F^j$.

Let $q$ denote the number of level-$(j+1)$ vertices of $F$. Let $\pi = [w_1, w_2, \ldots, w_q]$ be the witness of the embedding $\mathcal{E}_F^j$. We first show that for every vertex $w_i \neq v$, $i = 1, 2, \ldots, q$, the incoming edges of $w_i$ that belong to a reduced extended form $R_l^j$, $l \in \{1, 2, \ldots, p\}$, appear consecutively in the clockwise order of incoming edges of $w_i$ in the embedding $\mathcal{E}_F^j$. The following two cases may occur.

1. All $R_1^j, R_2^j, \ldots, R_p^j$ are primary.

   Let $R_y^j$ and $R_z^j$ $y, z \in \{1, 2, \ldots, p\}$, $y \neq z$, be two reduced extended forms with a $(j+1)$-level vertex $x$. Let $y_1, y_2, \ldots, y_c$, $c \geq 1$ be the set of level-$j$ neighbors of $x$ in $R_y^j$ and let $z_1, z_2, \ldots, z_d$, $d \geq 2$, be the set of level-$j$ neighbors of $x$ in $R_z^j$. Assume that in the clockwise order of neighbors of $x$ in the level planar embedding of $\mathcal{E}_F^j$ a vertex $y_l \in \{y_1, y_2, \ldots, y_c\}$ appears between the vertices $z_a$ and $z_b$ with $z_a, z_b \in \{z_1, z_2, \ldots, z_d\}$, $z_a \neq z_b$. Since $R_z^j$ is primary, there exists a path $P_z$ in $R_z^j$

connecting the vertices $z_a$ and $z_b$ neither using $x$ nor $v$. Since $R_y^j$ is primary and connected, there exists a path $P_y$ in $R_y^j$ connecting the vertices $y_l$ and $v$ not using $x$. Both paths cross each other but have no vertex in common, a contradiction to the level planar embedding $\mathcal{E}_F^j$. See Fig. 4.41 for illustration.

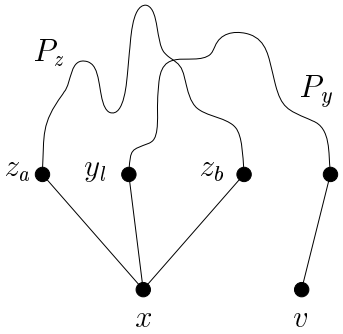



Figure 4.41: Illustration of the proof of Lemma 4.20. If in $\mathcal{E}_F^j$ the sequence of incoming edges of $x$ belonging to $R_z^j$ is separated by some edge $(y_l, x)$ with $y_l \notin V(R_z^j)$ two vertex disjoint paths exist that cross each other.

2. At least one $R_i^j$ is secondary.

   Let $R_i^j \in \{R_1^j, R_2^j, \ldots, R_p^j\}$ be a secondary reduced extended form. Thus there exist vertices $x_1, x_2, \ldots, x_l \in \{w_1, w_2, \ldots, w_q\}$ such that $R_i^j$ is $x_i$-connected for all $i = 1, 2, \ldots, l$, $(l \geq 1)$. The fact that $R_i^j$ is $x_i$-connected implies that all forms having a virtual vertex labeled $x_i$ in their frontier have been $x_i$-merged into $R_i^j$ in an earlier step. Therefore, $R_i^j$ is the only reduced extended component that contains the vertices $x_1, x_2, \ldots, x_l$ on level $j+1$. Hence the incoming edges of $x_i$, $i = 1, 2, \ldots, l$, corresponding to $R_i^j$ form a consecutive sequence in the clockwise order. For all other vertices $\{w_1, w_2, \ldots, w_q\} - \{x_1, x_2, \ldots, x_l\}$ the same argument as in the first case applies.

We construct from $\mathcal{E}_F^j$ an embedding $\mathcal{E}'$. Introduce for every reduced extended form $R_l^j$, $l \in \{1, 2, \ldots, p\}$ and for every vertex $w_i \neq v$, $i = 1, 2, \ldots, q$, a vertex $w_i^l$ if $S_l^{w_i} \neq \emptyset$. Replace each vertex $w_i \neq v$ by a sequence of virtual vertices $\{w_i^l \mid S_l^{w_i} \neq \emptyset, 1 \leq l \leq p\}$, such that $w_i^l$ is adjacent to the same vertices as $w_i$ in $R_l^j$. We do not change the order of the incoming edges of $w_i$, and label each vertex $w_i^l$ with $w_i$. See for an illustration the example shown in Fig. 4.42.

Since the incoming edges of $w_i$ that correspond to a reduced extended form $R_l^j$ appear consecutively around $w_i$, the embedding $\mathcal{E}'$ is level planar. Furthermore, the graph corresponding to $\mathcal{E}'$ is identical to $R_{\{1,2,\ldots,i\}}^j$, and we have by assumption that the witness $\pi'$ of $\mathcal{E}'$ is in $\mathrm{PERM}(T(R_{\{1,2,\ldots,i\}}^j))$. Since the witness $\pi$ arises from $\pi'$ by identifying all (consecutive) vertices with a common label $w \in V^{j+1}$, we have by construction that $\pi \in \mathrm{PERM}(T(F))$. □

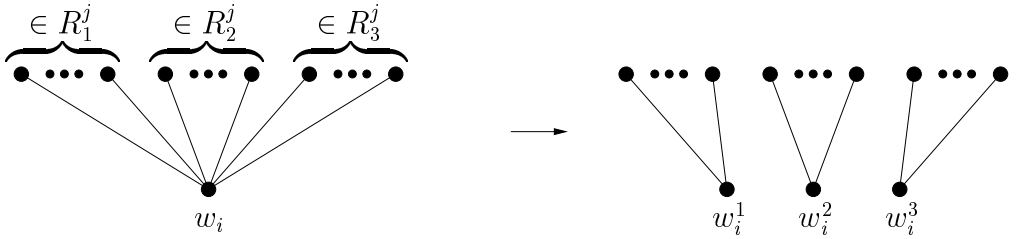



Figure 4.42: Illustration of the proof of Lemma 4.20. For the forms $R_1^j$, $R_2^j$, and $R_3^j$ we introduce virtual vertices $w_i^1$, $w_i^2$, and $w_i^3$. We replace $w_i$ by the sequence of virtual vertices, not reordering the incoming edges of $w_i$.

**Corollary 4.21.** *Let $G = (V, E)$ be a level graph with $k > 0$ levels. Let $F_i^j$, $i \in \{1, 2, \ldots, m_j\}$, be an arbitrary component of $G^j$. There exists a PQ-tree $T(R_i^j)$ representing all level planar embeddings of $F_i^j$.*

*Proof.* This follows immediately from Lemma 4.5 and the Lemmas 4.16, 4.19, and 4.20 by an inductive argument. □

The supposition of Corollary 4.21 does not require that $G$ is level planar. In case that $G$ is not level planar, there exists a level $j \leq k$ and a component $F_i^j$, $i \in \{1, 2, \ldots, m_j\}$, of $G^j$ such that $F_i^j$ is not level planar and the corresponding $PQ$-tree is the empty tree.

**Theorem 4.22.** *The algorithm LEVEL-PLANARITY-TEST tests a proper level graph $G = (V, E)$ for level planarity.*

*Proof.* Clear by Lemmas 4.16, 4.19, and 4.20. □

## 4.7 Bounding the Number of Reductions

When considering the algorithm LEVEL-PLANARITY-TEST, the interesting question comes up, how often the $PQ$-tree function REDUCE is called. Obviously, the number of calls for REDUCE in the first reduction phase is bounded by $m$ and therefore in $\mathcal{O}(n)$, while the number of calls of REDUCE performed upon a successful INSERT operation is bounded by $s - 1$, where $s$ denotes the number of sources of $G$. But how many extra REDUCE operations have to be executed in the second reduction phase? The following lemma will show that the number of these extra calls is also bounded by $s - 1$. We note that this is not a trivial result, as has been stated by Heath and Pemmaraju (1996). They observe that only one extra reduction is possible after every INSERT operation. This is only true for the first INSERT operation at a vertex $v$. If more forms have to be merged, their observation is not true in general.

The next two lemmas show that if the components are merged according to their size, the number of extra REDUCE operations per INSERT operation is bounded by 1. This however, is not true if the order of merge operations is changed. The first lemma shows a technical result needed in the proof of Lemma 4.24 that states the final result.

**Lemma 4.23.** *Let $R_i^j$, $1 \leq j \leq k-1$, $1 \leq i \leq m_j$, be an arbitrary reduced extended form. Let $w$ be an arbitrary level-$l$ vertex, $1 \leq l \leq j$, of $R_i^j$ and let $v$ be an arbitrary level-$(j+1)$ vertex in $R_i^j$. Then one of the following two conditions holds.*

1. *There exists a path $P$ connecting $w$ and $v$ such that $\text{lev}(u) < j+1$ for every vertex $u \neq v$ on $P$.*
2. *There exists a path $P'$ connecting $w$ and $v$ such that for every vertex $u \neq v$ on $P'$ the following condition holds.*

$$\text{lev}(u) = j+1 \text{ if and only if } R_i^j \text{ is } u\text{-connected.}$$

*Proof.* Since $R_i^j$ is connected there exists a path connecting $w$ and $v$. Assume that there is no path $P$ connecting $w$ and $v$ such that the first condition is satisfied. Hence, a path $P'$ exists traversing at least one vertex $u \neq v$ such that $\text{lev}(u) = j+1$ and $u$ is a cut vertex. Hence removing $u$ separates $R_i^j$ into several components, all being adjacent to the vertex $u$. Thus these components have been merged at vertex $u$ to $R_i^j$.

Assume now that there exists a vertex $u' \in P'$ with $u' \neq v$ and $\text{lev}(u') = j+1$ but $R_i^j$ is $u'$-unconnected. Hence $u'$ is not a cut vertex. Let $(y, u')$ and $(u', z)$ be the two edges of the path incident to $u'$. Since $u'$ is not a cut vertex, there exists a path $P_y^z$ connecting $y$ and $z$ using only vertices in $\bigcup_{l=1}^{j} V^l$. By successively replacing for every level $j+1$-vertex $y$ on $P'$ the path $P'$ by the symmetric difference $(P' - P_y^z) \cup (P_y^z - P')$, we get a path $P''$ connecting $w$ and $v$ such that for every vertex $u \in P$ with $\text{lev}(u) = j+1$ we have that $R_i^j$ is $u$-connected. □

**Lemma 4.24.** *Let $G$ be a level planar graph. Let $R_1^j, R_2^j, \ldots, R_p^j$, $F$, and the vertex $v$ be defined as in Lemma 4.20. Let $T(R^j_{\{1,2,\ldots,p\}})$ be the PQ-tree constructed as described in the second merge phase of CHECK-LEVEL, representing all level planar embeddings of $R^j_{\{1,2,\ldots,p\}}$, and let $T(F)$ be the PQ-tree constructed as described in the second merge phase of CHECK-LEVEL, by reducing in $T(R^j_{\{1,2,\ldots,p\}})$ all leaves with a common label $w$.*

*Then at most $p-1$ calls for the function REDUCE are necessary to construct $T(F)$ from $T(R^j_{\{1,2,\ldots,p\}})$.*

*Proof.* We show that if the forms $R_1^j, R_2^j, \ldots, R_p^j$ (and their corresponding $PQ$-trees) are $v$-merged according to their size, there can be at most one extra call for the function REDUCE after one merge operation. For simplicity, we prove this only for the first $v$-merge operation of $R_1^j$ and $R_2^j$. At the end of the proof we show how to obtain the same results for a $v$-merge operation of $R^j_{\{1,2,\ldots,l-1\}}$ and $R_l^j$.

Let $T(R_1^j)$ and $T(R_2^j)$ be the $PQ$-trees corresponding to $R_1^j$ and $R_2^j$. By assumption we have $\mathrm{LL}(R_1^j) \leq \mathrm{LL}(R_2^j)$. Assume that there exist virtual vertices with label $v$, $x$ and $y$ in $R_1^j$ and $R_2^j$ on level $j+1$. Let $v_1$, $x_1$ and $y_1$ be the virtual vertices with labels $v$, $x$, and $y$, respectively, in $R_1^j$. Let $v_2$, $x_2$ and $y_2$ be the virtual vertices with labels $v$, $x$, and $y$, respectively, in $R_2^j$. Let $\overline{S}_2^v$ be the set of virtual vertices of $R_2^j$ except for the vertex $v_2$. According to Lemma 4.18, the set $\overline{S}_2^v$ forms a consecutive sequence in every level planar embedding of $R_{\{1,2\}}^j$ and the vertex $v_2$ must be placed next to $\overline{S}_2^v$. Let $q = |\overline{S}_2^v|$, and $\overline{S}_2^v = \{w_1, w_2, \ldots, w_q\}$. Let $x_2 = w_\mu$ and $y_2 = w_\nu$ for some $w_\mu, w_\nu \in \overline{S}_2^v$, $1 < \mu < \nu \leq q$, and let without loss of generality

$$[v, w_1, w_2, \ldots, w_{\mu-1}, x_2, w_{\mu+1}, \ldots, w_{\nu-1}, y_2, w_{\nu+1}, \ldots, w_q]$$

be a witness for a level planar embedding of $R_2^j$. For the vertices $x_2$ and $y_2$, there exist paths $P_{x_2}$ and $P_{y_2}$ such that $P_{x_2}$ connects $v_2$ and $x_2$ and $P_{y_2}$ connects $v_2$ and $y_2$. The paths $P_{x_2}$ and $P_{y_2}$ can be constructed such that there exists a vertex $w \in P_{x_2} \cap P_{y_2}$, and splitting both paths at vertex $w$ (see Fig. 4.43 for an illustration) we get

- $P_{x_2}^{v_2} \subset P_{x_2}$ connecting the vertices $v_2$ and $w$,
- $P_{x_2}^{w} \subseteq P_{x_2}$ connecting the vertices $w$ and $x_2$,
- $P_{y_2}^{v_2} \subset P_{y_2}$ connecting the vertices $v_2$ and $w$,
- $P_{y_2}^{w} \subseteq P_{y_2}$ connecting the vertices $w$ and $y_2$,
- $P_{x_2}^{v_2}$ and $P_{y_2}^{v_2}$ are identical, and
- $P_{x_2}^{w}$ and $P_{y_2}^{w}$ are disjoint except for the vertex $w$.

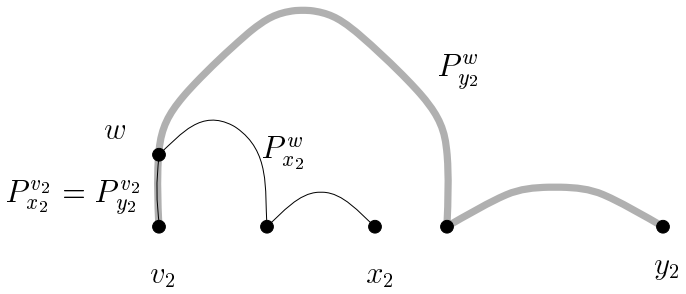



Figure 4.43: Illustration to the proof of Lemma 4.24. The path $P_{x_2}$ is drawn as thin black line. The path $P_{y_2}$ is drawn as thick grey line.

The paths $P_{x_2}$ and $P_{y_2}$ can be constructed according to Lemma 4.23 such that one of the following conditions holds for every path $P_\alpha$, $\alpha \in \{x_2, y_2\}$.

1. For every vertex $u \neq \alpha$, $u \neq v_2$ on the path $P_\alpha$, the inequality $\mathrm{lev}(u) \leq j$ holds.

2. If a vertex $u \neq \alpha$, $u \neq v_2$ on the path $P_\alpha$ exists such that $\text{lev}(u) = j + 1$ holds, then $R_2^j$ is $u$-connected.

Let $u_x$ be the first vertex on the path $P_{x_2}$ from $v_2$ to $x_2$ such that $\text{lev}(u_x) = j + 1$ and let $u_y$ be the first vertex on the path $P_{y_2}$ from $v_2$ to $y_2$ such that $\text{lev}(u_y) = j + 1$. Notice that if for every vertex $u$ on $P_{x_2} - \{v_2, x_2\}$, or on $P_{y_2} - \{v_2, y_2\}$ the inequality $\text{lev}(u) \leq j$ holds, then $u_x = x_2$, or $u_y = y_2$, respectively.

Let $z_2$ be a vertex on $P_{x_2}$ or $P_{y_2}$ such that for every vertex $a \in P_{x_2} \cup P_{y_2}$ the inequality $\text{lev}(z_2) \leq \text{lev}(a)$ holds. Since $\text{LL}(R_1^j) \leq \text{LL}(R_2^j)$, there exists a vertex $z_1$ in $R_1^j$ such that the inequality $\text{lev}(z_1) \leq \text{lev}(z_2)$ holds. There exists a path $P_{v_1}$ connecting $z_1$ and $v_1$. Since $R_1^j$ is $v$-unconnected, there exists path a $P_{x_1}$ connecting $z_1$ and $x_1$ and a path $P_{y_1}$ connecting $z_1$ and $y_1$ such that both paths $P_{x_1}$ and $P_{y_1}$ do not traverse $v_1$. According to Lemma 4.23 we can construct every path $P_\alpha$, $\alpha \in \{v_1, x_1, y_1\}$ such that one of following conditions holds.

1. For every vertex $u \neq \alpha$ on the path $P_\alpha$, the inequality $\text{lev}(u) \leq j$ holds.
2. If a vertex $u \neq \alpha$ on the path $P_\alpha$ exists such that $\text{lev}(u) = j + 1$ holds, then $R_1^j$ is $u$-connected.

Assume now that the path $P_{x_1}$ or $P_{y_1}$ traverses a level-$(j+1)$ vertex $u$ with the same label as $u_y$ or $u_x$. By construction, $R_1^j$ must be $u$-connected. This implies that $R_1^j$ and $R_2^j$ have been merged at $u$ in an earlier iteration. Hence we may assume that the paths $P_{x_1}$ and $P_{y_1}$ do not traverse a level-$(j + 1)$ vertex $u$ with the same label as $u_x$ or $u_y$.

We distinguish the following two cases.

1. $u_x \neq u_y$

   Let $P'_{y_2}$ be the partial path of $P_{y_2}$ from $v_2$ to $u_y$. By construction the path $P'_{y_2}$ connects the level-$(j + 1)$ vertices $v_2$ and $u_y$ traversing only vertices $a \in P'_{y_2}$ with $\text{lev}(a) \leq j$. By construction the paths $P_{x_1}$ and $P'_{y_2}$ are disjoint but do cross each other when merging the vertices labeled $x$, $y$, and $v$. Hence $G$ is not be level planar, which is a contradiction. See Fig. 4.44 for an illustration.

2. $u_x = u_y$

   From $u_x = u_y$ it follows that $u_x \neq x_2$ and $u_y \neq y_2$. Otherwise, $R_2^j$ would be $x$- or $y$-connected and $R_1^j$ and $R_2^j$ would have been merged at the vertex $x$ or $y$ first. By construction the paths $P_{x_1}$ and $P_{y_2}$ are disjoint but do cross each other when merging the vertices labeled $v$, $x$ and $y$. Hence $G$ is not level planar, which is a contradiction. See Fig. 4.45 as an illustration.

Now, let $v_{l-1}$, $x_{l-1}$ and $y_{l-1}$ be virtual vertices with labels $v$, $x$, and $y$, respectively, in $R^j_{\{1,2,\ldots,l-1\}}$, $1 < l \leq p$. Let $v_l$, $x_l$, and $y_l$ be virtual vertices with labels $v$, $x$, and $y$, respectively, in $R_l^j$. For every vertex $x_l$ and $y_l$, paths $P_{x_l}$ and $P_{y_l}$ are constructed analogously

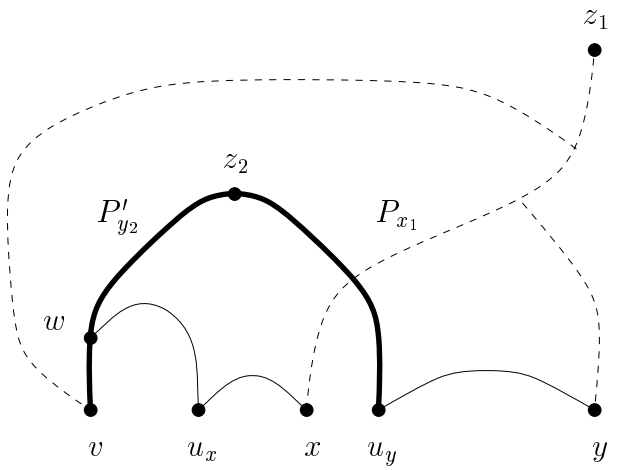



Figure 4.44: Illustration to the proof of Lemma 4.24. Case $u_x \neq u_y$. The fat line shows the path from the vertex labeled $v$ to $u_y$ that is crossed by $P_{x_1}$.

to the paths of $R_2^j$, thereby using the fact that according to Lemma 4.18 the set of virtual vertices of $R_l^j$ form a consecutive sequence in all level planar embeddings of $R^j_{\{1,2,\ldots,l\}}$, and the vertex labeled $v$ is the endmost vertex in the sequence.

Let $z_l$ be a vertex on $P_{x_l}$ or $P_{y_l}$ such that for every vertex $a \in P_{x_l} \cup P_{y_l}$ the inequality $\mathrm{lev}(z_l) \leq \mathrm{lev}(a)$ holds. Since $\mathrm{LL}(R_1^j) = \mathrm{LL}(R^j_{\{1,2\}}) = \cdots = \mathrm{LL}(R^j_{\{1,2,\ldots,l-1\}}) \leq \mathrm{LL}(R_l^j)$, there exist vertices $z_v$, $z_x$, and $z_y$ in $R^j_{\{1,2,\ldots,l-1\}}$ such that the inequalities

$$\begin{aligned} \mathrm{lev}(z_v) &\leq \mathrm{lev}(z_l)\ , \\ \mathrm{lev}(z_y) &\leq \mathrm{lev}(z_l)\ , \\ \mathrm{lev}(z_x) &\leq \mathrm{lev}(z_l) \end{aligned}$$

hold such that there exists a path $P_{z_v}$ connecting $z_v$ and $v_{l-1}$, a path $P_{z_x}$ connecting $z_x$ and $x_{l-1}$, a path $P_{z_y}$ connecting $z_y$ to $y_{l-1}$, and the paths $P_{z_x}$ and $P_{z_y}$ do not traverse $v$. Analogously to the case of $R_2^j$ we then construct a contradiction.

It follows that we have for every merge operation at most one extra call of the function REDUCE, yielding a total number of at most $p-1$ extra function calls. □

We note that the usage of the result of Lemma 4.18 is crucial for Lemma 4.24. If the reduced extended forms are not $v$-merged according to their sizes, and thus Lemma 4.18 does not hold, there might exist an arbitrary number of vertices that can be identified after a successful $v$-merge operation. The number of extra calls of the function REDUCE then may exceed one per merge operation. For the algorithm LEVEL-PLANARITY-TEST the Lemma 4.24 immediately yields the following Corollary.

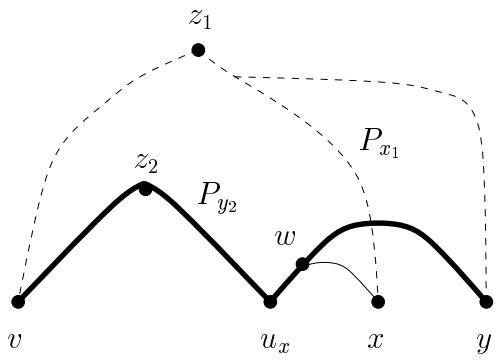



Figure 4.45: Illustration to the proof of Lemma 4.24. Case $u_x = u_y$. The fat line shows the path from the vertex labeled $v$ to the vertex labeled $y$ that is crossed by $P_{x_1}$.

**Corollary 4.25.** *Let $G$ be a level graph with $k > 1$ levels. Let $r$ be the number of calls of the function REDUCE, performed by LEVEL-PLANARITY-TEST on the graph $G$. Then $r \in \mathcal{O}(n)$.*

The proof of Lemma 4.24 reveals that if two $PQ$-trees are $v$-merged, obeying the order of merging according to their sizes, they both may have at most one leaf with the same label $w \neq v$ in their frontier. If both trees have a third leaf with similar label in their frontier, the graph is not level planar.

## 4.8 Proving $\mathcal{O}(n \log n)$ Running Time

After proving the correctness of the algorithm, we shall now determine its running time. The following theorem shows that LEVEL-PLANARITY-TEST can be implemented such that the running time is in $\mathcal{O}(n \log n)$. It turns out that all "difficult" operations such as reducing and merging can be performed in a total $\mathcal{O}(n)$ time. The linear-logarithmic running time is created by update operations on the leaves after $PQ$-trees have been merged.

**Theorem 4.26.** *The algorithm LEVEL-PLANARITY-TEST can be implemented, such that the running time for proper level graphs is in $\mathcal{O}(n \log n)$.*

*Proof.* The linear-logarithmic running time follows from an amortized analysis. Let $s$ denote the number of sources of $G$. Due to Corollary 2.2 we may assume that $m \leq 3n - 6$ holds. First, the number of operations that is performed in all calls of the function INSERT is proven to be in $\mathcal{O}(n)$. As has been explained in Section 3.1, an efficient implementation of $PQ$-trees implies that not every node of a $PQ$-tree has a valid parent pointer. Only children of $P$-nodes and the endmost children of $Q$-nodes have a pointer to their parents. The function INSERT traverses the path from the pertinent leaf towards the root of the

$PQ$-tree $T_{large}$ of the higher form, in order to find an appropriate position to place the smaller $PQ$-tree $T_{small}$ into the larger one. Thus INSERT uses existing parent pointers. We show that if INSERT detects a node with no parent pointer, and no appropriate position in $T_{large}$ has been found for $T_{small}$ yet, the graph $G$ is not level planar.

During the application of INSERT, ML-values of the nodes have to be checked. Since every child of a $P$-node $X$ has a pointer to its parent, the ML-value of $X$ is stored directly at $X$. The ML-values between the children of a $Q$-node $Y$ are not stored at $Y$ itself but stored at the children. Hence, no parent pointer is needed for accessing the ML-value between two sibling children of a $Q$-node. Assume now that INSERT passes a node $X'$ that has no pointer to its parent $X$. Then $X$ must be a $Q$-node and $X'$ is not an endmost child of $X$. If INSERT does not succeed in placing $T_{small}$ next to $X'$ (which implies that the ML-values of $X$ and its direct siblings are equal or larger than $\mathrm{LL}(T_{small})$) the graph is not level planar: placing $T_{small}$ next to $X$ creates crossings, and finding an appropriate place further up the tree will construct a $PQ$-tree $T_{merge}$ where in all permissible permutations the pertinent leaves of $T_{large}$ are separated by the endmost children of $X$ from the pertinent leaves of $T_{small}$.

Heath and Pemmaraju (1995, 1996) showed that the overall number of nodes traversed by INSERT does not exceed $\mathcal{O}(n)$. We cite their proof, since it contains valid information that is needed to estimate the number of steps performed by all calls of the function REDUCE. Furthermore we complete the proof by integrating the analysis of the data structure $PQ$-tree. Since LEVEL-PLANARITY-TEST stops traversing $T_{large}$ every time it detects a node without a valid parent pointer, the number of nodes that have to be visited in $T_{large}$ is proportional to the height of the form $F_{small}$ corresponding to $T_{small}$. Let $F_{large}$ be the form corresponding to $T_{large}$. There exists a path in $F_{large}$ that corresponds to the path in $T_{large}$ traversed by INSERT. We estimate how often an edge in $G$ can be traversed by such paths. Every edge $e$ belongs to the boundary of at most two faces. Traversing a path $P$ from a lowest level $j$ to a level $l$, $1 \leq l < j$, INSERT searches for a place to insert the smaller component on one side of the path $P$. After successfully inserting the component, the corresponding new $PQ$-tree is reduced, and the pertinent vertices are finally merged into one vertex in the merged form. Hence, INSERT is not able to place any other form on the same side of $P$ where $F_{small}$ has been placed (except for singular forms that require only constant time when inserted into a nonsingular form). Thus, every edge is traversed at most twice by such paths. Hence, the total number of nodes traversed by all calls of INSERT is bounded by $m \in \mathcal{O}(n)$.

With the analysis of the merge operations, Heath and Pemmaraju (1995, 1996) finish their proof on the claimed linear running time. But this is not sufficient since update operations have not been considered. We now show that the accumulated running time of all calls of the function REDUCE is linear. According to Theorems 3.4 and 3.8, the overall time needed by the calls of REDUCE for all calls in the first reduction phase during all executions of CHECK-LEVEL is in $\mathcal{O}(n)$.

We now determine the number of operations needed by all calls of REDUCE during the second phase. Since the number of leaves labeled $v$ that have to be reduced after a $v$-merge

operation is exactly two, the number of nodes traversed by REDUCE is bounded by the height of $T_{small}$. The overall time needed for these function calls is again in $\mathcal{O}(n)$. According to Lemma 4.24 there is at most one extra call for the function REDUCE for every call of the function INSERT to reduce leaves labeled $w \neq v$. Thus the overall time needed by these calls of REDUCE is also in $\mathcal{O}(n)$.

It follows that the running time of all calls of the function REDUCE and all calls of the function INSERT is in $\mathcal{O}(n)$. Every time $PQ$-trees have to be merged, the $PQ$-trees have to be sorted by their sizes. Since every merge operation reduces the number of $PQ$-trees by one, the total cost of sorting the $PQ$-trees is in $\mathcal{O}(s)$, applying bucket sort algorithms (see e.g. Cormen *et al.* (1990)).

However, the maintenance of the $PQ$-trees is more expensive. The merge operations are accompanied by update operations that sum up to linear-logarithmic runtime. We note that the analysis of these necessary updates has not been performed by Heath and Pemmaraju (1995, 1996). Leaves need to know to which $PQ$-tree they belong to since they need to access information such as the LL-value of the corresponding form. Furthermore, after two $PQ$-trees $T_i$ and $T_l$ have been $v$-merged, leaves labeled $w \neq v$ in both $PQ$-trees have to be detected and reduced. Thus we have to compare frontier($T_i$) and frontier($T_l$) for leaves with a same label, which sums up to $\mathcal{O}(m \log m)$ steps when scanning and updating the smaller set of frontier($T_i$) and frontier($T_l$). Hence, the following running time is obtained:

$$\mathcal{O}(n + s + m \log m).$$

Since $m$ is bounded by $n$ and $s \leq n$, the running time of LEVEL-PLANARITY-TEST can be bounded by $\mathcal{O}(n \log n)$. □

The next section discusses how the second reduction phase can be modified in order to obtain even linear running time for the level planarity test.

## 4.9 Improving to $\mathcal{O}(n)$ Running Time

As has been shown in the previous section, all operations can be performed using only linear time except the update operations. Two strategies are combined in the second reduction phase in order to achieve linear running time.

(i) We avoid updates of the leaves by showing how necessary information can be obtained when needed.

(ii) We avoid scanning for leaves with same label $w \neq v$ after a $v$-merge operation.

The previous approach made sure that after $v$-merging several $PQ$-trees, the new $PQ$-tree indeed represents all embeddings of its corresponding reduced extended form. This was obtained by reducing all leaves labeled $w \neq v$ in the new $PQ$-tree. The new approach does

not reduce any set of leaves labeled $w \neq v$. They are reduced only if necessary. In case that $PQ$-trees have to be $w$-merged, $w \neq v$, in a subsequent merge operation, we first reduce in every $PQ$-tree all leaves labeled $w$, replacing them by a single representative and then continue $w$-merging the $PQ$-trees. Thus, the new approach does not construct proper reduced extended forms when $v$-merging reduced extended forms. Such a reduced extended form, where not all virtual vertices with the same label are identified, is called *sloppy*.

The results of Section 4.6 imply that the vertices of level $j+1$, $1 \leq j < k$, may be reduced in any arbitrary order. Thus we may assume that the vertices of $V^{j+1}$ are numbered arbitrarily as $v^1, v^2, \ldots, v^{|V^{j+1}|}$. The $PQ$-trees are merged according to that numbering of the vertices.

If the leaves are not updated after merge operations, we need to discuss how to find out to which $PQ$-tree they belong to. Consider a set of leaves labeled by $v^i$, $2 \leq i \leq |V^{j+1}|$. Some of the leaves may have belonged to a $PQ$-tree $T$ that has been $v^l$-merged, $1 \leq l < i$, into another $PQ$-tree. A quick solution for obtaining the update information is the application of a *disjoint set forest* (see e.g. Cormen *et al.* (1990)) where every set represents all $PQ$-trees that have been merged into another $PQ$-tree. Since each $PQ$-tree $T$ is unique, the disjoint set forest can be implemented keeping a unique identification number for each tree using the well known operations *MakeSet*, *FindSet* and *Union*. This is easy to implement but yields only a $\mathcal{O}(n\alpha(m,n))$ time algorithm (with $\alpha(m,n)$ being the inverse Ackermann function). Thus maintaining the necessary information via dynamic set operations improves the running time but is still not linear. The following lemma finally leads to a linear time algorithm.

**Lemma 4.27.** *Let $T_1$ and $T_2$ be two $PQ$-trees such that there exists a leaf $v_1 \in \text{frontier}(T_1)$ labeled $v$ and a leaf $v_2 \in \text{frontier}(T_2)$ labeled $v$. Assume further that $\text{LL}(T_1) \leq \text{LL}(T_2)$ and let $X_2$ denote the root of $T_2$ and let $T_1'$ denote the tree that is constructed by $v$-merging $T_2$ into $T_1$. Assume further that $T_1'$ is reducible with respect to the leaves labeled $v$. Then the function call REDUCE($T_1'$,$S_1^v$) traverses every node on the unique path from $v_2$ to $X_2$ and every node on this path has a valid parent pointer.*

*Proof.* Consider a node on the path from the leaf $v_2$ to $X_2$ that has not a valid parent pointer. Hence, it must be the interior child of a $Q$-node. Thus $v_2$ does not form a consecutive sequence with $v_1$ in any permutation of $\text{PERM}(T_1')$, and $T_1'$ is not reducible with respect to the leaves labeled $v$. □

This observation leads to the following simple considerations. For every leaf labeled $v$ in a tree that corresponds to a smaller form, the information on the LL-value is stored in the root, and the leaf needs a pointer to the root. Hence, we go up the tree starting at the leaf labeled $v$ until we detect the root. Since the function-call REDUCE also traverses the path from a leaf labeled $v$ to the root, this extra traversal of the nodes does not have an effect on the asymptotic running time.

However, it is not allowed to traverse the path from a leaf labeled $v$ to the root if the leaf is contained in the frontier of the $PQ$-tree with the lower LL-value. Applying such traversals

to the $PQ$-trees with the lower LL-values results in a quadratic running time. The idea is to recognize the tree with the lower LL-value implicitly. In order to insert a tree $T_2$ into another tree $T_1$, we only need to guarantee that $\mathrm{LL}(T_1) \leq \mathrm{LL}(T_2)$ but it is not necessary to know the precise value of $\mathrm{LL}(T_1)$.

Figure 4.46 visualizes our strategy. Two $PQ$-trees $T_1$ and $T_2$ have to be merged. For a reduction of the merged $PQ$-tree with respect to the leaves $v_1$ and $v_2$, we need to traverse the paths corresponding to the thick grey lines. Our idea is to traverse these paths upwards until we detect the root of $T_2$ and the node $X$ in $T_1$.

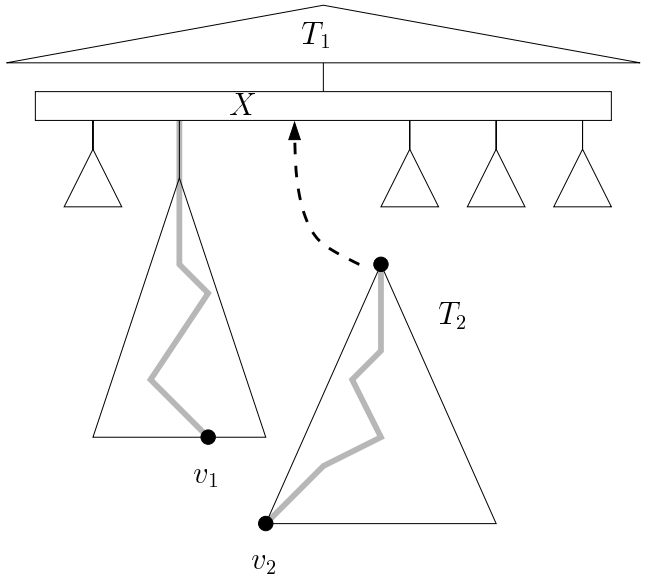



Figure 4.46: $PQ$-trees $T_1$ and $T_2$ have to be merged. The thick grey lines denote the paths that are traversed by the function REDUCE.

First of all, we assume for simplicity that no $PQ$-tree has more than one leaf labeled $v$ in its frontier, and that the $v$-merged $PQ$-trees are reducible. The idea is to detect for one of the two leaves its corresponding root while detecting an appropriate node $X$ in the other tree, such that the tree with larger LL-value can be inserted into the tree with smaller LL-value at node $X$.

For comparing nodes in $PQ$-trees while traversing the paths from the leaves towards the root their ML-values are used. The *level* of a node $X$ in a $PQ$-tree with respect to $v$ is defined to be the ML-value of $X$ on a path from a leaf labeled $v$ to the root. Three possible cases occur when determining the level of a node $X$.

- If $X$ is a $P$-node, the level is equal to the ML-value of $X$.
- If $X$ is an endmost child of a $Q$-node, the level is equal to the ML-value of the adjacent sibling of the node $X$.

- If $X$ is an interior child of a $Q$-node, the level is equal to the minimum ML-value of the two adjacent siblings of the node $X$.

Starting with two leaves $v_1$ and $v_2$ labeled $v$, we go up both paths from the leaves to the roots simultaneously, stopping as soon as the root of one of the trees has been reached. We remember the node $X$ of the tree with the smaller LL-value, and consider the next leaf $v_3$ labeled $v$. Starting at $v_3$, we traverse the path to the root, stopping as soon as either the root of the tree or a node $Y$ on at most the level of $X$ has been reached. In the latter case we continue going further up both paths simultaneously, starting at $X$ and $Y$ until a root node is detected.

The last node in the tree with the lower LL-value is stored in $C_{\min}$ and the corresponding ML-value of $C_{\min}$ is stored in LOWML. When processing a leaf $v_i$, $i > 2$, we first traverse the path from $v_i$ to the root until either a root node or a node that is on the same level as $C_{\min}$ or a lower level is detected. In the first case $C_{\min}$ and LOWML are left unchanged. In the second case we continue traversing both paths simultaneously, stopping when reaching a root node, updating $C_{\min}$ and LOWML. We first present the code fragment that handles the processing of two leaves. It uses a method GET-NEW-LOW that performs the traversal of the paths from the leaves to their root. Using a queue in the code has no specific meaning, only making the code more readable.

let $v_1, v_2, \ldots, v_\mu$ denote the leaves labeled $v$ in $V^{j+1}$;
store all $v_1, v_2, \ldots, v_\mu$ in a queue Q;
$C_{\min}$ := Q.pop();
LOWML := $j + 1$;
while Q is not empty do
  $C_{\text{new}}$ := Q.pop();
  GET-NEW-LOW($C_{\min}$,$C_{\text{new}}$,LOWML);

The function GET-NEW-LOW uses a trivial function MLVAL that returns the necessary ML-values according to the rules presented above. The input values of GET-NEW-LOW are $C_{\min}$ storing the node on the lowest detected level, $C_{\text{new}}$ storing the new leaf that has to be checked, and LOWML storing the lowest level reached so far. The values in $C_{\min}$ and LOWML may change during the application of GET-NEW-LOW. The variables $Z_{\min}$, $Z_{\text{cand}}$ and $Z_{\text{new}}$ are used for traversing the paths going further up the trees. The variable $Z_{\min}$ holds the currently known node with the lowest ML-value, $Z_{\text{cand}}$ is the next candidate that is checked, and $Z_{\text{new}}$ is an ancestor of $Z_{\text{cand}}$. The procedure starts traversing the path from $Z_{\text{cand}}$ to the root comparing the ML-values of the nodes on the path with the ML-value of $Z_{\min}$ until an ancestor $Z_{\text{new}}$ of $Z_{\text{cand}}$ is detected that is either the root of a $PQ$-tree or a node with ML-value lower than $Z_{\min}$.

```
void GET-NEW-LOW(C_min,C_new,LOWML)
begin
   Z_min := C_min;
   Z_cand := C_new;
   while no root has been reached do
      go further up the tree starting at Z_cand until either
         1. a node Z_new is reached that is the root of a PQ-tree;
         or
         2. a node Z_new is reached such that MLVAL(Z_new) < LOWML;
      if we stopped due to 1. then
         C_min := Z_min;
         LOWML := MLVAL(Z_min);
         return;
      else
         Z_cand := Z_min;
         Z_min := Z_new;
         LOWML := MLVAL(Z_new);
end.
```

If a node on one of these paths does not have a valid parent pointer, it follows from Lemma 4.27 that the graph is not level planar. Since every $PQ$-tree was assumed to have only one leaf labeled $v$ in its frontier, we now discuss how to handle several leaves labeled $v$ in a $PQ$-tree. When going up the path from a leaf labeled $v$ to the root $X$ of a tree $T$, we assign a flag to every node on this path. Since all the nodes on the path will be traversed during the reduction, these flags can be removed at no additional cost. If the tree $T$ has another leaf labeled $v$, one of the nodes with a flag is detected when traversing the path towards the root. The detection of such a node with a flag indicates the existence of at least one other leaf labeled $v$. All these leaves belong to the same $PQ$-tree and are memorized and reduced before the $PQ$-trees are $v$-merged. The following code fragment presents the new second reduction phase.

**New Second Reduction Phase**

```
for l = 1 to |V^{j+1}| do
   for every leaf labeled v^l do
      find the corresponding PQ-tree;
   for every found PQ-tree T(R_i^j) do
      if S_i^{v^l} ≥ 2 then
         if REDUCE(T(R_i^j),S_i^{v^l}) = ∅ then return ∅;
         else
            let ṽ^l be a single representative of S_i^{v^l};
            UPDATE(S_i^{v^l},ṽ^l);
            REPLACE(S_i^{v^l},ṽ^l);
```

reorder indices such that $S_1^{v^l}, S_2^{v^l}, \ldots, S_p^{v^l} \neq \emptyset$, and $S_{p+1}^{v^l}, S_{p+2}^{v^l}, \ldots, S_{m_j}^{v^l} = \emptyset$;
let $q$ be the number of $v^l$-singular reduced extended forms;
eliminate all $v^l$-singular $R_i^j$ except for the one with the lowest LL-value;
renumber the remaining $R_i^j$ from 1 to $p - q + 1$;
$p := p - q + 1$;
sort the $R_i^j$, such that $\mathrm{LL}(R_1^j) \leq \mathrm{LL}(R_2^j) \leq \mathrm{LL}(R_3^j) \leq \cdots \leq \mathrm{LL}(R_p^j)$;
$T(R_1^j) := T(R_1^j)$;
for $i = 2$ to $p$ do
    $T(R_1^j) :=$ INSERT$(T(R_1^j), T(R_i^j), v^l)$;
    $R_1^j := R_1^j \cup_{v^l} R_i^j$;
    if REDUCE$(T(R_1^j), S_1^{v^l}) = \emptyset$ then return $\emptyset$;
    else
        let $\tilde{v}^l$ be a single representative of $S_1^{v^l}$;
        UPDATE$(S_1^{v^l}, \tilde{v}^l)$;
        REPLACE$(S_1^{v^l}, \tilde{v}^l)$;
update the root pointers of the leaves;
add for every source a corresponding $PQ$-tree to $\mathcal{T}(G^j)$;
return $\mathcal{T}(G^{j+1})$;

Proving the correctness of this new second reduction phase is quite similar to the proof given in Section 4.6. Again, we need to show that throughout every iteration the $PQ$-trees are correctly maintained and the set of permissible permutations always represents exactly the set of level planar embeddings of the corresponding form.

In the $\mathcal{O}(n \log n)$ approach we first showed that $v$-merging a set of $PQ$-trees indeed produces a new $PQ$-tree $T$ that represents all level planar embeddings of its corresponding form. Then we proved that reducing all leaves labeled $w \neq v$ in this new $PQ$-tree $T$ is performed correctly. For proving the correctness of the new approach, we have to prove the correctness in inversed order. We first show that reducing in a $PQ$-tree all leaves with the same label $v$ constructs a $PQ$-tree that represents all level planar embeddings of the corresponding sloppy reduced extended form with the virtual vertices labeled $v$ identified. Then we prove the correctness of $v$-merging a set of $PQ$-trees.

The next lemma proves the reduction of leaves labeled $v$ in a sloppy reduced extended form to be correct. Its result corresponds to the result of Lemma 4.20, and most parts of the proof are analogous.

**Lemma 4.28.** *Let $G = (V, E)$, be a level graph with $k > 1$ levels. Let $v^l \in V^{j+1}$, $1 \leq j < k$, $1 \leq l \leq |V_{j+1}|$. Let $R_i^j$ be a level planar sloppy reduced extended form with $S_i^{v^l} \neq \emptyset$ and with $|S_i^w| \leq 1$ for all $w = v^1, v^2, \ldots, v^{l-1}$. Let $T(R_i^j)$ be the corresponding $PQ$-tree, representing all level planar embeddings of $R_i^j$.*

*Let $F$ be the subgraph constructed from $R_i^j$ by identifying all virtual vertices labeled $v^l$ to a single vertex $v^l$. Let $T(F)$ be the $PQ$-tree constructed as described in the "new" sec-*

*ond merge phase of CHECK-LEVEL, by reducing in $T(R_i^j)$ all leaves labeled $v^l$. Then* $\mathrm{PERM}(T(F))$ *is exactly the set of permutations of level-$(j+1)$ vertices that appear in level planar embeddings of $F$.*

*Proof.* The proof of $T(F)$ representing level planar embeddings of $F$ is done analogous to the proof of Lemma 4.20.

We now prove that for any $\mathcal{E}_F^j$ a $PQ$-tree equivalent to $T(F)$ exists that represents exactly the permutation of the level-$(j+1)$ vertices of $\mathcal{E}_F^j$. The idea is as in Lemma 4.20 to transform the level planar embedding $\mathcal{E}_F^j$ into a level planar embedding of $R_i^j$ giving us valid information on the $PQ$-tree. The transformation replaces in $\mathcal{E}_F^j$ the vertex $v^l$ by a sequence of virtual vertices. We associate every virtual vertex of $v^l$ with a reduced extended form that has been $w$-merged, $w \in \{v^1, v^2, \ldots, v^{l-1}\}$, into $R_i^j$ in an earlier iteration of the new second reduction phase.

Let $q$ be the number of level-$(j+1)$ vertices of $F$. Let $\pi = [w_1, w_2, \ldots, w_q]$ be a witness of the embedding $\mathcal{E}_F^j$. $F$ is either primary or it has been constructed by merging sloppy reduced extended forms at vertices $w \in \{v^1, v^2, \ldots, v^{l-1}\}$. Hence the set of incoming edges of $v^l$ can be partitioned into sets of incoming edges belonging to the sloppy reduced extended forms that have been $w$-merged to create $R_i^j$. Using the same argument as in the proof of Lemma 4.20, the incoming edges of $v^l$ that belong to those sloppy reduced extended forms appear consecutively in the clockwise order of incoming edges of $v^l$ in the embedding $\mathcal{E}_F^j$.

We construct from $\mathcal{E}_F^j$ an embedding $\mathcal{E}'$. Introduce for every sloppy reduced extended form $R_\nu^j$, $\nu \in \{1, 2, \ldots, m_j\}$, that has been $w$-merged into $F$ a vertex $v_\nu^l$ if $S_\nu^{v^l} \neq \emptyset$. Replace $v^l$ by a sequence of virtual vertices $v_\nu^l$, such that $v_\nu^l$ is adjacent to the same vertices as $v^l$ in $R_\nu^j$. Label each vertex $v_\nu^l$ with $v^l$.

Since the incoming edges of $v^l$ that correspond to a reduced extended form $R_\nu^j$ appear consecutively around $v^l$, the so constructed embedding $\mathcal{E}'$ is obviously level planar. Furthermore the graph corresponding to $\mathcal{E}'$ is identical to $R_i^j$, and we have by assumption that the witness $\pi'$ of $\mathcal{E}'$ is in $\mathrm{PERM}(T(R_i^j))$. Since the witness $\pi$ arises from $\pi'$ by identifying all (consecutive) leaves labeled $v^l$, we have by construction that $\pi \in \mathrm{PERM}(T(F))$. □

The next lemma proves the correctness of $v$-merging a set of $PQ$-trees where every $PQ$-tree has exactly one leaf labeled $v$.

**Lemma 4.29.** *Let $G = (V, E)$ be a level graph with $k > 1$ levels, and let $v^l \in V^{j+1}$ be a vertex with $1 \leq l \leq |V^{j+1}|$. Let $R_1^j, R_2^j, \ldots, R_p^j$, $p \geq 2$, be level planar sloppy reduced extended forms such*

*(i)* $S_i^{v^l} \neq \emptyset$ *for all* $i \in \{1, 2, \ldots, p\}$, *and*

*(ii)* $\mathrm{LL}(R_1^j) \leq \mathrm{LL}(R_2^j) \leq \mathrm{LL}(R_3^j) \leq \cdots \leq \mathrm{LL}(R_p^j)$.

*Suppose that the $PQ$-trees $T(R_1^j), T(R_2^j), \ldots, T(R_p^j)$ represent all level planar embeddings of $R_1^j, R_2^j, \ldots, R_p^j$. Let $T(R_{\{1,2,\ldots,p\}}^j)$ be the $PQ$-tree constructed as described in the "new"*

*second merge phase of CHECK-LEVEL. Then* $\mathrm{PERM}(T(R^j_{\{1,2,\ldots,p\}}))$ *is exactly the set of permutations of level-*$(j+1)$ *vertices that appear in level planar embeddings of* $R^j_{\{1,2,\ldots,p\}}$.

*Proof.* The proof is similar to the proof of Lemma 4.19 using Lemma 4.17 instead of Lemma 4.18. □

**Theorem 4.30.** *The algorithm LEVEL-PLANAR-TEST using the modified second reduction phase tests a given proper level graph* $G = (V, E)$ *for level planarity and can be implemented such that the running time is in* $\mathcal{O}(n)$.

*Proof.* The correctness follows immediately from Lemma 4.5 and the Lemmas 4.16, 4.28 and 4.29 by an inductive argument. For the running time, we have from the discussion above that the number of steps performed to identify the $PQ$-trees corresponding to pertinent leaves labeled $v$ is proportional to the number of steps performed in reducing these leaves. According to the proof of Theorem 4.26 the overall number of steps performed on these operations is bounded by $\mathcal{O}(n)$. We now consider the update operations of the leaves that have to be performed after all merge and reduce operations for a level have been completed. For every $PQ$-tree we keep its leaves stored in a doubly linked list. Every time two $PQ$-trees are merged, these lists are merged as well. This can be done without knowing the tree with the lower LL-value. We simply connect the lists at the new single representative that has to be introduced after the merge operation (followed by a reduction) is complete. After finishing all merge and reduce operations we scan for every remaining tree the doubly linked list of its leaves, doing the necessary updates. The total cost of these operations is in $\mathcal{O}(m)$. Together with the results of Theorem 4.26 this yields an $\mathcal{O}(n)$ algorithm. □

## 4.10 Testing Nonproper Level Graphs

For simplicity, we restricted our attention to the level planarity testing of proper level graphs. Of course, every nonproper level graph can be transformed into a proper one by inserting dummy vertices. This strategy should not be applied since the resulting number of vertices may be quadratic in the original number of vertices. The following theorem shows that our linear time level planarity test works on nonproper level graphs as well as on proper level graphs.

**Theorem 4.31.** *The algorithm LEVEL-PLANAR-TEST tests any, not necessarily proper, level graph* $G = (V, E)$ *in* $\mathcal{O}(n)$ *time for level planarity.*

*Proof.* Consider a long edge $e = (v, w)$, $v \in V_j$, $w \in V_l$, $1 \leq j < l-1 \leq k-1$, traversing one or more levels. Thus inserting dummy vertices for $e$ in order to construct a proper hierarchy would result in a graph $G'$ such that every dummy vertex $u^e_i$, $i \in \{j+1, j+2, \ldots, l-1\}$ has exactly one incoming edge and one outgoing edge. However, the reduction of a $PQ$-tree $T$ with respect to a set $S$ with $|S| = 1$, replacing the set by a new set $S'$ with $|S'| = 1$ is

trivial and does not modify the $PQ$-tree. Hence we do not need to consider the dummy vertices and therefore do not introduce them at all. Therefore, with no change our linear time algorithm correctly tests also nonproper level graphs for level planarity. □

# Chapter 5

# Level Planar Embedding

One can easily obtain the following naive embedding algorithm for level graphs, as has been suggested by Heath and Pemmaraju (1996). Choose any total order on $V^k$ that is consistent with the set $\mathcal{T}(G^k)$. Choose then any total order on $V^{k-1}$ that is consistent with $\mathcal{T}(G^{k-1})$ and that, together with the chosen order of $V^k$ implies a level planar embedding on the subgraph of $G$ induced by $V^{k-1} \cup V^k$. Extend this construction one level at a time until a level planar embedding of $G$ results.

However, to perform this algorithm, it is necessary to keep a copy of the set of $PQ$-trees of every level $l$, $1 \leq l \leq k$. Providing the copies of the $PQ$-trees easily sums up to an $\mathcal{O}(n^2)$ time algorithm for level graphs. Besides, an appropriate total order of the vertices of $V^j$, $1 \leq j < k$, can only be detected by reducing subsets of the leaves of $G^j$, where the subsets are induced by the adjacency lists of the vertices of $V^{j+1}$. More precisely, for every pair of consecutive edges $e_1 = (v_1, w)$, $e_2 = (v_2, w)$, $v_1, v_2 \in V^j$, in the adjacency list of a vertex $w \in V^{j+1}$, we have to reduce the set of leaves corresponding to the vertices $v_1$, $v_2$ in $\mathcal{T}(G^j)$. This immediately yields an $\Omega(n^2)$ algorithm for nonproper level graphs, with $\Omega(n^2)$ dummy vertices for long edges, since we are forced to consider for every long edge its exact position on the level that is traversed by the long edge.

In this chapter an algorithm is presented that is based on the level planarity test as it was given in the previous chapter. The idea is to augment the graph $G$ to a level planar $st$-graph $G_{st}$, compute a planar embedding of $G_{st}$, and use this planar embedding to construct a level planar embedding of $G$. This approach yields an $\mathcal{O}(n)$-time algorithm for (not necessarily proper) level graphs.

The first section presents the concept of the approach. The second section shows how to add edges to a level graph without destroying level planarity. Adding edges is the key strategy. The third section proves the correctness of the level planar embedder, and the fourth section proves the linear time bound. The chapter closes with some remarks on possible modifications of the level planar embedder.

## 5.1 Concept of an Embedding

In order to compute a level planar embedding of a level planar graph $G = (V, E)$ with a leveling $\text{lev}_G$, the graph $G$ is augmented to a planar directed acyclic $st$-graph $G_{st} = (V_{st}, E_{st})$ with $V_{st} = V \uplus \{s, t\}$ and $E \subset E_{st}$ such that $\text{lev}_G$ induces a topological numbering of $V_{st}$, numbering $s$ with 0 and $t$ with $k + 1$. The topological numbering of $G_{st}$ induces a leveling of $G_{st}$ where the vertices of $V \subset V_{st}$ are on the same levels as in $G$.

The graph $G_{st}$ is embedded planar with the edge $(s, t)$ on the boundary of the outer face. The level planar embedding is then constructed from the planar embedding. We present an algorithm "CONSTRUCT-LEVEL-EMBED" for constructing the level planar embedding $\mathcal{E}_l$ with respect to the planar embedding $\mathcal{E}_{st}$ of $G_{st}$. The algorithm executes once the well-known depth first search (see, e.g., Cormen *et al.* (1990)), starting at the sink $t$ of $G_{st}$. Throughout the algorithm a depth first search tree $T$ is constructed only for the analysis of the algorithm.

```
E_l CONSTRUCT-LEVEL-EMBED(G_st,E_st)
   begin
      mark all vertices w ∈ V_st "not visited";
      T := ∅;
      DFS(t);
      return E_l;
   end.

void DFS(w)
   begin
      mark vertex w "visited";
      for each incoming edge (v, w) in the clockwise order of E_st do
         if v has not been visited then
            put v at the right end of level lev_G(v) in E_l;
            add edge (v, w) to T;
            DFS(v);
   end.
```

Using the algorithm CONSTRUCT-LEVEL-EMBED, the following important theorem is shown stating that every level planar graph is a subgraph of a planar $st$-graph. The proof of the theorem reveals that the method CONSTRUCT-LEVEL-EMBED correctly constructs a level planar embedding. We construct a proper level graph from $G_{st}$ in the proof and show that CONSTRUCT-LEVEL-EMBED performs correctly on this graph. This is technical and makes the proof more simple. However, a transformation of $G_{st}$ into a proper level graph would lead to quadratic running time. This will be avoided in the algorithm and the corollary belonging to this theorem confirms that CONSTRUCT-LEVEL-EMBED performs correctly on a (not necessarily proper) level $st$-graph.

**Theorem 5.1.** *Let $G$ be a level graph with $k \in \mathbb{N}$ levels and a leveling $\mathrm{lev}_G$. Then the following two statements are equivalent.*

1. *$G$ is level planar.*
2. *$G$ is a subgraph of a planar directed acyclic st-graph $G_{st} = (V_{st}, E_{st})$ with $V_{st} = V \uplus \{s, t\}$ and $E \subset E_{st}$, such that $\mathrm{lev}_G$ induces a topological numbering of $V_{st}$, by numbering $s$ with $0$ and $t$ with $k + 1$.*

*Proof.* Di Battista and Tamassia (1988) and Kelly (1987) show that a directed acyclic graph is upward planar if and only if it is a subgraph of a planar *st*-graph. Given an upward planar graph, Di Battista and Tamassia (1988) construct in their proof a *st*-graph by adding two extra vertices $s$ and $t$ and an incoming edge for every source of $G$ and an outgoing edge for every sink of $G$. Since a level planar graph $G$ is trivially an upward planar graph, the graph $G_{st}$ constructed this way is a planar *st*-graph with $V_{st} = V \uplus \{s, t\}$ and $E \subset E_{st}$. Furthermore, they obtain that any topological numbering of $G$ is as well a topological numbering of $G_{st}$ due to the (geometrical) construction of $G_{st}$.

Now let $G_{st} = (V_{st}, E_{st})$ be a planar *st*-graph such that $\mathrm{lev}_G$ implies a topological numbering of $G_{st}$. For every long edge $e = (v, w)$ in $G_{st}$ with $\mathrm{lev}_G(w) - \mathrm{lev}_G(v) = i > 1$ replace the edge by a path resulting in a proper level graph. The proper level graph is obviously a level planar *st*-graph. (As mentioned earlier, this transformation has no effect on the running time.)

According to Platt (1976), a planar embedding of $G_{st}$ with the edge $(s, t)$ on the boundary of the outer face induces an upward planar embedding of $G_{st}$. Let $\mathcal{E}_{st}$ be the upward planar embedding of $G_{st}$.

We show that the algorithm CONSTRUCT-LEVEL-EMBED when applied to the proper level graph $G_{st}$ computes a level planar embedding of $G$. Clearly, every vertex is accessed and therefore placed onto its level since the strategy merely applies a depth first search strategy, starting at the only sink of $G_{st}$. It remains to show that the level embedding constructed this way is level planar. Assume there exist a pair of edges $(u_1, v_1), (u_2, v_2) \in E$ with $lev(u_1) = lev(u_2) = j$ and $u_1 \leq_j u_2$ and $v_2 \leq_{j+1} v_1$. (Thus $v_2$ has been placed onto level $j + 1$ before $v_1$ and $u_1$ has been placed onto level $j$ before $u_2$.) Let $P_{v_1}$ be the path from $t$ to $v_1$ in the depth first search tree $T$, let $P_{v_2}$ be the path from $t$ to $v_2$ in $T$ and let $P_{u_1}$ be the path from $t$ to $u_1$ in $T$. See Fig. 5.1 for an illustration. Let $z_{v_1}$ be the vertex at which path $P_{v_1}$ leaves path $P_{v_2}$ and let $z_{u_1}$ be the vertex at which path $P_{u_1}$ leaves path $P_{v_2}$. We may assume without loss of generality that $\mathrm{lev}_G(z_{u_1}) \leq \mathrm{lev}_G(z_{v_1})$. Let $P'_{u_1}$ be the subpath of $P_{u_1}$ starting at vertex $z_{u_1}$. Let $P'_{v_1}$ be the subpath of $P_{v_1}$ starting at vertex $z_{v_1}$. Then the paths $P'_{u_1}$ and $P'_{v_1}$ are vertex disjoint, except for the vertex $z_{u_1}$ if $z_{u_1} = z_{v_1}$. Thus combining the paths $P'_{u_1}$ and $P'_{v_1}$ together with the path from $z_{v_1}$ to $z_{u_1}$ in the DFS-tree $T$ and the edge $(u_1, v_1)$ yields an undirected cycle $C$ in $G_{st}$. Since $G_{st}$ is an *st*-graph, there exists a directed path $P$ connecting $v_2$ and $s$ using $u_2$ and $P$ is vertex disjoint from the cycle $C$. Two cases may occur.

$z_{u_1} \neq v_2$ Then all vertices that dominate $v_2$ must lie in the interior of cycle $C$. Since $s$ lies in the exterior of $C$, the path $P$ must intersect cycle $C$. Thus the embedding of $G_{st}$ is not planar. This is a contradiction.

$z_{u_1} = v_2$ Then the edge $(u_1, v_2)$ is in $P'_{u_1}$ and therefore contained in the cycle $C$. Thus, all vertices that dominate $v_2$ except $u_1$ must be in the interior of cycle $C$. Again, the path $P$ must intersect cycle $C$. Thus the embedding of $G_{st}$ is not planar. This is a contradiction as well.

Reconstructing the original graph $G$ from $G_{st}$ by removing all subdivisions of edges and removing $s$ and $t$ yields a level planar embedding of $G$. □

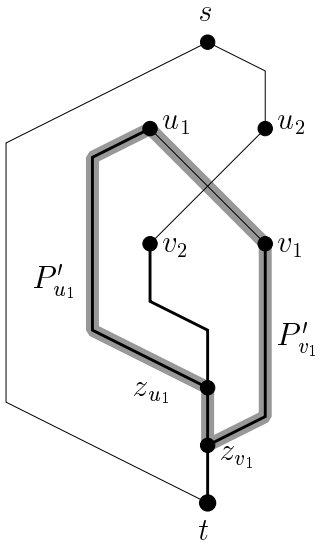



Figure 5.1: Illustration for the proof of Theorem 5.1. Paths on the depth first search tree are drawn fat. The cycle $C$ is drawn shaded.

The following corollary to the Theorem 5.1 shows that CONSTRUCT-LEVEL-EMBED computes a level planar embedding for a (not necessarily proper) level graph in linear time. The proof mainly shows that we can avoid transforming the $st$-graph $G_{st}$ into a proper level graph.

**Corollary 5.2.** *Let $G$ be a level planar graph with a leveling $\mathrm{lev}_G$, and $G_{st} = (V_{st}, E_{st})$ be a planar directed acyclic st-graph with $V_{st} = V \uplus \{s, t\}$ and $E \subset E_{st}$, such that*

*(i) every source in $G$ has exactly one incoming edge in $G_{st}$,*

*(ii) every sink in $G$ has exactly one outgoing edge in $G_{st}$, and*

*(iii) $\mathrm{lev}_G$ induces a topological numbering of $G_{st}$.*

*Then CONSTRUCT-LEVEL-EMBED computes a level planar embedding in $\mathcal{O}(n)$ time.*

*Proof.* Clearly, the algorithm terminates within linear time, since the algorithm executes the depth first search once and the number of extra edges $|E_{st} - E|$ is in $\mathcal{O}(n)$. Obviously, a long edge $e \in E_{st}$ is in the depth first tree if and only if all edges in the subdivision of $e$ are in the depth first search tree. Thus the correctness of the algorithm follows from the proof of Theorem 5.1. □

Using the planar embedding algorithm described in Chiba, Nishizeki, Abe, and Ozawa (1985), an upward planar embedding of an $st$-graph is easily computed. The nontrivial part is to construct an $st$-graph by adding edges to $G$ without destroying level planarity. First, two vertices $s$ and $t$ are added on extra levels. Then for every sink of $G$ an outgoing and every source of $G$ an incoming edge is added without destroying level planarity. Thus the embedding algorithm is sketched as follows

1. Set $G_{st} = G$.
2. Add an extra vertex $t$ on an extra level $k + 1$ and augment $G_{st}$ to a hierarchy by adding an outgoing edge to every sink of $G$ without destroying level planarity.
3. Add an extra vertex $s$ on an extra level 0 and augment $G_{st}$ to an $st$-graph by adding the edge $(s, t)$ and an incoming edge to every source of $G$ without destroying the level planarity.
4. Compute an upward planar embedding of $G_{st}$ using the algorithm presented by Chiba *et al.* (1985).
5. Construct a level planar embedding of $G$ from the planar embedding of $G_{st}$.

The difficult part is obviously to insert edges without destroying level planarity and is discussed in detail in the next section.

Figures 5.2, 5.3 and 5.4 demonstrate the strategy of constructing an $st$-graph from a level graph. We consider the level graph $G$ shown in Fig. 5.2 having 5 levels. (The same graph has been displayed already in Fig. 4.22 on Page 69.) The graph $G$ contains six sinks and five sources. The number of extra edges that we add in order to construct an $st$-graph is 12, one for every sink and every source, and the edge $(s, t)$.

Figure 5.3 shows the graph $G$ with an extra vertex $t$ on a level 6 and an extra vertex $s$ on a level 0. Adding the extra edges that are drawn as thick grey lines to $G$ constructs a hierarchy. The involved sinks in $G$ are drawn shaded.

In Fig. 5.4 the hierarchy of Fig. 5.3 is expanded to an $st$-graph by adding the edges that are drawn as thick grey lines (including the edge $(s, t)$). Here, the involved sources of $G$ are drawn shaded.

Once a level graph has been level planar embedded, we want to visualize it by producing a level planar drawing. This is very simple for proper graphs. Assign the vertices of every level integer $x$-coordinates according to the permutation that has been computed

by CONSTRUCT-LEVEL-EMBED, and draw the edges as straight line segments. This produces a level planar drawing and after applying some readjustments such a drawing can be aesthetically pleasing.

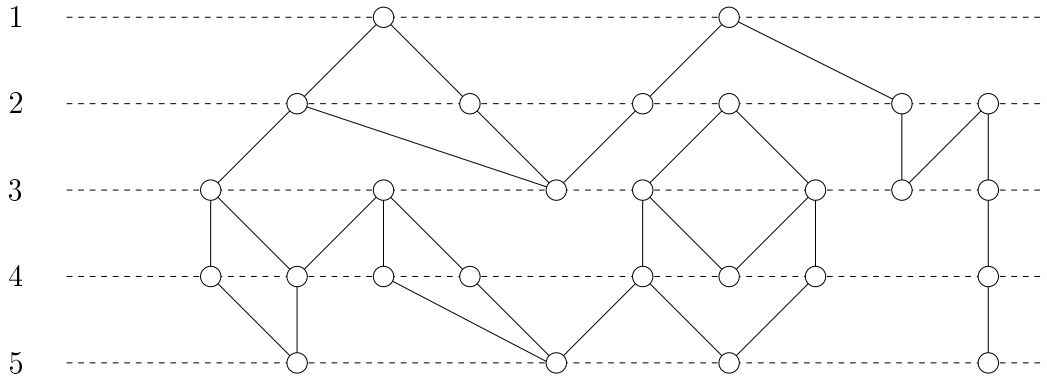



Figure 5.2: A level graph $G$.

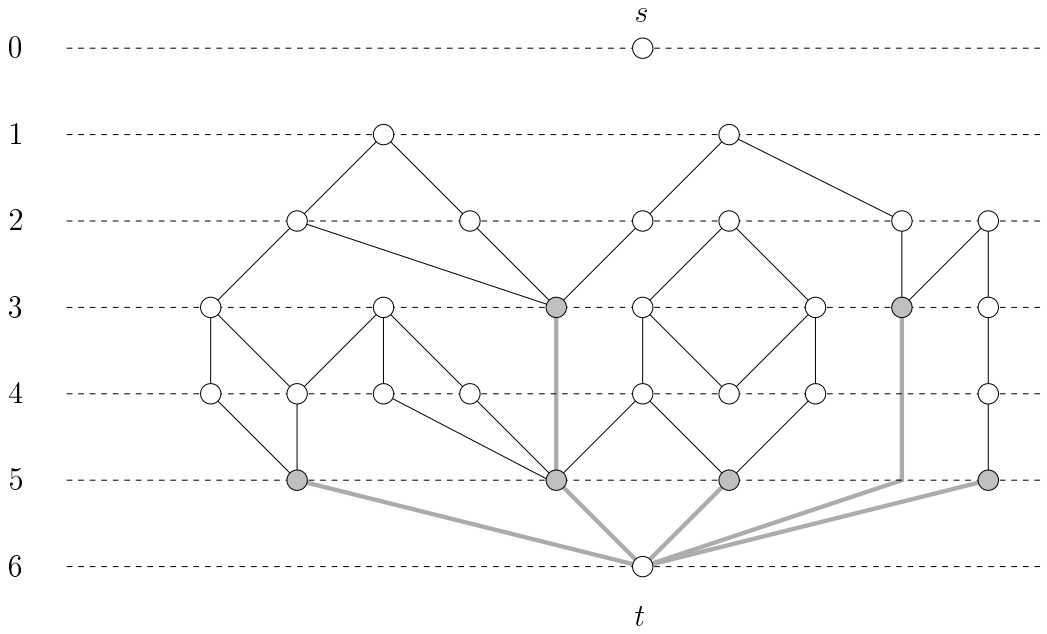



Figure 5.3: A hierarchy constructed from the level graph $G$ of Fig. 5.2.

For level graphs that are not necessarily proper, this approach is not applicable. It would be necessary to expand the level graph in horizontal direction for drawing the edges as straight line segments. If a lot of long edges exist in the graph, the area that is needed will be rather large, and the drawings are not aesthetically pleasing.

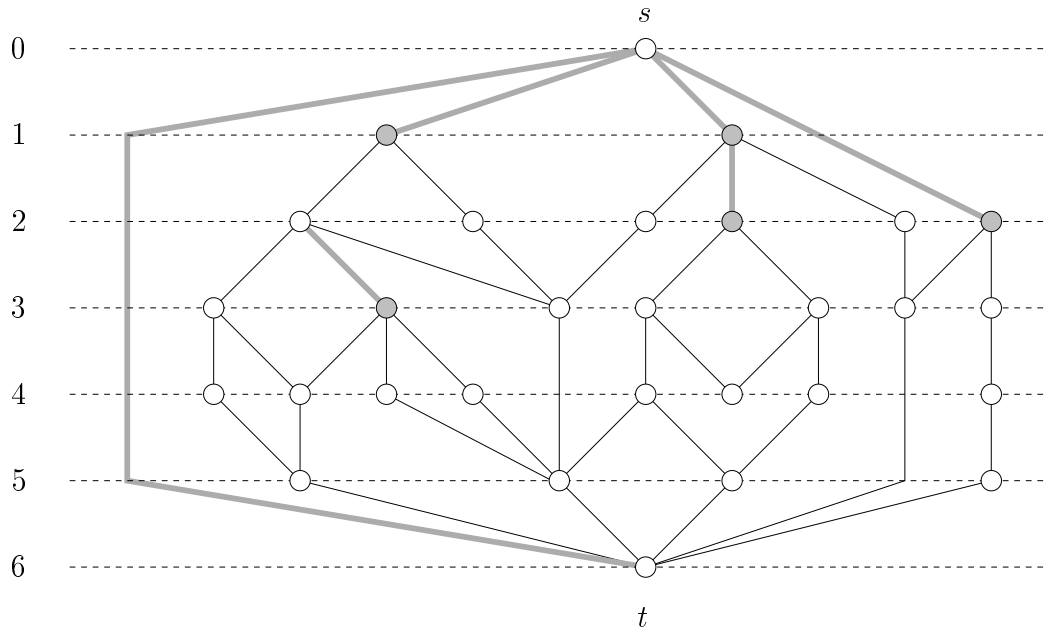



Figure 5.4: The *st*-graph constructed from the hierarchy of Fig. 5.3.

However, there is a nice and quick solution to it, using some extra information that is computed by our level planar embedding algorithm. Instead of drawing the graph $G$, we draw the *st*-graph $G_{st}$, and remove afterwards all edges and the vertices $s$ and $t$ that are not contained in $G$.

Drawing *st*-graphs has been extensively studied recently (see, e.g, Kant (1993), Luccio, Mazzone, and Wong (1987), Rosenstiehl and Tarjan (1986), Tamassia and Tollis (1986), and Tamassia and Tollis (1989)). Suitable approaches for drawing the *st*-graph $G_{st}$ have been presented by Di Battista and Tamassia (1988) and Di Battista, Tamassia, and Tollis (1992). These algorithms construct a planar upward polyline drawing of a planar *st*-graph according to a topological numbering of the vertices. The vertices of the *st*-graph are assigned to grid coordinates and the edges are drawn as polygonal chains. If we assign a topological numbering to the vertices according to their leveling, the algorithm presented by Di Battista and Tamassia (1988) produces in $\mathcal{O}(n)$ time a level planar polyline grid drawing of $G_{st}$ such that the number of edge bends is at most $6n - 12$ and every edge has at most two bends. This approach can be improved to produce in $\mathcal{O}(n)$ time a level planar polyline grid drawing of $G_{st}$ such that the drawing of $G_{st}$ has $\mathcal{O}(n^2)$ area, the number of edge bends is at most $(10n - 31)/3$, and every edge has at most two bends. Thus once we have augmented $G$ to the *st*-graph $G_{st}$, we can immediately produce a level planar drawing of $G$ in $\mathcal{O}(n)$ time.

## 5.2 Augmentation

Augmenting a level graph $G$ to an $st$-graph $G_{st}$ is divided into two phases. In the first phase an outgoing edge is added to every sink of $G$. Using the same algorithmic concept as in the first phase, an incoming edge is added to every source of $G$ in the second phase.

In order to add an outgoing edge for every sink of $G$ without destroying level planarity, we need to determine the position of a sink $v \in V^j$, $j \in \{1, 2, \ldots, k-1\}$, in the $PQ$-trees. This is done by inserting an indicator as a leaf into the $PQ$-trees. The indicator is ignored throughout the application of the level planarity test and will be removed either with the leaves corresponding to the incoming edges of some vertex $w \in V^l$, $l > j$, or it can be found in the final $PQ$-tree.

### 5.2.1 Sink Indicators

The idea of the approach can be explained best by an example. Figure 5.5 shows a small part of a level graph with a sink $v \in V^j$ and the corresponding part of the $PQ$-tree. Since $v$ is a sink, the leaf corresponding to $v$ will be removed from the $PQ$-tree before testing the graph $G^{j+1}$ for level planarity. Instead of removing the leaf, the leaf is kept in the tree ignoring its presence from now on in the $PQ$-tree. Such a leaf for keeping the position of a sink $v$ in a $PQ$-tree is called a *sink indicator* and denoted by $\mathrm{si}(v)$.

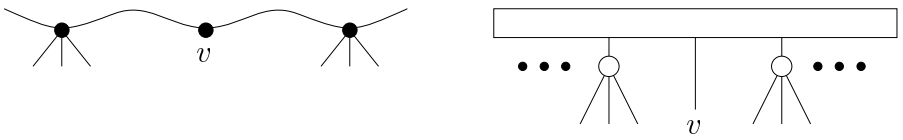



Figure 5.5: A sink $v$ in a level graph $G$ and the corresponding $PQ$-tree.

As shown in Fig. 5.6 the indicator of $v$ may appear within the sequence of leaves corresponding to incoming edges of a vertex $w \in V^l$. The indicator of $v$ is interpreted as a leaf corresponding to an edge $e = (v, w)$ and $G$ is augmented by $e$. Adding the edge $e$ to $G$ does not destroy the level planarity and provides an outgoing edge for the sink $v$.

When replacing a leaf corresponding to a sink by a sink indicator, a $P$- or $Q$-node $X$ may be constructed in the $PQ$-tree such that $\mathrm{frontier}(X)$ consists only of sink indicators. The presence of such a node is ignored in the $PQ$-tree as well. A node of a $PQ$-tree is an *ignored* node if and only if its frontier contains only sink indicators. By definition, a sink indicator is also an ignored node.

### 5.2.2 Sink Indicators in Template Reductions

In order to achieve linear time for the level planar embedder, we have to avoid searching for sink indicators that can be considered for augmentation. Consequently, only those

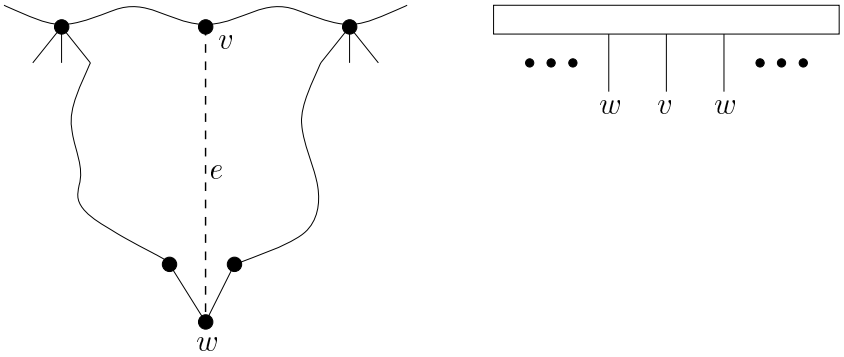



Figure 5.6: Adding an edge $e = (v, w)$ without destroying level planarity.

indicators $\mathrm{si}(v)$, $v \in V$, are considered for augmentation that appear within the pertinent subtree of a $PQ$-tree with respect to a vertex $w \in V$. We show that every edge added this way does not destroy level planarity. The first lemma considers sink indicators appearing within the sequence of pertinent leaves.

**Lemma 5.3.** *Let* $\mathrm{si}(v)$ *be a sink indicator of a vertex* $v \in V^j$, $1 < j < k$, *in a* $PQ$*-tree* $T$ *corresponding to an extended form* $H$. *Adding the edge* $e = (v, w)$ *to* $G$ *does not destroy level planarity if one of the following two conditions holds.*

*(i)* $\mathrm{si}(v)$ *is a descendant of a full node in the pertinent subtree with respect to a vertex* $w \in V^l$, $j < l \leq k$.

*(ii)* $\mathrm{si}(v)$ *is a descendant of a partial* $Q$*-node in the pertinent subtree with respect to a vertex* $w \in V^l$, $j < l \leq k$, *and* $\mathrm{si}(v)$ *appears within the pertinent sequence.*

*Proof.* Since $\mathrm{si}(v)$ is child of a full node or appears at least within a pertinent sequence of full nodes, adding the edge $e = (v, w)$ does not destroy level planarity of the reduced extended form $R$ corresponding to $H$. Thus it remains to show that adding the edge has no effect on merge operations.

For every embedding $\mathcal{E}$ of $R$, the edge $e$ is embedded either between two incoming edges of $w$ or next to the consecutive sequence of incoming edges of $w$. If $e$ is embedded between two incoming edges, the edge $e$ obviously does not affect the level planar embedding of any nonsingular form and $u$-singular form with $u \neq w$.

If $e$ is embedded next to the consecutive sequence of incoming edges of $w$, then $\mathrm{si}(v)$ must be a descendant of a full node $X$. If $X$ is a $P$-node, there exists an embedding of $R$ such that the edge $e$ can be embedded between two incoming edges of $w$. Thus adding the edge does not affect the level planar embedding of any nonsingular form and any $u$-singular form, with $u \neq w$.

Consider now a full $Q$-node $X$. By construction, $\mathrm{si}(v)$ is a descendant leaf at one end of $X$. The $Q$-node $X$ corresponds to a subgraph $B$. The vertex $v$ must be on the boundary of the outer face of the subgraph $B$ and there exists a path $P = (v = u_1, u_2, \ldots, u_\mu = w)$, $\mu \geq 2$,

on the boundary of the outer face of $B$ such that $\text{lev}(u_i) < l$ for all $i = 1, 2, \ldots, \mu - 1$. Thus none of the nodes $u_i$, $i = 1, 2, \ldots, \mu - 1$, is considered for a merge operation. Hence, replacing the path $P$ by an edge $(v, w)$ at the boundary of the outer face does not affect the level planar embedding of any nonsingular form and any $u$-singular form, with $u \neq w$. Figure 5.7 illustrates the insertion of an edge $e = (v, w)$ if $\text{si}(v)$ is the endmost child of a $Q$-node.

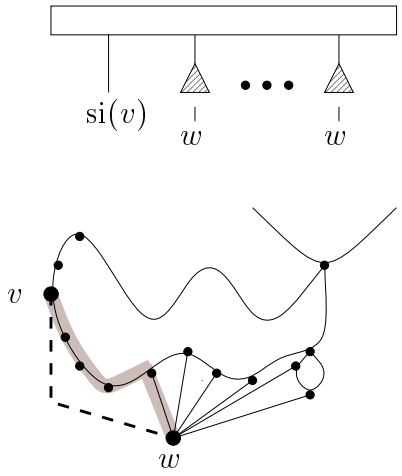



Figure 5.7: Sink indicator $\text{si}(v)$ is an endmost child of a $Q$-node. The path $P$ is drawn shaded, the edge $e = (v, w)$ is drawn as a dotted line.

Considering $w$-singular forms, we have that adding the edge $e$ produces one more face but the height of the largest $w$-cavity or the largest interior face remains valid. Thus a $w$-singular form that has to be embedded within an interior face or within a $w$-cavity can be embedded level planar after the insertion of $e$. □

Lemma 5.3 proves that an edge can be added to the graph without destroying level planarity if its sink indicator is found in the pertinent subtree. However, adding edges changes the structure of the graph and it is therefore not clear if for every sink indicator that is found in the pertinent subtree an edge can be added without destroying level planarity. However, Lemma 5.3 has been proven in a more general way immediately yielding the following corollary by applying an inductive argument.

**Corollary 5.4.** *Let $w \in V^{j+1}$, $1 < j < k$, be a virtual vertex in an extended form $H_i^j$, $1 \leq i \leq m_j$, and let $T$ be the corresponding PQ-tree. Let $S \subset V^1 \cup V^2 \cup \cdots \cup V^j$, be a set of sinks such that for every vertex $v \in S$ there exists a sink indicator $\text{si}(v)$, and one of the following two conditions holds.*

*(i)* $\text{si}(v)$ *is a descendant of a full node in the pertinent subtree of $T$ with respect to the vertex $w$.*

*(ii)* si($v$) *is a descendant of a partial* $Q$*-node in the pertinent subtree of* $T$ *with respect to the vertex* $w$*, and* si($v$) *appears within the pertinent sequence.*

*Then for every* $v \in S$ *the edge* $(v, w)$ *can be added to* $G$ *without destroying level planarity.*

*Proof.* By induction on the number of sinks in $T$, applying Lemma 5.3. □

Lemma 5.3 allows us to consider an edge for insertion if a sink indicator is a descendant of a full node or a descendant of a partial $Q$-node within the sequence of full children of the $Q$-node. The lemma does not consider a sink indicator si($v$) that appears as a child of a partial $Q$-node $X$ such that si($v$) is a sibling to the pertinent sequence. Although the following lemma shows that edges corresponding to sink indicators that are endmost children at the full end of a singly partial $Q$-node can be added without destroying level planarity, the case where sink indicators are between the sequence of full and the sequence of empty children reveals problems.

**Lemma 5.5.** *Let* si($v$) *be a sink indicator of a vertex* $v \in V^j$, $1 < j < k$, *in a* $PQ$*-tree* $T$ *and let* si($v$) *be a descendant of an ignored node* $X$ *that is a child of a singly partial* $Q$*-node* $Y$ *in the pertinent subtree with respect to a vertex* $w \in V^l$, $j < l \leq k$. *If* $X$ *appears at the full end of the singly partial* $Q$*-node, the edge* $e = (v, w)$ *can be added without destroying level planarity.*

*Proof.* Analogous to the proof of Lemma 5.3 for the case in which si($v$) is a descendant of a full $Q$-node. □

Consider now the situation of an extended reduced form $R$ as shown in Fig. 5.8. The sink indicator si($v$) is a child of a partial $Q$-node in the pertinent subtree of some vertex $w \in V^l$, $j < l \leq k$, and si($v$) is adjacent to a full and an empty node. Adding the edge $e$ does not a priori destroy level planarity in $R$, but it creates a new interior face, such that the large space between $w$ and the rightmost vertex of the subgraph corresponding to the subtree rooted at $X$ is destroyed. Now assume that a nonsingular form $R'$ has to be $w$-merged into $R$, applying merge operation D. Although the ML-value between the leaf $w$ and the node $X$ allows us to add the form $R'$ between $w$ and $X$, there is, due to the insertion of $e$, not enough space between $w$ and $X$. Hence a crossing is created and a nonlevel planar graph is constructed as is shown in Fig. 5.9. Consequently, a sink indicator that is found to be a sibling of a pertinent sequence and an empty sequence is never considered for edge augmentation.

By applying the results of Lemmas 5.3 and 5.5 during the template matching algorithm, not all sink indicators are considered for edge insertion. Some of the indicators remain in the final $PQ$-tree that represents all possible permutations of vertices of $V^k$ in the level planar embeddings of $G$. The following lemma allows us not only to insert edges $(w, t)$ for every $w \in V^k$ but also to insert an edge $(v, t)$ for every remaining sink indicator si($v$).

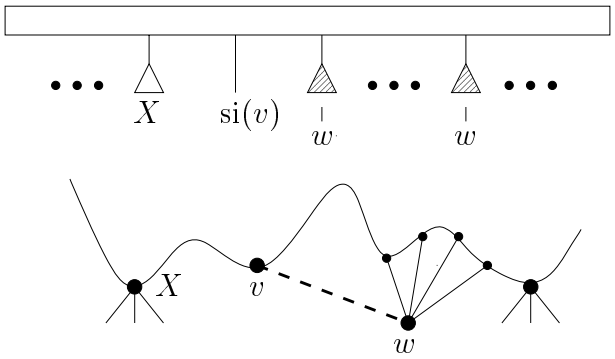



Figure 5.8: A doubly partial $Q$-node and its corresponding part of the form $R$. The new inserted edge $e = (v, w)$ is drawn as a dotted line.

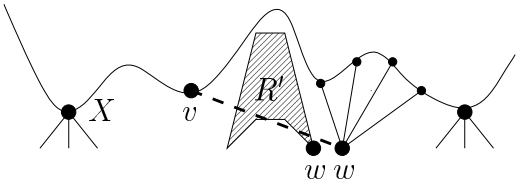



Figure 5.9: Merging $R'$ into $R$ with the new edge $e = (v, w)$ is not level planar.

**Lemma 5.6.** *Let* si$(v)$ *be a sink indicator of a vertex* $v \in V^j$, $1 < j < k$. *If* si$(v)$ *is in the final PQ-tree* $T$, *the edge* $e = (v, t)$ *can be added without destroying level planarity.*

*Proof.* Adding to every vertex $w \in V^k$ an edge $(w, t)$ does not affect the level planarity of the graph. Thus consider testing the level $V^{k+1}$ for level planarity. Obviously the pertinent subtree of $T$ is equal to $T$ and applying Corollary 5.4 proves Lemma 5.6. □

Since any leaf may eventually turn into a sink indicator, the introduction of the sink indicators has an effect on the templates of Booth and Lueker (1976). Although the general strategy is to ignore the sink indicators during the template matchings, there are quite a few problems that come up along with them. We discuss these problems in Chapter 6 and only mention one important feature now. Consider any of the templates P2, P3, ..., P6. These templates process a $P$-node $X$ with empty, full, and partial children. Obviously, all ignored nodes that are children of $X$ are allowed to be permuted into the pertinent subtree. Thus the sink indicators in the frontier of these ignored nodes are allowed to be considered for edge augmentation during the application of the template. However, the ignored children can only be considered if all children of $X$ are traversed in order to find the ignored children. This in turn implies that all empty children of $X$ have to be traversed as well. Since the template matching algorithm of Booth and Lueker (1976) does not process any of the empty children, the traversal of all children of $X$ has to be avoided for run

time reasons. Hence, not all of the children of $X$ are scanned. This is no drawback, since any of the involved sink indicators will be eventually in a pertinent subtree or in the final $PQ$-tree, where they finally will be considered for edge augmentation.

### 5.2.3 Sink Indicators in Merge Operations

While the treatment of sink indicators during the application of the template matching algorithm is rather easy in principle, this does not hold for merge operations. We consider all merge operations and discuss necessary adaptions in order to treat the sink indicators correctly.

If sink indicators and ignored nodes have to be manipulated correctly during the merge process, ML-values as they have been introduced for nonignored nodes have to be introduced for ignored nodes as well. Consider a node $X$ that becomes ignored. We make the following conventions.

(i) If $X$ is a child of a $P$-node $Y$, the corresponding ML-value for $X$ is $\mathrm{ML}(Y)$.

(ii) If $X$ is a child of a $Q$-node, we distinguish two cases:

(a) $X$ does not have an adjacent ignored sibling. Let $Z$ and $Y$ be its direct nonignored siblings. Then we leave the values $\mathrm{ML}(Z,X)$ and $\mathrm{ML}(X,Y)$ at $X$, and replace according to the level planarity test the values $\mathrm{ML}(Z,X)$ and $\mathrm{ML}(X,Y)$ by a new value $\mathrm{ML}(Z,Y) = \min\{\mathrm{ML}(Z,X), \mathrm{ML}(X,Y)\}$ at $Z$ and $Y$. The case where $X$ has just one nonignored sibling is solved analogously.

(b) $X$ has adjacent ignored direct siblings. Let $Z_I$ and $Y_I$ be the next ignored siblings and let $Z$ and $Y$ be its direct nonignored siblings with $Z$ at the side where $Z_I$ is, and $Y$ at the side where $Y_I$ is. Let $\mathrm{ML}(Z,X)$ and $\mathrm{ML}(X,Y)$ be the ML-values between $Z$ and $X$, and $X$ and $Y$, respectively. Let $\mathrm{ML}(Z_I,X)$ be the ML-value stored at $Z_I$, and let $\mathrm{ML}(X,Y_I)$ be the ML-value stored at $Y_I$. Then we replace at $X$ the values $\mathrm{ML}(Z,X)$ by $\mathrm{ML}(Z_I,X)$ and $\mathrm{ML}(X,Y)$ by $\mathrm{ML}(X,Y_I)$, and replace according to the level planarity test the values $\mathrm{ML}(Z,X)$ and $\mathrm{ML}(X,Y)$ by a new value $\mathrm{ML}(Z,Y) = \min\{\mathrm{ML}(Z,X), \mathrm{ML}(X,Y)\}$ at $Z$ and $Y$. The cases with only one nonignored or one ignored direct sibling are a handled analogously.

This strategy ensures that nonignored siblings $Z$ and $Y$ "know" the maximal height of the space between them, while the knowledge about the height of the space between the sinks and their corresponding indicators is left at the ignored nodes only.

**Lemma 5.7.** *Let $X$ be an ignored node that is a child of a $Q$-node and let $\mathrm{ML}_l$ and $\mathrm{ML}_r$ be the ML-values that have been assigned to $X$ by one of the rules (ii)(a) or (ii)(b) described above. Then the values $\mathrm{ML}_l$ and $\mathrm{ML}_r$ are valid for $X$.*

*Proof.* The sink indicators in $\mathrm{frontier}(X)$ can be interpreted as leaves corresponding to long edges. Thus the ML-values remain valid. □

**Lemma 5.8.** *Let $X$ be an ignored node that is a child of a $P$-node $Y$. Then the value* $\mathrm{ML}(Y)$ *is valid for $X$.*

*Proof.* Analogous to the proof of Lemma 5.7. □

Suppose now that we have two reduced forms $R_1$ and $R_2$ and their corresponding trees $T_1$ and $T_2$ with $\mathrm{LL}(T_1) \leq \mathrm{LL}(T_2)$ have to be $w$-merged. As described in 4.3.1, we start with the leaf labeled $w$ in $T_1$ and proceed upwards in $T_1$ until a node $X'$ and its parent $X$ are encountered such that one of the five merge conditions as described in Section 4.3.1 applies. The merge operations are discussed in an order according to the difficulties that are encountered when handling involved sink indicators. Before starting with the less problematic ones, one more convention is made. If $X$ is a node in a $PQ$-tree, $R_X$ denotes the subgraph corresponding to the subtree rooted at the node $X$.

### Merge Operation E

The tree $T_1$ is reconstructed by inserting a $Q$-node $X$ as new root of $T_1$ with two children $X'$ and the root of $T_2$. The following observation is trivial.

**Observation 5.9.** *There is no need to adapt the merge operation E in order to handle sink indicators correctly.*

### Merge Operation A

The root of $T_2$ is attached as a child to a $P$-node $X$ of $T_1$ thus we have that

$$\mathrm{ML}(X) < \mathrm{LL}(T_2) \ .$$

Obviously, all ignored nodes that are children of $X$ are allowed to be permuted in the pertinent subtree. Thus the sink indicators in their frontier are allowed to be considered for edge augmentation. However, as already mentioned for the templates P2, P3, ... , P6 in 5.2.2, the ignored children can only be considered if all children of $X$ are traversed in order to find the ignored children. This implies that all empty children of $X$ have to be traversed as well, yielding a quadratic time algorithm. Thus ignored children of $X$ are not considered for augmentation and we can make following observation.

**Observation 5.10.** *There is no need to adapt the merge operation A in order to handle sink indicators correctly.*

### Merge Operation D

Let $X$ be a $Q$-node of $T_1$ with ordered children $X_1, X_2, \ldots, X_\eta$, $\eta > 1$. Let $X' = X_\lambda$, $1 < \lambda < \eta$, and $\mathrm{ML}(X_{\lambda-1}, X_\lambda) < \mathrm{LL}(T_2) \leq \mathrm{ML}(X_\lambda, X_{\lambda+1})$. Thus $R_2$ has to be nested

between the subgraphs $R_{X_{\lambda-1}}$ and $R_{X_\lambda}$ and the root of $T_2$ is attached as a child to the $Q$-node $X$ between $X_{\lambda-1}$ and $X_\lambda$.

Let $I_1, I_2, \ldots, I_\mu$, $\mu \geq 0$, be the sequence of ignored nodes between $X_{\lambda-1}$ and $X_\lambda$ with $X_{\lambda-1}$ and $I_1$ being direct siblings, and $X_\lambda$ and $I_\mu$ being direct siblings. As illustrated in Fig. 5.10 there may exist a $\nu \in \{1, 2, \ldots, \mu\}$ such that for every sink indicator

$$\mathrm{si}(v) \in \bigcup_{i=\nu}^{\mu} \mathrm{frontier}(I_i) \ , \quad v \in \bigcup_{i=1}^{\mathrm{lev}(w)-1} V^i \ ,$$

the graph has to be augmented by an edge $e = (v, w)$. Adding these edges does not destroy level planarity. Furthermore, augmenting the graph $G$ for every $\mathrm{si}(v) \in \bigcup_{i=\nu}^{\mu} \mathrm{frontier}(I_i)$ by an edge $e' = (v, u)$, $u \in \bigcup_{i=\mathrm{lev}(w)}^{k} V^i$, $u \neq w$ destroys level planarity.

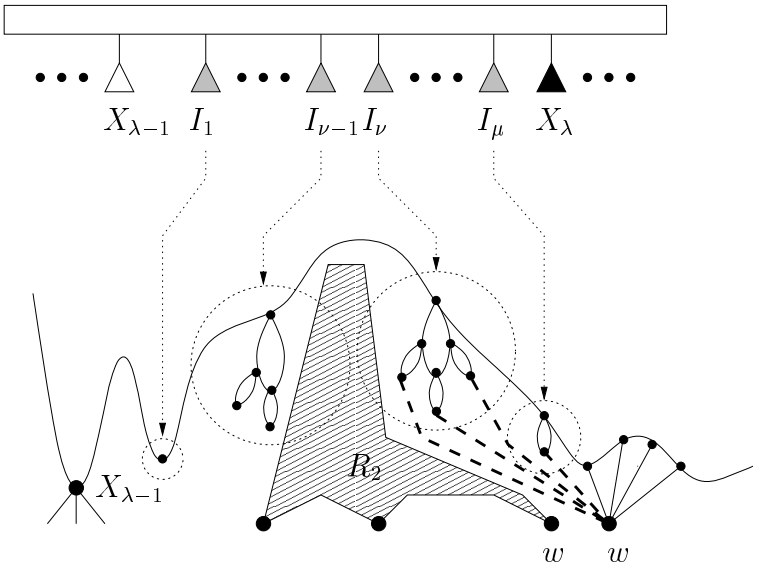



Figure 5.10: Merging the form $R_2$ into $R_1$ using the merge operation D forces us to augment $G$ by the edges drawn as dotted lines.

Using the following lemma we are able to find all the sink indicators that have to be considered for edge insertion when applying the merge operation D.

**Lemma 5.11.** *Let $X$ be a child of a $Q$-node and let $Y$ be a direct nonignored sibling of $X$. Let $I_1, I_2, \ldots, I_\mu$, $\mu \geq 0$, be the sequence of ignored nodes between $X$ and $Y$ with $X$ and $I_1$ being direct siblings, and $Y$ and $I_\mu$ being direct siblings. There exists a $\nu \in \{1, 2, \ldots, \mu+1\}$ such that $\mathrm{ML}(X, Y) = \mathrm{ML}(I_{\nu-1}, I_\nu)$, with $I_0 = X$ and $I_{\mu+1} = Y$.*

*Proof.* The lemma follows immediately from Lemma 5.7. □

For correct handling of the sink indicators while applying the merge operation D we scan the ignored nodes in reverse order $I_\mu, I_{\mu-1}, \ldots, I_1$, $\mu \geq 0$, until we detect a $\nu$, $1 \leq \nu \leq \mu+1$,

such that $\mathrm{ML}(I_{\nu-1}, I_\nu) < \mathrm{LL}(T_2)$, with $I_{\mu+1} = X_\lambda$ and $I_0 = X_{\lambda-1}$. Placing the root of $T_2$ between $I_{\nu-1}$ and $I_\nu$ makes the ignored nodes $I_\nu, I_{\nu+1}, \ldots, I_\mu$ appearing within the pertinent subtree. This allows to augment the graph $G$ by an edge $e = (v, w)$ for every sink indicator $\mathrm{si}(v) \in \bigcup_{i=\nu}^{\mu} \mathrm{frontier}(I_i)$ during the reduction with respect to $w$.

### Merge Operation C

Let $X$ be a $Q$-node with ordered children $X_1, X_2, \ldots, X_\eta$, $X' = X_\lambda$, $1 < \lambda < \eta$, and $\mathrm{ML}(X_{\lambda-1}, X_\lambda) < \mathrm{LL}(T_2)$ and $\mathrm{ML}(X_\lambda, X_{\lambda+1}) < \mathrm{LL}(T_2)$. The node $X_\lambda$ is replaced by a $Q$-node $Y$ with two children, $X_\lambda$ and the root of $T_2$.

Let $I_1, I_2, \ldots, I_\mu$, $\mu \geq 0$, be the sequence of ignored nodes between $X_{\lambda-1}$ and $X_\lambda$ with $X_{\lambda-1}$ and $I_1$ being direct siblings, and $X_\lambda$ and $I_\mu$ being direct siblings. Let $J_1, J_2, \ldots, J_\rho$, $\rho \geq 0$, be the sequence of ignored nodes between $X_\lambda$ and $X_{\lambda+1}$ with $X_\lambda$ and $J_1$ being direct siblings, and $X_{\lambda+1}$ and $J_\rho$ being direct siblings.

As illustrated in Fig. 5.11 there may exist a $\nu$, $1 \leq \nu \leq \mu$, such that for every sink indicator

$$\mathrm{si}(v) \in \bigcup_{i=\nu}^{\mu} \mathrm{frontier}(I_i) \ , \quad v \in \bigcup_{i=1}^{\mathrm{lev}(w)-1} V^i \ ,$$

$G$ has to be augmented by an edge $e = (v, w)$ if $R_2$ is embedded between $R_{X_{\lambda-1}}$ and $R_{X_\lambda}$.

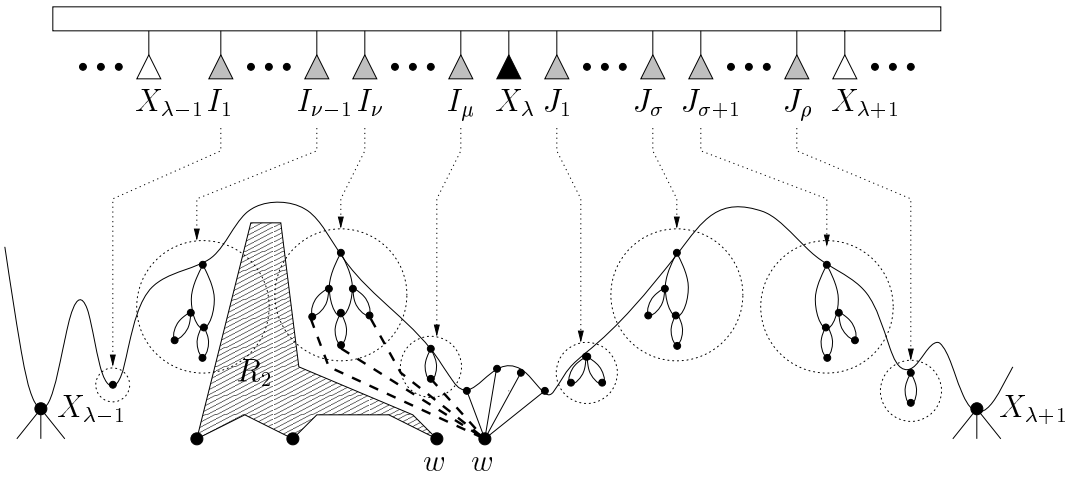



Figure 5.11: Merging the form $R_2$ into $R_1$ using the merge operation C and embedding it between $R_{X_{\lambda-1}}$ and $R_{X_\lambda}$ forces $G$ to be augmented by the edges drawn as dotted lines.

As is illustrated in Fig. 5.12, $R_2$ can be embedded between $R_{X_\lambda}$ and $R_{X_{\lambda+1}}$, and there may exist a $\sigma$, $1 \leq \sigma \leq \rho$, such that for every sink indicator

$$\mathrm{si}(v) \in \bigcup_{i=1}^{\sigma} \mathrm{frontier}(J_i) \ , \quad v \in \bigcup_{i=1}^{\mathrm{lev}(w)-1} V^i \ ,$$

$G$ has to be augmented by an edge $e = (v, w)$.

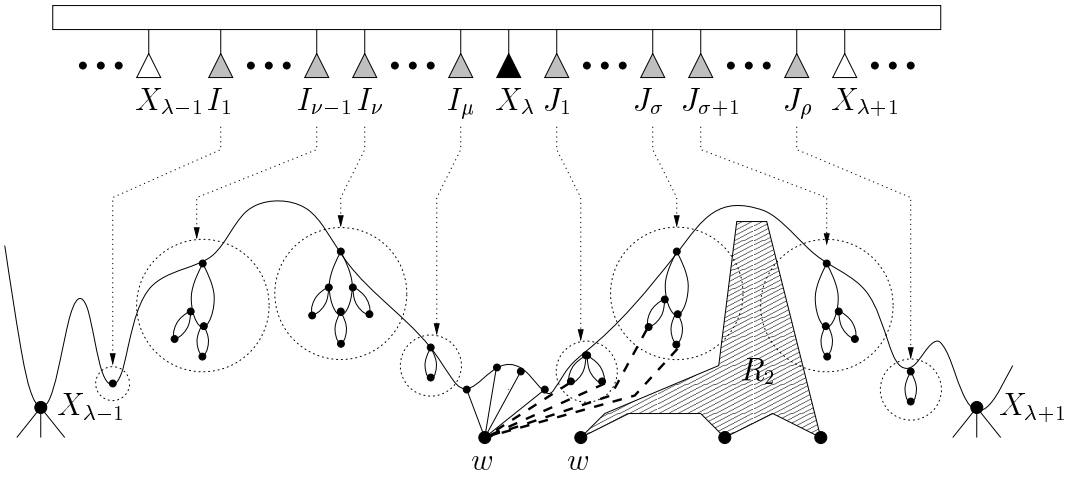



Figure 5.12: Merging the form $R_2$ into $R_1$ using the merge operation C and embedding it between $R_{X_\lambda}$ and $R_{X_{\lambda+1}}$ forces $G$ to be augmented by the edges drawn as dotted lines.

It is not possible to consider both sets of ignored nodes for edge augmentation. Consider for instance the example shown in Fig. 5.13, where edges for both sets $\bigcup_{i=\nu}^{\mu} \text{frontier}(I_i)$ and $\bigcup_{i=1}^{\sigma} \text{frontier}(J_i)$ have been added, yielding immediately a nonlevel planar graph.

However, deciding which set of sink indicators has to be considered for edge augmentation is not possible unless $X_\lambda$ is a full node. Proceeding the level planarity test down the levels $V^{\text{lev}(w)+1}$ to $V^k$ may embed the component $R_2$ on either of the two sides of $R_{X_\lambda}$. Since the side is unknown during the merge operation, we have to keep the affected sink indicators in mind. Furthermore, we must devise a method that allows to recognize the correct embedding during subsequent reductions.

The sequences $I_\nu, I_{\nu+1}, \ldots, I_\mu$ and $J_1, J_2, \ldots, J_\sigma$ are called the *reference sequence* of $R_2$ and denoted by $\text{rseq}(R_2)$. We refer to $I_\nu, I_{\nu+1}, \ldots, I_\mu$ as the *left reference sequence* of $R_2$ denoted by $\text{rseq}(R_2)^{left}$, and to $J_1, J_2, \ldots, J_\sigma$ as the *right reference sequence* denoted by $\text{rseq}(R_2)^{right}$. The union $\bigcup_{i=\nu}^{\mu} \text{frontier}(I_i) \cup \bigcup_{i=1}^{\sigma} \text{frontier}(J_i)$ is called the *reference set* of $R_2$ and denoted by $\text{ref}(R_2)$. The *left* and *right reference set* $\text{ref}(R_2)^{left}$ and $\text{ref}(R_2)^{right}$, respectively, are defined analogously to the left and right reference sequence.

The following lemma shows that considering both the left and the right reference set for augmentation is allowed if $X_\lambda$ is a full node.

**Lemma 5.12.** *Let $X$ be a Q-node of a PQ-tree $T_1$ with ordered children $X_1, X_2, \ldots, X_\eta$. Let $T_2$ be a PQ-tree that is w-merged at $X_\lambda$, $\lambda \in \{1, 2, \ldots, \eta\}$, and $X$ by the merge operation C. Let $\text{rseq}(R_2)^{left}$ be the reference sequence between $X_{\lambda-1}$ and $X_\lambda$, and let $\text{rseq}(R_2)^{right}$ be the reference sequence of $R_2$ between $X_\lambda$ and $X_{\lambda+1}$. Then $\text{ref}(R_2)^{left}$ and $\text{ref}(R_2)^{right}$ both can be considered for edge augmentation simultaneously if $X_\lambda$ is a full node.*

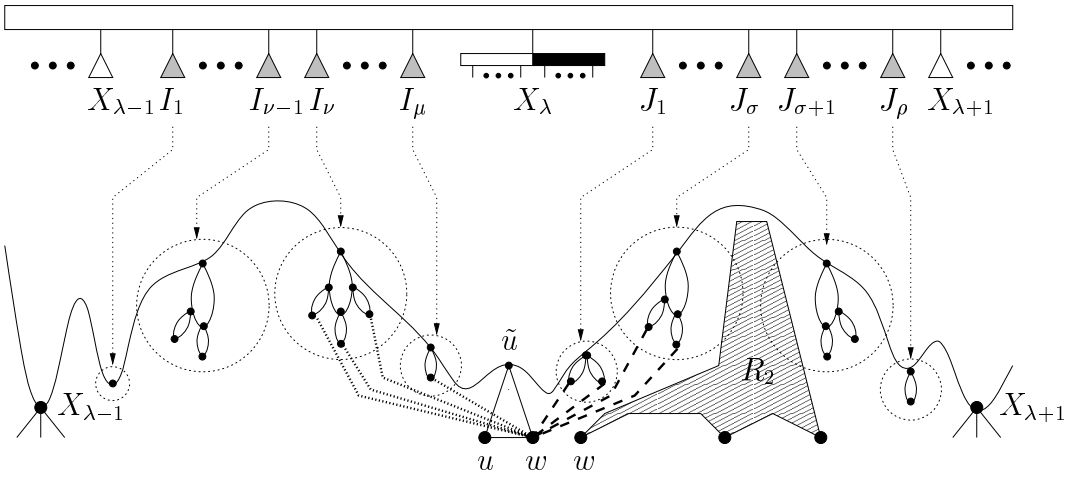



Figure 5.13: Merging the form $R_2$ into $R_1$ using the merge operation C does not allow to consider sinks on both sides of $R_{X_\lambda}$ for edge augmentation. Independently on the chosen embedding of $R_2$ there are always crossings between a path connecting $\tilde{u}$ and $u$ and the new edges.

*Proof.* Let $X_\lambda$ be a full node. Then

$$R_{X_\lambda} \cap V^{\text{lev}(w)} \;=\; \{w\}$$

and no long edge $e = (u, u')$ with tail $u \in R_{X_\lambda}$ traverses level $\text{lev}(w)$. Thus for every sink $v \in \bigcup_{i=1}^{\text{lev}(w)-1} V^i$ corresponding to a sink indicator $\text{si}(v) \in \text{ref}(R_2)$ we have that every path $P_v$ connecting $v$ and $w$ on the boundary of the outer face of $R_1$ that does not traverse $R_{X_{\lambda-1}}$ and $R_{X_{\lambda+1}}$ uses only vertices $\bigcup_{i=1}^{\text{lev}(w)-1} V^i$. Hence, an edge $(v, w)$ can be added for every sink $v$ without destroying level planarity of $R_1$. It follows that the form constructed from merging $R_1$ and $R_2$ is also level planar. Furthermore, $\text{rseq}(R_2)^{\mathit{left}}$ and $\text{rseq}(R_2)^{\mathit{right}}$ have been chosen such that the ML-values $\text{ML}(X_{\lambda-1}, X_\lambda)$ and $\text{ML}(X_\lambda, X_{\lambda+1})$ remain unaffected. Thus augmenting the graph $G$ by an edge $(v, w)$ for every $\text{si}(v) \in \text{ref}(R_2)$ has no effect on subsequent merge operations. □

The example in Fig. 5.11 and Fig. 5.12 is constructed such that the left and the right sets $\text{ref}(R_2)^{\mathit{left}}$ and $\text{ref}(R_2)^{\mathit{right}}$ both can be considered for edge augmentation.

In Section 5.2.4 a method using a special ignored indicator is developed for deciding which subset of $\text{ref}(R_2)$ has to be considered for edge augmentation. Before continuing with the algorithmic solution, we finish by considering the merge operation B where exactly the same problem occurs as has been encountered for the merge operation C.

### Merge Operation B

Let $X$ be a $Q$-node with ordered children $X_1, X_2, \ldots, X_\eta$, and let $X' = X_1$, and $\text{ML}(X_1, X_2) < \text{LL}(T_2)$. The node $X_1$ is replaced by a $Q$-node $Y$ having two children, $X_1$ and the root of $T_2$.

Let $I_1, I_2, \ldots, I_\mu$, $\mu \geq 0$, be the sequence of ignored nodes at one end of $X$ with $X_1$ and $I_\mu$ being direct siblings and $I_1$ being an endmost child of $X$. Let $J_1, J_2, \ldots, J_\rho$, $\rho \geq 0$, be the sequence of ignored nodes between $X_1$ and $X_2$ with $X_1$ and $J_1$ being direct siblings, and $X_2$ and $J_\rho$ being direct siblings.

Analogously to the merge operation C, there may exist sink indicators affected by merging $R_2$ into $R_1$ in both sets $I_1, I_2, \ldots, I_\mu$, $\mu \geq 0$, and $J_1, J_2, \ldots, J_\rho$, $\rho \geq 0$. Again it is not possible to decide if the left reference set $\mathrm{ref}(R_2)^{left}$ or the right reference set $\mathrm{ref}(R_2)^{right}$ has to be considered for edge augmentation.

### 5.2.4 Contacts

In order to solve the decision problem of the merge operations B and C, we examine how $R_2$ is fixed to either side of the vertex $w \in V$ in a level planar embedding of $G$. For the rest of this subsection we consider two $PQ$-trees $T_1$ and $T_2$, such that $T_2$ has been $w$-merged into $T_1$ using a merge operation B or C. Let $X$ be the $Q$-node with children $X_1, X_2, \ldots, X_\eta$, $\eta \geq 2$, and let $X_\lambda$, $\lambda \in \{1, 2, \ldots, \eta\}$, be its child that is replaced by a new $Q$-node having two children $X_\lambda$ and the root of $T_2$. Let $R_{X_\lambda}$ be the subgraph of $R_1$ corresponding to the subtree rooted at $X_\lambda$ before merging $R_1$ and $R_2$. Let $R_X$ be the subgraph corresponding to the subtree rooted at $X$ before merging $R_1$ and $R_2$.

**Definition 5.13.** *Define $\vec{R}_X$ to be the set of all vertices $u \in V$ such that there exists a vertex $v \in R_X$ and a (not necessarily directed) path $P$ connecting $u$ and $v$ not using the connective cut vertex of $X$. Define further $D(R_{X_\lambda} \cup R_2) \subset \bigcup_{i=\mathrm{lev}(w)}^{k} V^i$ to be the set of vertices $u \in \bigcup_{i=\mathrm{lev}(w)}^{k} V^i$ such that the following two conditions hold.*

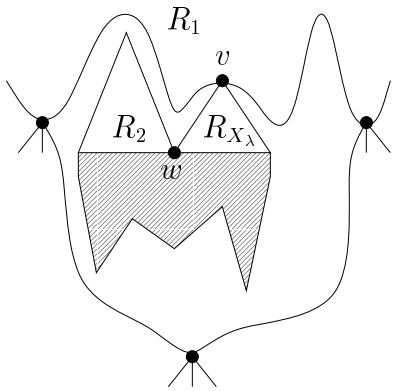
1. *There exists a directed path $P = (u_1, u_2, \ldots, u_\xi = u)$, $\xi > 1$, with $u_1 \in R_{X_\lambda} \cup R_2$.*
2. *There exists a vertex $\tilde{u} \in \bigcup_{i=\mathrm{lev}(w)}^{k} V^i$ and a directed path $\tilde{P} = (\tilde{u}_1, \tilde{u}_2, \ldots, \tilde{u}_\iota = \tilde{u})$, $\iota > 1$, with $\tilde{u}_1 \in R_{X_\lambda} \cup R_2$, such that $\mathrm{lev}(\tilde{u}) \geq \mathrm{lev}(u)$ and the paths $P$ and $\tilde{P}$ are vertex disjoint except for possibly $u$ and $\tilde{u}$.*

*The vertex set $D(R_{X_\lambda} \cup R_2)$ is called* dependent set *of $R_{X_\lambda} \cup R_2$.*

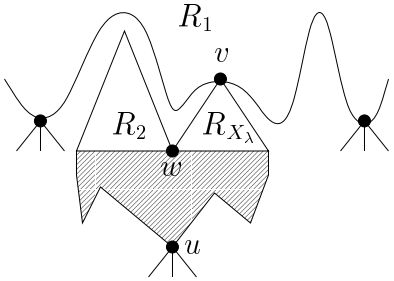
Figure 5.14 illustrates different kinds of dependent sets $D(R_{X_\lambda} \cup R_2)$. The dependent set $D(R_{X_\lambda} \cup R_2)$ is drawn shaded in all four cases. The vertex $v$ in all four subfigures denotes the connective cut vertex of $X_\lambda$ in $G^{\mathrm{lev}(w)}$ that allows to reverse the subgraph $R_{X_\lambda} \cup R_2$ with respect to $R_X$.

For simplicity, we make the overall assumption for the rest of this section that no vertex $u \in D(R_{X_\lambda} \cup R_2)$ is involved in a merge operation. This matter is discussed in the next section, handling concatenations of merge operations. However, subsequent merge operations to any other vertex not contained in $D(R_{X_\lambda} \cup R_2)$ are allowed after $w$-merging $R_2$ into $R_1$. This includes merge operations involving vertices of the subgraph corresponding to the tree $T_1$, except of course for the dependent set.

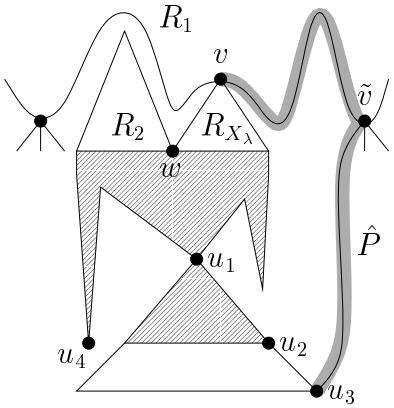
Figure 5.14(a) illustrates the case, where $v$ is not only a cut vertex in $G^{\text{lev}(w)}$ but a also a cut vertex in the graph $G$. Consequently, $R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2)$ will be embedded within an interior face or the outer face with the option to chose its embedding unaffected from the embedding of the rest of the graph. Hence $R_2$ may be embedded on an arbitrary side of $R_{X_\lambda}$ with respect to $R_X$.



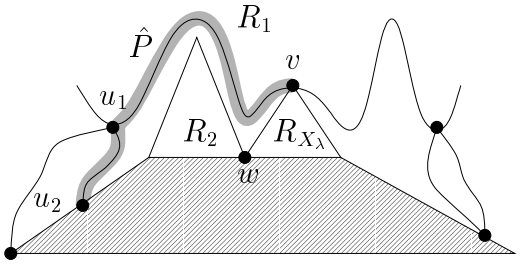

(a) Cut vertex $v$ allows free embedding of $R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2)$.




(b) Split pair $u$, $v$ allows a free embedding of $R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2)$.




(c) Split pair $u$, $v$ allows a free embedding of $R_2$.




(d) Fixed embedding of $R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2)$.

Figure 5.14: The figure illustrates different dependent sets $D(R_{X_\lambda} \cup R_2)$. The dependent sets are drawn shaded and path $\hat{P}$ is drawn grey.

Figure 5.14(b) illustrates the case, where $v$ is not a cut vertex in the graph $G$ but there exists a vertex $u \in D(R_{X_\lambda} \cup R_2)$ such that $u$ and $v$ form a split pair and we have that

$$\text{lev}(\tilde{u}) < \text{lev}(u) \text{ if } \tilde{u} \in D(R_{X_\lambda} \cup R_2) - \{u\} \ .$$

Thus $R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2)$ forms a split component and its embedding may be chosen freely. Hence $R_2$ may be embedded on an arbitrary side of $R_{X_\lambda}$ with respect to $R_X$.

Figure 5.14(c) illustrates a more delicate situation involving a split pair $v$ and $u_1$. According to the definition of the dependent set, the vertex $u_2$ is contained in $D(R_{X_\lambda} \cup R_2)$ since there exists a vertex $u_4$ with $\text{lev}(u_2) = \text{lev}(u_4)$ and two directed paths $P$ and $\tilde{P}$, with

(i) $P$ connecting a vertex of $R_{X_\lambda} \cup R_2$ and $u_2$,

(ii) $\tilde{P}$ connecting a vertex of $R_{X_\lambda} \cup R_2$ and $u_4$, and

(ii) $P$ and $\tilde{P}$ being disjoint.

Although $u_2 \in D(R_{X_\lambda} \cup R_2)$, the vertex $u_2$ is not contained in the split component of $v$ and $u_1$. The vertex $u_3$, however, does not belong to the dependent set $D(R_{X_\lambda} \cup R_2)$ since any directed path connecting a vertex of $R_{X_\lambda} \cup R_2$ and a vertex in $\bigcup_{i=\text{lev}(u_3)}^{k} V^i$ must contain the vertex $u_1$. Hence, the paths are not disjoint and we have that $u_3 \notin D(R_{X_\lambda} \cup R_2)$. Figure 5.14(c) shows a (not necessarily directed) path $\hat{P}$ connecting $v$ and $u_3$ via $\tilde{v}$, such that $\hat{P}$ and $R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2)$ are disjoint. This leads to the interesting fact that $u_3$ and therefore $u_2$ are fixed in their embedding to the side where $\tilde{v}$ is, while we are still able to flip the split component of $u_1$ and $v$ around, choosing an arbitrary side where to embed $R_2$ next to $R_{X_\lambda}$ with respect to $R_X$.

However, the existence of a split component does not guarantee a free choice of the embedding. In case that a (not necessarily directed) path $\tilde{P}$ exists, connecting the vertices $v$ and $u_2$ via $\tilde{v}$ such that the path $\tilde{P}$ and $R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2)$ are disjoint, and the path $\tilde{P}$ uses only vertices in $\bigcup_{i=1}^{\text{lev}(u_2)} V^i$, we cannot flip the split component of $v$ and $u_1$ anymore.

While Fig. 5.14(a),(b),(c) describe examples of dependent sets such that an embedding of $R_2$ can be freely chosen, Fig. 5.14(d) gives an example of a dependent set that has to be embedded such that $R_2$ is forced to be embedded on exactly one side of $R_{X_\lambda}$ with respect to $R_X$. Consider a vertex $u_1 \in V - (R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2))$ and a vertex $u_2 \in D(R_{X_\lambda} \cup R_2)$ such that there exists path $\hat{P}$ disjoint to $R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2)$, connecting $v$ and $u_2$ via $u_1$, and the path $\hat{P}$ uses only vertices in $\bigcup_{i=1}^{\text{lev}(u_2)} V^i$. If there exists a vertex $u_3 \in D(R_{X_\lambda} \cup R_2)$ with $\text{lev}(u_3) \geq \text{lev}(u_2)$, the path $\hat{P}$ forces $R_2$ to be embedded on one side of $R_{X_\lambda}$ with respect to $R_X$.

Figure 5.14 implicitly assumes that the $Q$-node $X$ remains a node with at least two nonignored children, one being the $Q$-node $Y$ (the node that has been introduced when merging $T(R_1)$ and $T(R_2)$). The example of Fig. 5.15 shows a subgraph corresponding to the subtree rooted at $X$, where $X$ has become a $Q$-node with only one nonignored child that is the

node $Y$. Thus, there exists a split pair $v$ and $\tilde{v}$ in $G$ with $\tilde{v}$ being the connective cut vertex of $R_X$ that allows reversing the split component containing $\vec{R}_X - (R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2))$. This implies that $R_2$ may be embedded on either side of $R_{X_\lambda}$ with respect to $R_X$. We note that a path $P = (v = u_1, u_2, \ldots, u_\mu = u)$, $\mu \geq 2$, may exist, connecting $v$ and a vertex $u \in D(R_{X_\lambda} \cup R_2)$ such that $P$ is disjoint to $D(R_{X_\lambda} \cup R_2)$, and the path $P$ uses only vertices in $\bigcup_{i=1}^{\text{lev}(u)} V^i$. Such a path has no effect on the embedding of $R_2$ next to $R_{X_\lambda}$ with respect to $R_X$ since $P$ must traverse the connective cut vertex $\tilde{v}$ of $R_X$. Figure 5.15 shows the path $P$ as a dotted line.

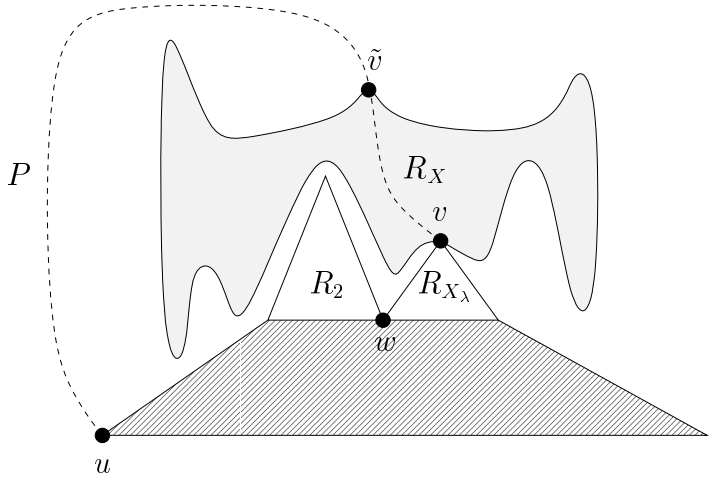



Figure 5.15: Graph corresponding to the situation where $X$ became a $Q$-node with one nonignored child. The embedding of $R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2)$ with respect to $R_X$ may be chosen freely.

However, if there exists a vertex $\tilde{u} \in \vec{R}_X - (R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2))$ such that for every vertex $u_i \in P$ the inequality $\text{lev}(u_i) \leq \text{lev}(\tilde{u})$ holds, the embedding of $R_2$ is fixed next to $R_{X_\lambda}$ with respect to $R_X$. Figure 5.16 shows such a situation that does not allow to flip the split component of $v$ and $\tilde{v}$ without creating a crossing. This case occurs if the node $Y$ becomes an endmost child of the $Q$-node $X$.

Obviously two possible scenarios may occur.

- The subgraph $R_2$ is fixed to one side of $R_{X_\lambda}$ with respect to $R_X$ in the final level planar embedding. The set of sink indicators that has to be considered for edge augmentation is predetermined but is unknown during the merge operation.
- The subgraph is not fixed to any side of $R_{X_\lambda}$ with respect to $R_X$ in the final level planar embedding. Thus we may chose arbitrarily either the left or the right reference set for edge augmentation. However, also this fact is unknown during the merge operation.

We now gather our observations on the coherence between the paths connecting $v$ and any vertex $u \in D(R_{X_\lambda} \cup R_2)$ and the embedding of $R_2$.

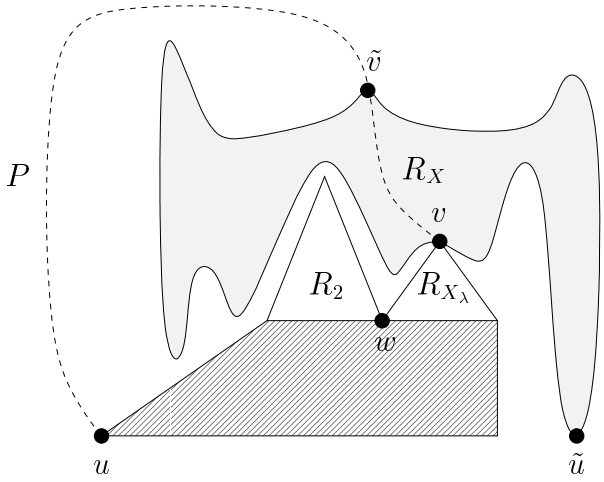



Figure 5.16: Graph corresponding to the situation where $X$ is a $Q$-node with $Y$ being an endmost child. If a path $P$ exists, the embedding of $R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2)$ with respect to $R_X$ is fixed.

**Observation 5.14.** *Let $v$ be the connective cut vertex of $R_{X_\lambda}$ and let $\tilde{v}$ be the connective cut vertex of $R_X$ if $X$ has a parent. The subgraph $R_2$ is not fixed to any side of $R_{X_\lambda}$ with respect to $R_X$ if and only if for every vertex $u$ in the dependent set $D(R_{X_\lambda} \cup R_2)$ and every undirected path $P = (v = u_1, u_2, \ldots, u_\mu = u)$, $\mu \geq 2$, with $u_i \in \bigcup_{i=1}^{\mathrm{lev}(u)} V^i$ for all $i = 1, 2, \ldots, \mu$, one of the following conditions holds.*

*(i)* $u_{\mu-1} \in R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2)$.

*(ii)* $\tilde{v} \in P$ *and for all* $v' \in \vec{R}_X - (R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2))$ *the inequality* $\mathrm{lev}(v') < \mathrm{lev}(u)$ *holds.*

*(iii)* $v$ *and* $u$ *form a split pair in* $G$ *and for all* $v' \in D(R_{X_\lambda} \cup R_2) - \{u\}$ *the inequality* $\mathrm{lev}(v') < \mathrm{lev}(u)$ *holds.*

**Observation 5.15.** *Let $v$ be the connective cut vertex of $R_{X_\lambda}$ and let $\tilde{v}$ be the connective cut vertex of $R_X$ if $X$ has a parent. The subgraph $R_2$ is fixed to a side of $R_{X_\lambda}$ with respect to $R_X$ if and only if there exists a vertex $u$ in the dependent set $D(R_{X_\lambda} \cup R_2)$ and an undirected path $P = (v = u_1, u_2, \ldots, u_\mu = u)$, $\mu \geq 2$, with $u_i \in \bigcup_{i=1}^{\mathrm{lev}(u)} V^i$ for all $i = 1, 2, \ldots, \mu$, and all of the following three conditions hold.*

*(i)* $u_{\mu-1} \notin R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2)$.

*(ii)* *(a)* $\tilde{v} \notin P$, *or*

*(b)* $\tilde{v} \in P$ *and there exists a* $v' \in \vec{R}_X - (R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2))$ *such that* $\mathrm{lev}(v') \geq \mathrm{lev}(u)$.

*(iii) There exists a vertex* $v' \in D(R_{X_\lambda} \cup R_2) - \{u\}$ *such that the inequality* $\text{lev}(v') \geq \text{lev}(u)$ *holds.*

The path $P$ connecting the vertex $v$ and a vertex $u \in D(R_{X_\lambda} \cup R_2)$ uses only level-$i$ vertices with $i \leq \text{lev}(u)$. This implies that the last edge $(u_{\mu-1}, u)$ on the path $P$ must be an incoming edge of $u$. We use this fact to determine to which side of $R_{X_\lambda}$ the form $R_2$ is fixed with respect to $R_X$. During the reduction of the leaves corresponding to the vertex $u$ we analyze the incoming edges of $u$, determining for each edge if it is the last edge of a path that is treated in one of the Observations 5.14 and 5.15. The following two lemmas help us to perform the case distinction in a very efficient way. We note that the parent of $Y$ ($Y$ is the $Q$-node that has been inserted by the merge operation) does not need to be the node $X$ throughout the algorithm, e.g., it may have been removed from the $PQ$-tree when applying a reduction using one of the templates Q2 and Q3.

**Lemma 5.16.** *The subgraph* $R_2$ *has to be fixed in its embedding at one side of* $R_{X_\lambda}$ *with respect to* $R_X$ *if and only if the* $Q$*-node* $Y$ *is removed from the tree* $T$ *during the application of the template matching algorithm using template Q2 or template Q3, and the parent of* $Y$ *did not become a node with* $Y$ *as the only nonignored child.*

*Proof.* Let $R_2$ be fixed to a side of $R_{X_\lambda}$ with respect to $R_X$. According to Observation 5.15, there exists a vertex $u$ in the dependent set $D(R_{X_\lambda} \cup R_2)$ and an undirected path $P = (v = u_1, u_2, \ldots, u_\mu = u)$, $\mu \geq 2$, with $u_i \in \bigcup_{i=1}^{\text{lev}(u)} V^i$ for all $i = 1, 2, \ldots, \mu$. The last edge $e = (u_{\mu-1}, u)$ on $P$ is therefore an incoming edge of $u$, and the following holds:

$$u_{\mu-1} \notin R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2) \ .$$

Since $u$ is in $D(R_{X_\lambda} \cup R_2)$, it must have a second incoming edge $\tilde{e}$, with $\tilde{e}$ being incident to a vertex $\tilde{u} \in \bigcup_{i=1}^{\text{lev}(u)-1} V^i \cap (R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2))$. Thus for the leaf $\tilde{l}$ in $T$ corresponding to $\tilde{e}$ it follows that

$$\tilde{l} \in \text{frontier}(Y) \ . \tag{5.1}$$

Furthermore, the condition 5.15(i) guarantees that for the leaf $l$ in $T$ corresponding to $e$ we have

$$l \notin \text{frontier}(Y) \ . \tag{5.2}$$

Let $Z$ be the smallest common ancestor of $l$ and $\tilde{l}$ in the $PQ$-tree. According to 5.1 and 5.2, the $Q$-node $Y$ is a descendant of $Z$ and we have $Y \neq Z$.

Let $\tilde{X}$ be the parent of $Y$. If condition 5.15(ii)(a) holds, then $l \in \text{frontier}(\tilde{X})$, and $\tilde{v}$ (the connective cut vertex of $R_X$) and $v$ (the connective cut vertex of $R_{X_\lambda}$) do not form a split pair in $G$. Thus the parent of $Y$ did not become a node with $Y$ as its only nonignored child.

If on the other hand condition 5.15(ii)(b) holds, then $l \notin \text{frontier}(\tilde{X})$, but there exists at least one empty child of $\tilde{X}$ containing a leaf in its frontier corresponding to a vertex $v' \in \vec{R}_X - (R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2))$. Thus again, the parent of $Y$ did not become a node with $Y$ as the only nonignored child.

The node $Y$ was a child of the $Q$-node $X$ when it was introduced into the $PQ$-tree. Since the parent of $Y$ did not become a node with $Y$ as the only nonignored child, we have according to Lemma 3.5 that $Y$ remains a child of a $Q$-node throughout the applications of the template matching algorithm. Due to the overall assumption that no vertex in $D(R_{X_\lambda} \cup R_2)$ is involved in another merge operation, $Y$ remains a child of a $Q$-node throughout every merge operation.

Due to condition 5.15(iii) there exists an empty leaf in the frontier of the node $Y$. Thus $Y$ is a partial node, and $Y$ and its parent $\tilde{X}$ are traversed during the reduction with respect to the vertex $u$. Since $\tilde{X}$ is a $Q$-node that is contained in the pertinent subtree with respect to $u$, either template Q2 or template Q3 is applied to $Y$ and $\tilde{X}$, removing $Y$ from the $PQ$-tree.

Now let $Y$ be removed from the tree during the reduction with respect to some vertex $u$ by applying template Q2 or Q3 and and let the parent of $Y$ never become a node with $Y$ being its only nonignored child.

Since the parent of $Y$ always has at least two children, condition 5.15(ii)(a) or (ii)(b) must hold. Furthermore, the application of template Q2 or Q3 implies that the template matching algorithm has traversed $Y$ and its parent, which is a $Q$-node as well. Hence the root of the pertinent subtree must be a proper ancestor of $Y$. Thus there exists a pertinent leaf $l$ not in the subtree of $Y$, and a path $P = (v = u_1, u_2, \ldots, u_\mu = u)$, $\mu \geq 2$, with $u_i \in \bigcup_{i=1}^{\text{lev}(u)} V^i$ for all $i = 1, 2, \ldots, \mu$, and the following holds:

$$u_{\mu-1} \notin R_{X_\lambda} \cup R_2 \cup D(R_{X_\lambda} \cup R_2) \ .$$

Since one of the templates Q2 and Q3 has been applied in order to remove $Y$ from the tree, $Y$ itself must have been partial, and therefore must have had at least one empty leaf in its frontier. Thus condition 5.15(iii) holds. It follows that $R_2$ is fixed on one side of $R_{X_\lambda}$ with respect to $R_X$. □

**Lemma 5.17.** *The subgraph $R_2$ is not fixed to any side of $R_{X_\lambda}$ with respect to $R_X$ if and only if one of the following cases occurs during the application of the template matching algorithm.*

*(i) The Q-node $Y$ gets ignored.*

*(ii) The Q-node $Y$ is not ignored and can be found in the final PQ-tree.*

*(iii) The Q-node $Y$ has only one nonignored child.*

*(iv) The parent of $Y$ has only $Y$ as a nonignored child.*

*Proof.* Let $R_2$ be a subgraph not fixed to any side of $R_{X_\lambda}$. According to Observation 5.14 the cases 5.14(i), 5.14(ii), or 5.14(iii) apply. If there exists a path $P$ in $G$ that satisfies condition 5.14(ii), it follows that the $Q$-node $X$ was transformed into a node with only one nonignored child and possibly some ignored children. Then the case (iv) follows immediately. If there exists a vertex $u \in D(R_{X_\lambda} \cup R_2)$ that satisfies 5.14(iii) then there exists a level $l$, $\text{lev}(w) < l \leq k$, ($w$ being the vertex involved in merging $R_{X_\lambda}$ and $R_2$) such that

$$D(R_{X_\lambda} \cup R_2) \cap \bigcap_{i=l}^{k} V^i \quad = \quad \emptyset$$

and

$$|D(R_{X_\lambda} \cup R_2) \cap V^{l-1}| \quad = \quad 1 \ .$$

Thus after completing the level planarity test for $G^{l-1}$ the node $Y$ is a $Q$-node with just one nonignored child.

Now assume that 5.14(i) holds for all paths in $G$ connecting $v$ and a vertex $u \in D(R_{X_\lambda} \cup R_2)$ and no path matches condition 5.14(ii) and 5.14(iii). It follows from 5.14(i) that 5.15(i) cannot hold. According to Lemma 5.16, the $Q$-node $Y$ is not removed from the tree using one of the templates Q2 and Q3. Therefore, one of the following two cases must hold.

1. There exists a level $l$, $\text{lev}(w) < l \leq k$, such that

$$D(R_{X_\lambda} \cup R_2) \cap \bigcap_{i=l}^{k} V^i \quad = \quad \emptyset$$

   and

$$|D(R_{X_\lambda} \cup R_2) \cap V^{l-1}| \quad \geq \quad 1 \ .$$

   Thus after completing the level planarity test for $G^{l-1}$ the node $Y$ is a $Q$-node with nonignored children. Two subcases occur

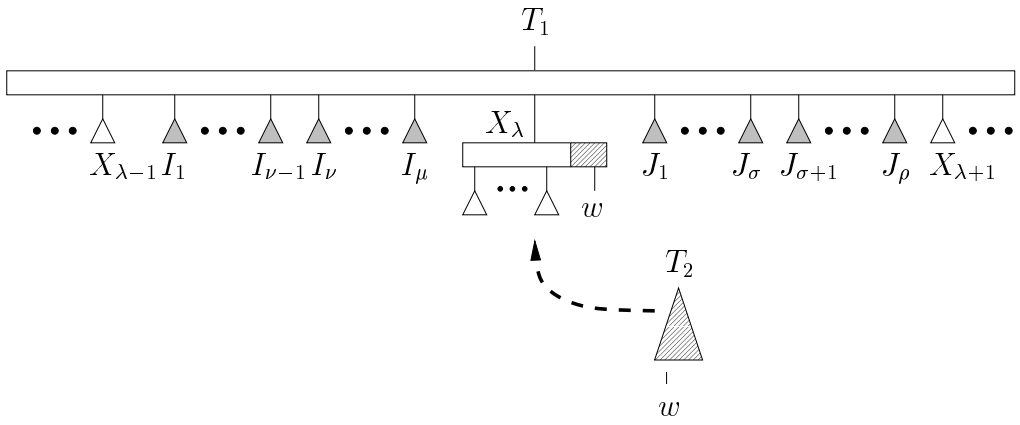
   (a) Every leaf in the frontier of the nonignored children of $Y$ is replaced by a sink indicator before testing $G^l$ for level planarity. It follows that case (i) applies.

   (b) All leaves in the frontier of $Y$ except for the leaves in the frontier of one child of $Y$ become ignored. Thus case (iii) applies

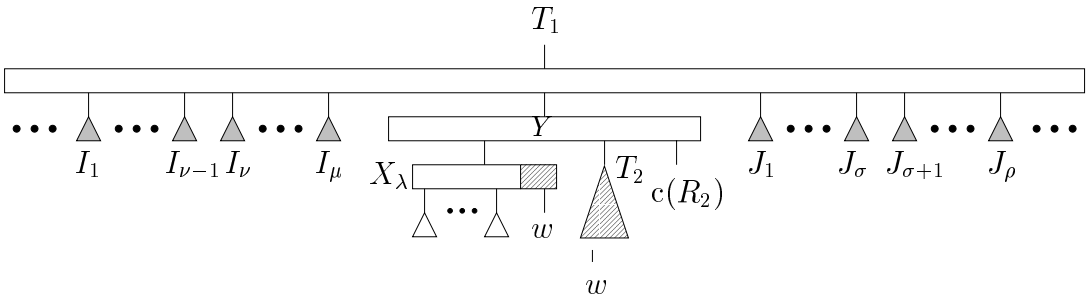
2. The node $Y$ is found in the final $PQ$-tree.

In reversion, if one of the four cases applies to the $Q$-node $Y$, we have by Observation 5.14 that any embedding may be chosen. □

Lemmas 5.16 and 5.17 reveal a solution for solving the problem of deciding whether $R_2$ is fixed to one side of $R_{X_\lambda}$ with respect to $R_X$. A strategy is developed for detecting on which side of $R_{X_\lambda}$ the subgraph $R_2$ has to be embedded. One endmost child of $Y$ clearly can be identified with the side where the root of $T_2$ has been placed, while the other endmost child of $Y$ can be identified with the side were $X_\lambda$ is. Every reversion of the $Q$-node $Y$ corresponds to changing the side were $R_2$ has to be embedded and all we need to do is to detect the side of $Y$ that belongs to $R_2$, when finally removing $Y$ from the tree applying one of the templates Q2 or Q3. The strategy is to mark the end of $Y$ belonging to $R_2$ with a special ignored node. Such a special ignored node is called a *contact* of $R_2$ and denoted by $\mathrm{c}(R_2)$. It is placed as endmost child of $Y$ during the merge operation B or C next to the root of $T_2$. Thus the $Q$-node $Y$ has now three children instead of two. See Fig. 5.17 for an illustration.




(a) $\mathrm{ML}(X_{\lambda-1}, X_\lambda) < \mathrm{LL}(T_2)$ and $\mathrm{ML}(X_\lambda, X_{\lambda+1}) < \mathrm{LL}(T_2)$.




(b) Contact $\mathrm{c}(R_2)$ is added as a child to $Y$ next to the root of $T_2$.

Figure 5.17: Adding a contact during the merge operation C.

Since the contact $\mathrm{c}(R_2)$ is related to a $w$-merge operation, the vertex $w$ is called *related vertex* of $\mathrm{c}(R_2)$ and denoted by $\omega(\mathrm{c}(R_2))$. The corresponding $w$-merge operation is said to

be *associated* with $c(R_2)$. Before gathering some observations about contacts, it is necessary to show that the involved ignored nodes remain in the relative position of $Y$ within the $Q$-node, and are therefore not moved or removed.

**Lemma 5.18.** *The ignored nodes of* $\text{rseq}(R_2)^{left}$ *and* $\text{rseq}(R_2)^{right}$ *stay siblings of* $Y$ *until one of the templates Q2 or Q3 is applied to* $Y$ *and its parent.*

*Proof.* The ignored nodes of $\text{rseq}(R_2)^{left}$ and $\text{rseq}(R_2)^{right}$ are children of a $Q$-node, and therefore remain children of a $Q$-node keeping their order throughout the application of the template matching algorithm, unless either $\text{rseq}(R_2)^{left}$ or $\text{rseq}(R_2)^{right}$ are found to be within a pertinent sequence. However, this can only happen if the node $Y$ becomes pertinent, provided that the node $Y$ does not become ignored itself. □

A contact has some special attributes that are immediately clear and very useful for our approach. In the following observations we again assume that $Y$ and its parent have not been an object of another merge operation B or C. Concatenation of contacts is discussed in the next subsection.

**Observation 5.19.** *Since the contact is an endmost child of a* $Q$*-node* $Y$*, it will remain an endmost child of the same* $Q$*-node* $Y$*, unless the node* $Y$ *is eliminated applying one of the templates Q2 or Q3.*

**Observation 5.20.** *If the node* $Y$ *is eliminated applying the templates Q2 or Q3, the contact* $c(R_2)$ *determines the side were* $R_2$ *has to be embedded next to* $R_{X_\lambda}$ *with respect to* $R_X$*. The contact is then a direct sibling to* $\text{rseq}(R_2)^i$*, for some* $i \in \{left, right\}$ *and* $\text{ref}(R_2)^i$ *has to be considered for edge augmentation.*

**Observation 5.21.** *If one of the four cases mentioned in 5.17 applies to* $Y$ *or its parent,* $R_2$ *can be embedded on any side* $R_{X_\lambda}$ *with respect to* $R_X$ *and therefore either* $\text{ref}(R_2)^{left}$ *or* $\text{ref}(R_2)^{right}$ *has to be considered for edge augmentation.*

Besides placing $c(R_2)$ as endmost child next to the root of $T_2$, $c(R_2)$ is equipped with a set of four pointers, denoting the beginning and the end of both the left and the right reference sequence of $R_2$. Let $I_1, I_2, \ldots, I_\mu$, $\mu \geq 0$, be the sequence of ignored nodes on the left side of $Y$ with $Y$ and $I_\mu$ being direct siblings, and let $J_1, J_2, \ldots, J_\rho$, $\rho \geq 0$, be the sequence of ignored nodes on the right side of $Y$ with $Y$ and $J_1$ being direct siblings. Let

$$\text{ref}(R_2) \;=\; \left(\bigcup_{i=\nu}^{\mu} \text{frontier}(I_i)\right) \;\cup\; \left(\bigcup_{i=1}^{\sigma} \text{frontier}(J_i)\right)$$
$$\text{for } 1 \leq \nu \leq \mu + 1,\; 0 \leq \sigma \leq \rho$$

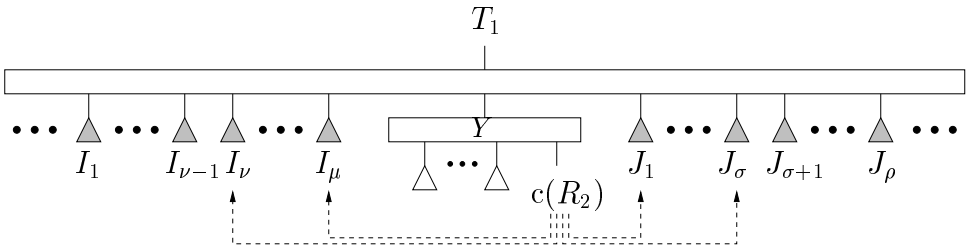
with

$$\left(\bigcup_{i=\nu}^{\mu} \text{frontier}(I_i)\right) = \emptyset \qquad \text{if } \nu = \mu + 1$$

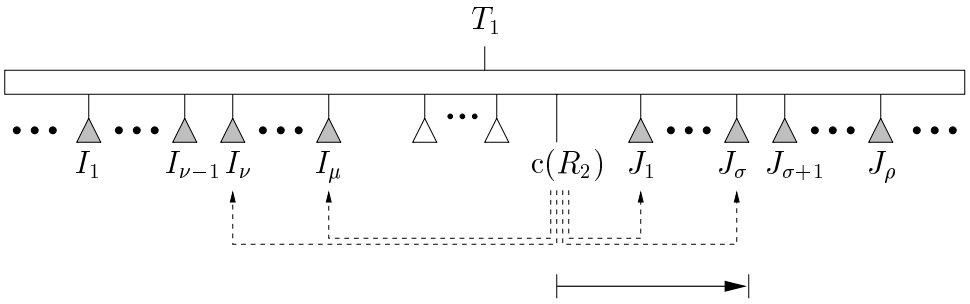
and

$$\left(\bigcup_{i=1}^{\sigma} \text{frontier}(J_i)\right) = \emptyset \qquad \text{if } \sigma = 0 \ .$$

After performing a reduction applying template Q2 or Q3 to the node $Y$, the contact is either a direct sibling of $I_\mu$ or a direct sibling of $J_1$. In the first case, we scan the sequence of ignored siblings starting at $I_\mu$ until the ignored node $I_\nu$ is detected. In the latter case, the sequence of ignored siblings is scanned by starting at $J_1$ until the node $J_\sigma$ is detected. Figure 5.18 illustrates this strategy for the latter case. Storing pointers of the ignored nodes $I_\nu, I_\mu, J_1, J_\sigma$ at $\text{c}(R_2)$, we are able to identify the reference set $\text{ref}(R_2)$. The nodes $I_\nu, I_\mu, J_1, J_\sigma$ are called the *reference points* of the contact $\text{c}(R_2)$. Analogously to the definition of a reference set of $R_2$, $\text{ref}(R_2)$ is said to be the reference set of $\text{c}(R_2)$ and denoted by $\text{ref}(\text{c}(R_2))$.




(a) Node $Y$ with contact $\text{c}(R_2)$ before the application of a template Q2 or Q3.




(b) Contact $\text{c}(R_2)$ is adjacent to the ignored node $J_1$. We chose $\text{ref}(R_2)^{right}$ for augmentation.

Figure 5.18: Identification of the reference set that has to be chosen for augmentation. The dotted lines denote the pointers of $\text{c}(R_2)$ to its reference points.

The section closes with a summary of the results.

**Lemma 5.22.** *Let* $\mathrm{c}(R_2)$ *be a contact related to a vertex* $w$ *and let* $\mathrm{ref}(R_2)^{left}$ *be the left reference set of* $\mathrm{c}(R_2)$ *with reference points* $I_\nu, I_\mu$ *and* $\mathrm{ref}(R_2)^{right}$ *be the right reference set of* $\mathrm{c}(R_2)$ *with reference points* $J_1, J_\sigma$*. Then the following statements are true.*

*(i) If* $\mathrm{c}(R_2)$ *is adjacent to* $I_\mu$*, then augmenting* $G_{st}$ *by an edge* $(u, w)$ *for every* $\mathrm{si}(u) \in \mathrm{ref}(R_2)^{left}$ *does not destroy level planarity.*

*(ii) If* $\mathrm{c}(R_2)$ *is adjacent to* $J_1$*, then augmenting* $G_{st}$ *by an edge* $(u, w)$ *for every* $\mathrm{si}(u) \in \mathrm{ref}(R_2)^{right}$ *does not destroy level planarity.*

*Proof.* The lemma immediately follows from Lemmas 5.16 and 5.18. □

**Remark 5.23.** *It is important to note that a contact and its reference set do not have an effect on the results in the Lemmas 5.3, 5.5, and 5.6. A contact is nonexisting for these lemmas and either the left or the right reference set of the contact can be regarded as nonexisting as well, while the other set stays in the PQ-tree. If a reference set and therefore also its contact is contained within a pertinent sequence, we are always able to decide by the application of the template matching algorithm which part of the reference set is in fact not contained in the PQ-tree.*

### 5.2.5 Concatenation of Contacts

For clarity, the previous section omitted the concatenation of merge operations applied to the vertices of the dependent set corresponding to a merge operation B or C. This section deals with the subject of concatenating merge operations.

Let $R_1$ be a reduced extended form, that has been $w_1$-merged into a reduced extended form $R$ applying a merge operation B or C. Let $T$ and $T_1$ be the $PQ$-trees corresponding to $R$ and $R_1$. Let $X$ be the $Q$-node with children $X_1, X_2, \ldots, X_\eta$, $\eta \geq 2$, and let $X_\lambda$, $\lambda \in \{1, 2, \ldots, \eta\}$, be the child that is replaced by a new $Q$-node having two children $X_\lambda$ and the root of $T_1$. Let $R_i$, $i = 2, 3, \ldots, \mu$, be reduced extended forms where every $R_i$ has to be $w_i$-merged into $R$, and $R_i$ is $w_i$-merged into $R$ before $R_{i+1}$ is $w_{i+1}$-merged into $R$, for all $i = 2, 3, \ldots, \mu - 1$.

**Definition 5.24.** *Let* $D(R_{X_\lambda} \cup R_1) \subset \bigcup_{\nu=\mathrm{lev}(w_1)}^{k} V^\nu$ *be the dependent set of* $R_{X_\lambda} \cup R_1$*. The dependent set of* $R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_i$*,* $i \in \{2, 3, \ldots, \mu\}$*, is denoted by* $D(R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_i) \subset \bigcup_{\nu=\mathrm{lev}(w_1)}^{k} V^\nu$*, and is recursively defined to be the set of all vertices* $u \in \bigcup_{\nu=\mathrm{lev}(w_1)}^{k} V^\nu$ *such that the following conditions hold.*

1. $w_\nu \in D(R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_{i-1})$.
2. *There exists a directed path* $P = (u_1, u_2, \ldots, u_\xi = u)$*,* $\xi > 1$*, with* $u_1 \in R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_i$.

3. *There exists a vertex* $\tilde{u} \in \bigcup_{\nu=\mathrm{lev}(w_1)}^{k} V^{\nu}$ *and a directed path* $\tilde{P} = (\tilde{u}_1, \tilde{u}_2, \ldots, \tilde{u}_\iota = \tilde{u})$, $\iota > 1$, *with* $\tilde{u}_1 \in R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_i$, *such that* $\mathrm{lev}(\tilde{u}) \geq \mathrm{lev}(u)$ *and the paths* $P$ *and* $\tilde{P}$ *are vertex disjoint except possibly for* $u$ *and* $\tilde{u}$.

**Definition 5.25.** *A sequence of* $w_i$*-merge operations,* $i = 1, 2, \ldots \mu$, *of reduced extended forms* $R_i$ *into a reduced extended form* $R$ *is said to be a* concatenation *of merge operations if the following three conditions hold.*

*(i)* $R_1$ *is* $w_1$*-merged using a merge operation B or C.*

*(ii) For all* $w_i$*-merge operations we have* $w_i \in D(R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_{i-1})$.

*(iii)* $R_1$ *has not been fixed to one side of* $R_{X_\lambda}$ *with respect to* $R_X$ *and it is unknown if its embedding can be chosen freely.*

Interestingly, a concatenation of merge operations does not really affect the results of Observations 5.14 and 5.15 and the Lemmas 5.16 and 5.17. This is immediately clear for the observations that we now give for the concatenated case.

**Observation 5.26.** *Let* $v$ *be the connective cut vertex of* $R_{X_\lambda}$ *and let* $\tilde{v}$ *be the connective cut vertex of* $R_X$ *if* $X$ *has a parent. Let* $R_i$, $i = 1, 2, \ldots, \mu$, *be a sequence of reduced extended forms that are* $w_i$*-merged into* $R$ *and their merge operations are concatenations. The subgraph* $R_1$ *is not fixed to any side of* $R_{X_\lambda}$ *with respect to* $R_X$ *if and only if for every vertex* $u$ *in the dependent set* $D(R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_\mu)$ *and every undirected path* $P = (v = u_1, u_2, \ldots, u_\xi = u)$, $\xi \geq 2$, *with* $u_i \in \bigcup_{\nu=1}^{\mathrm{lev}(u)} V^{\nu}$ *for all* $i = 1, 2, \ldots, \xi$, *one of the following conditions holds.*

*(i)* $u_{\xi-1} \in R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_\mu \cup D(R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_\mu)$.

*(ii)* $\tilde{v} \in P$ *and for all* $v' \in \vec{R}_X - (R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_\mu \cup D(R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_\mu))$ *the inequality* $\mathrm{lev}(v') < \mathrm{lev}(u)$ *holds.*

*(iii)* $v$ *and* $u$ *form a split pair in* $G$ *and for all* $v' \in D(R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_\mu) - \{u\}$ *the inequality* $\mathrm{lev}(v') < \mathrm{lev}(u)$ *holds.*

**Observation 5.27.** *Let* $v$ *be the connective cut vertex of* $R_{X_\lambda}$ *and let* $\tilde{v}$ *be the connective cut vertex of* $R_X$ *if* $X$ *has a parent. Let* $R_i$, $i = 1, 2, \ldots, \mu$, *be a sequence of reduced extended forms that are* $w_i$*-merged into* $R$ *and their merge operations are concatenations. The subgraph* $R_1$ *is fixed to a side of* $R_{X_\lambda}$ *with respect to* $R_X$ *if and only if there exists a vertex* $u$ *in the dependent set* $D(R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_\mu)$ *and an undirected path* $P = (v = u_1, u_2, \ldots, u_\xi = u)$, $\xi \geq 2$, *with* $u_i \in \bigcup_{\nu=1}^{\mathrm{lev}(u)} V^{\nu}$ *for all* $i = 1, 2, \ldots, \xi$, *and all three of the following conditions hold.*

*(i)* $u_{\xi-1} \notin R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_\mu \cup D(R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_\mu)$.

*(ii) (a)* $\tilde{v} \notin P$, *or*

*(b)* $\tilde{v} \in P$ *and there exists a* $v' \in \vec{R}_X - (R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_\mu \cup D(R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_\mu))$ *such that* $\text{lev}(v') \geq \text{lev}(u)$.

*(iii) There exists a vertex* $v' \in D(R_{X_\lambda} \cup R_1 \cup R_2 \cup \cdots \cup R_\mu) - \{u\}$ *such that the inequality* $\text{lev}(v') \geq \text{lev}(u)$ *holds.*

In order to see that the results of Lemmas 5.16 and 5.17 (up to minor differences) still hold, we show that the "local structure" of the $PQ$-tree at the $Q$-node $Y$ and its parent $X$ either does not change or, if it changes, the embedding of $R_1$ is fixed on one side of $R_{X_\lambda}$ with respect to $R_X$. With an "unchanged local structure" we express (informally) that throughout concatenated merge operations the node $Y$ (or any node that replaces $Y$), and $X$ (or any node that replaces $X$) stay $Q$-nodes with $Y$ (or its replacing node) remaining unchanged in the position of its siblings. The following lemma formally describes how the $Q$-node $Y$ is changed during subsequent concatenated merge operations. The results of the lemma then immediately lead to results similar to the ones in Lemmas 5.16 and 5.17.

**Lemma 5.28.** *Let* $Y$ *be a* $Q$*-node that has been introduced by* $w_1$*-merging the* $PQ$*-tree* $T_1$ *into* $T$ *using a merge operation B or C, replacing a child* $X_\lambda$ *of a* $Q$*-node* $X$*. Let* $T_i$*,* $i = 2, 3, \ldots, \mu$*,* $\mu \geq 2$*, be a sequence of* $PQ$*-trees that are* $w_i$*-merged into* $T$ *such that the* $w_i$*-merge operations are concatenations. Let* $Y'$ *be the node that occupies the position of* $Y$ *in the* $PQ$*-tree after the* $w_\mu$*-merge operation is complete. Then* $Y'$ *and its parent are* $Q$*-nodes.*

*Proof.* Let $R_i$, $i = 1, 2, \ldots, \mu$, be the forms corresponding to the $PQ$-trees $T_i$. We prove the lemma by induction.

According to the definition of a concatenation $R_1$ has not been embedded at one side of $R_{X_\lambda}$ with respect to $R_X$ and it is unknown if its embedding can be chosen freely. Lemma 5.17 therefore implies that the parent of $Y$ did not become a node with $Y$ as the only nonignored child, and according to Corollary 3.5 the parent of $Y$ must be a $Q$-node. Furthermore, Lemma 5.17 implies that $Y$ must have at least two leaves in its frontier both corresponding to different vertices in $G$.

When applying a $w_2$-merge operation to a vertex $w_2 \in D(R_{X_\lambda} \cup R_1)$ three cases are possible.

(i) Only descendants of $Y$ are affected by the merge operation.

(ii) The node $Y$ and its parent are affected by the merge operation.

(iii) Proper ancestors of $Y$ are affected by the merge operation.

Consider the first case. If only proper descendants of $Y$ are involved, neither $Y$ nor its parent are affected. If $Y$ and a child of $Y$ are affected, the merge operations B, C or D are applied to the child. Thus $Y$ remains a $Q$-node with unchanged position in the $PQ$-tree.

Consider the second case. Since the parent of $Y$ is a $Q$-node, the only allowed merge operations are B, C, and D. The operations B and C insert a new $Q$-node $Y'$ at the

position of $Y$. Reducing the leaves labeled $w_2$ after the merge operation does only affect the children of $Y'$, since $Y'$ is the root of the pertinent subtree. Therefore, the $Q$-node $Y'$ remains unchanged in its position. However, if the merge operation D is applied, the template matching algorithm performed directly after the merge operation removes $Y$ from the tree by applying one of the templates Q2 or Q3. Hence, according to Lemma 5.16, $R_1$ is embedded at one side of $R_{X_\lambda}$ with respect to $R_X$. Therefore, the $w_1$-merge operation of $T_1$ and the $w_2$-merge operation of $T_2$ are not a concatenation.

If proper ancestors of $Y$ are involved, the reduction of the $PQ$-tree with respect to $w_2$ removes $Y$ from the $PQ$-tree by applying one of the templates Q2 or Q3. Again, the $w_1$-merge operation of $T_1$ and the $w_2$-merge operation of $T_2$ are not concatenated.

The lemma then follows by a simple inductive argument. □

The following lemmas are almost identical to the Lemmas 5.16 and 5.17, taking into account that the subgraph induced by the subtree rooted at $Y$ (or any $Q$-node that replaces $Y$ due to a merge operation) may have grown by concatenated merge operations.

**Lemma 5.29.** *Let $Y$ be the $Q$-node that was introduced by a $w_1$-merge operation B or C of a $PQ$-tree $T_1$ into a tree $T$, replacing a node $X_\lambda$ that was a child of a $Q$-node $X$ in $T$. Let $Y'$ be a $Q$-node occupying the position of $Y$, and $Y'$ has been introduced during a merge operation concatenating the $w_1$-merge operation. The subgraph $R_1$ corresponding to $T_1$ has to be embedded at exactly one side of $R_{X_\lambda}$ with respect to $R_X$ if and only if the $Q$-node $Y'$ is removed from the tree $T$ during the application of the template matching algorithm using template Q2 or template Q3, and the parent of $Y$ did not become a node with $Y$ as the only nonignored child.*

*Proof.* The lemma follows from Lemma 5.16 and Lemma 5.28. □

**Lemma 5.30.** *Let $Y$ be the $Q$-node that was introduced by a $w_1$-merge operation B or C of a $PQ$-tree $T_1$ into a tree $T$, replacing a node $X_\lambda$ that was a child of a $Q$-node $X$ in $T$. Let $Y'$ be a $Q$-node occupying the position of $Y$, and $Y'$ has been introduced during a merge operation concatenating the $w_1$-merge operation. The subgraph $R_1$ corresponding to $T_1$ is not fixed to any side of $R_{X_\lambda}$ if and only if one of the following cases occurs during the application of the template matching algorithm.*

*(i) The $Q$-node $Y'$ gets ignored.*

*(ii) The $Q$-node $Y'$ is not ignored and can be found in the final $PQ$-tree.*

*(iii) The $Q$-node $Y'$ has only one nonignored child.*

*(iv) The parent of $Y'$ has only $Y'$ as a nonignored child.*

*Proof.* The lemma follows from Lemma 5.17 and Lemma 5.28. □

Let $c$ be a contact that is a child of the $Q$-node $Y$. As long as concatenated merge operations only affect descendants of the $Q$-node $Y$, they have no effect on $c$ and its reference sequence. However, if $Y$ and its parent are subject to a merge operation B or C, there exists a coherence between the existing contact $c$ and the new contact that is introduced by the merge operation.

Consider a node $X_\lambda$ and its parent $X$ that are subject to several merge operations. Due to Lemma 5.28, the merge operations A, D, and E can be performed one after another without worrying about the correct treatment of involved sink indicators. However, the merge operations B and C may "affect" each other. Consider for instance the example shown in Fig. 5.19, presenting three forms $R_1$,$R_2$, and $R_3$ that have been successively merged into a form $R$ at the vertices $w_1, w_2, w_3$. For every form $R_i$, the example also gives the set of edges that have to be added as incoming edges to $w_i$, $i \in \{1, 2, 3\}$, in the given embedding. On the other hand, Fig. 5.20 gives the same example only with a different embedding showing different sets of edges that have to be added as incoming edges to $w_i$. In particular, the edge set joining $w_1$ is now empty. The vertices $w_1, w_2, w_3$ do not necessarily have to be on the same level.

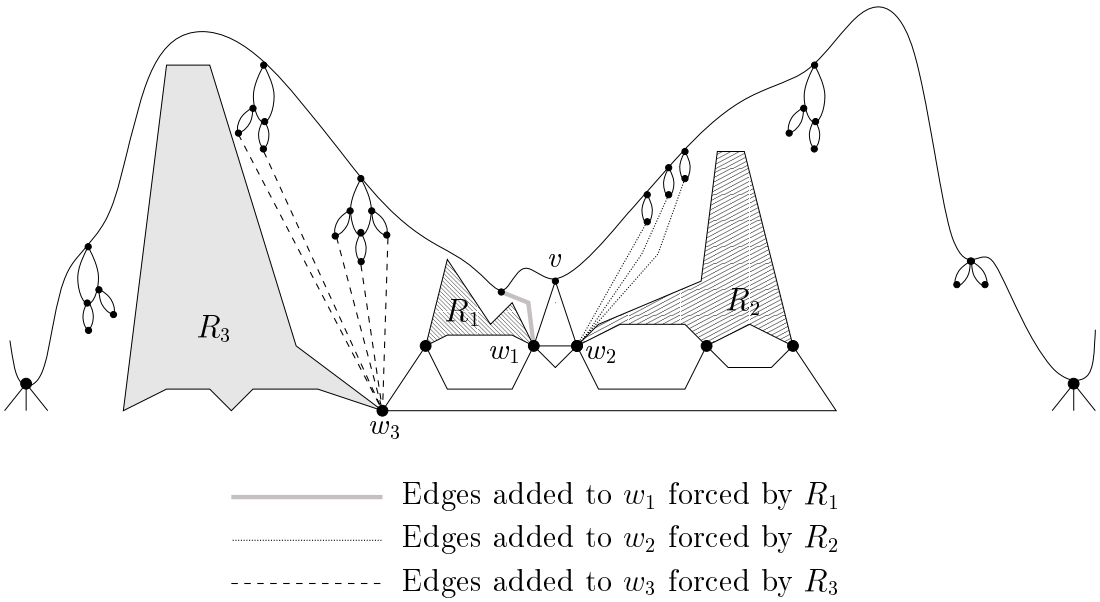



Figure 5.19: Concatenation of merge operations. First possible embedding of the forms $R_1$, $R_2$, and $R_3$ next to $R_{X_\lambda}$ with respect to $R_X$.

In the rest of this section we discuss how to handle sequences of the merge operations B and C that may affect each other. We say that two contacts $c_1$ and $c_2$ *mutually influence* each other if $\text{ref}(c_1) \cap \text{ref}(c_2) \neq \emptyset$. Two merge operations B or C *mutually influence* each other if their corresponding contacts mutually influence each other. Consider a $Q$-node $Y$ that has been introduced as a child of a $Q$-node $X$ applying one of the operations B or C. The following lemma shows when merge operations mutually influence each other.

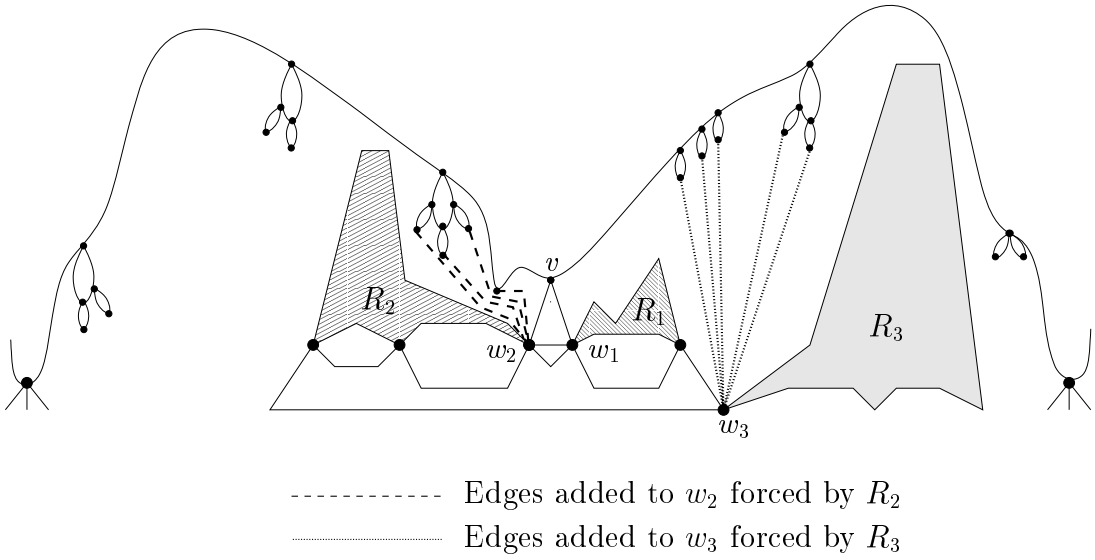



Figure 5.20: Concatenation of merge operations. Second possible embedding for the forms $R_1$, $R_2$, and $R_3$ next to $R_{X_\lambda}$ with respect to $R_X$.

**Lemma 5.31.** *Let $X$ be a Q-node and let $Y$ be a Q-node that is a child of $X$ such that $Y$ has been introduced by applying a merge operation B or C. Let $c_1$ be the contact that is associated with the merge operation. Let $c_2$ be another contact that is associated with a later merge operation B or C. Then $c_1$ and $c_2$ mutually influence each other if and only if the node $Y$ and its parent $X$ have been object of the merge operation associated with $c_2$.*

*Proof.* Let $Y$ and its parent $X$ be object of a merge operation associated with $c_2$. If $\mathrm{ref}(c_2) \neq \emptyset$ (otherwise, $c_2$ is not installed) we have that $\mathrm{ref}(c_1) \cap \mathrm{ref}(c_2) \neq \emptyset$. Therefore $c_1$ and $c_2$ mutually influence each other. (Notice that $\mathrm{ref}(c_1) \subseteq \mathrm{ref}(c_2)$ does not necessarily hold.)

To show the "only if" part it is convenient to prove that a contact associated with a merge operation that affects nodes other than $Y$ and $X$ does not influence $c_1$. Consider a merge operation B or C that is associated with descendants of $Y$. Then every node in $\mathrm{rseq}(c_2)$ is also a descendant of $Y$. Therefore, $c_1$ and $c_2$ do not influence each other.

Next, consider a merge operation that is associated with an ancestor $Z$ of $X$ and the parent $Z'$ of $Z$. Since $Z'$ is a proper ancestor of $X$, $c_1$ and $c_2$ do not refer to the same ignored nodes.

Finally, any other pair of a node $Z$ and its parent $Z'$ that are affected by a merge operation obviously does not have an effect on $Y$ and its contact $c_1$ unless $Z$ is a direct nonignored sibling of $Y$ and $Z' = X$. Let $I_1, I_2, \ldots, I_\mu$, $\mu \geq 0$, be the sequence of ignored nodes between $Z$ and $Y$ with $Z$ and $I_1$ being direct siblings, and $Y$ and $I_\mu$ being direct siblings. According to Lemma 5.11 there exists a $\nu \in \{1, 2, \ldots, \mu + 1\}$ such that $\mathrm{ML}(Z, X) = \mathrm{ML}(I_{\nu-1}, I_\nu)$, with $I_0 = Z$ and $I_{\mu+1} = Y$. Thus $I_{\nu-1}$ is not in $\mathrm{rseq}(c_1)$ and $I_\nu$ is not in $\mathrm{rseq}(c_2)$. It follows that $\mathrm{ref}(c_1) \cap \mathrm{ref}(c_2) = \emptyset$. □

**Remark 5.32.** *Let $X$ be a $Q$-node in a $PQ$-tree $T$ and let $Y$ be a $Q$-node that is a child of $X$ such that $Y$ has been introduced by applying a merge operation $B$ or $C$. The merge operations $A$ and $E$ obviously do not affect $Y$ and $X$ since the conditions are not satisfied. The merge operation $D$ does not affect $X$ and $Y$ either, since a $PQ$-tree $T'$ that is merged into $T$ involving the nodes $X$ and $Y$ either is placed between $Y$ and its left nonignored sibling, or between $Y$ and its right nonignored sibling. If* frontier$(Y)$ *contains one of the pertinent leaves that is considered for the merge operation, the application of the template matching algorithm will lead directly to the removal of $Y$ and therefore to the removal of its endmost contacts. If not, we have according to Lemma 5.11 no conflict concerning the involved ignored nodes.*

Lemma 5.31 allows an easy detection of the existence of contacts that are influenced by the introduction of a new contact. In case of a merge operation B or C, we only need to check if the node $Y$ that has to be replaced by a new $Q$-node does have a contact as an endmost child. However, the contact is then separated from its reference sequence since $Y$ is not a child of the $Q$-node $X$ anymore. This seems to destroy the strategy of handling the contact and its reference sequence correctly. The following lemma shows how to deal with this problem.

**Lemma 5.33.** *Let $X$ be a $Q$-node and let $Y$ be a $Q$-node that is a child of $X$ such that $Y$ has been introduced by applying a merge operation $B$ or $C$. Let $c_1$ be the contact that is associated with the merge operation. Assume further that $X$ and $Y$ are object of a concatenated $w$-merge operation $B$ or $C$ and that $Y$ is replaced by a new $Q$-node $Y'$, where $Y'$ gets as children (by construction) the root of some $PQ$-tree $T'$ and the node $Y$. Then $Y$ is removed from the $PQ$-tree during the reduction with respect to $w$.*

*Proof.* The new $Q$-node is obviously the root of the pertinent subtree with respect to $w$. Since the two merge operations are concatenated, Lemma 5.17 does not apply to $Y$. (Otherwise, simply remove the contact and either the left or right reference set.) It follows that the node $Y$ must be a partial $Q$-node. Therefore, template Q2 or Q3 is applied to $Y$ and its parent $Y'$, and $Y$ is removed from the tree, and the children of $Y$ become children of $Y'$. □

Lemma 5.33 ensures that after the reduction with respect to $w$, $c_1$ is again a child of a $Q$-node $Y'$, where $Y'$ is a child of $X$ occupying the former position of $Y$. We consider the position of $c_1$ within the sequence of children of $Y'$. Let $c_2$ be the contact associated with the merge operation that introduced $Y'$. Two cases may occur during the reduction with respect to $w$.

1. The contact $c_1$ was at the empty end of $Y$ and since $Y$ was an endmost child of $Y'$, $c_1$ is now an endmost child of $Y'$.

2. The contact $c_1$ was at the full end of $Y$, and appears within the sequence of full children of $Y'$.

After having finished the reduction with respect to $w$ one of the following two rules is applied to the contact $c_1$.

**Rule I** If $c_1$ is an endmost child of $Y'$, $c_1$ remains in its position as an endmost child of $Y'$.

**Rule II** If $c_1$ is found within the sequence of pertinent nodes, $c_1$ is placed as a new endmost child of $Y'$ next to $c_2$.

**Lemma 5.34.** *Let $R$, $R_1$, and $R_2$ be reduced extended forms of a level planar graph $G$. Let $T$, $T_1$, and $T_2$ be the PQ-trees corresponding to $R$, $R_1$, $R_2$. Let $w_1, w_2 \in V$ with $\text{lev}(w_1) \leq \text{lev}(w_2)$. Let $R_1$ be first $w_1$-merged into $R$. Let $R_2$ be $w_2$-merged into $R$ after $R_1$ has been $w_1$-merged into $R$ such that the two merge operations are concatenated. Let $X$, $X_\lambda$, $Y$ and $Y'$ be Q-nodes such that*

*(a) $X_\lambda$ was a child of $X$ in a PQ-tree $T$ before $w_1$-merging $T_1$ into $T$.*

*(b) $X_\lambda$ was replaced by $Y$ when $w_1$-merging $T_1$ into $T$ using the merge operation B or C.*

*(c) $Y$ was replaced by $Y'$ when $w_2$-merging $T_2$ into $T$ using the merge operation B or C.*

*Let $c(R_1)$ be the contact that is associated with the introduction of $Y$ and let $c(R_2)$ be the contact that is associated with the introduction of $Y'$. Let $R_{X_\lambda}$ be the subgraph corresponding to $X_\lambda$, and assume that $c(R_1)$ has been replaced by applying one of the Rules I or II. Then exactly one of the following statements holds.*

*(i) The contacts $c(R_1)$ and $c(R_2)$ are both endmost children at the same end of the Q-node $Y'$ if and only if their corresponding forms have to be embedded on the same side of $R_{X_\lambda}$ with respect to $R_X$.*

*(ii) The contacts $c(R_1)$ and $c(R_2)$ are both endmost children on opposite sides of the Q-node $Y'$ if and only if their corresponding forms have to be embedded on opposite sides of $R_{X_\lambda}$ with respect to $R_X$.*

*Proof.* Since the two merge operations are concatenated, Lemma 5.17 does not apply to $Y$. (Otherwise, simply remove the contact and either the left or right reference set.) It follows that $Y$ is a partial $Q$-node. Thus there exist at least two leaves, one corresponding to an incoming edge of $w_2$ and one corresponding to either an incoming edge of a vertex $u \in V^{\text{lev}(w_2)}$, $u \neq w_2$ or an edge traversing level $\text{lev}(w_2)$. Since $Y$ is a partial $Q$-node there exists an embedding of $R_Y$ and two paths

$$\begin{array}{ll} P = (u_1, u_2, \ldots, u_\mu) & \mu \geq 2 \\ u_1 \in R_1 & \\ u_i \notin R_{X_\lambda} - \{w_1\} & i = 1, 2, \ldots, \mu \\ \text{lev}(u_i) < \text{lev}(w_2) & i = 1, 2, \ldots, \mu - 1 \\ \text{lev}(u_\mu) \geq \text{lev}(w_2) & \end{array}$$

and

$$\begin{array}{ll} P' = (u'_1, u'_2, \ldots, u'_\nu) & \nu \geq 2 \\ u'_1 \in R_{X_\lambda} & \\ u_i \notin R_1 & i = 1, 2, \ldots, \nu \\ \mathrm{lev}(u'_i) < \mathrm{lev}(w_2) & i = 1, 2, \ldots, \nu - 1 \\ \mathrm{lev}(u'_\nu) \geq \mathrm{lev}(w_2) & \end{array}$$

such that

(a) $P$ and $P'$ are disjoint,

(b) both $P$ and $P'$ are on the boundary outer face of the embedding of $R_Y$, and

(c) either $u_\mu = w_2$ or $u'_\nu = w_2$, but not both.

See Fig. 5.21 for an illustration.

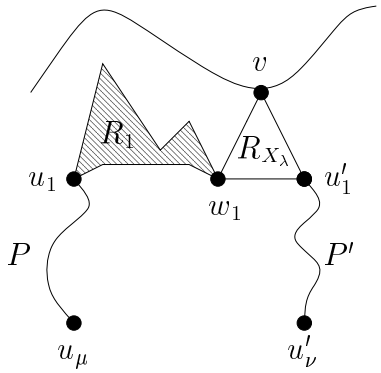



Figure 5.21: Illustration of the proof of Lemma 5.34.

First, case (i) is proven. Let $R_2$ and $R_1$ be embedded on the same side of $R_{X_\lambda}$. It follows that $w_2 \in P$, otherwise $R_2$ and $P$ would cross each other. Let $Z$ be the child of $Y$ that is an ancestor of the leaf labeled $w_2$. Since the path $P$ is on the outer face of $R_Y$ on the side where $R_1$ is embedded, $Z$ must be an endmost nonignored child of $Y$ on the side where $\mathrm{c}(R_1)$ is an endmost child. Since $Y$ is partial, $\mathrm{c}(R_1)$ will appear within the pertinent sequence of leaves labeled $w_2$ after the reduction with respect to $w_2$ is complete. Therefore Rule II is applied and $\mathrm{c}(R_1)$ and $\mathrm{c}(R_2)$ are both endmost children at the same end of $Y'$.

Now let $\mathrm{c}(R_1)$ and $\mathrm{c}(R_2)$ be endmost children on the same side, and assume that $R_1$ and $R_2$ have to be embedded on opposite sides of $R_{X_\lambda}$ with respect to $R_X$. It follows that $w_2 \in P'$, otherwise $R_2$ and $P'$ would cross each other. By construction, $\mathrm{c}(R_1)$ was found within in the pertinent sequence with respect to the vertex $w_2$ after $R_2$ was $w_2$-merged into $R$. So there exists a path $P''$ on the boundary of $R_Y$, and $P''$ connects a vertex $u \in R_1$ and $w_2$, not using any vertices $u' \in \bigcup_{i=\mathrm{lev}(w_2)}^{k} V^i$. However, $P''$ must cross $P$, a contradiction.

The case (ii) is proven analogously to the case (i). □

**Remark 5.35.** *Lemma 5.34 also holds if $w_1 = w_2$. It is interesting to note that in the case of $w_1 = w_2$ the form $R_1$ must be a $w_1$-singular form. If $R_1$ is not singular and $w_1 = w_2$, it follows that the PQ-tree $T_2$ is merged into the subtree rooted at $Y$, and merging $R_2$ into $R$ does not influence the $w_1$-merge operation.*

If two contacts $c_1$ and $c_2$ influence each other, and therefore $\mathrm{ref}(c_1) \cap \mathrm{ref}(c_2) \neq \emptyset$ holds, we need to redefine their reference sets such that no conflicts appear when sink indicators for edge augmentation are considered. A situation, where we can chose for a sink indicator to which reference set it belongs has to be avoided.

Again let $R$, $R_1$, $R_2$, $X$, $X_\lambda$, $Y$, $Y'$, $c(R_1)$, and $c(R_2)$ be defined as in Lemma 5.34. The idea is to leave the reference set of $c(R_1)$ (the contact associated to the "first" merge operation) unchanged, and adapt the reference set of $c(R_2)$ (the contact associated to the "second" merge operation). Let $I_1, I_2, \ldots, I_\mu$, $\mu \geq 0$, be the sequence of ignored nodes on the left side of $X_\lambda$ with $X_\lambda$ and $I_\mu$ being direct siblings, and let $J_1, J_2, \ldots, J_\rho$, $\rho \geq 0$, be the sequence of ignored nodes on the right side of $X_\lambda$ with $X_\lambda$ and $J_1$, being direct siblings. For simplicity assume that

$$\mathrm{ref}(c(R_2)) \subseteq \left(\bigcup_{i=1}^{\mu} \mathrm{frontier}(I_i)\right) \cup \left(\bigcup_{i=1}^{\rho} \mathrm{frontier}(J_i)\right) .$$

The case where the direct nonignored siblings of $X_\lambda$ become ignored before $w_2$-merging $T_2$ into $T$, is handled by a straightforward adaption of the strategy. Let

$$\mathrm{ref}(c(R_1)) = \left(\bigcup_{i=\nu_1}^{\mu} \mathrm{frontier}(I_i)\right) \cup \left(\bigcup_{i=1}^{\sigma_1} \mathrm{frontier}(J_i)\right)$$
$$\text{for } 1 \leq \nu_1 \leq \mu + 1,\ 0 \leq \sigma_1 \leq \rho$$

be the reference set of $c(R_1)$, where we assume without loss of generality that none of the two subsets is empty. The reference points of $c(R_1)$ are $I_{\nu_1}, I_\mu, J_1, J_{\sigma_1}$. Assume further that

$$\mathrm{ref}(c(R_2)) = \left(\bigcup_{i=\nu_2}^{\mu} \mathrm{frontier}(I_i)\right) \cup \left(\bigcup_{i=1}^{\sigma_2} \mathrm{frontier}(J_i)\right)$$
$$\text{for } 1 \leq \nu_2 \leq \mu + 1,\ 0 \leq \sigma_2 \leq \rho\ .$$

After performing the second merge operation including the reduction of the leaves labeled $w_2$, the contacts $c(R_1)$ and $c(R_2)$ occupy two relative positions at the their parent $Y'$.

(i) $c(R_1)$ and $c(R_2)$ are endmost children on different ends of $Y'$. Due to Lemma 5.34, $R_1$ and $R_2$ are embedded on opposite sides of $R_{X_\lambda}$ with respect to $R_X$. Thus $c(R_1)$ and $c(R_2)$ do not interfere when finally determining the sets of sink indicators that are considered for edge augmentation. We scan the sequences for the reference points $I_{\nu_2}, I_\mu, J_1, J_{\sigma_2}$ and store them at $c(R_2)$.

(ii) $c(R_1)$ and $c(R_2)$ are at the same end of $Y'$ with $c(R_1)$ being (by construction) an endmost child. Due to Lemma 5.34, $R_1$ and $R_2$ are embedded at the same side of $R_{X_\lambda}$. Thus $c(R_1)$ and $c(R_2)$ interfere when we finally determine the sets of sink indicators that are considered for edge augmentation.

The new reference set of $c(R_2)$ is determined as follows.

$$\text{ref}(c(R_2))^{left} = \begin{cases} \bigcup_{i=\nu_2}^{\nu_1-1} \text{frontier}(I_i) & \text{if } \nu_2 < \nu_1 \\ \emptyset & \text{otherwise} \end{cases} \tag{5.3}$$

$$\text{ref}(c(R_2))^{right} = \begin{cases} \bigcup_{i=\sigma_1+1}^{\sigma_2} \text{frontier}(J_i) & \text{if } \sigma_2 > \sigma_1 \\ \emptyset & \text{otherwise} \end{cases} \tag{5.4}$$

Then the reference set of $c(R_2)$ is

$$\text{ref}(c(R_2)) = \text{ref}(c(R_2))^{right} \cup \text{ref}(c(R_2))^{left} .$$

Hence, the ignored nodes $I_{\nu_2}, I_{\nu_1-1}, J_{\sigma_1+1}, J_{\sigma_2}$ are stored as reference points at $c(R_2)$.

Consider the second case when removing the $Q$-node $Y'$ during the application of the template Q2 or Q3. The contact $c(R_1)$ is an endmost child of $Y'$. Thus, after the application of the template Q2 or Q3 the contact $c(R_1)$ is a direct sibling of either $I_\mu$ or $J_1$. Therefore, the identification of sink indicators that have to be considered for edge augmentation joining the vertex $w_1$ is a straightforward matter. After this identification is finished, the contact $c(R_1)$ and the set of ignored siblings that were considered for augmentation are removed from the $PQ$-tree, leaving the contact $c(R_2)$ as a direct sibling of either $I_{\nu_1-1}$ or $J_{\sigma_1+1}$. Again, the identification of the sink indicators that have to be considered for edge augmentation joining the vertex $w_2$ is straightforward.

**Lemma 5.36.** *Let* $c_i$, $i = 1, 2, \ldots, \mu$, $\mu \geq 1$, *be contacts that are endmost children of a* $Q$*-node* $Y$ *in a* $PQ$*-tree* $T$. *Let contact* $c_i$ *be related to vertex* $w_i \in V$, *such that* $\text{lev}(w_i) \leq \text{lev}(w_{i+1})$, $i = 1, 2, \ldots, \mu - 1$. *In case that* $\text{lev}(w_i) = \text{lev}(w_{i+1})$ *holds, let the* $PQ$*-tree* $T_i$ *corresponding to* $c_i$ *be* $w_i$*-merged into* $T$ *before the tree* $T_{i+1}$ *corresponding to* $w_{i+1}$ *is* $w_{i+1}$*-merged into* $T$. *Let* $\text{ref}(c_i)^{left}$ *be the left reference set of* $c_i$ *with reference points* $I_i^b, I_i^e$ *and* $\text{ref}(c_i)^{right}$ *be the right reference set of* $c_i$ *with reference points* $J_i^b, J_i^e$. *For every* $c_i$, *the nodes* $I_i^b$ *and* $J_i^b$ *denote the first ignored node in the reference sequence* $\text{rseq}(c_i)^{left}$ *and* $\text{rseq}(c_i)^{right}$, *respectively, and* $I_i^e$ *and* $J_i^e$ *denote the last ignored node in the reference sequence* $\text{rseq}(c_i)^{left}$ *and* $\text{rseq}(c_i)^{right}$, *respectively. Then the following statements hold true.*

*(i) If* $c_i$ *is adjacent to* $I_i^b$, *then augmenting* $G_{st}$ *by an edge* $(u, w)$ *for every* $\text{si}(u) \in \text{ref}(c_i)^{left}$ *does not destroy level planarity.*

*(ii) If* $c_i$ *is adjacent to* $J_i^b$, *then augmenting* $G_{st}$ *by an edge* $(u, w)$ *for every* $\text{si}(u) \in \text{ref}(c_i)^{right}$ *does not destroy level planarity.*

*Proof.* By construction, the sequence of children of $Y$ is partitioned into the three sets: $C_1$, $N$ and $C_2$ where

- $C_1 = \mathrm{c}_1^1, \mathrm{c}_2^1, \ldots, \mathrm{c}_\nu^1$, $0 \leq \nu \leq \mu$, is the sequence of contacts on one end of $Y$ with $\mathrm{c}_1^1$ being an endmost child of $Y$.
- $N$ is a sequence of ignored and nonignored nodes.
- $C_2 = \mathrm{c}_1^2, \mathrm{c}_2^2, \ldots, \mathrm{c}_\varphi^2$, $\varphi = \mu - \nu$, is the sequence of contacts on the opposite side of $C_1$ with $\mathrm{c}_1^2$ as an endmost child of $Y$.

By construction, we have for $C_\xi$, $\xi = 1, 2$,

$$\mathrm{lev}(\omega(\mathrm{c}_i^\xi)) \;\leq\; \mathrm{lev}(\omega(\mathrm{c}_{i+1}^\xi)) \quad i = 1, 2, \ldots, |C_\xi| - 1$$

where $\omega(\mathrm{c}_i^\xi)$ denotes the vertex related to $\mathrm{c}_i^\xi$. For any $i \in \{1, 2, \ldots, |C_\xi| - 1\}$ with $\mathrm{lev}(\omega(\mathrm{c}_i^\xi)) = \mathrm{lev}(\omega(\mathrm{c}_{i+1}^\xi))$ the $PQ$-tree corresponding to $\mathrm{c}_i^\xi$ has been $\omega(\mathrm{c}_i^\xi)$-merged into $T$ before the tree corresponding to $\mathrm{c}_{i+1}^\xi$ has been $\omega(\mathrm{c}_{i+1}^\xi)$-merged.

It follows that either $\mathrm{c}_1^1 = \mathrm{c}_1$ or $\mathrm{c}_1^2 = \mathrm{c}_1$ and $\mathrm{c}_1^1$ and $\mathrm{c}_1^2$ do no interfere, since their corresponding forms are placed on opposite sides with respect to $R_{X_\lambda}$ and $R_X$. We may assume that $\mathrm{c}_1^1 = \mathrm{c}_1$. Due to Observation 5.20, $\mathrm{c}_1^1$ is either adjacent to $I_1^b$ or to $J_1^b$. Assume without loss of generality that $\mathrm{c}_1^1$ is adjacent to $I_1^b$. It follows from Lemma 5.22 that considering $\mathrm{ref}(\mathrm{c}_1)^{left}$ for edge augmentation does not destroy level planarity. Removing $\mathrm{c}_1^1$ and $\mathrm{ref}(\mathrm{c}_1)^{left}$ from the tree, the correctness of the lemma follows by a simple inductive argument. □

## 5.3 Complete Algorithm

The last section referred to the details that have to be taken into consideration when constructing a level planar embedding of a level graph $G = (V, E)$. The idea was to augment the level graph to a level $st$-graph $G_{st} = (V_{st}, E_{st})$, $V_{st} = V \uplus \{s, t\}$, $E \subseteq E_{st}$, compute a planar embedding for $G_{st}$ and construct a level planar embedding of $G$ from the embedding of $G_{st}$. This section combines the results and shows the correctness of the level planar embedder. Using a function AUGMENT that augments a given level graph by adding an outgoing edge to every sink in the graph without destroying level planarity, the level planar embedding algorithm is formulated as follows.

$\mathcal{E}_l$ **LEVEL-PLANAR-EMBED**$(G = (V^1, V^2, \ldots, V^k; E))$

begin
  ignore all isolated vertices;
  expand $G$ to $G_{st}$ by adding $s$ to $V^0$ and $t$ to $V^{k+1}$;

```
  AUGMENT(G_st);
  if AUGMENT failed then
    return E_l = ∅;
  //G_st is now a hierarchy;
  orient the graph G_st from the bottom to the top;
  AUGMENT(G_st);
  add edge (s, t);
  //G_st is now an st-graph;
  orient the graph G_st from the top to the bottom;
  compute a topological sorting of V_st;
  compute a planar embedding E_st using the topological sorting
    as an st-numbering;
  E_l = CONSTRUCT-LEVEL-EMBED(E_st, G_st);
  return E_l;
end.
```

As sketched in Section 5.1, the level planar embedder first expands the graph $G$ to $G_{st}$ by an extra level $V^0$ adjacent to $V^1$, and an extra level $V^{k+1}$ adjacent to $V^k$ with a source $s$, $\{s\} = V^0$ and a sink $t$, $\{t\} = V^{k+1}$. It then augments $G_{st}$ to a hierarchy by applying AUGMENT, adding an outgoing edge to every sink of $G$. Applying the function AUGMENT to $G_{st}$ from the bottom to the top, $G_{st}$ is augmented by adding an incoming edge to every source of $G$. When finally adding the edge $(s, t)$, the graph $G_{st}$ is an $st$-graph. By construction, the leveling of $G$ is a topological numbering of $G_{st}$. The computation of any topological sorting on the $st$-graph yields an $st$-numbering. Using this $st$-numbering, a planar embedding $\mathcal{E}_{st}$ is computed using the algorithm of Chiba, Nishizeki, Abe, and Ozawa (1985). Finally, the function CONSTRUCT-LEVEL-EMBED presented in Section 5.1 computes a level planar embedding $\mathcal{E}_l$.

The function AUGMENT is almost identical to the function LEVEL-PLANAR, except that it does not call the function CHECK-LEVEL but a function EMBED-LEVEL. However, the function EMBED-LEVEL is almost identical to the function CHECK-LEVEL that has been described in Section 4.9. We only need the following modifications.

(i) If $v$ is a sink in $V^j$, $1 < j < k$, replace the corresponding leaf by a sink indicator si($v$) before processing $G^{j+1}$. If this replacement constructs a node $X$ having only sink indicators in its frontier, mark $X$ as ignored and update the ML-values as described in Section 5.2.3.

(ii) When reducing a set of leaves with respect to a vertex $w$ in a $PQ$-tree, ignore all sink indicators and ignored nodes during the application of the template matching algorithm. After the reduction is complete, including updates for the PML- and QML-values, the pertinent subtree is removed from the tree and replaced by a single representative. During the removal of the pertinent subtree with respect to $w$, we check for sink indicators in the pertinent subtree. For every si($v$) that is found in

the pertinent subtree, we add an edge $(v, w)$ to $G_{st}$, unless si$(v)$ is affected by the existence of a contact c in the pertinent subtree. If the latter applies, add an edge $(v, w')$, with $w'$ being the vertex related to c.

(iii) When $w$-merging a $PQ$-tree $T'$ into a $PQ$-tree $T$, necessary adjustments as described in the Sections 5.2.4 and 5.2.5 have to be applied to the merge operations B or C. If necessary, a contact is introduced as a third child of the new $Q$-node. Furthermore, if the new contact mutually influences existing ones, the Rules I or II (see 5.2.5) have to be applied after reducing $T$ with respect to $w$.

(iv) After processing the level $k$, an edge $(v, t)$ is added for every vertex $v \in V^k$. Furthermore we scan the final $PQ$-tree $T$ for remaining sink indicators, and add for every indicator si$(v)$ an edge $(v, t)$ to $G_{st}$, unless si$(v)$ is affected by the existence of a contact c in the pertinent subtree. If the latter applies we add an edge $(v, w')$, with $w'$ being the vertex related to c.

**Theorem 5.37.** *The algorithm LEVEL-PLANAR-EMBED computes a level planar embedding of a level planar graph $G = (V, E)$.*

*Proof.* From Lemmas 5.3, 5.5, 5.6 and 5.36 it follows that augmenting $G_{st}$ to a hierarchy using the function AUGMENT does not destroy level planarity. Consequently, the augmentation of $G_{st}$ to a single source, single sink graph does not destroy level planarity either. The vertices $s$ and $t$ are on the outer face of a level planar embedding of $G_{st}$. Hence, adding the edge $(s, t)$ to $G_{st}$ does not destroy level planarity. Thus a planar $st$-graph has been constructed. Computing a topological sorting of $G_{st}$ yields a numbering $\mathcal{N} : V_{st} \to \{1, 2, \ldots, n + 2\}$ of the vertices with

(i) $\mathcal{N}(s) = 1$ and $\mathcal{N}(t) = n + 2$, and

(ii) for every $v \in V$ there exist vertices $u, w \in V_{st}$ such that $\mathcal{N}(u) < \mathcal{N}(v) < \mathcal{N}(w)$.

Thus $\mathcal{N}$ is an $st$-numbering of $V_{st}$ and applying the embedding algorithm of Chiba *et al.* (1985) to $G_{st}$ we can construct according to Theorem 5.1 a level planar embedding of $G$. □

## 5.4 Proving $\mathcal{O}(n)$ Running Time

Equipping the $PQ$-trees with sink indicators and contacts, and marking internal nodes as ignored nodes seems to be a massive blow up of the data structure, easily consuming quadratic running time. However, as we will prove with the next theorem, we get a linear running time algorithm. The linearity of the algorithm follows mainly from two facts.

1. The number of sources and sinks is bounded by $n$. Thus the number of sink indicators, contacts and internal ignored nodes is bounded by a constant multiple of $n$.

2. We scan for ignored nodes only if necessary. The number of accesses of ignored nodes is bounded by a constant factor of $n$ as well.

Before proving the main theorem of this section, we show that the augmentation of the graph $G$ into a hierarchy only needs $\mathcal{O}(n)$ time.

**Lemma 5.38.** *Given a level planar graph $G = (V, E)$ with $k$ levels and an extra sink $t$ on an extra level $k+1$, the function AUGMENT constructs in $\mathcal{O}(n)$ time a level planar hierarchy by augmenting $G$ for every sink $v \in V$ by an edge $e = (v, w)$, $w \in V \uplus \{t\}$, $\mathrm{lev}(v) < \mathrm{lev}(w)$.*

*Proof.* The function AUGMENT performs as the function LEVEL-PLANAR, with certain modifications. It is sufficient to show that the amount of extra work performed by these modifications consumes $\mathcal{O}(n)$ time.

The number of sinks is bounded by $\mathcal{O}(n)$. Thus the number of sink indicators and contacts is bounded by $\mathcal{O}(n)$. Since every $P$-node corresponds to a cut vertex and every $Q$-node to a reversible subgraph, the number of ignored $P$- and $Q$-nodes is also bounded by $\mathcal{O}(n)$. Marking a node as ignored is done only once per node, and is performed if the last remaining nonignored child becomes ignored. Therefore, the amount of time needed to discover nodes that have become ignored is bounded by $\mathcal{O}(n)$ for all ignored nodes throughout the application of the algorithm.

According to Theorem 5.37, AUGMENT constructs a level planar graph by adding at most $n$ edges. Thus a sink indicator $\mathrm{si}(v)$ of a sink $v \in V$ can be interpreted as a leaf corresponding to an outgoing edge of $v$. The tree is not scanned explicitly for sink indicators. (Recall that we do not scan for ignored nodes during the application of templates P2, P3, ..., P6.) Thus an ignored node is scanned only a constant number of times during the application of the template matching algorithm. Therefore we have that the number of extra operations in all calls of REDUCE is in $\mathcal{O}(n)$.

The amount of work that has to be done during the merge operations B and C in order to install a contact and to identify the reference sets of the contact is bounded by a constant plus the number of ignored nodes that are contained in the reference sequence. We combine three facts:

(i) Every ignored node may appear at most twice in a reference sequence (twice in case that two contacts are endmost children on opposite sides of the same $Q$-node).

(ii) Let $\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_\mu$ be the sequence of contacts on the left end of a $Q$-node $Y$, and let $\mathrm{c}_{\mu+1}, \mathrm{c}_{\mu+2}, \ldots, \mathrm{c}_\nu$ be the sequence of contacts on the right end of $Y$. Let

$$\mathrm{rseq}_1^{\mathit{left}} \;=\; \bigcup_{i=1}^{\mu} \mathrm{rseq}(\mathrm{c}_i)^{\mathit{left}} \;,$$

$$\mathrm{rseq}_2^{left} = \bigcup_{i=\mu+1}^{\nu} \mathrm{rseq}(\mathrm{c}_i)^{left} ,$$

$$\mathrm{rseq}_1^{right} = \bigcup_{i=1}^{\mu} \mathrm{rseq}(\mathrm{c}_i)^{right} ,$$

$$\mathrm{rseq}_2^{right} = \bigcup_{i=\mu+1}^{\nu} \mathrm{rseq}(\mathrm{c}_i)^{right} .$$

We keep for each of the four sequences a pointer to the first and a pointer the last ignored node, including information on the minimal ML-value between ignored siblings of a sequence. Using these pointers, a new contact $c_{\nu+1}$ can be concatenated at $\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_\nu$ using only constant time to check if its left or right reference sequence is empty. If the left or the right reference sequence is not empty, the reference sequence consists due to equations 5.3 and 5.4 only of ignored nodes that are either not contained in $\mathrm{rseq}_1^{left}$, and $\mathrm{rseq}_1^{right}$ (in case that $\mathrm{c}_{\nu+1}$ is added to the same side as $\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_\nu$), or not contained in $\mathrm{rseq}_2^{left}$, and $\mathrm{rseq}_2^{right}$ (in case that $\mathrm{c}_{\nu+1}$ is added to the same side as $\mathrm{c}_{\mu+1}, \mathrm{c}_{\mu+2}, \ldots, \mathrm{c}_\nu$). Thus the total amount of work needed for determining the reference sequence of every contact is bounded by the number of ignored nodes and therefore in $\mathcal{O}(n)$.

(iii) Consider a sequence of contacts $\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_\mu$, $\mu \geq 1$, at one end of a $Q$-node $Y$, with $\mathrm{c}_1$ being an endmost child of $Y$. Thus $\mathrm{c}_i$ has been introduced into the $PQ$-tree before $\mathrm{c}_{i+1}$, $i = 1, 2, \ldots, \mu - 1$. Let $\mathrm{c}_{\mu+1}$ be a new contact that has to be introduced to the tree such that $\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_\mu$ and $\mathrm{c}_{\mu+1}$ mutually influence each other. Assume that $\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_\mu$ appear within the pertinent sequence with respect to $\omega(\mathrm{c}_{\mu+1})$. According to Rule II the complete sequence $\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_\mu$ has to be moved to the end of the new $Q$-node, where $\mathrm{c}_{\mu+1}$ is added as an endmost child. This constructs a new sequence $\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_\mu, \mathrm{c}_{\mu+1}$. The children of a $Q$-node are organized in a doubly linked list. Hence, we only need to access the contacts $\mathrm{c}_1$ and $\mathrm{c}_\mu$ for moving the sequence $\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_\mu$. Since the contact $\mathrm{c}_1$ is an endmost child of $Y$, it can be accessed in constant time. Keeping a pointer at $\mathrm{c}_1$ towards $\mathrm{c}_\mu$, the contact $\mathrm{c}_\mu$ can be accessed in constant time as well. When the sequence $\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_\mu$ and $\mathrm{c}_{\mu+1}$ have been combined to a new sequence, the pointer to the end of this sequence stored at $\mathrm{c}_1$ is updated. The number of operations to apply Rule II is therefore bounded by a constant per merge operation.

We deduce that the amount of extra work performed during all merge operations B and C is bounded by a constant number of operations per merge operation plus the total number of ignored nodes and therefore is in $\mathcal{O}(n)$.

The amount of extra work that has to be performed for removing contacts and their corresponding reference sets from the tree is proportional to the number of ignored nodes.

Contacts, for which the embedding of the associated subgraph is fixed (that is, Lemma 5.29 applies to these contacts) are detected implicitly during a reduction with respect to some vertex $v$. The total amount of work for detecting and removing these contacts and their reference sets is therefore in $\mathcal{O}(n)$.

We now estimate the amount of work needed for contacts for which the embedding of the associated subgraph may be chosen freely (that is, Lemma 5.30 applies to these contacts). The Lemma 5.30 does not provide a useful strategy for detecting a contact $\mathrm{c}(R)$ with a free choice for the embedding for $R$. Scanning the $PQ$-trees in order to find such contacts results in a $\mathcal{O}(n^2)$ time algorithm. We show how to find these contacts during the application of the template matching algorithm.

Consider several contacts as children of a $Q$-node $Y$, and $Y$ is a child of a $Q$-node $X$. According to Lemma 5.17 the following four cases may apply to $Y$.

(i) The $Q$-node $Y$ is ignored.

(ii) The $Q$-node $Y$ is not ignored and can be found in the final $PQ$-tree.

(iii) The $Q$-node $Y$ has only one nonignored child.

(iv) The $Q$-node $X$ has only $Y$ as nonignored child.

**Case (i)** The $Q$-node $Y$ will eventually be found within the pertinent subtree of some vertex $w$. Since $Y$ is child of a $Q$-node $X$, $X$ must be itself full, partial or ignored. Since the ignored node $Y$ is in the pertinent sequence, and the reference sequences of the contacts of $Y$ are ignored nodes adjacent to $Y$, the reference sets are also contained in the pertinent subtree. When scanning the subtree for sink indicators $\mathrm{si}(v)$, $v \in V$, in order to augment the graph by edges $e = (v, w)$, these contacts are detected. We chose an arbitrary reference point for a contact c that is an endmost child of $Y$, and remove it and its corresponding reference set from the tree, adding for every $\mathrm{si}(\tilde{v}) \in \mathrm{ref}(\mathrm{c})$ an edge $(\tilde{v}, \omega(\mathrm{c}))$ to $G$. All other contacts that are children of $Y$ are removed together with their reference set in correspondence to the choice we took for the first contact.

**Case (ii)** We proceed as in case (i), when the final $PQ$-tree is scanned for sink indicators $\mathrm{si}(v)$, $v \in V$, in order to augment the graph by edges $e = (v, t)$.

**Case (iii)** Two subcases occur.

(a) There exists a vertex $w$ such that $Y$ and its parent $X$ are both in the pertinent subtree of $w$. The node $Y$ may be either full or partial. In the latter case, $\mathrm{frontier}(Y)$ contains full and empty leaves.

If $Y$ is full, choose for one of its endmost contacts an arbitrary reference point, and remove the corresponding reference set from the tree. Continue removing

the remaining contacts, taking the choice of the first removed contact into consideration. This is done while scanning the pertinent subtree for sink indicators $\mathrm{si}(v)$, $v \in V$, in order to augment the graph by edges $e = (v, w)$.

If $Y$ is partial, $Y$ will become a $Q$-node with at least two children, one of them being empty and one of them being full. Thus templates Q2 and Q3 have to be applied to $Y$ and its parent $X$. Since the reference points may be chosen freely for the endmost contacts, we apply one of the templates Q2 or Q3 first. This places the endmost contacts next to one of their reference points. We remove the contacts and their reference sets, taking into consideration the situation caused by the application of the template Q2 or Q3.

(b) There does not exist a vertex such that both $Y$ and $X$ become pertinent. Then nodes $Y$ and $X$ are both contained in the final $PQ$-tree and we may freely choose the reference sets for the contacts.

It is interesting to note that due to the minimality of the pertinent subtree the node $Y$ never becomes the root of a pertinent subtree in case (iii).

**Case(iv)** This case is treated analogously to the case (iii).

The work that has to be done in the final $PQ$-tree for finding contacts providing a free choice for embedding their reference set, is bounded by the number of nodes in $T$ and therefore in $\mathcal{O}(n)$. Moreover all other contacts providing a free choice are detected implicitly during a reduction with respect to some vertex $w$. Thus the total amount of work for detecting and removing contacts with a free choice and their reference sets is in $\mathcal{O}(n)$.

The results of the proof summarize to $\mathcal{O}(n)$ running time for the function AUGMENT. □

**Theorem 5.39.** *The algorithm LEVEL-PLANAR-EMBED computes a level planar embedding of a level planar graph $G = (V, E)$ in $\mathcal{O}(n)$ time.*

*Proof.* Clearly, expanding $G$ to $G_{st}$ by adding $s$ to $V^0$ and $t$ to $V^{k+1}$ needs only constant time. Augmenting $G_{st}$ to a hierarchy is in $\mathcal{O}(n)$ according to Lemma 5.38. Since the number of edges added to $G_{st}$ is bound by $\mathcal{O}(n)$, the graph $G_{st}$ is augmented to an $st$-graph in $\mathcal{O}(n)$ time. Computing a topological sorting can be done in linear time as well and computing a planar embedding is performed in $\mathcal{O}(n)$ time according to Chiba *et al.* (1985). Finally, computing a level planar embedding of $G$ using the planar embedding of $G_{st}$ can be done in $\mathcal{O}(n)$ time according to Corollary 5.2. □

## 5.5 Remarks

It is of course possible to apply the following strategy for embedding a level planar graph. First, augment the graph to a level planar hierarchy. Second, compute an embedding of

the hierarchy using the algorithm LPTH by Di Battista and Nardelli (1988). Since the augmentation adds at most $n$ edges, the algorithm of Di Battista and Nardelli (1988) will need $\mathcal{O}(n)$ time as well, yielding a linear time algorithm for embedding a level planar graph. However, this is of no practical value, since both implementations, an implementation of AUGMENT and an implementation of the algorithm of Di Battista and Nardelli, have to be provided. None of the two algorithms is easy to implement. Thus it is preferable for any programmer just to implement AUGMENT and to use an existing implementation of the planar embedding algorithm of Chiba *et al.* (1985). This avoids putting extra effort in the development of LPTH, and good implementations of the embedding algorithm are available for commercial as well as noncommercial use (see, e.g., Mehlhorn and Näher (1998)).

Our approach for embedding a level planar graph does not make use of the strategy presented by Chiba *et al.* (1985), except of course for embedding a planar graph. The idea of direction indicators has not been considered in the function AUGMENT for two reasons.

(a) An approach using only direction indicators, not augmenting the graph by edges, computes for every vertex the sequence of incoming edges as they appear in a planar embedding. However, according Carpano (1980) the planar embedding is not a level planar embedding. The approach does not yield any information where to place sinks and sources of the levels $V^2, V^3, \ldots, V^{k-1}$. Thus a stronger algorithmic concept providing more information is needed.

(b) If $R_1$ and $R_2$ are components of $G^j$ that have to be $w$-merged at a vertex $w \in V^{j+1}$, the merge operations, especially the B and C operations, flip subsequences of the incoming edges of $w$. To keep track of the reversions of the subsequences of the incoming edges of a vertex $w$, several direction indicators have to be provided for every vertex $w$. However, such an approach demands a more involved adaption of the algorithm of Chiba *et al.* (1985).

We conclude that an adaption of direction indicators not using our approach appears to be difficult since it requires some sort of post processing for constructing a level planar embedding from the planar embedding, while the approach presented in this chapter transforms a level planar embedding problem into a planar embedding problem, yielding a straightforward adaption of the algorithm of Chiba *et al.* (1985).

A possible modification of our approach would be a combination of AUGMENT and direction indicators. While our approach needs three phases (twice the application of AUGMENT and once the application of the planar embedder), a combination can be used to construct a one phase algorithm, using linear time. But such an approach would be even more difficult to implement than the three phase algorithm. Besides, it is not clear if an implementation of a one phase approach would perform better than the three phase approach since the amount of extra work for the investigation of the direction indicators is estimated to be more complicated than in the "normal" planar embedding algorithm.

Finally we mention how the level planar embedder should be modified in order to work on an unconnected level graph. Instead of computing for every connected component its own

$st$-graph, augment every component with respect to the same super source $s$ and super sink $t$ and add the edge $(s, t)$ only once. For the resulting graph, we need to make only one call of the planar embedding algorithm of Chiba *et al.* (1985). This in turn allows to embed all components level planar at once, with the components all lined up in a row.

# Chapter 6

# Implementation of a Level Planar Embedder

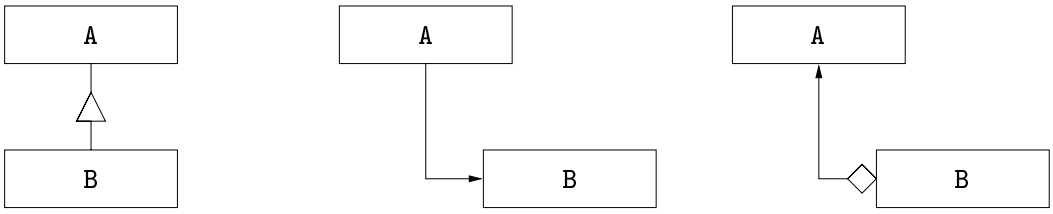
An object oriented prototype of a level planarity testing and embedding algorithm has been implemented in C++. This chapter reviews the concept of the software and pays attention to some of its details. The implementation of LEVEL-PLANAR-EMBED is based on an implementation of $PQ$-trees as template class by Leipert (1997). The planar embedding algorithm of Chiba *et al.* (1985) that is needed in the third phase of LEVEL-PLANAR-EMBED was provided by Buchheim (1997) and is also based on the $PQ$-tree implementation of Leipert (1997).

In order to display inheritance, acquaintance and aggregation of objects in subsequent figures of this chapter, we use a representation as it has been introduced by Gamma, Helm, Johnson, and Vlissides (1996). If a class `B` inherits a class `A`, we say `B` is a derived class of `A`. A subclasses as `B` can refine and redefine the behavior of its parent `A`, more specifically class `B` may override an operation defined by its parent class `A`.

Acquaintance means that a class `B` knows of another class `A` and we say that `B` uses class `A`. Acquainted objects (an object is an instance of a class) may request operations on each other, but they are not responsible for each other.

Aggregation means that a class `B` owns another class `A` and we say that `B` has class `A`. Aggregation implies that one object is responsible of another object. An aggregated object and its owner have identical lifetimes. Figure 6.1 gives an overview on the used symbols.

The first section reviews the concept of an implementation of the $PQ$-tree data structure as class template in C++. The second section describes the concept of the implementation of a level planar embedder based on the implementations of the $PQ$-tree data structure. We have implemented the level planarity test as described in section 4.5 including the necessary modifications as described in 5.2 and 5.3. The implementation embeds a level planar graph in $\mathcal{O}(n \log n)$ time. A version using $\mathcal{O}(n)$ time has not been implemented yet since the implementation of the level planarity test according to 4.5 was already finished and an implementation of the level planar embedding algorithm was halfway finished at




(a) Class `B` inherits class `A`.

(b) Class `B` knows class `A`.

(c) Class `B` aggregates class `A`.

Figure 6.1: Different relationships between classes according to Gamma *et al.* (1996).

the time we figured out how to obtain a linear running time.

Since the implementation contains 4037 statements, 698 `if` queries and 497 `else` and `else if` queries, we abstained from presenting the complete implementation and concentrated on a few details of the implementation. The last three sections contain such details, namely the implementation of the merge operation, and the implementation of two templates. We decided to present the implementation of template P5 since it is the most problematic one of the templates P1 – P6. Furthermore, we give template Q2 since it includes the handling of contacts.

## 6.1 $PQ$-trees

Since the $PQ$-trees obtain more and more attention throughout the literature, the $PQ$-trees and the reduction algorithm have been implemented as a template class in C++. This allows easier adaption of the data structure to different algorithms. We pursued two main goals when implementing $PQ$-trees and the template matching algorithm:

- Easy application of the $PQ$-tree data structure to the consecutive ones property problem and other related problems. Specific for such a feature is to let a user not worry about any internal details of the $PQ$-tree. The user just specifies a set $U$ and its subsets that have to be reduced.
- The design of the code should be reusable. Applications of more sophisticated algorithms to the data structure should be possible and easy.

Except for its implementation, the first task is conceptually not difficult. Solving the second task is more difficult since possible applications of sophisticated algorithms are almost unpredictable. A careful study of existing algorithms based on $PQ$-trees led to the following requirements on a reusable implementation of the $PQ$-trees.

(i) The data structure should be implemented as class template. This allows very easy adaption to the consecutive one property problem.

(ii) It should be possible to add new types of nodes. Applications known in the literature are the algorithms described in Chiba *et al.* (1985), and Feng *et al.* (1995).

(iii) Different treatment of nodes apart from the original template matching algorithm should be possible. Applications known in the literature are the algorithms described in Chiba *et al.* (1985), and Feng *et al.* (1995).

(iv) It should be possible to associate different kinds of information to the nodes of the $PQ$-tree. Applications known in the literature are the algorithms described in Jayakumar *et al.* (1989), Karabeg (1990), Applegate *et al.* (1994), and Hundack *et al.* (1996).

(v) It must be allowed to manipulate a $PQ$-tree at will. An application known in the literature is the algorithm described in Applegate *et al.* (1994).

Constructing $PQ$-trees as a class template is not problematic. The addition of new types of nodes to the $PQ$-trees can be achieved by the concept of inheritance using an abstract base class for nodes. Polymorphism allows different treatment of nodes apart from the original template matching algorithm by keeping most of the functions as protected virtual member functions. If any functionality has to be provided in the template matching algorithm that differs from the original concern, it can be obtained in derived classes by overloading the corresponding virtual function. Since ignored nodes are used in several $PQ$-tree algorithms, the implementation already provides a set of tools for a straightforward adaption of ignored nodes.

For allowing all sorts of manipulations to the $PQ$-tree (e.g., as it is needed in the algorithm of Applegate *et al.* (1994)), the implementation of the $PQ$-tree data structure contains several functions that are called *constructive functions*. These functions enable the user to construct or manipulate $PQ$-trees and they handle the complete pointer and flag setup in the $PQ$-tree, always providing a proper $PQ$-tree. However, the constructive functions do not update a parent of a modified node, if the node is an interior child of a $Q$-node. Such a node does not have a valid pointer to its parent in general, and updating its parent needs linear time in the number of nodes in the $PQ$-tree. There are, however, exceptions from this rule. A few extra functions are included to meet the needs of algorithms that always have to update the parent pointer. They can be applied in algorithms that do not use the $PQ$-trees in the classical sense (see, e.g., Applegate *et al.* (1994)).

Figure 6.2 gives an overview on the design of the class structure that has been used. We distinguish in the implementation the following types of objects:

(i) Classes designed to handle information that is associated with leaves or internal nodes of the $PQ$-tree. These classes are dark grey shaded in Fig. 6.2.

(ii) Classes designed to represent the nodes of a $PQ$-tree. These classes are light grey shaded in Fig. 6.2.

(iii) Classes managing the data structure $PQ$-tree and providing the reduction algorithm REDUCE. These classes are left unshaded in Fig. 6.2.

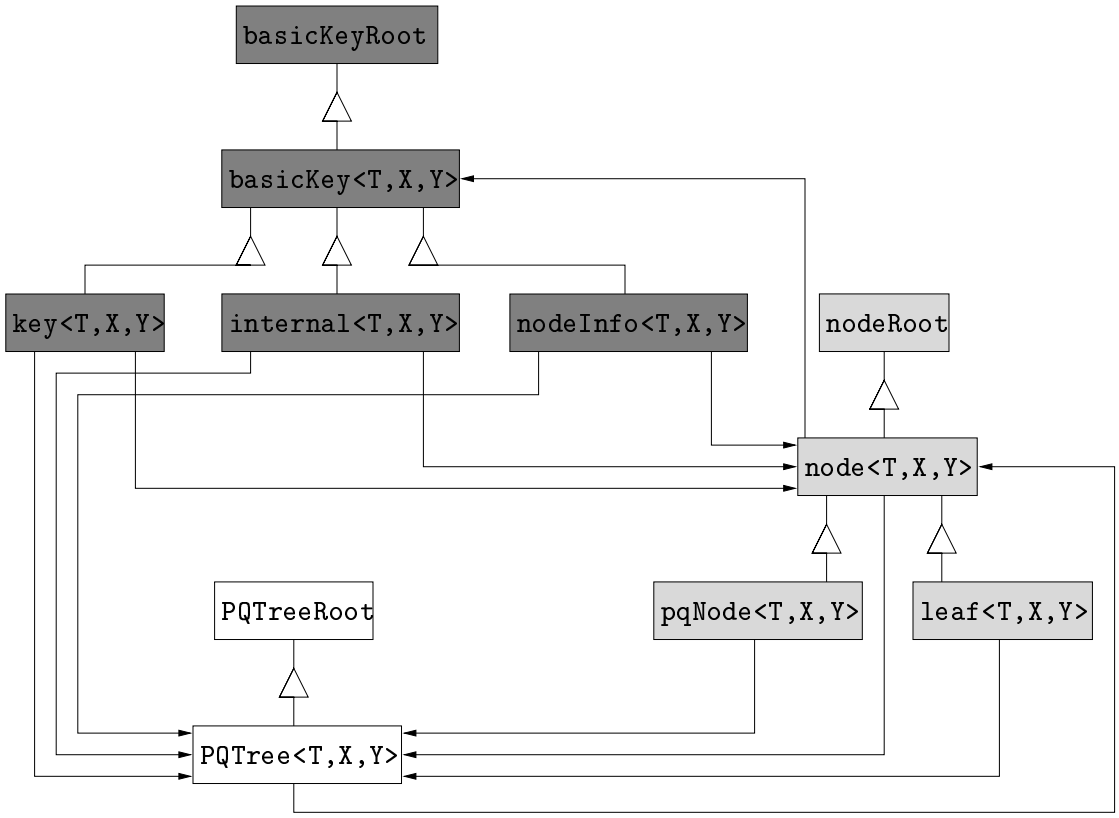



Figure 6.2: Overview on the design of the class structure for the implementation of the data structure $PQ$-tree

The implementation of a $PQ$-tree is organized as a class template with three types of parameters `T`, `X`, and `Y` that are used to represent different kinds of information. Information is stored in special container class templates and every instance of a container class is associated with exactly one object representing a node in the $PQ$-tree.

The abstract base class template `basicKey<T,X,Y>` is the base class for all container classes that carry information of the nodes. The class `key<T,X,Y>`, `nodeInfo<T,X,Y>`, and `internal<T,X,Y>` are derived classes of `basicKey<T,X,Y>`. These three classes carry information of nodes and an object of such a class is able to identify its corresponding node in constant time and vice versa. We distinguish three types of information:

- Information concerning only leaves are stored in the class template `key<T,X,Y>`. The parameter `T` belongs to this type of information and an object of type `T` is stored at an object of type `key<T,X,Y>`. An object of type `key<T,X,Y>` can be only assigned to a leaf.

- Information concerning any node are stored in the class template `nodeInfo<T,X,Y>`. The parameter `X` belongs to this type of information and an object of type `X` is stored at an object of type `nodeInfo<T,X,Y>`. An object of type `nodeInfo<T,X,Y>` can be assigned to any node.
- Information concerning only internal nodes are stored in the class template `internal<T,X,Y>`. The parameter `Y` is assigned to this type of information and an object of type `Y` is stored at an object of type `internal<T,X,Y>`. An object of type `internal<T,X,Y>` can be only assigned to an internal node.

For less sophisticated applications such as the consecutive ones property problem, the last two information types are usually not needed. However, it is expected to store information at every leaf, since by Definition 3.1 every leaf in a $PQ$-tree corresponds to exactly one element of the universal set $U$.

The class template `node<T,X,Y>` is the abstract base class for all nodes in the $PQ$-tree. This base class implements almost all functionality needed to handle the nodes. There exist two derived classes of `node<T,X,Y>`, the class template `leaf<T,X,Y>` and the class template `pqNode<T,X,Y>`. Objects of `leaf<T,X,Y>` correspond to leaves in a $PQ$-tree. Objects of `pqNode<T,X,Y>` correspond to internal nodes in the $PQ$-tree, and are used for $P$-nodes as well as for $Q$-nodes. The reason for using one class template for two different types of internal nodes is the similar treatment in basic operations such as replacing or exchanging nodes. The pointer treatment for the children differs only marginal between $P$- and $Q$-nodes.

The interface of the class template `PQTree<T,X,Y>` provides the reduction algorithm of Booth and Lueker (1976) and the constructive functions. A function for initializing a $PQ$-tree with a set $U$, a function that evokes a reduction with respect to a subset $S \subseteq U$ and a function for removing all allocated memory are public. All constructive functions and most of the functions needed to handle the template matching algorithm are virtual protected member functions. This allows necessary adaptions to different algorithmic needs.

The implementation is easily applied to the consecutive ones property problem. A user only needs to allocate an array `A` of `key<T,X,Y>` using the parameter `T` to represent the type of the elements of $U$. An instance of `PQTree<T,X,Y>` is allocated and initialized with `A`. Reducing a subset $S$ in $U$ is performed by calling the corresponding public function, specifying the elements of the subset $S$ in `A`. If the $PQ$-tree was reducible with respect to $S$, the frontier of the $PQ$-tree can be obtained by an extra call. If all reductions are complete, the class template `PQTree<T,X,Y>` does all necessary deallocation, except for the array `A`.

Ignored nodes can be adapted very easy by implementing a new class `B` that inherits the abstract class template `node<T,X,Y>` and a new class `C` that inherits the class template `PQTree<T,X,Y>`. Special virtual functions that perform no action in `PQTree<T,X,Y>` and are called by default within the template matching algorithms need to be overridden in `C`. These virtual functions perform basic operations as getting the next (nonignored) sibling to the left or right. Overriding these functions is simple as only the new type of node needs to

be considered. The ignored nodes can be introduced into a $PQ$-tree using the constructive functions.

The class template `PQTree<T,X,Y>` provides a file interface to the Tree Interface (see Leipert (1996)) that can be used to visualize the $PQ$-trees and all information that are stored at the nodes. New types of nodes are easily adapted to this interface by overriding a virtual protected member function that is called by default. The visualization of the $PQ$-trees is useful during the development of new implementations.

It is difficult to obtain a large number of "good" instances of the consecutive ones property problem in order to test the implementation. We observed that even when we used very large randomly generated instances, a lot of cases in the template matching algorithm remained untested. The $PQ$-trees that were produced by these instances were rather simple with only a very few $P$-nodes and even fewer $Q$-nodes. Hence, we decided to implement the planarity test of Booth and Lueker (1976) for testing the implementation. Two reasons led to this decision:

1. The set $U$ changes every time after leaves with respect to a vertex $v$ of a graph $G$ are reduced. Thus, the $PQ$-trees tend to be more complex, with a lot of $P$- and $Q$-nodes.

2. There exist good graph generators for both, planar and nonplanar graphs, e.g., the graph generator by Stamm-Wilbrandt (1996) that constructs maximal planar graphs and nonplanar graphs with one or two edges causing nonplanarity. These generators allow to produce a huge amount of test instances, and to verify not only the results of the planarity test but especially the template matching algorithm.

The implementation of the planarity test was provided by Buchheim (1997). Other implementations based on our implementation of $PQ$-trees are:

1. Physical mapping with end probes by Christof (1997), implemented by Oswald (1997).

2. Detection of obstructions to planarity by Karabeg (1990), implemented by Vollen (1997).

3. Detection of violated comb inequalities when solving TSP instances with branch and cut by Applegate *et al.* (1994), our implementation used by Störmer (1998).

4. The $c$-planarity test of Feng *et al.* (1995), implemented by Liebel (1998).

5. The algorithm PLANARIZE by Jayakumar *et al.* (1989) for computing a spanning planar subgraph, our implementation.

6. Level planar testing and embedding as discussed in this work.

## 6.2 Level Planar Embedder

The level planarity testing and embedding algorithm is encapsulated in a class `LevelPlanarTest`. Its public interface consists, besides a constructor and a destructor, of a function

```
int PlanarityTest(LeveledGraph* graph)
```

with a parameter of type `LeveledGraph*`, storing any level graph. The graph does not have to be proper, nor does it need to be connected. The class `LeveledGraph` can be used as an adapter to different graph representations.

An overview on the design of the class structure is given in Fig. 6.3. Except for the class `LeveledGraph`, all classes are aggregated by `LevelPlanarTest`. The class `planGraph` provides an interface to the embedding algorithm of Chiba *et al.* (1985) and was implemented by Buchheim (1997) using the $PQ$-tree implementation (see Section 6.1). The class `levelPlanGraph` has been derived from `planGraph` and works as an adapter to the class `planGraph`.

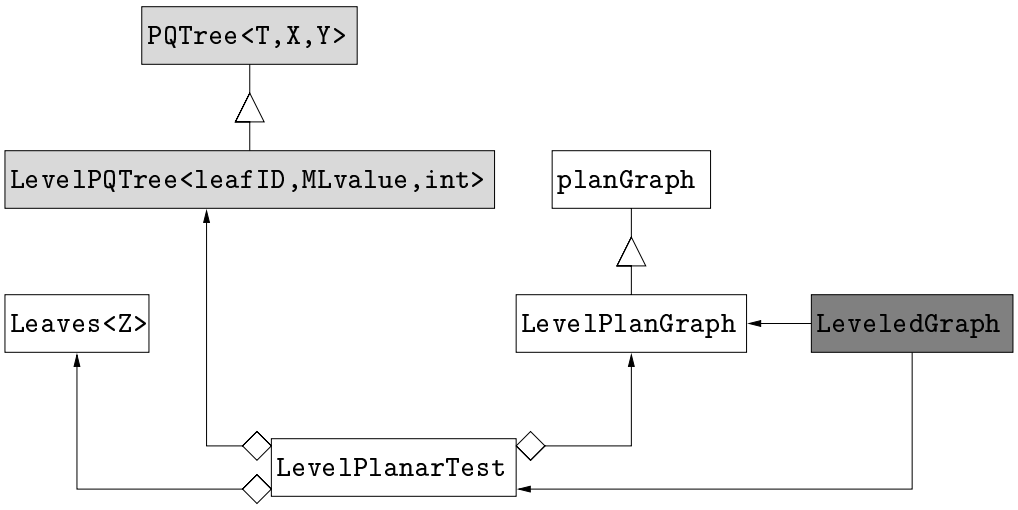



Figure 6.3: Overview on the design of the class structure for a level planar embedding algorithm.

The extended functionality of the $PQ$-trees that is needed in the level planar embedding algorithm is encapsulated in the class `LevelPQTree<leafID,MLvalue,int>`, a derived class from the class template `PQTree<T,X,Y>`. The parameter `T` associated with the information stored at the leaves of a $PQ$-tree is overloaded by a struct `leafID` that contains information on the edge corresponding to a leaf. The parameter `X` is overloaded by a class `MLvalue` that is associated with the information stored at the nodes (leaves as well as interior nodes) of a $PQ$-tree. The class `MLvalue` is designed to handle the ML-values of the nodes of a $PQ$-tree.

The class `LevelPlanarTest` allocates for every nonisolated source of a level graph $G = (V, E)$ an instance of `LevelPQTree<leafID,MLvalue,int>`. The access of the leaves in all $PQ$-trees is managed by using a class template `Leaves<Z>`, where the parameter `Z` is overloaded by `leveledGraphKey<MLvalue,int>*`. The class `leveledGraphKey<MLvalue,int>` is explained below.

The class `LevelPlanarTest` manages the complete level embedding algorithm as it has been presented in Chapter 5.3, controlling all three phases of LEVEL-PLANAR-EMBED, including all reductions and merge operations. Since all nodes of one $PQ$-tree have to be added into another $PQ$-tree in a merge operation, `LevelPlanarTest` and `LevelPQTree<leafID,MLvalue,int>` coexist in a tight binding.

A set of classes that is not shown in Fig. 6.3 accompanies the class `LevelPQTree<leafID,MLvalue,int>`. Figure 6.4 gives an overview on the design of the class structure associated with `LevelPQTree<leafID,MLvalue,int>`. Although not shown in the figure, the class `LevelPlanarTest` aggregates all of these classes.

The class templates `sinkFlag<leafID,X,Y>` and `Contact<leafID,X,Y>` are derived from the abstract base class template `node<T,X,Y>` and are associated with sink indicators and contacts. Instances of these classes are leaves in a $PQ$-tree and are considered as ignored nodes during the template matching algorithm. The class template `leveledGraphInfo<T,MLvalue,Y>`, derived from `nodeInfo<T,X,Y>` is a container class designed to carry an instance of type `MLvalue`, representing the ML-value of a node in the tree. The class template `leveledGraphKey<X,Y>` is derived from `graphKey<leafID,X,Y>` that itself is a derived class of `key<T,X,Y>`. A `leveledGraphKey<X,Y>` is a container class designed to carry an instance of type `leafID`, containing the information on the edge of the graph $G$ that is associated with an instance of `leaf<T,X,Y>`. It is also used to keep information on the vertex that is associated with an instance of `sinkFlag<leafID,X,Y>` or `Contact<leafID,X,Y>`. The usage of an intermediate class `graphKey<leafID,X,Y>` is conditioned by other implementations based on `PQTree<T,X,Y>`.

## 6.3 Preface to Code Examples

Before we present the three example procedures of our implementation, a short introduction to some basic pointer structure is given. Furthermore, we give an overview on some of the functions of the classes `node<T,X,Y>`, `nodeInfo<T,X,Y>`, `PQTree<T,X,Y>`, and `LevelPQTree<leafID,MLvalue,int>`. The overview is far from being complete. It lists only those functions that are used in the example procedures.

The children of a $P$-node $X$ are organized in a circular doubly linked list. The $P$-node contains a pointer to one of its children, called the *reference child*. Via this pointer, the node $X$ is able to access its children. Every child of the $P$-node $X$ has a valid pointer to its parent $X$.

The children $Y_1, Y_2, \ldots, Y_l$ of a $Q$-node $Y$ are organized in a doubly linked list as well. This list, however, is not circular. The nodes $Y_1$ and $Y_l$ are the two end nodes of the list. The

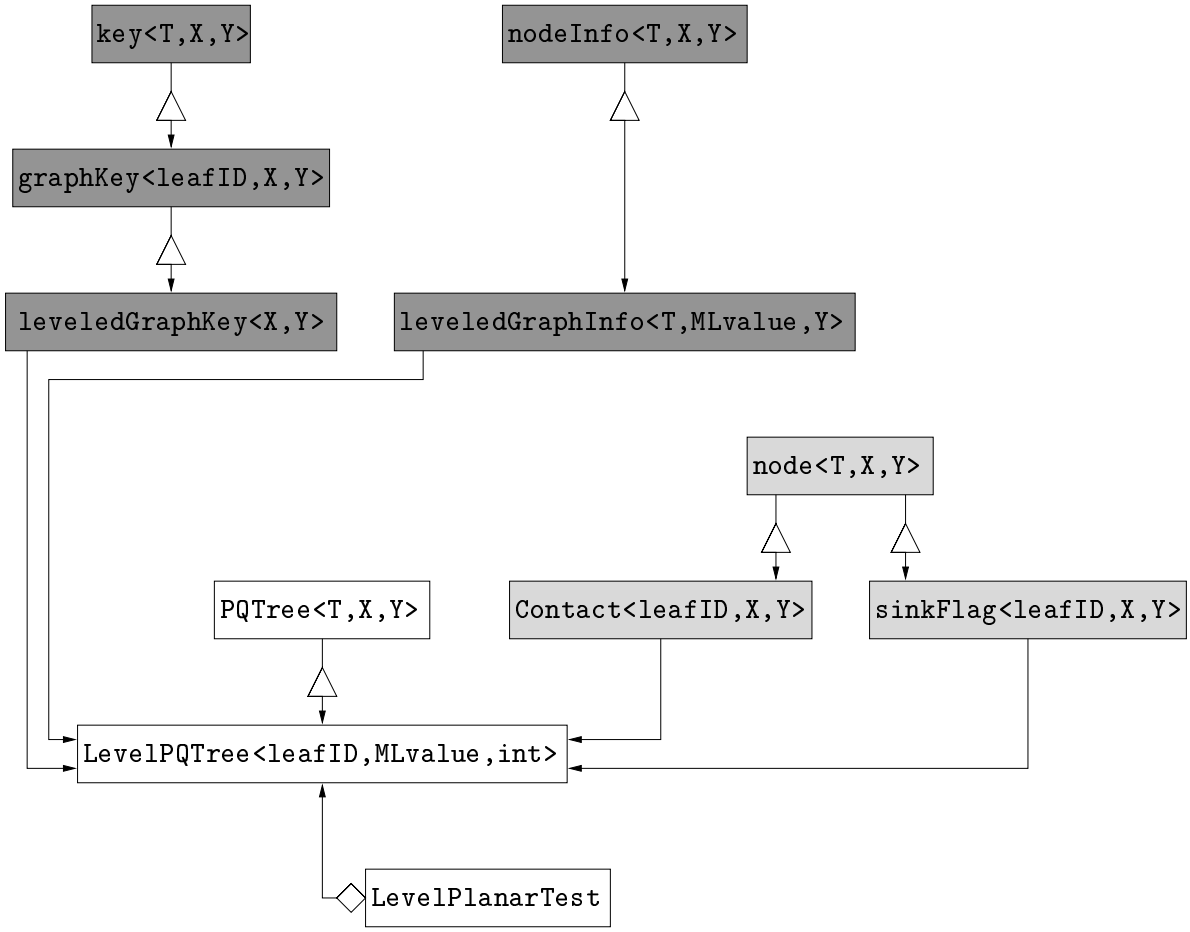



Figure 6.4: Overview on the design of the class structure associated with `LevelPQTree<leafID,MLvalue,int>`.

$Q$-node $Y$ has a pointer to $Y_1$ and a pointer to $Y_l$. The children $Y_1$ and $Y_l$ are said to be the endmost children of $Y$ and both have a valid pointer to their parent $Y$. All other children $Y_2, Y_3, \ldots, Y_{l-1}$, do not have a valid pointer to their parent. The pointers of these children have not been updated in order to achieve linear running time in the implementation of the $PQ$-trees. In general, the parent pointer of a node $Y_i$, $2 \leq i \leq l-1$, points to a node that does not exist anymore, and whose memory has been freed. Using such a pointer leads to runtime errors. However, it is very easy to recognize if a node $Z$ is an interior child of a $Q$-node, and therefore has no valid pointer to its parent. The node $Z$ becomes a child of a $Q$-node either in the application of the template matching algorithm or due to an external operation such as adding a node to the tree. In either case, it will be recognized if $Z$ is a child of a $Q$-node. Due to Corollary 3.5, we know that $Z$ stays a child of a $Q$-node during all subsequent applications of the template matching algorithm if no external operation such as removing a node is performed. Thus, we mark the node once as a child of a $Q$-node.

Every node has a pointer to its left sibling `_sibLeft` and a pointer to its right sibling `_sibRight`. However, left and right has essentially absolutely no meaning in a $PQ$-tree. We could also have used the terms yellow and green to denote the two different pointers. This becomes more evident since by construction of the template matching algorithm the following situation may occur. A node $X$ points to it sibling $Z$ via `_sibRight` and $Z$ itself also points to $X$ via `_sibRight`. Thus $Z$ is right sibling of $X$ and $X$ is right sibling of $Z$. The class templates `PQTree<T,X,Y>` and `node<T,X,Y>` cover the usual handling of the sibling pointers, provide iterators, and let a user not worry on the correct treatment of the sibling pointers.

In an implementation of a level planar embedder, the handling of siblings is slightly more complicated. Let $Y$ be a $Q$-node with children $Y_1, Y_2, \ldots, Y_l$. As mentioned in the proof of Theorem 4.26, the values $\mathrm{ML}(Y_i, Y_{i+1})$, $i = 1, 2, \ldots, l-1$, are not maintained at the parent $Y$ since the node $Y$ cannot be accessed by any of its internal children $Y_2, Y_3, \ldots, Y_{l-1}$ via a parent pointer. These ML-values are maintained at the children $Y_1, Y_2, \ldots, Y_l$. For every node $Y_i$, $2 \leq i \leq l-1$, we maintain two values $\mathrm{ML}(Y_{i-1}, Y_i)$, and $\mathrm{ML}(Y_i, Y_{i+1})$. At $Y_1$, and $Y_l$ we maintain the values $\mathrm{ML}(Y_1, Y_2)$, and $\mathrm{ML}(Y_{l-1}, Y_l)$, respectively. The values $\mathrm{ML}(Y_{i-1}, Y_i)$, and $\mathrm{ML}(Y_i, Y_{i+1})$ stored at $Y_i$ are associated with $Y_{i-1}$, and $Y_{i+1}$, respectively. We store the ML-value that corresponds to the left sibling `_sibLeft` of $Y_i$ in `_leftML` and ML-value that corresponds to the right sibling `_sibRight` of $Y_i$ in `_rightML`. This allows a clear identification. However, since left and right have no meaning in a $PQ$-tree, every modification of a $PQ$-tree involving a change at the ML-values is always accompanied by a constant number of case distinctions in order to modify the ML-values correctly.

As mentioned in the previous section, the ML-values stored at a node $Y$ are maintained in an object of type `MLvalue`. This object is stored in an instance of type `nodeInfo<leafID,MLvalue,int>` that is associated with the node $Y$. The `MLvalue` is kept in a member variable of `nodeInfo<leafID,MLvalue,int>` called `_userStructInfo`. The variable `_userStructInfo` is the only public member of `nodeInfo<leafID,MLvalue,int>` allowing direct access.

For the rest of Chapter 6 we make the following conventions for the documentation of the example code. When dereferencing functions of base classes, we omit the usage of the template parameters. The used template parameters are evident by context and this convention allows easier reading. Furthermore, we usually omit the parameters of the function, writing instead `(...)`. The used parameters are easily obtained from the displayed code fragments. Thus instead of writing, e.g.,

```
LevelPQTree<leafID,MLvalue,int>::CheckIgnoredSiblings(
node<leafID,MLvalue,int>* nodePtr, int LL, node<leafID,MLvalue,int>* left,
node<leafID,MLvalue,int>* right, int vertexNum, stack<edge*>* newEdges)
```

we write

```
LevelPQTree::CheckIgnoredSiblings(...) .
```

The code itself has been implemented using the literate programming system `noweb` by Ramsey (1992) which allowed a nice documentation of the program in LaTeX2e.

### 6.3.1 Functions of `node<T,X,Y>`

`int childCount()` returns the number of children of a node. Only valid for $P$-nodes.

`int childCount(int)` sets the number of children of a node, and returns the new value.

`node<T,X,Y>* getEndmost(int)` returns a pointer to either the left or the right endmost child of a $Q$-node. The side is specified by an integer. If the node is a $P$-node, the function returns a `NULL` pointer.

`node<T,X,Y>* getSib(int)` returns the adjacent sibling of a node either to its left or to its right. The side is specified by an integer. The returned sibling is allowed to be either ignored or nonignored.

`node<T,X,Y>* getNextSib(node<T,X,Y>*)` returns an adjacent sibling of a node. The returned sibling is specified by an adjacent sibling on the opposite side of the node. The returned sibling is allowed to be either ignored or nonignored.

`nodeInfo<T,X,Y>* getNodeInfo()` returns a pointer to the information container class associated with a node.

`node<T,X,Y>* parent()` returns a pointer to the parent of a node. If the node is an internal child of a $Q$-node, the returned value might be an invalid pointer.

`node<T,X,Y>* parent(node<T,X,Y>*)` sets a pointer to the parent of a node. The function returns the new pointer.

`int parentType()` returns an integer specifying if the parent of a node is a $P$- or a $Q$-node.

`int parentType(int)` sets an integer specifying if the parent of a node is a $P$- or a $Q$-node and returns the value.

`node<T,X,Y>* referenceChild()` returns a pointer to a child of a $P$-node. If the node is a $Q$-node, the function returns a `NULL` pointer.

`int setNodeInfo(nodeInfo<T,X,Y>*)` sets a pointer of type `nodeInfo<T,X,Y>` to the information container class associated with a node. The function returns 1 for success, and 0 otherwise.

`int status()` returns the status of a node. The status may be empty, full, partial, or any user defined status such as ignored.

`void status(int)` sets the status of a node.

### 6.3.2 Functions of `basicKey<T,X,Y>`

`void setNodePointer(node<T,X,Y>*)` sets a pointer to the node associated with the information class.

### 6.3.3 Functions of `PQTree<T,X,Y>`

`int addNodeToNewParent(node<T,X,Y>* parent,node<T,X,Y>* child)` adds a node to a new parent. The function returns 1 for success, and 0 otherwise.

`int addNodeToNewParent(...)` adds a node to a new parent specifying the left and the right sibling of the node. The input parameters are (in this order) `node<T,X,Y>* parent`, `node<T,X,Y>* child`, `node<T,X,Y>* leftBrother`, and `node<T,X,Y>* rightBrother`. No siblings need to be specified if the parent is a $P$-node. The function returns 1 for success, and 0 otherwise.

`node<T,X,Y>* clientNextSib(node<T,X,Y>* nodePtr,node<T,X,Y>* other)` is a virtual function returning an adjacent nonignored sibling of a node `nodePtr`. The returned sibling is specified by an adjacent nonignored sibling `other` on the opposite side of the node.

`node<T,X,Y>* clientLeftEndmost(node<T,X,Y>*)` is a virtual function returning a nonignored endmost child on the left side of a $Q$-node.

`node<T,X,Y>* clientRightEndmost(node<T,X,Y>*)` is a virtual function returning a nonignored endmost child on the right side of a $Q$-node.

`node<T,X,Y>* clientSibLeft(node<T,X,Y>*)` is a virtual function that returns the adjacent nonignored sibling of a node to its left.

`node<T,X,Y>* clientSibRight(node<T,X,Y>*)` is a virtual function that returns the adjacent nonignored sibling of a node to its right.

`void exchangeNodes(node<T,X,Y> *oldNode,node<T,X,Y> *newNode)` replaces a node `oldNode` by a node `newNode` in the $PQ$-tree.

`stack<node<T,X,Y>*>* partialChildrenStack(node<T,X,Y>*)` allows to get information on the partial children of a node.

`void removeChildFromSiblings(node<T,X,Y>*)` removes a node from the doubly linked list of its children. This does not affect the parent, unless the child was endmost child of a $Q$-node or child of a $P$-node.

`int template_P5(node<T,X,Y>*)` is a virtual function that performs the template matching P5.

`int template_Q2(node<T,X,Y> *nodePtr,int isRoot)` is a virtual function that performs the template matching Q2. The integer `isRoot` is a flag signalizing if `nodePtr` is the root of the pertinent subtree.

### 6.3.4 Functions of `LevelPQTree<leafID,MLvalue,int>`

`int CheckIgnoredSiblings(...)` is a function used for merge operations. The parameters are (in this order) `node<leafID,MLvalue,int>* nodePtr`, `int LL`, `node<leafID,MLvalue,int>* left`, `node<leafID,MLvalue,int>* right`, `int vertexNum`, `stack<edge*>* newEdges`. The function is used in a merge operation B or C, and is applied to a node `nodePtr` that is subject to the merge operation if `nodePtr` is a full node, i.e., the function applies Lemma 5.12. The parameters `left` and `right` denote the direct siblings to the left and to the right of `nodePtr`, respectively. The parameter `vertexNum` is an integer associated with the merge operation, and `newEdges` is a stack that is used to store for every sink indicator that is considered for edge augmentation the corresponding new edge. `LL` holds the LL-value of the smaller $PQ$-tree that is inserted as sibling to `nodePtr`. The return value is the number of new edges.

`int CheckIgnoredSiblings(...)` is a function used for merge operations. The parameters are (in this order) `node<leafID,MLvalue,int>* nodePtr`, `int LL`, `node<leafID,MLvalue,int>* dir`, `int vertexNum`, `stack<edge*>* newEdges`. The function is used in a merge operation D, and is applied to a node `nodePtr` that is subject to the merge operation, i.e., the function applies Lemma 5.11. The parameter `dir` denotes the direct sibling of `nodePtr` on the side where the root of the subtree has to be placed. The parameters `vertexNum`, `newEdges`, `LL` and the return value are defined as in the previous function.

`node<leafID,MLvalue,int>* CheckIgnoredSiblings(...)` is a function used for merge operations. The parameters are (in this order) `node<leafID,MLvalue, int>* nodePtr`, `int LL`, `node<leafID,MLvalue,int>* left`, `node<leafID, MLvalue,int>* right`, `int vertexNum`. The function is used in a merge operation B or C, and is applied to a node `nodePtr` that is subject to the merge operation if `nodePtr` is a partial node and we need to add a contact to the $PQ$-tree associated with the merge operation. All used parameters have the same functionality as described above. The function returns a contact of type `node<leafID,MLvalue,int>*`. The contact stores all necessary information.

`void setContactReductionValues(int)` informs a $PQ$-tree that a merge operation B or C has been applied. In the subsequent reduction after the merge operation, the $PQ$-tree applies Rules I or II to contacts that become children of the root of the pertinent subtree.

`void connectContact(...)` is a function used in templates Q2 and Q3. The parameters are (in this order) `node<leafID,MLvalue,int>** contact`, `node<leafID,MLvalue,int>** contactSib`, `node<leafID,MLvalue,int>** leftCheck`, `node<leafID,MLvalue,int>** rightCheck`. The function removes a sequence of contacts and its adjacent reference sequence from the tree. The

function `connectContact` adds for every sink indicator that is detected in the frontier of the reference sequence an edge directed towards the associated vertex of the corresponding contact. The function returns the direct siblings of the removed sequence. The pointers to these nodes are needed in order to perform correct update operations on the ML-values.

`void moveContact(...)` is function used in template Q2. The parameters are (in this order) `node<leafID,MLvalue,int>* contact`, `node<leafID, MLvalue,int>* contactSib`, `node<leafID,MLvalue,int>* Qnode`. The function moves a sequence of contacts that appears between a pertinent sequence to its new position according to Rule II.

## 6.4 Code Example I: `Merge`

The function `Merge(...)` is a protected member function of the class `LevelPlanarTest`. Given two $PQ$-trees `PQmax` and `PQapp` and a pertinent leaf in both $PQ$-trees, the function performs a merge operation on the trees. The function traverses the path from the pertinent leaf towards the root in the tree `PQmax` with the lower LL-value in order to find an appropriate position to place the tree `PQapp` with the larger LL-value into it.

The function applies the merge operations as described in 4.3.1. It does not check if the form corresponding to `PQapp` is singular and can be added within an interior face or cavity to the form corresponding to `PQmax`. This is done by the calling function (which corresponds to the function INSERT as described in 4.5).

If necessary, the function `Merge(...)` adds contacts to the root of the pertinent subtree as described in 5.2.5. `Merge(...)` does not reduce the pertinent leaves after successfully applying one of the five merge operations.

### 6.4.1 Input Values

`PQmax` is a pointer of type `LevelPQTree<leafID,MLvalue,int>`. It denotes the $PQ$-tree with the lower LL-value.

`PQapp` is a pointer of type `LevelPQTree<leafID,MLvalue,int>`. It denotes the $PQ$-tree with the higher LL-value. Thus the following inequality holds.

$$\mathrm{LL}(\texttt{PQmax}) \leq \mathrm{LL}(\texttt{PQapp}) \ .$$

`flagMax` is a pointer of type `leveledGraphKey<MLvalue,int>`. It denotes the information container of the pertinent leaf in the $PQ$-tree `PQmax` allowing the access of the pertinent leaf in constant time.

`flagApp` is a pointer of type `leveledGraphKey<MLvalue,int>`. It denotes the information container of the pertinent leaf in the $PQ$-tree `PQapp` allowing the access of the pertinent leaf in constant time.

`vertexNum` is an integer describing the vertex associated with the merge operation.

### 6.4.2 Return Values

1 if a merge operation was performed, 0 if no appropriate position in the $PQ$-tree `PQmax` for inserting `PQapp` was found. In the latter case, the graph is not level planar.

`PQmax` is a pointer of type `LevelPQTree<leafID,MLvalue,int>`. It denotes the $PQ$-tree with the lower LL-value. If the merge operation was successful, `PQmax` contains the complete tree `PQapp` as subtree. A reduction of the pertinent leaves associated with `flagMax` and `flagApp` is not performed by the function `Merge(...)`.

`PQapp` is a pointer of type `LevelPQTree<leafID,MLvalue,int>`. It denotes an empty $PQ$-tree with no nodes. Its memory is deallocated by the function `Merge(...)`.

`flagMax` is a pointer of type `leveledGraphKey<MLvalue,int>` that is left unchanged by `Merge(...)`.

`flagApp` is a pointer of type `leveledGraphKey<MLvalue,int>` that is left unchanged by `Merge(...)`.

### 6.4.3 Variables

`_nodePtr` is a pointer of type `node<leafID,MLvalue,int>`. It denotes a node on the path from the leaf corresponding to `flagMax` to the root of `PQmax`.

`_parent` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the parent of `_nodePtr`.

`_sibling` is a pointer of type `node<leafID,MLvalue,int>`. It denotes a sibling of `_nodePtr`, if `_nodePtr` is a child of a $Q$-node.

`_contact` is a pointer of type `node<leafID,MLvalue,int>`. It denotes a new contact that eventually has to be introduced during the merge operations B or C.

`_newQnode` is a pointer of type `pqNode<leafID,MLvalue,int>`. It denotes a new $Q$-node that is introduced into the $PQ$-tree `PQmax` by the merge operation.

`_nodeInfoPtr` is a pointer of type `leveledGraphInfo<leafID,int>`. In case a new $Q$-node `_newQnode` is allocated by `Merge(...)`, the `_newQnode` needs to be equipped with an information container class of type `leveledGraphInfo<leafID,int>`.

`_found` is an integer that is initialized with 0 and is set to 1 if the merge operation was performed successfully.

`_leftML` is an integer holding the ML-value between `_nodePtr` and the sibling of `_nodePtr` that is accessible via the left sibling pointer of `_nodePtr`.

`_rightML` is an integer holding the ML-value between `_nodePtr` and the sibling of `_nodePtr` that is accessible via the right sibling pointer of `_nodePtr`.

`_nodeID` is an integer used to determine the identification number of `_newQnode`.

`_onlyLeaf` is an integer that is 1 as long as `_nodePtr` holds a full node. The value of `_onlyLeaf` is changed to 0 if `_nodePtr` becomes partial. In case that `_onlyLeaf` is 1 during a merge operation B or C, Lemma 5.12 is applied and no contact needs to be added.

`_dummyMLvalue` is an instance of the class `MLvalue` storing the ML-values of `_newQnode`.

### 6.4.4 Code-Body

The function `Merge(...)` executes a `while`-loop that traverses the path from the leaf associated with `flagMax` towards the root of `PQmax`. The variable `_nodePtr` denotes the current node on this path. The function exits from the while loop if one of the following two cases applies.

- A merge operation can be performed on `_nodePtr` and its `_parent`.
- `_nodePtr` is an interior child of a $Q$-node and neither merge operation C nor D can be performed. According to the proof of Theorem 4.26, the graph $G$ is not level planar.

If a merge operation has been applied successfully, function `Merge(...)` exits the `while` loop returning 1. If no merge operation was performed successfully, `Merge(...)` exits the `while` loop returning 0.

⟨Merge⟩≡

```
/*****************************************************************************
                                    Merge
*****************************************************************************/

int LevelPlanarTest::Merge(LevelPQTree* PQmax,
                           LevelPQTree* PQapp,
                           leveledGraphKey<MLvalue,int>* flagMax,
                           leveledGraphKey<MLvalue,int>* flagApp,
                           int vertexNum)
{
   node<leafID,MLvalue,int>*      _nodePtr = flagMax->nodePointer();
```

```
    node<leafID,MLvalue,int>*      _parent      = NULL;
    node<leafID,MLvalue,int>*      _sibling     = NULL;
    pqNode<leafID,MLvalue,int>*    _newQnode    = NULL;
    node<leafID,MLvalue,int>*      _contact     = NULL;
    leveledGraphInfo<leafID,int>*  _nodeInfoPtr = NULL;
    int                            _found       = 0;
    int                            _leftML      = 0;
    int                            _rightML     = 0;
    int                            _nodeID      = 0;
    int                            _onlyLeaf    = 1;
    MLvalue                        _dummyMLvalue;


    _nodeID = max(PQmax->_identificationNumber,PQapp->_identificationNumber);

    while (!_found)
    {
        _leftML = _nodePtr->getNodeInfo()->_userStructInfo._leftML;
        _rightML= _nodePtr->getNodeInfo()->_userStructInfo._rightML;
        _parent = _nodePtr->parent();
        if (_parent == NULL)
        {
            ⟨Merge: Perform merge operation E⟩
        }
        else if (_nodePtr->parentType() == P_NODE)
        {
            ⟨Merge: _nodePtr is a child of P-node⟩
        }
        else if (PQmax->clientSibLeft(_nodePtr) == NULL ||
                 PQmax->clientSibRight(_nodePtr) == NULL)
        {
            ⟨Merge: _nodePtr is an endmost child of Q-node⟩
        }
        else
        {
            ⟨Merge: _nodePtr is an interior child of Q-node⟩
        }
        if (_found)
        {
            Update(PQmax,PQapp,flagMax,flagApp);
            return 1;
        }
    }
    return 0;
}
```

### 6.4.5 Processing _nodePtr

The application of one of the merge operations is dependent on the situation found at _nodePtr. The following three code fragments examine if _nodePtr is a child of a $P$-node, an endmost child of $Q$-node, or an internal child of a $Q$-node. Depending on the case that applies, we either need to proceed further up the tree, or a valid decisions can be made on the merge operation that has to be applied, or we detect that the graph $G$ is not level planar.

#### _nodePtr is a child of $P$-node

The parent of _nodePtr is a $P$-node. If

$$\text{ML}(\texttt{\_parent}) < \text{LL}(\texttt{PQapp})$$

holds, merge condition A applies. If

$$\text{ML}(\texttt{\_parent}) \geq \text{LL}(\texttt{PQapp})$$

holds, the tree PQapp cannot be merged into PQmax at this position. We continue moving up towards the root of PQmax in order to find an appropriate location for merging the smaller $PQ$-tree into the larger one.

⟨Merge: _nodePtr *is a child of $P$-node*⟩≡
```
if (_parent->getNodeInfo()->_userStructInfo._PnodeML < PQapp->_LL)
{
  ⟨Merge: Perform merge operation A⟩
}
else
{
  if (_parent->childCount() > 1)
      _onlyLeaf = 0;
  _nodePtr = _parent;
}
```

#### _nodePtr is an endmost child of $Q$-node

The parent of _nodePtr is a $Q$-node and _nodePtr is an endmost child of its parent. The code fragment checks if merge condition B applies. In case condition B does not apply, we proceed further up the tree.

⟨Merge: _nodePtr *is an endmost child of $Q$-node*⟩≡
```
if (_leftML != 0 && _rightML != 0)
    cerr << "ERROR in MERGE: endmost child "
         << "with illegal NONZERO ML-values detected." << endl
         << "Appending large PQ-Tree: " << PQmax->_ID
```

```
          << " and small PQ-Tree " << PQapp->_ID << endl;
  else if ((_leftML != 0 && _leftML < PQapp->_LL) ||
           (_rightML != 0 && _rightML < PQapp->_LL))
  {
      ⟨Merge: Perform merge operation B⟩
  }
  else
  {
      if (PQmax->clientSibLeft(_nodePtr) != NULL ||
          PQmax->clientSibRight(_nodePtr) != NULL)
          _onlyLeaf = 0;
      _nodePtr = _parent;
  }
```

**_nodePtr is an interior child of $Q$-node**

The parent of _nodePtr is a $Q$-node and _nodePtr is an interior child of its parent. Thus, _nodePtr does not know its parent. If it is not possible to apply either merge operation C or D to _nodePtr, the graph $G$ is not level planar.

⟨Merge: _nodePtr *is an interior child of $Q$-node*⟩≡

```
  if (_leftML < PQapp->_LL && _rightML < PQapp->_LL)
  {
      ⟨Merge: Perform merge operation C⟩
  }
  else if (_leftML < PQapp->_LL)
  {
      ⟨Merge: Perform merge operation D, left⟩
  }
  else if (_rightML < PQapp->_LL)
  {
      ⟨Merge: Perform merge operation D, right⟩
  }
  else
      return 0;
```

### 6.4.6 Merge Operations

#### Merge Operation A

The parent of _nodePtr is a $P$-node such that

$$\mathrm{ML}(\texttt{\_parent}) < \mathrm{LL}(\texttt{PQapp}) \ .$$

The root of PQapp is added to _nodePtr. The Merge(...) function finishes successfully.

⟨Merge: *Perform merge operation A*⟩≡
```
PQmax->addNodeToNewParent(_parent,PQapp->_root,_nodePtr,NULL);
_found = 1;
```

### Merge operation B

Node _nodePtr is an endmost child of a $Q$-node and either

$$\mathrm{ML}(\texttt{\_nodePtr},\texttt{\_nodePtr->getSib(LEFT)}) < \mathrm{LL}(\texttt{PQapp})$$

or

$$\mathrm{ML}(\texttt{\_nodePtr},\texttt{\_nodePtr->getSib(RIGHT)}) < \mathrm{LL}(\texttt{PQapp})$$

holds. The node _nodePtr is replaced by a new $Q$-node _newQnode. The _newQnode gets as children _nodePtr, the root of PQapp, and if necessary a _contact. The Merge(...) function then finishes successfully.

In case that _nodePtr is a full node, the value of _onlyLeaf is 1, and Lemma 5.12 is applied. This is done by calling the member function LevelPQTree::CheckIgnoredSiblings(...) of PQmax. One of the parameters is _newEdges that is a class member of LevelPlanarTest collecting new edges that are added to the graph in order to construct a hierarchy.

In case that _nodePtr is not a full node, the value of _onlyLeaf is 0, and we possibly have to add a contact that is associated with this merge operation. Calling the member function LevelPQTree::CheckIgnoredSiblings(...) of PQmax allocates a new contact, and determines the reference sequence of the contact, equipping it with the necessary information.

The code differs whether _nodePtr is the left endmost or the right endmost child of its _parent.

⟨Merge: *Perform merge operation B*⟩≡
```
if (_onlyLeaf)
    PQmax->CheckIgnoredSiblings(_nodePtr,PQapp->_LL,
                                _nodePtr->getSib(LEFT),
                                _nodePtr->getSib(RIGHT),
                                vertexNum,_newEdges);
else
    _contact = PQmax->CheckIgnoredSiblings(_nodePtr,PQapp->_LL,
                                           _nodePtr->getSib(LEFT),
                                           _nodePtr->getSib(RIGHT),
                                           vertexNum);

_newQnode = new pqNode<leafID,MLvalue,int>(_nodeID++,Q_NODE,EMPTY);
PQmax->exchangeNodes(_nodePtr,_newQnode);
PQmax->addNodeToNewParent(_newQnode,_nodePtr);
```

```
PQapp->removeChildFromSiblings(PQapp->_root);
if (_leftML != 0)
    PQmax->addNodeToNewParent(_newQnode,PQapp->_root,NULL,_nodePtr);
else
    PQmax->addNodeToNewParent(_newQnode,PQapp->_root,_nodePtr,NULL);
```

The node `_nodePtr` is now the right endmost child (if `_leftML != 0`) or the left endmost child (if `_rightML != 0`) of `_newQnode` and `PQapp->_root` is the left or right endmost child of `_newQnode`. In case that a contact has been created by a call of `LevelPQTree::CheckIgnoredSiblings(...)`, we add the `_contact` as an endmost child to `_newQnode` next to `PQapp->_root`.

If `_nodePtr` is a $Q$-node having already contacts, the Rules I or II described in Section 5.2.5 have to be applied to these contacts during the application of template Q2. However, the template Q2 removes every contact found in the pertinent sequence, unless the $PQ$-tree `PQmax` is informed to apply Rules I or II to `_nodePtr`. A function `LevelPQTree::setContactReductionValues(int)` is called. The values `LEFT` or `RIGHT` describe the side of `_newQnode`, where the root of `PQapp` has been added as an endmost child. If a sequence of contacts that have been children of `_nodePtr` is found within the pertinent sequence as children of `_newQnode` after reducing the $PQ$-tree, these contacts are moved to the prescribed endmost side applying Rule II. The function `LevelPQTree::setContactReductionValues(int)` is called independent on the introduction of a new `_contact`. By Definition 5.24 the current merge operation and the merge operation associated with any contact that is a child of `_nodePtr` are concatenations.

⟨Merge: *Perform merge operation B*⟩+≡

```
if (_contact)
{
    if (_leftML)
        PQmax->addNodeToNewParent(_newQnode,_contact,NULL,PQapp->_root);
    else
        PQmax->addNodeToNewParent(_newQnode,_contact,PQapp->_root,NULL);
}
if (_leftML)
    PQmax->setContactReductionValues(LEFT);
else
    PQmax->setContactReductionValues(RIGHT);

_nodeInfoPtr = new leveledGraphInfo<leafID,int>(_dummyMLvalue);
_newQnode->setNodeInfo(_nodeInfoPtr);
_nodeInfoPtr->setNodePointer((node<leafID,MLvalue,int>*) _newQnode);
_newQnode->getNodeInfo()->_userStructInfo.set(_leftML,_rightML,0);
_newQnode->getNodeInfo()->_userStructInfo._vertex =
    _nodePtr->getNodeInfo()->_userStructInfo._vertex;
if (_leftML)
    PQapp->_root->getNodeInfo()->_userStructInfo._rightML = _leftML;
```

```
else
    PQapp->_root->getNodeInfo()->_userStructInfo._leftML = _rightML;
_found = 1;
```

### Merge Operation C

The node `_nodePtr` is an interior child of a $Q$-node and the following inequalities both hold.

$$\begin{aligned} \text{ML}(\texttt{\_nodePtr,\_nodePtr->getSib(LEFT)}) &< \text{LL}(\texttt{PQapp}) \ , \\ \text{ML}(\texttt{\_nodePtr,\_nodePtr->getSib(RIGHT)}) &< \text{LL}(\texttt{PQapp}) \ . \end{aligned}$$

The node `_nodePtr` is replaced by a new $Q$-node `_newQnode`. The `_newQnode` gets as children `_nodePtr`, the root of `PQapp`, and if necessary a `_contact`. The `Merge(...)` function then finishes successfully.

In case that `_nodePtr` is a full node, the value of `_onlyLeaf` is 1, and Lemma 5.12 is applied. This is done by calling the member function `LevelPQTree::CheckIgnoredSiblings(...)` of `PQmax`. One of the parameters is `_newEdges` that is a class member of `LevelPlanarTest` collecting all new edges that have to be added to the graph in order to construct a hierarchy.

In case that `_nodePtr` is not a full node, the value of `_onlyLeaf` is 0, and we possibly have to add a contact that is associated with this merge operation. Calling the member function `LevelPQTree::CheckIgnoredSiblings(...)` of `PQmax` allocates a new contact, and determines the reference sequence of the contact, equipping it with the necessary information.

⟨`Merge`*: Perform merge operation C*⟩≡

```
if (_onlyLeaf)
    PQmax->CheckIgnoredSiblings(_nodePtr,PQapp->_LL,
                                _nodePtr->getSib(LEFT),
                                _nodePtr->getSib(RIGHT),
                                vertexNum,_newEdges);
else
    _contact = PQmax->CheckIgnoredSiblings(_nodePtr,PQapp->_LL,
                                           _nodePtr->getSib(LEFT),
                                           _nodePtr->getSib(RIGHT),
                                           vertexNum);

_newQnode = new pqNode<leafID,MLvalue,int>(_nodeID++,Q_NODE,EMPTY);
PQmax->exchangeNodes(_nodePtr,_newQnode);
PQapp->removeChildFromSiblings(PQapp->_root);
PQmax->addNodeToNewParent(_newQnode,_nodePtr);
PQmax->addNodeToNewParent(_newQnode,PQapp->_root,_nodePtr,NULL);
```

The node `_nodePtr` is the new left endmost child of `_newQnode`, and `PQapp->_root` is the right endmost child of `_newQnode`. In case that a contact has been created by a call of

`LevelPQTree::CheckIgnoredSiblings(...)` we add the `_contact` as right endmost child to `_newQnode` next to the root of `PQapp`.

If `_nodePtr` is a $Q$-node having already contacts, the Rules I or II as described in Section 5.2.5 have to be applied to these contacts during the application of template Q2. However, the template Q2 removes every contact found in the pertinent sequence, unless the $PQ$-tree `PQmax` is informed to apply Rules I or II to `_nodePtr`. A function `LevelPQTree::setContactReductionValues(int)` is called. The value `RIGHT` holds the side of `_newQnode`, where the root of `PQapp` has been added as an endmost child. If a sequence of contacts that have been children of `_nodePtr` is found within the pertinent sequence as children of `_newQnode` after reducing the $PQ$-tree, these contacts are moved to the right endmost side applying Rule II. The function `LevelPQTree::setContactReductionValues(int)` is called independent on the introduction of a new `_contact`. By Definition 5.24 the current merge operation and the merge operation associated with any contact that is a child of `_nodePtr` are concatenations.

⟨`Merge`*: Perform merge operation C*⟩+≡

```
if (_contact)
    PQmax->addNodeToNewParent(_newQnode,_contact,PQapp->_root,NULL);

PQmax->setContactReductionValues(RIGHT);

_nodeInfoPtr = new leveledGraphInfo<leafID,int>(_dummyMLvalue);
_newQnode->setNodeInfo(_nodeInfoPtr);
_nodeInfoPtr->setNodePointer((node<leafID,MLvalue,int>*) _newQnode);
_newQnode->getNodeInfo()->_userStructInfo.set(_leftML,_rightML,0);
_newQnode->getNodeInfo()->_userStructInfo._vertex
    = _nodePtr->getNodeInfo()->_userStructInfo._vertex;
_nodePtr->getNodeInfo()->_userStructInfo._leftML = 0;
if (_rightML < _leftML)
{
    _nodePtr->getNodeInfo()->_userStructInfo._rightML = _leftML;
    _PQapp->_root->getNodeInfo()->_userStructInfo._leftML = _leftML;
}
else
    _PQapp->_root->getNodeInfo()->_userStructInfo._leftML =_rightML;
_found = 1;
```

### Merge Operation D, left

The node `_nodePtr` is an interior child of a $Q$-node and the following inequalities both hold.

$$\begin{aligned} \mathrm{ML}(\texttt{\_nodePtr},\texttt{\_nodePtr->getSib(LEFT)}) &< \mathrm{LL}(\texttt{PQapp}) \ , \\ \mathrm{ML}(\texttt{\_nodePtr},\texttt{\_nodePtr->getSib(RIGHT)}) &\geq \mathrm{LL}(\texttt{PQapp}) \ . \end{aligned}$$

The root of PQapp is placed between _nodePtr and its left sibling. The application of Lemma 5.11 associated with the merge operation D is performed by calling the member function LevelPQTree::CheckIgnoredSiblings(...) of PQmax. The function Merge(...) then finishes successfully.

⟨Merge: *Perform merge operation D, left*⟩≡

```
PQmax->CheckIgnoredSiblings(_nodePtr,PQapp->_LL,
                            _nodePtr->getSib(LEFT),
                            vertexNum,_newEdges);

_sibling = _nodePtr->getSib(LEFT);
PQapp->removeChildFromSiblings(PQapp->_root);
PQmax->addNodeToNewParent(NULL,PQapp->_root,_sibling,_nodePtr);
PQapp->_root->getNodeInfo()->_userStructInfo._leftML = _leftML;
PQapp->_root->getNodeInfo()->_userStructInfo._rightML = _leftML;
PQapp->_root->parentType(Q_NODE);
_found = 1;
```

### Merge Operation D, right

The node _nodePtr is an interior child of a $Q$-node and the following inequalities both hold.

$$\begin{aligned} \text{ML}(\texttt{\_nodePtr},\texttt{\_nodePtr->getSib(RIGHT)}) &< \text{LL}(\texttt{PQapp}) \ , \\ \text{ML}(\texttt{\_nodePtr},\texttt{\_nodePtr->getSib(LEFT)}) &\geq \text{LL}(\texttt{PQapp}) \ . \end{aligned}$$

The root of PQapp is placed between _nodePtr and its right sibling. The application of Lemma 5.11 associated with the merge operation D is performed by calling the member function LevelPQTree::CheckIgnoredSiblings(...) of PQmax. The function Merge(...) then finishes successfully.

⟨Merge: *Perform merge operation D, right*⟩≡

```
PQmax->CheckIgnoredSiblings(_nodePtr,PQapp->_LL,
                            _nodePtr->getSib(RIGHT),
                            vertexNum,_newEdges);

_sibling = _nodePtr->getSib(RIGHT);
PQapp->removeChildFromSiblings(PQapp->_root);
PQmax->addNodeToNewParent(NULL,PQapp->_root,_nodePtr,_sibling);
PQapp->_root->getNodeInfo()->_userStructInfo._leftML =  _rightML;
PQapp->_root->getNodeInfo()->_userStructInfo._rightML = _rightML;
PQapp->_root->parentType(Q_NODE);
_found = 1;
```

**Merge Operation E**

The node `_nodePtr` is a pointer to the root of the $PQ$-tree `PQmax`. Thus merge condition E applies. A new $Q$-node `_newQnode` is allocated. The roots of both trees `PQmax` and `PQapp` are added as children to `_newQnode`. The new $Q$-node then becomes root of `PQmax`. The function `Merge(...)` then finishes successfully.

⟨`Merge`*: Perform merge operation E*⟩≡

```
_newQnode = new pqNode<leafID,MLvalue,int>(_nodeID++,Q_NODE,EMPTY);
PQmax->exchangeNodes(_nodePtr,_newQnode);
PQmax->_root = _newQnode;
PQapp->removeChildFromSiblings(PQapp->_root);
PQmax->addNodeToNewParent(_newQnode,_nodePtr);
PQmax->addNodeToNewParent(_newQnode,PQapp->_root,_nodePtr,NULL);

_nodeInfoPtr = new leveledGraphInfo<leafID,int>(_dummyMLvalue);
_newQnode->setNodeInfo(_nodeInfoPtr);
_nodeInfoPtr->setNodePointer((node<leafID,MLvalue,int>*) _newQnode);
_newQnode->getNodeInfo()->_userStructInfo.set(0,0,0);
_found = 1;
```

## 6.5 Code Example II: `Template P5`

The function `template_P5(...)` is a protected member function of the class `LevelPQTree<leafID,MLvalue,int>`. The function overloads the virtual protected function `PQTree::template_P5(...)`. The class `LevelPQTree<leafID,MLvalue,int>` overloads the function in order

- to guarantee correct application of template P5, and
- to update ML-values.

The function collects pointers to all nodes of the $PQ$-tree that are affected by the template P5. After calling the function `PQTree::template_P5(...)`, it applies all necessary update operations.

### 6.5.1 Input Values

`nodePtr` is a pointer of type `node<leafID,MLvalue,int>` and the function `template_P5(...)` tries to apply the template P5 to `nodePtr` and its children.

### 6.5.2 Return Values

1 if the the template P5 was applied successfully to `nodePtr` and its children, 0 otherwise.

### 6.5.3 Variables

`_partialChild` is a pointer of type `node<leafID,MLvalue,int>`. It denotes a partial child of `nodePtr` before template P5 is applied. By construction, `_partialChild` is a $Q$-node.

`_fullNode` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the endmost full node of `_partialChild` after template P5 has been applied.

`_emptyNode` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the endmost empty node of `_partialChild` after template P5 has been applied.

`_fakeEndmost` is a pointer of type `node<leafID,MLvalue,int>`. It denotes a fake endmost empty node of `_partialChild` after template P5 has been applied. The pointer is needed to cover a special case. During the application of template P5, the empty children of `nodePtr` remain children of `nodePtr`, and `nodePtr` is placed at the empty end of `_partialChild`. This prevents the template P5 from scanning the empty children. If `nodePtr` is a $P$-node having only one empty child the function call `PQTree::template_P5(nodePtr)` does not add `nodePtr` at the empty end of `_partialChild`. It places the only empty child of `nodePtr` at the empty end of `_partialChild`. The template P5 applies this strategy in order to ensure that only proper $PQ$-trees are constructed. However, in our level planar embedding algorithm, `nodePtr` may have ignored children. The function `PQTree::template_P5(...)` notices that `nodePtr` has just one empty child, and it adds an arbitrary child (due to the technique of not scanning the empty children this might be an ignored node) to the empty end of `_partialChild`. We access this child via the pointer `_fakeEndmost`, and apply a correction step.

`_nodeInfoF` is a pointer of type `leveledGraphInfo<leafID,int>`. In case `_fullNode` is a new $P$-node that has been allocated by `PQTree::template_P5(...)` in order to gather all full children of `nodePtr`, the `_fullNode` needs to be equipped with an information container class of type `leveledGraphInfo<leafID,int>`.

`_dummyMLvalue` is an instance of type `MLvalue` and stores the information of `_fullNode` if `_fullNode` was allocated by `PQTree::template_P5(...)`.

`_ml` is an integer storing the ML-value of the $P$-node `nodePtr`. This value will be the ML-value between a new empty endmost child and the former empty endmost child of `_partialChild` as well as between the a new full endmost child of `_partialChild` and the former full endmost child of `_partialChild`. It is also the ML-value that is stored in `_dummyMLvalue`. Although the full nodes are removed from the tree after successfully completing the reduction, the ML-values of the full children need to be updated properly since the PML- and the QML-values need to be computed or updated.

`_fullCount` is an integer denoting the number of full children of `nodePtr` before template P5 is applied. This number may be 0.

`_emptyCount` is an integer denoting the number of empty children of `nodePtr` before template P5 is applied. This number may be 0.

`_side` is an integer denoting the full end of `_partialChild`.

### 6.5.4 Code-Body

The function `template_P5(...)` keeps certain information before executing the function `PQTree::template_P5(...)`. If the template reduction P5 was performed successfully, function `template_P5(...)` applies the necessary update operations.

⟨template_P5⟩≡

```
/*************************************************************************
                                template_P5
*************************************************************************/

int LevelPQTree::template_P5(node<leafID,MLvalue,int>*  nodePtr)
{
    node<leafID,MLvalue,int>*      _partialChild  = NULL;
    node<leafID,MLvalue,int>*      _fullNode      = NULL;
    node<leafID,MLvalue,int>*      _emptyNode     = NULL;
    node<leafID,MLvalue,int>*      _fakeEndmost   = NULL;
    leveledGraphInfo<leafID,int>*  _nodeInfoF     = NULL;
    int                            _ml            = 0;
    int                            _emptyCount    = 0;
    int                            _fullCount     = 0;
    int                            _side          = 0;
    MLvalue                        _dummyMLvalue;

    ⟨template_P5: Preparation⟩
    if (PQTree<leafID,MLvalue,int>::template_P5(nodePtr))
    {
        ⟨template_P5: Update⟩
    }
    else
        return 0;
}
```

### 6.5.5 Preparation of Template P5

Before calling the function `PQTree::template_P5(...)` for applying the template P5 to `nodePtr`, the following code fragment collects information on:

- the ML-value of `nodePtr` in `_ml`,
- a partial child of `nodePtr` in `_partialChild`,
- the number of full children of `nodePtr` in `_fullCount`, and
- the number of empty children of `nodePtr` in `_emptyCount`.

⟨`template_P5`*: Preparation*⟩≡
```
_ml = nodePtr->getNodeInfo()->_userStructInfo._PnodeML;
partialChildrenStack(nodePtr)->startAtBottom();
if (!partialChildrenStack(nodePtr)->readLast())
    _partialChild = partialChildrenStack(nodePtr)->readNext();
_fullCount = fullChildrenStack(nodePtr)->count();
_emptyCount = nodePtr->childCount() - partialChildrenStack(nodePtr)->count()
              - _fullCount;
```

### 6.5.6 Update after Template P5

After `PQTree::template_P5(...)` has been successfully applied, the necessary updates are performed. The node `_partialChild` now occupies the position of `nodePtr` in the *PQ*-tree. The full and empty children of `nodePtr` are gathered at the full and empty end of `_partialChild`. We first determine the full and the empty endmost child of `_partialChild`.

⟨`template_P5`*: Update*⟩≡
```
if (clientLeftEndmost(_partialChild)->status() == FULL)
{
    _fullNode = clientLeftEndmost(_partialChild);
    _emptyNode = clientRightEndmost(_partialChild);
    _side = LEFT;
}
else
{
    _fullNode = clientRightEndmost(_partialChild);
    _emptyNode = clientLeftEndmost(_partialChild);
    _side = RIGHT;
}
```

If the number of full children of `nodePtr` (stored in `_fullCount`) was at least 1, the `_partialChild` has a new full endmost child and the ML-values of this child have to be set or updated. Since `_partialChild` occupies the position of `nodePtr` in the *PQ*-tree, the ML-values of `_partialChild` are updated as well. The case where `_partialChild` has a new empty endmost child is considered in the next code fragment.

⟨`template_P5`*: Update*⟩+≡
```
if (_fullCount > 0)
```

```
{
    ⟨template_P5: Update new full endmost node⟩
}
_partialChild->getNodeInfo()->_userStructInfo._leftML =
    nodePtr->getNodeInfo()->_userStructInfo._leftML;
_partialChild->getNodeInfo()->_userStructInfo._rightML =
    nodePtr->getNodeInfo()->_userStructInfo._rightML;
```

If the number of empty children of `nodePtr` (stored in `_emptyCount`) was at least 1, the `_partialChild` has a new empty endmost child, and the ML-values of this child are updated.

If this case does not apply, `nodePtr` did not have any empty children. However, `nodePtr` might have had some ignored children. This is tested using the reference pointer of `nodePtr` to the doubly linked list of children. If the pointer is not the `NULL` pointer, there exists at least one ignored child. The node `nodePtr` was considered by the function `PQTree::template_P5(...)` to be removed from the tree. We reinsert `nodePtr` at the full end of the partial child. By applying this strategy, we make sure that the sink indicators in the frontier of `nodePtr` are in the pertinent subtree. This is a legal operation, since they were children of the $P$-node `nodePtr`. Therefore, it was possible to permute them within the sequence of pertinent leaves before applying template P5. Thus these sink indicators can be considered for edge augmentation according to Corollary 5.4.

⟨template_P5: *Update*⟩+≡

```
if (_emptyCount >= 1)
{
    ⟨template_P5: Update new empty endmost node⟩
}
else if (nodePtr->status() == TO_BE_DELETED &&
         nodePtr->referenceChild() != NULL)
{
    if (_side == LEFT)
        addNodeToNewParent(_partialChild,nodePtr,NULL,
                           _partialChild->getEndmost(_side));
    else if (_side == RIGHT)
        addNodeToNewParent(_partialChild,nodePtr,
                           _partialChild->getEndmost(_side),NULL);
    nodePtr->status(IGNORED);
}
return 1;
```

The partial node `_partialChild` has a new full endmost child stored in `_fullNode` and the ML-values of `_fullNode` have to be updated. Two cases apply.

(i) The node `nodePtr` had only one full child and no full $P$-node has been allocated by `PQTree::template_P5(...)`. The ML-values of `_fullNode` are updated.

(ii) The node `nodePtr` had at least two full children and a new full $P$-node has been

allocated by `PQTree::template_P5(...)`. The new $P$-node `_fullNode` is equipped with a container class of type `leveledGraphInfo<leafID,int>` and the ML-values of `_fullNode` are set.

The first case can be recognized by the existence of a container class of type `leveledGraphInfo<leafID,int>` associated with `_fullNode` since `_fullNode` was an existing node in the $PQ$-tree when entering the function `PQTree::template_P5(...)`. The second case is recognized by the absence of an information class associated with `_fullNode`.

⟨`template_P5`*: Update new full endmost node*⟩≡
```
if (_fullNode->getNodeInfo() != NULL)
{
    if (clientSibLeft(_fullNode) != NULL)
        _fullNode->getNodeInfo()->_userStructInfo._leftML = _ml;
    else
        _fullNode->getNodeInfo()->_userStructInfo._rightML = _ml;
}
else
{
    _nodeInfoF = new leveledGraphInfo<leafID,int>(_dummyMLvalue);
    _fullNode->setNodeInfo(_nodeInfoF);
    _nodeInfoF->setNodePointer(_fullNode);
    if (clientSibLeft(_fullNode) != NULL)
        _fullNode->getNodeInfo()->_userStructInfo.set(_ml,0,_ml);
    else
        _fullNode->getNodeInfo()->_userStructInfo.set(0,_ml,_ml);
}
```

If `_partialChild` has a new full endmost child (stored in `_fullNode`) the ML-values of `_fullNode`s adjacent nonignored sibling is updated.

⟨`template_P5`*: Update new full endmost node*⟩+≡
```
if (clientSibLeft(_fullNode) != NULL)
{
    if (clientSibRight(clientSibLeft(_fullNode)) == _fullNode)
        clientSibLeft(_fullNode)->getNodeInfo()->
            _userStructInfo._rightML = _ml;

    else
        clientSibLeft(_fullNode)->getNodeInfo()->
            _userStructInfo._leftML = _ml;
}
else
{
    if (clientSibLeft(clientSibRight(_fullNode)) == _fullNode)
        clientSibRight(_fullNode)->getNodeInfo()->
            _userStructInfo._leftML = _ml;
```

```
    else
        clientSibRight(_fullNode)->getNodeInfo()->
            _userStructInfo._rightML = _ml;
}
```

The partial node `_partialChild` has a new empty endmost child stored in `_emptyNode` and the ML-values of `_emptyNode` have to be updated. Two cases apply.

(i) The node `nodePtr` had only one empty child and at least one ignored child. The function `PQTree::template_P5(...)` only noticed that `nodePtr` has just one empty child, and added an arbitrary child to the empty end of `_partialChild`. We access this child via the pointer `_fakeEndmost`, and apply a correction step by replacing `_fakeEndmost` with `nodePtr`, and adding `_fakeEndmost` as a child back to `nodePtr`. After the correction is complete, the ML-values are updated by setting `_emptyNode = nodePtr`, applying case (ii).

(ii) Two subcases apply.

a) Node `nodePtr` is an endmost child of `_partialChild`. Thus `nodePtr == _emptyNode` and either `nodePtr` has at least two empty, nonignored children or the correction step of case (i) was applied.

b) An empty child of `nodePtr` is an endmost child of `_partialChild`. Thus `_emptyNode` equals the empty, nonignored child of `nodePtr` and `nodePtr` did have exactly one empty child and no ignored children.

We update the ML-values of `_emptyNode` and its sibling.

⟨template_P5*: Update new empty endmost node*⟩≡

```
if (_emptyCount == 1 && nodePtr->referenceChild() != NULL)
{
    if (_side == RIGHT)
        _fakeEndmost = _partialChild->getEndmost(LEFT);
    else if (_side == LEFT)
        _fakeEndmost = _partialChild->getEndmost(RIGHT);
    exchangeNodes(_fakeEndmost,nodePtr);
    if (nodePtr->referenceChild() !=
        nodePtr->referenceChild()->getNextSib(NULL))
        // nodePtr has two children
        addNodeToNewParent(nodePtr,_fakeEndmost,
                           nodePtr->referenceChild(),
                           nodePtr->referenceChild()->getNextSib(NULL));
    else
        // nodePtr has only one child
        addNodeToNewParent(nodePtr,_fakeEndmost,
                           nodePtr->referenceChild(),NULL);
    nodePtr->childCount(1);
```

```
    nodePtr->status(EMPTY);
    _emptyNode = nodePtr;
}
if (clientSibLeft(_emptyNode) != NULL)
{
    _emptyNode->getNodeInfo()->_userStructInfo._leftML = _ml;
    _emptyNode->getNodeInfo()->_userStructInfo._rightML = 0;
    if (clientSibRight(clientSibLeft(_emptyNode)) == _emptyNode)
        clientSibLeft(_emptyNode)->getNodeInfo()->
            _userStructInfo._rightML = _ml;
    else
        clientSibLeft(_emptyNode)->getNodeInfo()->
            _userStructInfo._leftML = _ml;
}
else
{
    _emptyNode->getNodeInfo()->_userStructInfo._rightML = _ml;
    _emptyNode->getNodeInfo()->_userStructInfo._leftML = 0;
    if (clientSibLeft(clientSibRight(_emptyNode)) == _emptyNode)
        clientSibRight(_emptyNode)->getNodeInfo()->
            _userStructInfo._leftML = _ml;
    else
        clientSibRight(_emptyNode)->getNodeInfo()->
            _userStructInfo._rightML = _ml;
}
```

## 6.6 Code Example III: `Template Q2`

The function `template_Q2(...)` is a protected member function of the class `LevelPQTree<leafID,MLvalue,int>`. The function overloads the virtual protected function `PQTree::template_Q2(...)`. The class `LevelPQTree<leafID,MLvalue,int>` overloads the function in order

- to guarantee correct application of template Q2,
- to update ML-values, and
- to handle contacts.

The function collects pointers to all nodes of the $PQ$-tree that are affected by the template Q2. After calling the function `PQTree::template_Q2(...)`, it applies all necessary update operations.

### 6.6.1 Input Values

`nodePtr` is a pointer of type `node<leafID,MLvalue,int>` and the function `template_Q2(...)` tries to apply the template Q2 to `nodePtr` and its children.

`isRoot` is an integer that is 1 if `nodePtr` is the root of the pertinent subtree and 0 otherwise.

### 6.6.2 Return Values

1 if the the template Q2 was applied successfully to `nodePtr` and its children, 0 otherwise.

### 6.6.3 Variables

`_partialChild` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the partial child of `nodePtr` before template Q2 is applied. By construction, `_partialChild` is a $Q$-node.

`_partialLeft` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the adjacent nonignored sibling on the left side of `_partialChild` before template Q2 is applied.

`_partialLeftIgn` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the adjacent ignored sibling on the left side of `_partialChild` before template Q2 is applied.

`_partialRight` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the adjacent nonignored sibling on the right side of `_partialChild` before template Q2 is applied.

`_partialRightIgn` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the adjacent ignored sibling on the right side of `_partialChild` before template Q2 is applied.

`_fullEnd` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the full endmost nonignored child of `_partialChild` before template Q2 is applied.

`_fullSib` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the adjacent nonignored sibling of `_fullEnd` before template Q2 is applied.

`_fullIgn` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the endmost ignored child at the full end of `_partialChild` before template Q2 is applied.

`_fullIgnSib` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the adjacent sibling of `_fullIgn` before template Q2 is applied. Node `_fullIgnSib` may be either ignored or nonignored.

`_emptyEnd` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the empty endmost nonignored child of `_partialChild` before template Q2 is applied.

`_emptySib` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the adjacent nonignored sibling of `_emptyEnd` before template Q2 is applied.

`_emptyIgn` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the endmost ignored child at the empty side of `_partialChild` before template Q2 is applied.

`_emptyIgnSib` is a pointer of type `node<leafID,MLvalue,int>`. It denotes the adjacent sibling of `_emptyIgn` before template Q2 is applied. The node `_emptyIgnSib` may be either ignored or nonignored.

`_ignoredEmpty` is a pointer of type `node<leafID,MLvalue,int>`. It denotes an ignored child of `nodePtr` that is marked empty in case `nodePtr` has only one nonignored child (which is then the `_partialChild`). Marking an ignored child as empty allows us to apply the function `PQTree::template_Q2(...)`.

`_mlLeft` is an integer holding the ML-value between `_partialChild` and its adjacent left sibling.

`_mlRight` is an integer holding the ML-value between `_partialChild` and its adjacent right sibling.

`_ignoredStatus` is the original status of the node `_ignoredEmpty`. After the template Q2 has been applied, the original status is returned to `_ignoredEmpty`.

`_success` is an integer that is 1 if the template Q2 and the necessary update operations have been applied successfully.

### 6.6.4 Code-Body

The function `template_Q2(...)` keeps certain information before executing the function `PQTree::template_Q2(...)`. If the template reduction Q2 was performed successfully, function `template_Q2(...)` applies the necessary update operations.

⟨template_Q2⟩≡

```
/*****************************************************************************
                                  template_Q2
*****************************************************************************/


int LevelPQTree::template_Q2(node<leafID,MLvalue,int>*  nodePtr,int isRoot)
{
    node<leafID,MLvalue,int>*      _partialChild    = NULL;
    node<leafID,MLvalue,int>*      _partialLeft     = NULL;
    node<leafID,MLvalue,int>*      _partialLeftIgn  = NULL;
    node<leafID,MLvalue,int>*      _partialRight    = NULL;
    node<leafID,MLvalue,int>*      _partialRightIgn = NULL;
```

```
    node<leafID,MLvalue,int>*      _fullEnd         = NULL;
    node<leafID,MLvalue,int>*      _fullSib         = NULL;
    node<leafID,MLvalue,int>*      _fullIgn         = NULL;
    node<leafID,MLvalue,int>*      _fullIgnSib      = NULL;
    node<leafID,MLvalue,int>*      _emptyEnd        = NULL;
    node<leafID,MLvalue,int>*      _emptySib        = NULL;
    node<leafID,MLvalue,int>*      _emptyIgn        = NULL;
    node<leafID,MLvalue,int>*      _emptyIgnSib     = NULL;
    node<leafID,MLvalue,int>*      _ignoredEmpty    = NULL;
    int                            _mlLeft          = 0;
    int                            _mlRight         = 0;
    int                            _ignoredStatus   = 0;
    int                            _success         = 1;

    ⟨template_Q2: Preparation⟩
    if (_success &&
        PQTree<leafID,MLvalue,int>::template_Q2(nodePtr,isRoot))
    {
        ⟨template_Q2: Update⟩
    }
    else
        _success = 0;
    return _success;
}
```

### 6.6.5 Preparation of Template Q2

Before calling the function `PQTree::template_Q2(...)` for applying template Q2 to `nodePtr`, pointers to all involved nodes need to be stored. The nodes involved in template Q2 are

- endmost ignored and nonignored children of `_partialChild`, and
- ignored and nonignored siblings of `_partialChild`.

The information on the nodes is needed in order to update ML-values between endmost children of `_partialChild` and their new siblings. If `nodePtr` does not have a partial child, the template Q2 does not modify `nodePtr` and no update is necessary.

⟨template_Q2: Preparation⟩≡

```
partialChildrenStack(nodePtr)->startAtBottom();
if (!partialChildrenStack(nodePtr)->readLast())
{
   _partialChild = partialChildrenStack(nodePtr)->readNext();
   if (clientLeftEndmost(_partialChild)->status() == FULL)
```

```
{
    _fullEnd = clientLeftEndmost(_partialChild);
    if (_partialChild->getEndmost(LEFT)->status()== CONTACT)
        _fullIgn = _partialChild->getEndmost(LEFT);

    _emptyEnd = clientRightEndmost(_partialChild);
    if (_partialChild->getEndmost(RIGHT)->status() == IGNORED ||
        _partialChild->getEndmost(RIGHT)->status() == SINKFLAG ||
        _partialChild->getEndmost(RIGHT)->status() == CONTACT)
        _emptyIgn = _partialChild->getEndmost(RIGHT);
}
else
{
    _fullEnd = clientRightEndmost(_partialChild);
    if (_partialChild->getEndmost(RIGHT)->status()== CONTACT)
        _fullIgn = _partialChild->getEndmost(RIGHT);

    _emptyEnd = clientLeftEndmost(_partialChild);
    if (_partialChild->getEndmost(LEFT)->status() == IGNORED ||
        _partialChild->getEndmost(LEFT)->status() == SINKFLAG ||
        _partialChild->getEndmost(LEFT)->status() == CONTACT)
        _emptyIgn = _partialChild->getEndmost(LEFT);
}
_fullSib = clientNextSib(_fullEnd,NULL);
_emptySib = clientNextSib(_emptyEnd,NULL);
if (_fullIgn)
    _fullIgnSib = _fullIgn->getNextSib(NULL);
if (_emptyIgn)
    _emptyIgnSib = _emptyIgn->getNextSib(NULL);

_mlLeft = _partialChild->getNodeInfo()->_userStructInfo._leftML;
_mlRight = _partialChild->getNodeInfo()->_userStructInfo._rightML;
_partialLeft = clientSibLeft(_partialChild);
_partialRight = clientSibRight(_partialChild);
```

If the ignored sibling of the left or the right side of `_partialChild` is a contact, the variable `_partialLeftIgn` or `_partialRightIgn`, respectively, is not set. This contact belongs by construction to a sequence of endmost contacts of `nodePtr`, and no ML-values have to be set between these contacts and `_emptyIgn` or between the contacts and `_fullIgn`.

⟨`template_Q2`: *Preparation*⟩+≡

```
if (_partialChild->getSib(LEFT) && (
    _partialChild->getSib(LEFT)->status() == IGNORED ||
    _partialChild->getSib(LEFT)->status() == SINKFLAG))
    _partialLeftIgn = _partialChild->getSib(LEFT);
if (_partialChild->getSib(RIGHT) && (
```

```
        _partialChild->getSib(RIGHT)->status() == IGNORED ||
        _partialChild->getSib(RIGHT)->status() == SINKFLAG))
        _partialRightIgn = _partialChild->getSib(RIGHT);
}
```

If the node `nodePtr` has only one nonignored child, this nonignored child is apparently the partial child stored in `_partialChild`. Mark one of the ignored children of `nodePtr` as full. There must be at least one ignored child since the implementation avoids chains. This allows the function `PQTree::template_Q2(...)` to perform correctly on `nodePtr`. We remember to update the parent pointer of `_fullEnd` after the template matching has been performed since this is not done by the template Q2 due to our trick. The operation is legal, since the subgraph corresponding to the $Q$-node `nodePtr` can be reversed without affecting a level planar embedding of the rest of the graph. Thus, sink indicators and contacts that are descendants of `nodePtr` but not descendants of the `_partialChild` are allowed to appear within the pertinent sequence.

⟨`template_Q2`*: Preparation*⟩+≡

```
if (clientRightEndmost(nodePtr) && clientLeftEndmost(nodePtr) &&
    clientRightEndmost(nodePtr) == clientLeftEndmost(nodePtr))
{
    if (nodePtr->getEndmost(LEFT)->status() == IGNORED ||
        nodePtr->getEndmost(LEFT)->status() == SINKFLAG)
    {
        _ignoredEmpty = nodePtr->getEndmost(LEFT);
        _ignoredStatus = nodePtr->getEndmost(LEFT)->status();
        _ignoredEmpty->status(FULL);
    }
    else if (nodePtr->getEndmost(RIGHT)->status() == IGNORED ||
             nodePtr->getEndmost(RIGHT)->status() == SINKFLAG)
    {
        _ignoredEmpty = nodePtr->getEndmost(RIGHT);
        _ignoredStatus = nodePtr->getEndmost(RIGHT)->status();
        _ignoredEmpty->status(FULL);
    }
    else
    {
        _success = 0;
        cerr << "ERROR 0: LevelPQTree : template_Q2: "
             << "called on a node with apparently only "
             << "one child. " << endl;
    }
}
```

### 6.6.6 Update after Template Q2

After `PQTree::template_Q2(...)` has been successfully applied, the necessary updates are performed. First, we check if an ignored child of `nodePtr` has been marked as empty in order to allow `PQTree::template_Q2(...)` to perform correctly on `nodePtr`. If such a node exists, we reset its status back from `EMPTY` to the status it had before it was marked as `EMPTY`.

⟨`template_Q2`*: Update*⟩≡
```
if (_ignoredEmpty)
{
    _ignoredEmpty->status(_ignoredStatus);
    _fullEnd->parent(nodePtr);
}
```

If `_partialChild` existed, the ML-values between the endmost nonignored children of `_partialChild` and their new siblings have to be updated. This holds for both the full and empty nodes. The ML-values between the ignored endmost child at the empty end of `_partialChild` and its new ignored sibling (if it exists) need to be updated as well.

The ML-values between ignored nodes in the pertinent subtree are not needed for computing or updating the PML- and QML-values. Furthermore, the ignored nodes in the pertinent subtree are removed from the $PQ$-tree after the reduction is complete. Thus, the ignored endmost child at the full end of `_partialChild` and its new ignored sibling do not need to be updated.

⟨`template_Q2`*: Update*⟩+≡
```
if (_partialChild != NULL)
{
    ⟨template_Q2: Update nonignored full endmost children⟩
    ⟨template_Q2: Update nonignored empty endmost children⟩
    ⟨template_Q2: Update contacts⟩
    if (_emptyIgn)
    {
        ⟨template_Q2: Update ignored empty endmost children⟩
    }
}
```

This code fragment updates the ML-values between the full endmost child of `_partialChild` and its new adjacent nonignored sibling.

⟨`template_Q2`*: Update nonignored full endmost children*⟩≡
```
if (_partialRight == clientNextSib(_fullEnd,_fullSib))
{
    if (_partialRight == clientSibRight(_fullEnd))
        _fullEnd->getNodeInfo()->_userStructInfo._rightML =_mlRight;
    else if (_partialRight == clientSibLeft(_fullEnd))
        _fullEnd->getNodeInfo()->_userStructInfo._leftML = _mlRight;
}
```

```
else if (_partialLeft == clientNextSib(_fullEnd,_fullSib))
{
    if (_partialLeft == clientSibLeft(_fullEnd))
        _fullEnd->getNodeInfo()->_userStructInfo._leftML = _mlLeft;
    else if (_partialLeft == clientSibRight(_fullEnd))
        _fullEnd->getNodeInfo()->_userStructInfo._rightML = _mlLeft;
}
else
{
    _success = 0;
    cerr << "ERROR 1: LevelPQTree : template_Q2: "
         << "doubly linked list of children messed up." << endl;
}
```

This code fragment updates the ML-values between the empty endmost child of `_partialChild` and its new adjacent nonignored sibling.

⟨`template_Q2`*: Update nonignored empty endmost children*⟩≡

```
if (_partialRight == clientNextSib(_emptyEnd,_emptySib))
{
    if (_partialRight == clientSibRight(_emptyEnd))
        _emptyEnd->getNodeInfo()->_userStructInfo._rightML =_mlRight;
    else if (_partialRight == clientSibLeft(_emptyEnd))
        _emptyEnd->getNodeInfo()->_userStructInfo._leftML = _mlRight;

}
else if (_partialLeft == clientNextSib(_emptyEnd,_emptySib))
{
    if (_partialLeft == clientSibLeft(_emptyEnd))
        _emptyEnd->getNodeInfo()->_userStructInfo._leftML = _mlLeft;
    else if (_partialLeft == clientSibRight(_emptyEnd))
        _emptyEnd->getNodeInfo()->_userStructInfo._rightML = _mlLeft;
}
else
{
    _success = 0;
    cerr << "ERROR 2: LevelPQTree : template_Q2: "
         << " doubly linked list of children messed up." << endl;
}
```

The following code fragment updates contacts. There are two cases two be considered

(i) contacts at the full end of `_partialChild`,

(ii) contacts at the empty end of `_partialChild`.

Case (i) is considered first. Thus we have at least one contact within the pertinent sequence. Three subcases apply.

(i) a) A merge operation B or C has been applied right before the reduction that applies function `template_Q2(...)` to `nodePtr` and `nodePtr` is the new $Q$-node that has been introduced by the function `LevelPlanarTest::Merge(...)`. Thus Rule II has to be applied to the contacts within the pertinent sequence. The function `template_Q2(...)` notices this fact by checking if `nodePtr` is the root of the pertinent subtree (then `isRoot == 1`) and by checking if the private member variable `_contactFlag` of the class `LevelPQTree<leafID,MLvalue,int>` is unequal to 0. A function `moveContact(...)` is then called that places the sequence of `contacts` at its predetermined position.

(i) b) No merge operation B or C was applied before the reduction corresponding to this function call. By Observation 5.20 the contacts are adjacent to a reference sequence. We apply Lemma 5.36 by calling the function `connectContact(...)` that removes the sequence of contacts and the adjacent reference sequence from the tree. The function `connectContact(...)` adds for every sink indicator that is detected in the frontier of the reference sequence an edge directed towards the associated vertex of the corresponding contact.

(i) c) A merge operation B or C has been applied right before the reduction that applies function `template_Q2(...)` to `nodePtr` but `nodePtr` is not the root of the pertinent subtree. We proceed as in case (i) b).

The parameters of the function `connectContact(...)` are references to the pointers of the involved nodes since the contacts and their reference sequences are removed from the $PQ$-tree. The function `connectContact` returns the siblings of the contacts and the ignored nodes in the reference sequences. This feature is needed for the case (ii).

⟨`template_Q2`*: Update contacts*⟩≡

```
if (_fullIgn && _fullIgn->status() == CONTACT)
{
    if (isRoot && _contactFlag)
        moveContact(_fullIgn,_fullIgnSib,nodePtr);

    else
        connectContact(&_fullIgn,&_fullIgnSib,
                       &_partialRightIgn,&_partialLeftIgn);
}
```

The case (ii) considers a sequence of contacts at the empty end of `_partialChild`. Three subcases apply.

(ii) a) A merge operation B or C has been applied right before the reduction that applies function `template_Q2(...)` to `nodePtr` and `nodePtr` is the new $Q$-node that has been introduced by the function `LevelPlanarTest::Merge(...)`. Thus Rule I has to be applied to the contacts within the pertinent sequence and nothing has to be done.

(ii) b) No merge operation B or C was applied before the reduction corresponding to this function call. By Observation 5.20 the contacts are adjacent to a reference sequence. We apply Lemma 5.36 by calling the function `connectContact(...)`. Here we make use of the parameters of the function `connectContact(...)` being references to the pointers of the involved nodes. The ML-values of the ignored siblings on both sides of the removed sequence of nodes (contacts consecutively followed or preceded by their left or right reference sequence) need to be updated. The function `connectContact(...)` returns these ignored nodes allowing us to update them as ignored empty endmost children.

(ii) c) A merge operation B or C has been applied right before the reduction that applies function `template_Q2(...)` to `nodePtr` but `nodePtr` is not the root of the pertinent subtree. We proceed as in case (ii) b).

⟨`template_Q2`*: Update contacts*⟩+≡

```
if (_emptyIgn && _emptyIgn->status() == CONTACT && !(isRoot && _contactFlag))
    connectContact(&_emptyIgn,&_emptyIgnSib,
                   &_partialRightIgn,&_partialLeftIgn);
```

The following code fragment updates the ML-value of the endmost ignored child `_emptyIgn` of `_partialChild`. The node `_emptyIgn` was on the empty side of `_partialChild`. We do not update the endmost ignored child of `_partialChild` on the full side of `_partialChild` since this ignored child is within the pertinent subtree and will be removed after the reduction is finished successfully.

The code fragment is only accessed if `_emptyIgn` exists. It checks if `_partialRightIgn` and `_emptyIgn` are siblings. If the nodes are not siblings, the code fragment checks if `_partialLeftIgn` and `_emptyIgn` are siblings. If this also does not hold, it is checked if `_emptyIgn` is adjacent to one of the nonignored nodes `_partialRight` or `_partialLeft`. The node `_emptyIgn` must be adjacent to one of the four nodes. This also holds if `_emptyIgn` is the new endmost child of `nodePtr`, and therefore has no sibling on one side. In this case, either `_partialRight` or `_partialLeft` contains a null pointer. Thus either

```
_partialRight == _emptyIgn->getNextSib(_emptyIgnSib)
```

or

```
_partialLeft == _emptyIgn->getNextSib(_emptyIgnSib)
```

holds, setting a correct ML-value at `_emptyIgn`. The used strategy immediately allows to check if the template matching was performed correctly. If it was not performed correctly, an error message is printed.

Consider now the first if query. We check if `_partialRightIgn` exists. If it does not exist, and `_emptyIgn` is the endmost child of `nodePtr`, the if query

```
_partialRightIgn == _emptyIgn->getSib(LEFT)
```

might return 1 and wrong ML-values would be considered.

If `_partialRightIgn` exists, it is checked if `_partialRightIgn` and `_emptyIgn` are adjacent. This is done by examining the left and the right sibling of `_emptyIgn`. If, e.g.,

```
_partialRightIgn == _emptyIgn->getSib(LEFT)
```

holds, we need to set the ML-value of `_emptyIgn` associated with the left sibling pointer of `_emptyIgn`.

The ML-value that we assign to `_emptyIgn` depends on the pointer that `_partialRightIgn` uses to dereference its sibling `_emptyIgn`. If, e.g., `_partialRightIgn` dereferences `_emptyIgn` via its left sibling pointer, we use the corresponding left ML-value of `_partialRightIgn`.

All other cases are handled analogous.

⟨`template_Q2`*: Update ignored empty endmost children*⟩≡

```
if (_partialRightIgn && _partialRightIgn == _emptyIgn->getSib(LEFT))
{
    if (_partialRightIgn->getSib(LEFT) == _emptyIgn)
        _emptyIgn->getNodeInfo()->_userStructInfo._leftML =
        _partialRightIgn->getNodeInfo()->_userStructInfo._leftML;
    else if (_partialRightIgn->getSib(RIGHT) == _emptyIgn)
        _emptyIgn->getNodeInfo()->_userStructInfo._leftML =
        _partialRightIgn->getNodeInfo()->_userStructInfo._rightML;
}
else if (_partialRightIgn && _partialRightIgn == _emptyIgn->getSib(RIGHT))
{
    if (_partialRightIgn->getSib(LEFT) == _emptyIgn)
        _emptyIgn->getNodeInfo()->_userStructInfo._rightML =
        _partialRightIgn->getNodeInfo()->_userStructInfo._leftML;
    else if (_partialRightIgn->getSib(RIGHT) == _emptyIgn)
        _emptyIgn->getNodeInfo()->_userStructInfo._rightML =
        _partialRightIgn->getNodeInfo()->_userStructInfo._rightML;
}
else if (_partialLeftIgn && _partialLeftIgn == _emptyIgn->getSib(LEFT))
{
    if (_partialLeftIgn->getSib(LEFT) == _emptyIgn)
        _emptyIgn->getNodeInfo()->_userStructInfo._leftML =
        _partialLeftIgn->getNodeInfo()->_userStructInfo._leftML;
    else if (_partialLeftIgn->getSib(RIGHT) == _emptyIgn)
        _emptyIgn->getNodeInfo()->_userStructInfo._leftML =
        _partialLeftIgn->getNodeInfo()->_userStructInfo._rightML;
}
else if (_partialLeftIgn && _partialLeftIgn == _emptyIgn->getSib(RIGHT))
```

```
{
    if (_partialLeftIgn->getSib(LEFT) == _emptyIgn)
        _emptyIgn->getNodeInfo()->_userStructInfo._rightML =
        _partialLeftIgn->getNodeInfo()->_userStructInfo._leftML;
    else if (_partialLeftIgn->getSib(RIGHT) == _emptyIgn)
        _emptyIgn->getNodeInfo()->_userStructInfo._rightML =
        _partialLeftIgn->getNodeInfo()->_userStructInfo._rightML;
}
else if (_partialRight == _emptyIgn->getNextSib(_emptyIgnSib))
{
    if (_partialRight == _emptyIgn->getSib(RIGHT))
        _emptyIgn->getNodeInfo()->_userStructInfo._rightML =_mlRight;
    else if (_partialRight == _emptyIgn->getSib(LEFT))
        _emptyIgn->getNodeInfo()->_userStructInfo._leftML = _mlRight;

}
else if (_partialLeft == _emptyIgn->getNextSib(_emptyIgnSib))
{
    if (_partialLeft == _emptyIgn->getSib(RIGHT))
        _emptyIgn->getNodeInfo()->_userStructInfo._rightML = _mlLeft;
    else if (_partialLeft == _emptyIgn->getSib(LEFT))
        _emptyIgn->getNodeInfo()->_userStructInfo._leftML = _mlLeft;
}
else
     cerr << "ERROR 3: LevelPQTree : template_Q2: "
          << "doubly linked list of ignored children messed up." << endl;
```

# Chapter 7

# Discussion

The method for producing level drawings of a digraph $G = (V, E)$ that was presented by Sugiyama *et al.* (1981) is highly intuitive and can be applied to any directed graph, regardless of its graph theoretic properties. Thus, the approach of Sugiyama *et al.* (1981) (and subsequent methods of Gansner, North, and Vo (1988), Eades and Sugiyama (1991), Messinger, Rowe, and Henry (1991), and Gansner, Koutsofios, North, and Vo (1993) that are closely related) is not restricted to the drawing of digraphs with a predefined leveling. Besides, its implementation is rather easy if only heuristics are used for the various problems that appear. This makes the approach very attractive in practice and variations of it are not only found in almost all graph drawing systems, but also in a lot of other systems that need to visualize information. The hierarchical approach consists of three steps.

(i) (a) If the graph $G$ does not have a leveling, the vertices of $G$ are assigned to levels.
(b) The level graph $G$ is transformed into a proper level graph.

(ii) The vertices within each level are ordered to obtain a small the number of edge crossings.

(iii) A horizontal coordinate is assigned to each vertex.

Transforming the level graph $G$ into a proper graph is done since it is difficult to handle crossings involving long edges. However, the number of dummy vertices that are added during the transformation is in $\mathcal{O}(n^2)$ with $n$ being the number of vertices in $G$.

By using the linear time level planarity test and the linear time level planar embedding algorithm that have been presented in this work, a drawing of a level planar graph can be produced in $\mathcal{O}(n)$ time without transforming the level graph into a proper one first. Thus the usual hierarchical approach as presented above is expanded by an extra step where we check in $\mathcal{O}(n)$ if the graph is level planar, and if so, produce the corresponding drawing. If not, we apply the usual techniques for minimizing the number of crossings.

Clearly, level graphs that need to be visualized are not level planar in general. Thus we expect research to continue on concentrating on the subject of minimizing the number of

edge crossings. Since almost all approaches known in the literature only attack the problem of 2-level crossing minimization, studies on more general approaches are desirable in order to obtain a global view on the graph while reducing the number of edge crossings.

From our point of view, two main directions are worth to be investigated. The first is to solve the *multi-level crossing minimization problem* by branch-and-cut methods as suggested in Jünger, Lee, Mutzel, and Odenthal (1997a). This demands deeper polyhedral studies of the associated polytope. The second direction is to study an alternative method, the *$k$-level planarization problem*, suggested by Mutzel (1996). This method removes a minimum number of edges such that the resulting graph is $k$-level planar. For the final diagram, the removed edges are reinserted into a $k$-level planar drawing. In order to apply the $k$-level planarization method, the level planar embedding algorithm as it has been presented in this work can be applied. This makes our results also interesting for nonlevel planar graphs.

Mutzel (1996) studies the $k$-level planarization problem for the case $k = 2$. However, extracting a 2-level planar graph with maximum number of edges from a given 2-level graph is $\mathcal{NP}$-hard, as has been shown by Eades and Whitesides (1994). Based on a characterization of 2-level planar graphs by Harary and Schwenk (1972),Tomii *et al.* (1977), and Eades *et al.* (1986) (see 4.1), Mutzel (1996) gives an integer linear programming formulation for the 2-level planarization problem and defines and investigates the polytope associated with the set of all 2-level planar subgraphs of a 2-level graph. The polytope has full dimension and the inequalities occurring in the integer linear description are facet defining for the polytope. Moreover, Mutzel (1996) showed that these inequalities can be separated in polynomial time and therefore can be used efficiently in a branch-and-cut method for solving practical instances of the 2-level planarization problem.

In order to attack the $k$-level planarization problem for $k \geq 2$, an integer linear programming formulation has to be found, and the polytope associated with the set of all $k$-level planar subgraphs of a $k$-level graph needs to be described. Besides, polynomial time separation algorithms need to be developed for practical application. We build our hope that this can be achieved on two facts, namely our algorithm for recognizing $k$-level planarity, and the recent results of Healy and Kuusik (1998) who give a characterization of level planar graphs in terms of minimal forbidden subgraphs called *minimal nonlevel planar subgraph patterns* (MNLP-patterns). Such a MNLP-pattern is defined to have the property that the removal of any edge in the pattern makes the pattern level embeddable without edge crossings.

One important task in the $k$-level planarization problem is the detection of MNLP-patterns in nonlevel planar graphs. Further investigations are desired in order to expand our level planarity test such that it outputs a minimal nonlevel planar subgraph if the tested graph is not level planar. Karabeg (1990) and Hundack *et al.* (1996) have successfully installed a method in the planarity test of Booth and Lueker (1976) for detecting subdivisions of $K_{3,3}$ and $K_5$ in nonplanar graphs. With respect to the approach of Karabeg, we hope that similar methods can be found for nonlevel planar graphs.

Detecting a MNLP-pattern is not only interesting for the $k$-level planarization problem. Such methods can also be used to verify the result of the level planarity test. The need for the ability of verification needs to be mentioned in context with the fact that planarity tests are nontrivial programs and therefore it is not unlikely that an implementation is faulty. It is therefore desirable for a program to have the ability of self-checking. For the level planarity test that was presented in this work, two cases occur. Either it outputs that the level graph is planar. Then the test should be able to compute an embedding and/or a drawing. This can be done using the level planar embedding algorithm presented in this work. Or the test should be able to exhibit a MNLP-pattern.

These considerations reveal that there is a large number of open problems closely related to the topics of this work on level planarity. Other topics deal with the layout of level graphs. A lot of effort was spent on the second step of the hierarchical approach of Sugiyama *et al.* (1981), trying to minimize the number of crossings. However, the third phase that actually produces the drawing of a graph has been underestimated in the past. Problems come along with long edges that usually tend to have a lot of bends. The effect is that even if the number of edge crossings is small, the drawings are almost never aesthetically pleasing. The only work that deals with the third phase in full detail is by Gansner, Koutsofios, North, and Vo (1993).

A lot of problems arise when using the hierarchical approach for graphs that do not have a leveling. Here, vertices are assigned to certain levels. Although this is done obeying certain requirements (e.g., the level graph should be compact, the leveling has to be proper, and 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 number of crossings that appear in a level graph is highly dependent on the chosen leveling. Therefore, it would be preferable to develop methods that try to combine the three phases of the hierarchical approach.

We hope that the tools and results we have presented in this work will contribute to a deeper understanding in the drawing of level graphs.

# Bibliography


Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Applegate, D., Bixby, R., Chvàtal, V., and Cook, W. (1994). Finding cuts in the TSP. Technical report, AT&T Bell Labs.

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

Batini, C., Furlani, L., and Nardelli, E. (1985). What is a good diagram? A pragmatic approach. In *4th Int. Conf. Entity-Relationship-Approach*. IEEE.

Bertolazzi, P., Di Battista, G., Mannino, C., and Tamassia, R. (1998). Optimal upward planarity testing of single-source digraphs. *SIAM Journal on Computing*, **27**(1), 132–169.

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

Buchheim, C. (1997). Personal communications. Institut für Informatik, Universität zu Köln.

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

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

Catarci, C. (1995). The assignment heuristic for crossing reduction. *IEEE Transaction on System, Man, and Cybernetics*, **25**(3).

Chandramouli, M. and Diwan, A. A. (1995). Upward numbering testing for triconnected graphs. In F. J. Brandenburg, editor, *Proc. Graph Drawing '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 140–151. Springer Verlag.

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

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

Chiba, T., Nishioka, I., and Shirakawa, I. (1979). An algorithm for maximal planarization of graphs. In *Proceedings on the 1979 IEEE International Symposium on Circuits and Systems*, pages 336–441.

Christof, T. (1997). *Low-Dimensional 0/1-Polytopes and Branch-and-Cut in Combinatorial Optimization*. Ph.D. thesis, Ruprecht-Karls-Universität Heidelberg.

Christof, T., Jünger, M., Kecegioglu, J., Mutzel, P., and Reinelt, G. (1997). A branch and cut approach to physical mapping by unique end-probes. *Journal of Computational Biology*, **4**(4), 433–447.

Cormen, T., Leiserson, C., and Rivest, R. (1990). *Introduction to Algorithms*. MIT-Press.

Corneil, D. G., Kim, H., Natarajan, S., Olariu, S., and Sprague, A. P. (1995). Simple linear time recognition of unit interval graphs. *Information Processing Letters*, **55**, 99–104.

Dahlhaus, E. (1998a). A linear time algorithm to recognize clustered graphs and its parallelization. Technical Report 98.325, Institut für Informatik, Universität zu Köln.

Dahlhaus, E. (1998b). Personal communications.

de Figueiredo, C. M., Meidanis, J., and Mello, C. P. (1995). A linear-time algorithm for proper interval graph recognition. *Information Processing Letters*, **56**, 179–184.

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

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

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

Di Battista, G. and Tamassia, R. (1989). Incremental planarity testing. In *Proceedings on the 30th Annual IEEE Symposium on Foundations of Computer Science, North Carolina*, pages 436–441.

Di Battista, G. and Tamassia, R. (1996). On-line palnarity testing. *SIAM Journal on Computing*, **25**(5), 956–997.

Di Battista, G., Tamassia, R., and Tollis, I. G. (1992). Constrained visibility representations of graphs. *Information Processing Letters*, **41**, 1–7.

Di Battista, G., Eades, P., Tamassia, R., and Tollis, I. G. (1998). *Graph Drawing*. Prentice Hall.

Djidjev, H. N. (1995). A linear algorithm for the maximal planar subgraph problem. In *4th Workshop Algorithms Data Struct.*, volume 955 of *Lecture Notes in Computer Science*, pages 369–380. Springer-Verlag.

Djidjev, H. N. (1998). Personal communication.

Dresbach, S. (1994). A new heuristic layout algorithm for dags. In *Operations Research Proceedings*, pages 121–126. Springer-Verlag.

Eades, P. and Kelly, D. (1986). Heuristics for drawing 2-layered networks. *ARS Combinatoria*, **21-A**, 89–98.

Eades, P. and Sugiyama, K. (1991). How to draw a directed graph. *J. Inform. Process.*, **13**, 424–253.

Eades, P. and Whitesides, S. (1994). Drawing graphs in two layers. *Theoretical Computer Science*, **131**, 361–374.

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

Eades, P., McKay, B. D., and Wormald, N. C. (1986). On an edge crossing problem. In *Proc. 9th Australian Computer Science Conference*, pages 327–334. Australian National University.

Even, S. (1979). *Graph Algorithms*. Computer Science Press, Potomac, Maryland.

Even, S. and Tarjan, R. E. (1976). Computing an st-numbering. *Theoretical Computer Science*, **2**, 339–344.

Feng, Q.-W., Cohen, R. F., and Eades, P. (1995). Planarity for clustered graphs. In P. Spirakis, editor, *Algorithms – ESA '95*, volume 979 of *Lecture Notes in Computer Science*, pages 213–226. Springer Verlag.

Foulds, L. R. and Robinson, D. F. (1976). A strategy for solving the plant layout problem. *Operational Research Quaterly*, **27**(4), 845–855.

Fulkerson, D. R. and Gross, O. A. (1965). Incidence matrices and interval graphs. *Pacific J. Mathematics*, **15**, 835–855.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1996). *Design Patterns*. Addison-Wesley.

Gansner, E. R., North, S. C., and Vo, K.-P. (1988). Dag – a program that draws directed graphs. *Software – Practice and Experience*, **17**(1), 1047–1062.

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

Garey, M. R. and Johnson, D. S. (1983). Crossing number is $\mathcal{NP}$-complete. *SIAM Journal on Algebraic and Discrete Methods*, **4**(3), 312–316.

Garg, A. and Tamassia, R. (1994). On the computational complexity of upward and rectilinear planarity testing. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing '94*, volume 894 of *Lecture Notes in Computer Science*, pages 286–297. Springer Verlag.

Goldstein, A. J. (1963). An efficient and constructive algorithm for testing whether a graph can be embedded in the plane. In *Graph and Combinatorics Conference*.

Harary, F. (1969). *Graph Theory*. Addison Wesley.

Harary, F. and Schwenk, A. (1972). A new crossing number for bipartite graphs. *Utilitas Mathematica 1*.

Healy, P. and Kuusik, A. (1998). Characterisation of level non-planar graphs by minimal patterns. Technical Report UL-CSIS-98-4, Department of Computer Science & Information, University of Limerick.

Heath, L. S. and Pemmaraju, S. V. (1995). Recognizing leveled-planar dags in linear time. In F. J. Brandenburg, editor, *Proc. Graph Drawing '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 300–311. Springer Verlag.

Heath, L. S. and Pemmaraju, S. V. (1996). Stack and queue layouts of directed acyclic graphs: Part II. Technical report, Department of Computer Science, Virginia Polytechnic Institute & State University.

Hopcroft, J. and Tarjan, R. E. (1971). Planarity testing in $v \log v$ steps: Extended abstract. In *IFIP Congress '71*, Foundations of Information Processing, pages 18–22. North-Holland.

Hopcroft, J. and Tarjan, R. E. (1974). Efficient planarity testing. *ACM*, **21**, 549–568.

Hsu, W. L. (1992). A simple test for the consecutive ones property. In T. Ibaraki, Y. Inagaki, K. Iwama, T. Nishizeki, and M. Yamashita, editors, *Algorithms and Computation, 3rd International Symposium, ISAAC '92*, volume 650 of *Lecture Notes in Computer Science*, pages 459–468. Springer Verlag.

Hundack, C., Mehlhorn, K., and Näher, S. (1996). A simple linear time algorithm for identifying Kuratowski graphs. Technical report, Max-Planck-Institut für Informatik, Saarbrücken.

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

Jayakumar, R., Thulasiraman, K., and Swamy, M. N. S. (1986). On maximal planarization of non-planar graphs. *IEEE Transactions on Circuits Systems*, **33**(8), 843–844.

Jayakumar, R., Thulasiraman, K., and Swamy, M. N. S. (1989). On $O(n^2)$ algorithms for graph planarization. *IEEE Transactions on Computer-Aided Design*, **8**(3), 257–267.

Jünger, M. and Mutzel, P. (1996). Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica*, **16**, 33–59.

Jünger, M. and Mutzel, P. (1997). 2–layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications, 1997*, **1**, 1–25.

Jünger, M., Lee, E. K., Mutzel, P., and Odenthal, T. (1997). A polyhedral approach to the multi–layer crossing minimization problem. In G. Di Battista, editor, *Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, pages 13–24. Springer Verlag.

Jünger, M., Leipert, S., and Mutzel, P. (1997). Pitfalls of using PQ-trees in Automatic Graph Drawing. In G. Di Battista, editor, *Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, pages 193–204. Springer Verlag.

Jünger, M., Leipert, S., and Mutzel, P. (1998a). A note on computing a maximal planar subgraph using PQ–trees. *IEEE Transactions On Computer-Aided Design*, **17**(7), 609–612.

Jünger, M., Leipert, S., and Mutzel, P. (1998b). Level planarity testing in linear time. Technical Report 98.321, Institut für Informatik, Universität zu Köln. To appear in: Graph Drawing '98, Lecture Notes in Computer Science, Springer Verlag.

Kant, G. (1992). An $O(n^2)$ maximal planarization algorithm based on PQ-trees. Technical Report RUU-CS-92-03, Department of Computer Science, Utrecht University.

Kant, G. (1993). A more compact visibillity representation. In J. van Leeuwen, editor, *Proc. 19th International Workshop on Graph-Theoretical Concepts in Computer Science*, Lecture Notes in Computer Science. Springer Verlag.

Karabeg, A. (1990). Classification and detection of obstructions to planarity. *Linear and multilinear Algebra*, **26**, 15–38.

Kelly, D. (1987). Fundamentals of planar ordered sets. *Discrete Mathematics*, **63**, 197–216.

Korte, N. and Möhring, R. H. (1988). An incremental linear time algorithm for recognizing interval graphs. *SIAM Journal on Computing*, **18**, 68–81.

Kuratowski, K. (1930). Sur le problème des courbes gauches en topologie. *Fund. Math.*, **15**, 271–283.

La Poutré, J. A. (1994). Alpha-algorithms for incremental testing. In *26th Annual ACM Symp. on Theory of Comput.*, pages 706–715.

Leipert, S. (1995). *Berechnung maximal planarer Untergraphen mit Hilfe von PQ-Bäumen*. Master's thesis, Institut für Informatik der Universität zu Köln.

Leipert, S. (1996). The Tree Interface – Version 1.0. Technical Report 96.242, Institut für Informatik, Universität zu Köln.

Leipert, S. (1997). PQ–trees, an implementation as template class in C++. Technical Report 97.259, Institut für Informatik, Universität zu Köln. available as LEDA Extension Package at `http://www.mpi-sb.mpg.de/LEDA/`.

Lemke, I. and Sander, G. (1995). *The VCG Tool, a visualization tool for compiler graphs*. Compare Consortium and Universität des Saarlandes. Software is available per anonymous ftp at `ftp.cs.uni-sb.de`.

Lempel, A., Even, S., and Cederbaum, I. (1967). An algorithm for planarity testing of graphs. In *Theory of Graphs: International Symposium: Rome, July 1966*, pages 215–232. Gordon and Breach, New York.

Lengauer, T. (1989). Hierarchical planarity testing algorithm. *Journal of the ACM*, **36**(3), 474–509.

Liebel, C. (1998). *Analyse und Implementation eines c-planaren Einbettungsalgorithmus für zusammenhängende Cluster-Graphen*. Master's thesis, Institut für Informatik, Universität zu Köln, Germany.

Liu, F. C. and Geldmacher, R. C. (1977). On the deletion of nonplanar edges of a graph. In *10th S-E Conf. Combinatorics, Graph Theory and Computing*, pages 727–738.

Luccio, F., Mazzone, S., and Wong, C. (1987). A note on visibility graphs. *Discrete Mathematics*, **64**, 209–219.

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

Mehlhorn, K. and Näher, S. (1998). *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press. Software available through Algorithmic Solutions GmbH at `http://www.mpi-sb.mpg.de/LEDA/`.

Meidanis, J. and Munuera, E. G. (1996). A theory for the consecutive ones property. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *III South American Workshop on String Processsing*, pages 194–202.

Meidanis, J., Porto, O., and Telles, G. (1997). On the consecutive ones property. Technical report, Institute of Computing, University of Campinas, Brazil.

Messinger, E. B., Rowe, L. A., and Henry, R. H. (1991). A divide and conquer algorithm for the automatic layout of large directed graphs. *IEEE Transaction on System, Man, and Cybernetics*, **1**(21), 1–12.

Mutzel, P. (1992). A fast $\mathcal{O}(n)$ embedding algorithm based on the Hopcroft–Tarjan planarity test. Technical Report 92.107, Institut für Informatik, Universität zu Köln.

Mutzel, P. (1994). *The Maximum Planar Subgraph Problem*. Ph.D. thesis, Universität zu Köln.

Mutzel, P. (1996). An alternative method to crossing minimization on hierachical graphs. In S. North, editor, *Graph Drawing '96*, volume 1190 of *Lecture Notes in Computer Science*, pages 318–333.

Novick, M. B. (1989). Generalized PQ-trees. Technical Report 89-1074, Department of Computer Science, Cornell University, Ithaca, NY.

Oswald, M. (1997). *PQ-Bäume im Branch & Cut-Ansatz für das Physical-Mapping-Problem mit Endprobes*. Master's thesis, Universität Heidelberg, Germany.

Ozawa, T. and Takahashi, H. (1981). A graph-planarization algorithm and its application to random graphs. In *Graph Theory and Algorithms*, volume 108 of *Lecture Notes in Computer Science*, pages 95–107. Springer Verlag.

Platt, C. (1976). Planar lattices and planar graphs. *Journal of Combinatorial Theory (B)*, **21**, 30–39.

Purchase, H. (1997). Which aesthetic has the greatest effect on human understanding. In G. Di Battista, editor, *Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer Verlag.

Purchase, H., Cohen, R. F., and James, M. (1995). Validating graph drawing aesthetics. In F. J. Brandenburg, editor, *Graph Drawing '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 435–447. Springer Verlag.

Ramsey, N. (1992). *The* `noweb` *Hacker's Guide*. Department of Computer Science, Princeton University.

Rosenstiehl, P. and Tarjan, R. E. (1986). Rectilinear planar layouts and bipolar orientations of planar graphs. *Discrete and Computational Geometry*, **1**, 343–353.

Shirey, R. W. (1969). *Implementation and Analysis of Efficient Graph Planarity Testing Algorithms*. Ph.D. thesis, University of Wisconsin.

Stamm-Wilbrandt, H. (1996). Personal communications.

Störmer, P. (1998). *A Parallel Branch & Cut Algorithm for the Solution of Large-Scale Traveling Salesman Problems*. Ph.D. thesis, Institut für Informatik, Universität zu Köln.

Sugiyama, K., Tagawa, S., and Toda, M. (1981). Methods for visual understanding of hierarchical systems. *IEEE Trans. on Systems, Man, and Cybernetics*, **11**(2), 109–125.

Tamassia, R. and Preparata, F. P. (1990). Dynamic maintenance of planar digraphs with applications. *Algorithmica*, **5**, 509–527.

Tamassia, R. and Tollis, I. G. (1986). A unified approach to visibillity representations of planar graphs. *Discrete and Computational Geometry*, **1**, 321–341.

Tamassia, R. and Tollis, I. G. (1989). Tessellation representations of planar graphs. In *Proc. 27th Allerton Conf. Communication Control Computing*, pages 48–57.

Tarjan, R. E. (1969). Implementation of an efficient algorithm for planarity testing of graphs. Comput. Sci. Department, Stanford University.

Tarjan, R. E. (1971). *An Efficient Planarity Algorithm.* Ph.D. thesis, Comput. Sci. Department, Stanford University.

Tomii, N., Kambayashi, Y., and Shuzo, Y. (1977). On planarization algorithms of 2-level graphs. *Papers of the Tech. Group on Electronic Comp., IECEJ*, **EC77-38**, 1–12.

Valls, V., Marti, R., and Lino, P. (1996a). A tabu tresholding algorithm for arc crossing minimization in bipartite graphs. *Annals of Operations Research*, **63**, 223–251.

Valls, V., Marti, R., and Lino, P. (1996b). A branch and bound algorithm for minimizing the number of crossing arcs in bipartite graphs. *Journal of Operational Research*, **90**, 303–319.

Vollen, G. (1997). *PQ-trees and maximal planarization – an approach to skewness*. Master's thesis, Department of Informatics, University of Oslo, Sweden.

Warfield, J. N. (1977). Crossing theory and hierarchy mapping. *IEEE Trans. on Systems, Man, and Cybernetics*, **7**(7), 505–523.

Westbrook, J. (1992). Fast incremental planarity testing. In *Int. Col. ob Automata, Languages, and Programming*, pages 342–353.

Whitney, H. (1933). 2-isomorphic graphs. *Amer. J. Math.*, **55**, 245–254.

# List of Symbols

# Index

# Deutsche Zusammenfassung

Planarität ist eine Eigenschaft von Graphen, die viel beachtet und untersucht wird. Sie erlaubt es, gewisse Probleme, die sich bei allgemeinen Graphen als schwer erweisen, bei planaren Graphen effizient zu lösen. Klassische Beispiele sind die Bestimmung der Chromatischen Zahl eines Graphen oder die Bestimmung von Multicomodity Flüssen.

Ein Graph $G = (V, E)$ ist planar genau dann, wenn er sich ohne Kantenüberkreuzungen in der Ebene zeichnen läßt. Auf dem Gebiet des automatischen Zeichnens von Graphen hat daher die Planarität eine große Bedeutung, da das wesentlichste Kriterium für eine ästhetisch ansprechende Darstellung eines Graphen eine möglichst geringe Zahl von Kreuzungen oder, falls der Graph planar ist, eine kreuzungsfreie Darstellung ist. Für das automatische Zeichnen von Graphen ist daher, neben der Überprüfung eines Graphen auf Planarität mittels eines Planaritätstests, insbesondere auch die Bestimmung einer planaren Einbettung mittels eines Einbettungsalgorithmus von Bedeutung.

Hier haben sich in jüngerer Vergangenheit zwei Ansätze erfolgreich durchsetzen können. Der erste Ansatz basiert auf einer „Divide-and-Conquer“ Strategie, bei der in einem auf Planarität zu testenden Graphen $G$ ein Kantenkreis $C$ gesucht wird, dessen Entfernen den Graphen in mindestens zwei Zusammenhangskomponenten zerlegt. Anschließend werden die Zusammenhangskomponenten von $G - C$ rekursiv auf Planarität überprüft. Ist jede der Zusammenhangskomponenten von $G - C$ planar, so wird anschließend versucht, die Komponenten so um $C$ zu gruppieren, daß der resultierende Graph planar ist. Dieses Konzept wurde von Hopcroft und Tarjan (1974) erfolgreich in einen Planaritätstest umgesetzt, dessen Laufzeit linear in der Zahl der Ecken $V$ eines Graphen ist. Ein darauf basierender Einbettungsalgorithmus mit ebenfalls linearer Laufzeit wurde von Mehlhorn und Mutzel (1996) entwickelt.

Der zweite Ansatz basiert auf der Konstruktion einer Folge von induzierten Untergraphen. Ausgehend von einem Untergraphen, der durch eine Ecke des Graphen induziert und somit trivialerweise planar ist, werden sukzessive alle Ecken zu dem Untergraphen addiert und bei jeder Addition überprüft, ob der daraus resultierende induzierte Untergraph planar ist. Mit Hilfe der $PQ$-Baum Datenstruktur konnte dieses Konzept von Booth und Lueker (1976) erfolgreich in $\mathcal{O}(n)$ Zeit mit $n = |V|$ realisiert werden. Chiba, Nishizeki, Abe und Ozawa (1985) entwickelten einen auf diesem Ansatz basierenden Einbettungsalgorithmus mit linearer Laufzeit.

Neben dem ästhetischen Kriterium der kreuzungsfreien Zeichnung für planare Graphen gibt es eine Reihe weiterer Kriterien, die häufig nicht gleichzeitig zu realisieren sind. So ist es zum Beispiel wünschenswert, bei der Darstellung gerichteter azyklischer Graphen alle Kanten des Graphen monoton in eine Richtung verlaufen zu lassen. Gerichtete azyklische Graphen, die eine planare Zeichnung besitzen, in der alle Kanten monoton in eine Richtung gezeichnet sind, werden als aufwärtsplanare Graphen bezeichnet. Bei der Aufwärtsplanarität handelt es sich um eine Eigenschaft, die noch restriktiver ist als die der Planarität. Gerichtete azyklische Graphen mit der Eigenschaft planar zu sein, sind in der Regel nicht aufwärtsplanar. Überdies handelt es sich bei der Aufwärtsplanarität um ein $\mathcal{NP}$-vollständiges Problem, wie Garg und Tamassia (1994) nachweisen konnten. Lediglich für den Fall, daß der zu testende Graph genau nur eine Quelle (respektive eine Senke) besitzt, ist es bislang möglich, einen Graphen auf Aufwärtsplanarität zu testen (siehe u.a. Bertolazzi, Di Battista, Mannino und Tamassia (1998)).

Häufig anzutreffen ist der Wunsch nach geschichteten Zeichnungen oder hierarchischen Darstellungen von gerichteten azyklischen Graphen. Solche Darstellungstechniken haben ihren Ursprung in der Netzplantechnik, in PERT-Diagrammen und bei der Darstellung von Programmabläufen. Vorgegeben wird dabei zusätzlich zu dem Graph $G$ eine Funktion $\text{lev} : V \to \mathbb{Z}$, die jeder Ecke eine ganze Zahl so zuordnet, daß für jede gerichtete Kante $(u, v) \in E$ gilt: $\text{lev}(v) \geq \text{lev}(u) + 1$. Eine solche Funktion wird als die Schichtung eines Graphen $G$ bezeichnet. Ein Graph mit einer Schichtung heißt Schichtgraph. Die Menge der Knoten $V^j = \text{lev}^{-1}(j)$ ist eine Schicht des Graphen. Eine geschichtete Zeichnung eines Schichtgraphen ist eine Zeichnung des Graphen in der Ebene, bei der die Ecken der Schicht $j$ auf der Geraden $l_j = \{(x, k-j) \mid x \in \mathbb{R}\}$ plaziert werden und jede Kante $(u, v) \in E$, $u \in V^i$, $v \in V^j$, $1 \leq i < j \leq k$, als monoton fallende Kurve gezeichnet wird. Den Ecken des Graphen werden also feste $y$-Koordinaten zugewiesen, lediglich die $x$-Koordinaten sind frei wählbar. Besitzt nun ein Schichtgraph eine geschichtete Zeichnung ohne Kantenüberkreuzungen, so ist der Graph *schichtplanar*.

Der übliche Ansatz um eine solche geschichtete Zeichnung zu erzeugen, ist ein Drei-Phasen-Modell von Sugiyama, Tagawa und Toda (1981). In einer ersten Phase wird, sofern nicht bereits durch die darzustellende Probleminstanz vorgegeben, eine Einteilung der Ecken in $k$ Schichten vorgenommen. In der zweiten Phase wird versucht, die Zahl der Kantenüberkreuzungen zu minimieren, um anschließend in der dritten Phase die Zeichnung zu erzeugen.

Bei der zweiten Phase wird versucht, das *$k$-Schichten-Kreuzungsminimierungsproblem* zu lösen. Geht man davon aus, daß jeder Schichtgraph in einen Schichtgraphen transformiert werden kann, in dem jede Kante nur zwei Ecken auf benachbarten Schichten verbindet, so besteht das $k$-Schichten-Kreuzungsminimierungsproblem darin, für die Knoten von jeder Schicht geeignete Permutationen zu finden, so daß die Zahl der entstehenden Kreuzungen möglichst gering ist. Allerdings konnten Garey und Johnson (1983) nachweisen, daß es sich hierbei schon für $k = 2$ Schichten um ein $\mathcal{NP}$-schweres Problem handelt. Eades, McKay und Wormald (1986) wiesen nach, daß das 2-Schichten-Kreuzungsminimierungsproblem selbst dann noch $\mathcal{NP}$-schwer ist, wenn die Ecken auf einer der beiden Schichten in ihren Positionen fixiert werden.

Aufgrund der Schwere der Probleme wurde bislang lediglich versucht, durch lokale Kreuzungsminimierung die Zahl aller Kreuzungen zu verringern. Dabei wird der Graph Schicht für Schicht durchlaufen und unter Verwendung einer „guten“ Heuristik zur Minimierung der Kreuzungen in 2-Schicht-Graphen versucht, die Zahl der Kreuzungen zwischen je zwei benachbarten Schichten zu minimieren. Die anfänglichen Heuristiken in diesem Verfahren sind inzwischen durch reifere Ansätze wie dem „Branch-and-Cut“ Verfahren von Jünger und Mutzel (1997) ersetzt worden, die für den Fall, daß eine der beiden Schichten fest gewählt wird, vielfach nicht nur beweisbar gute, sondern auch optimale Lösungen bestimmen.

Der Ansatz von Sugiyama, Tagawa und Toda (1981) erfreut sich ungebrochener Beliebtheit, was sich zum einen mit der für praxisrelevante Probleme geeigneten Darstellung erklären läßt. Zum anderen aber zeichnet sich dieser Ansatz vor allem durch eine für Fachfremde leichte Verständlichkeit aus und ist überdies mit Hilfe einer einfachen Kreuzungsminimierungsheuristik leicht zu implementieren. Allerdings können für den Fall von drei und mehr Schichten Zeichnungen der geschichteten Graphen erstellt werden, in denen die Zahl der auftretenden Kreuzungen ein Vielfaches der minimal möglichen Zahl von Kreuzungen übersteigt. Dies ist natürlich bedingt durch die stark eingeschränkte lokale Sicht während der Kreuzungsminimierung zwischen zwei benachbarten Schichten. Im schlimmsten Fall werden so Graphen mit Kreuzungen gezeichnet, die sich tatsächlich kreuzungsfrei zeichnen lassen. Ansätze zur Minimierung der Zahl der Kreuzungen unter gleichzeitiger Berücksichtigung aller Schichten stecken noch in den Anfängen (ein erster Ansatz dazu ist bei Jünger, Lee, Mutzel und Odenthal (1997) verzeichnet).

Ein weiterer möglicher Ansatz, um ästhetisch ansprechende Zeichnungen zu erzeugen, wurde von Mutzel (1996) aufgezeigt. Statt das $k$-Schichten-Kreuzungsminimierungsproblem zu betrachten, soll versucht werden das $\mathcal{NP}$-schwere *$k$-Schichten-Planarisierungsproblem* mit Hilfe von Branch-and-Bound Methoden zu lösen. Dabei muß die minimale Zahl von Kanten bestimmt werden, deren Entfernen den Graphen (unter Berücksichtigung der Schichtung) planarisiert. Anschließend wird eine kreuzungsfreie Zeichnung des Graphen bestimmt und die entfernten Kanten werden so in die Zeichnung eingefügt, daß die Zahl der dadurch entstehenden Kreuzungen klein bleibt. Um diese Verfahren anwenden zu können, muß allerdings ein Einbettungsalgorithmus existieren, der für einen geschichteten Graphen eine planare Einbettung bestimmt, die die Schichtung des Graphen berücksichtigt.

Hier setzt die vorliegende Arbeit an. Zum einen ist es wünschenswert, einen geschichteten Graphen, der sich kreuzungsfrei zeichnen läßt, der somit schichtplanar ist, als solchen zu erkennen und eine entsprechende Einbettung zu bestimmen. Zum anderen eröffnet ein Einbettungsalgorithmus für schichtplanare Graphen über eine Verwendung im $k$-Schichten-Planarisierungsproblem eine sinnvolle Alternative zum $k$-Schichten-Kreuzungsminimierungsproblem.

Einen ersten Schichtplanaritätstest entwickelten Di Battista und Nardelli (1988) für die eingeschränkte Klasse der Hierarchien. Eine Hierarchie ist ein Schichtgraph mit genau einer Quelle. Beginnend mit der ersten Schicht überprüft der Algorithmus sukzessive, ob

der durch die ersten $j$ Schichten induzierte Untergraph $G^j$ schichtplanar ist. Dieses Vorgehen kann unter Verwendung der Datenstruktur $PQ$-Baum so implementiert werden, daß ein Test in $\mathcal{O}(n)$ Zeit durchführbar ist. Di Battista und Nardelli (1988) gaben ferner eine Modifikation ihres Algorithmus an, die zusätzlich die Bestimmung einer schichtplanaren Einbettung erlaubt. Der Ansatz kann allerdings nicht bei generellen Schichtgraphen appliziert werden, da $G^j$ im allgemeinen nicht zusammenhängend ist.

Eine Erweiterung dieses Schichtplanaritätstestes wurde von Heath und Pemmaraju (1995) präsentiert. Diese Erweiterung sollte es erlauben, auch allgemeine Schichtgraphen auf Schichtplanarität zu testen. Dabei wird für jede Komponente $F$ von $G^j$ ein $PQ$-Baum eingeführt, der die Menge der schichtplanaren Einbettungen von $F$ repräsentiert. Sind zwei Komponenten $F_1$ und $F_2$ von $G^j$ inzident zu derselben Ecke in Schicht $j+1$, müssen die zu $F_1$ und $F_2$ korrespondierenden $PQ$-Bäume $T_1$ und $T_2$ zu einem neuen $PQ$-Baum kombiniert werden.

In dem erweiterten Schichtplanaritätstest konnten wir allerdings tiefgreifende Defizite feststellen, die dazu führen, daß sich schichtplanare Graphen durch den Algorithmus nicht als solche identifizieren lassen. Ferner behaupten Heath und Pemmaraju (1995), daß der von ihnen präsentierte Ansatz nur $\mathcal{O}(n)$ Zeit benötigt. Dies ist allerdings nicht nachvollziehbar, da gewisse Aspekte (z.B. Update-Operationen) nicht analysiert werden und für den dargestellten Algorithmus nur eine Laufzeit von $\mathcal{O}(n \log n)$ erreichbar ist. Gänzlich unbearbeitet ließen Heath und Pemmaraju (1995) einen Einbettungsalgorithmus für schichtplanare Graphen.

In dieser Arbeit analysieren wir die in dem Schichtplanaritätstest von Heath und Pemmaraju (1995) auftretenden Defizite. Durch die Entwicklung neuer Methoden können wir die auftretenden Defizite umgehen und einen $\mathcal{O}(n \log n)$ Schichtplanaritätstest entwickeln. Durch eine konzeptionelle Änderung im Ablauf des Algorithmus sowie der Verwendung weiterer neuer Methoden ist es außerdem gelungen, einen Schichtplanaritätstest mit $\mathcal{O}(n)$ Laufzeit zu entwickeln.

Des weiteren wurde ein Einbettungsalgorithmus für schichtplanare Graphen mit linearer Laufzeit entwickelt. Dem Einbettungsalgorithmus liegt, basierend auf dem Schichtplanaritätstest, ein aus drei Phasen bestehendes Konzept zugrunde. In den ersten beiden Phasen wird unter Berücksichtigung der Schichtplanarität durch Einfügen von zusätzlichen Kanten ein planarer $st$-Graph erzeugt. Eine beliebige topologische Ordnung der Ecken dieses $st$-Graphen induziert eine $st$-Numerierung und basierend auf dieser Numerierung wird eine herkömmliche planare Einbettung mit Hilfe des Algorithmus von Chiba, Nishizeki, Abe und Ozawa (1985) erzeugt. Diese herkömmliche planare Einbettung läßt sich dann auf eine schichtplanare Einbettung zurückführen.

Im folgenden geben wir eine Übersicht über die einzelnen Kapitel der Arbeit. Nach einer Einführung im ersten Kapitel werden im zweiten Kapitel die für die Arbeit grundlegenden graphentheoretischen Begriffe eingeführt.

Im dritten Kapitel wird die für den Schichtplanaritätstest und Einbettungsalgorithmus sehr wichtige Datenstruktur der $PQ$-Bäume eingeführt. Mit Hilfe der von Booth und Lueker

(1976) entwickelten $PQ$-Bäume kann man für eine endliche Menge $U$ genau die Permutationen über $U$ darstellen, in denen für $n$ Teilmengen $S_i \subseteq U$, $i = 1, 2, \ldots, n$, alle Elemente von $S_i$ eine zusammenhängende Teilfolge bilden. Für jede Probleminstanz bestehend aus einer Menge $U$ und Teilmengen $S_i$, $i = 1, 2, \ldots, n$, kann ein solcher $PQ$-Baum in linearer Zeit bestimmt werden. Eine solche Konstruktion wird als Reduktion der Mengen $S_i$ bezeichnet. Des weiteren wird im dritten Kapitel der auf den $PQ$-Bäumen basierende Planaritätstest für herkömmliche Graphen vorgestellt, um die grundlegende Strategie der „Ecken-Addition“ des in dieser Arbeit vorgestellten Planaritätstests darzulegen. Der auf diesem Planaritätstest für herkömmliche Graphen basierende Einbettungsalgorithmus von Chiba, Nishizeki, Abe und Ozawa (1985) wird im Hinblick auf seine Verwendung in der dritten Phase des Einbettungsalgorithmus für schichtplanare Graphen ebenfalls vorgestellt. Da sich in jüngerer Zeit gezeigt hat, daß die Applikation der $PQ$-Baum Datenstruktur nicht immer unproblematisch ist, werden noch zwei weitere Algorithmen vorgestellt, die ebenfalls $PQ$-Bäume verwenden. Dabei handelt es sich zum einen um einen $c$-Planaritätstest für Clustergraphen von Feng, Cohen und Eades (1995) als eine gelungene Adaption der Datenstruktur. Zum anderen handelt es sich um einen Ansatz von Jayakumar, Thulasiraman und Swamy (1989) und Kant (1992) zur Bestimmung eines maximal planaren Untergraphen in einem nicht planaren Graphen, bei dem wir in dieser Arbeit ein grundsätzliches Defizit nachweisen können, das nahelegt, zur Lösung dieses Problems keine $PQ$-Bäume zu verwenden.

Im vierten Kapitel wird ein Schichtplanaritätstest für allgemeine Schichtgraphen entwickelt. Dazu wird zunächst die von Di Battista und Nardelli (1988) gefundene Beschreibung der Schichtplanarität von Hierarchien mittels verbotener Untergraphen eingeführt, von der Healy und Kuusik (1998) nachweisen konnten, daß diese Beschreibung ebenfalls eine Beschreibung der Schichtplanarität für allgemeine, geschichtete Graphen ist. Der Schichtplanaritätstest für Hierarchien verwaltet im wesentlichen einen $PQ$-Baum. Mit Hilfe dieses $PQ$-Baumes lassen sich für einen durch die ersten $j$ Schichten induzierten Untergraphen $G^j$ alle Permutationen der Ecken auf der Schicht $j$, die in einer beliebigen schichtplanaren Einbettung existieren, darstellen. Im Übergang vom $PQ$-Baum für $G^j$ zum $PQ$-Baum für $G^{j+1}$ wird die von Booth und Lueker (1976) entwickelte Reduktion auf Mengen von eingehenden Kanten, die zu derselben Ecke in $V^{j+1}$ inzident sind, angewandt. Im Ansatz von Heath und Pemmaraju (1995) wird für jede Komponente von $G^j$ ein solcher $PQ$-Baum verwaltet. Falls Komponenten zu derselben Ecke in $V^{j+1}$ inzident sind, so werden die korrespondierenden $PQ$-Bäume zu einem neuen $PQ$-Baum zusammengefaßt. Im folgenden wird dies als „Merge“-Operation bezeichnet. Wir stellen in diesem Ansatz zwei grundlegende Defizite fest. Zum einen werden „singuläre“ Komponenten von $G^j$, das sind Komponenten, die genau nur zu einer Ecke aus $V^{j+1}$ inzident sind, nicht korrekt bearbeitet. Des weiteren fassen Heath und Pemmaraju (1995) $PQ$-Bäume in beliebiger Reihenfolge zusammen. Dadurch entstehen $PQ$-Bäume, die nicht alle schichtplanaren Einbettungen der korrespondierenden Komponenten darstellen. Durch diese Defizite werden schichtplanare Graphen nicht als solche identifiziert. Die Behebung der Defizite gelingt uns durch die Verwendung zweier neuer Konzepte. Durch die Speicherung und Verwaltung gewisser Informationen über

innere Länder sowie Gebiete des äußeren Landes in möglichen schichtplanaren Einbettungen können wir eine korrekte Behandlung singulärer Komponenten garantieren. Ferner ist es gelungen zu beweisen, daß bei einer Sortierung der $PQ$-Bäume nach Größe und anschließender Zusammenfassung der $PQ$-Bäume gemäß dieser Größe, die so erzeugten $PQ$-Bäume tatsächlich alle schichtplanaren Einbettungen ihrer Komponenten repräsentieren. In Kombination mit den Merge-Operationen von Heath und Pemmaraju (1995) ergibt sich dadurch ein $\mathcal{O}(n \log n)$-Schichtplanaritätstest. Notwendige Update-Operationen in diesem $\mathcal{O}(n \log n)$-Ansatz können wir durch eine Veränderung im Ablauf des Algorithmus sowie weiteren konzeptionellen Neuerungen vermeiden. Dadurch verbessern wir die Laufzeit des Planaritätstests auf $\mathcal{O}(n)$ Zeit.

Die Entwicklung eines $\mathcal{O}(n)$-Einbettungsalgorithmus für schichtplanare Graphen wird im fünften Kapitel behandelt. Dem Algorithmus liegt ein Drei-Phasen-Modell zugrunde. Nach Hinzufügen einer weiteren Quelle $s$ auf einer zusätzlichen Schicht oberhalb eines schichtplanaren Graphen $G = (V, E)$ mit $k$-Schichten und einer weiteren Senke $t$ auf einer zusätzlichen Schicht unterhalb von $G$ wird in einer ersten Phase der Graph $G$ zu einer Hierarchie augmentiert. Dabei wird zu jeder Senke aus $V$ eine ausgehende Kante addiert, so daß die Schichtplanarität des Graphen nicht verletzt wird. Die zweite Phase addiert zu jeder Quelle aus V eine eingehende Kante, ohne die Schichtplanarität zu verletzen. Nach der Addition einer weiteren Kante $(s, t)$ handelt es sich bei dem so erzeugten Graphen um einen planaren $st$-Graphen, bei dem die topologische Ordnung der Ecken eine $st$-Numerierung induziert. Unter Verwendung dieser $st$-Numerierung wird in einer dritten Phase mittels des Einbettungsalgorithmus für herkömmliche Graphen eine planare Einbettung bestimmt, die sich anschließend auf eine schichtplanare Einbettung des ursprünglichen Graphen $G$ zurückführen läßt. Der für die zweite Phase zu verwendende Algorithmus ist, unter Umkehrung der Kantenrichtung in dem Graphen, mit dem Algorithmus der ersten Phase identisch. Die Augmentation in der ersten Phase basiert auf dem $\mathcal{O}(n)$-Schichtplanaritätstest. Um eine ausgehende Kante inzident an eine Senke einzufügen, wird für jede Senke $v \in V^j$, $1 \leq j < k$, in dem $PQ$-Baum $T$, in dem das zur Senke $v$ korrespondierende Blatt letztmalig auftritt, ein „Senkenindikator“ $\mathrm{si}(v)$ eingefügt. Dieser Senkenindikator $\mathrm{si}(v)$ ist ein Blatt in $T$, das in allen $PQ$-Baum Operationen zu ignorieren ist. Dadurch wird die Klasse der Permutationen des $PQ$-Baumes $T$ nicht verändert. Wir finden Bedingungen, unter denen $\mathrm{si}(v)$ als ein Blatt korrespondierend zu einer neuen Kante $(v, w)$, mit $w \in V^l$, $j < l \leq k + 1$, interpretiert werden kann, bei denen die Addition dieser Kante die Schichtplanarität nicht verletzt. Dabei müssen wir unterscheiden zwischen Bedingungen, die bei Operationen innerhalb eines $PQ$-Baumes zur Anwendung kommen und Bedingungen, die bei Merge-Operationen eingesetzt werden müssen. Bei Operationen innerhalb eines $PQ$-Baumes ist die Verwaltung der Senkenindikatoren unproblematisch. Anders jedoch stellt sich die Situation bei Merge-Operationen dar. Hier treten bei der Wahl von Senkenindikatoren für eine Kantenaugmentierung gewisse Freiheiten auf, die zu dem Zeitpunkt der Merge-Operation nicht entschieden werden können. Die Entscheidung über die Wahl der Senkenindikatoren kann mit Hilfe von „Kontakten“ auf einen Zeitpunkt verschoben werden, bei der während der Bearbeitung einer späteren Schicht die möglichen schichtplanaren

Einbettungen so verringert werden, daß zulässige Aussagen über die korrekte Wahl getroffen werden können. Mit Hilfe der Kontakte, bei denen es sich ebenfalls um Blätter im $PQ$-Baum handelt, die in allen Operationen zu ignorieren sind, konnten wir ein System entwickeln, das in $\mathcal{O}(n)$ Zeit einen schichtplanaren Graphen zu einer schichtplanaren Hierarchie augmentiert. Da die Zahl der so zusätzlich eingefügten Kanten beschränkt durch die Zahl der Ecken des Graphen ist, ergibt sich sowohl für die Augmentierung zum $st$-Graphen, als auch die anschließende planare Einbettung und somit für die schichtplanare Einbettung eine lineare Laufzeit.

Das sechste Kapitel gibt einen Überblick über unsere Implementierung eines Einbettungsalgorithmus für schichtplanare Graphen. Die objektorientierte Implementierung erfolgte in C++, wobei die Datenstruktur $PQ$-Baum als Klassentemplate in die Implementierung eingefügt wurde. Da die Implementierung selbst sehr umfangreich ist, konzentriert sich das Kapitel 6 nach einer Einführung in das verwendete Klassenkonzept auf die Darstellung dreier Prozeduren. Diese wurden so gewählt, daß sie eine sinnvolle und ergänzende Darstellung zu den im vierten und fünften Kapitel vorgestellten Operationen bilden, überdies viele Spezialfälle abdecken und somit einen Einblick in die Details der Implementierung ermöglichen.

Wir beschließen die Arbeit im siebten Kapitel mit einer Beschreibung der erreichten Resultate sowie mit einer Diskussion über die Möglichkeiten und Aufgaben kommender Forschung. Diese schließen insbesondere die praktische Verwendung des Einbettungsalgorithmus im $k$-Schichten-Planarisierungsverfahren ein. Eine weitere Aufgabe ist die Entwicklung von Prädikaten für nicht schichtplanare Graphen. Entsprechende Verfahren geben Benutzern eines Planaritätstests Sicherheit über das Resultat eines Tests.

# ERKLÄRUNG

Ich versichere, daß ich die von mir vorgelegte Dissertation selbständig angefertigt, die benutzten Quellen und Hilfsmittel vollständig angegeben und die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, in jedem Einzelfall als Entlehnung kenntlich gemacht habe; daß diese Dissertation noch keiner anderen Fakultät oder Universität zur Prüfung vorgelegen hat; daß sie – abgesehen von unten angegebenen Teilpublikationen – noch nicht veröffentlicht worden ist sowie, daß ich eine solche Veröffentlichung vor Abschluß des Promotionssverfahrens nicht vornehmen werde. Die Bestimmungen dieser Promotionsordnung sind mir bekannt. Die von mir vorgelegte Dissertation ist von Herrn Prof. Dr. Michael Jünger betreut worden.

Sebastian Leipert

## Teilpublikationen

- Jünger, M., Leipert, S., and Mutzel, P. (1997). Pitfalls of using PQ-trees in Automatic Graph Drawing. In G. Di Battista, editor, *Graph Drawing '97*, volume 1353 of *Lecture Notes in Computer Science*, pages 193–204. Springer Verlag.

- Jünger, M., Leipert, S., and Mutzel, P. (1998a). A note on computing a maximal planar subgraph using PQ–trees. *IEEE Transactions On Computer-Aided Design*, **17**(7), 609–612.

- Jünger, M., Leipert, S., and Mutzel, P. (1998b). Level planarity testing in linear time. Technical Report 98.321, Institut für Informatik, Universität zu Köln. To appear in: Graph Drawing '98, Lecture Notes in Computer Science, Springer Verlag.

# Lebenslauf

| | |
|---|---|
| **Name** | Sebastian Leipert |
| **Geburtsdatum** | 11.1.1970 |
| **Geburtsort** | München |
| **Nationalität** | deutsch |
| **Familienstand** | verheiratet |

## Schulausbildung

| | |
|---|---|
| **1976 – 1980** | Gemeinschaftsgrundschule Wittenbergstraße, Bergisch Gladbach |
| **1980 – 1989** | Otto-Hahn-Gymnasium, Bergisch Gladbach |

## Hochschulausbildung

| | |
|---|---|
| **1989–1992** | Studium der Mathematik mit Nebenfach Betriebswirtschaftslehre an der Universität zu Köln |
| **April 1992** | Vordiplom |
| **April 1992** | Wechsel im Nebenfach zum Studienfach Informatik |
| **Oktober 1995** | Abschluß: Diplom-Mathematiker |
| **1996–1998** | Promotionsstudium im Fach Informatik an der Universität zu Köln |
| **seit 1995** | Wissenschaftlicher Mitarbeiter am Institut für Informatik der Universität zu Köln, Lehrstuhl Prof. Dr. Michael Jünger |