Planar Graphs and their Duals on Cylinder Surfaces

Christopher Auer

Dissertation

Supervisor

Prof. Dr. Franz J. Brandenburg

January 2014

Abstract

In this thesis, we investigates plane drawings of undirected and directed graphs on cylinder surfaces. In the case of undirected graphs, the vertices are positioned on a line that is parallel to the cylinder's axis and the edge curves must not intersect this line. We show that a plane drawing is possible if and only if the graph is a double-ended queue (deque) graph, i. e., the vertices of the graph can be processed according to a linear order and the edges correspond to items in the deque inserted and removed at their end vertices. A surprising consequence resulting from these observations is that the deque characterizes planar graphs with a Hamiltonian path. This result extends the known characterization of planar graphs with a Hamiltonian cycle by two stacks. By these insights, we also obtain a new characterization of queue graphs and their duals. We also consider the complexity of deciding whether a graph is a deque graph and prove that it is $\mathcal{NP}$-complete. By introducing a split operation, we obtain the splittable deque and show that it characterizes planarity. For the proof, we devise an algorithm that uses the splittable deque to test whether a rotation system is planar.

In the case of directed graphs, we study upward plane drawings where the edge curves follow the direction of the cylinder's axis (standing upward planarity; **SUP**) or they wind around the axis (rolling upward planarity; **RUP**). We characterize **RUP** graphs by means of their duals and show that **RUP** and **SUP** swap their roles when considering a graph and its dual. There is a physical interpretation underlying this characterization: A **SUP** graph is to its **RUP** dual graph as electric current passing through a conductor to the magnetic field surrounding the conductor. Whereas testing whether a graph is **RUP** is $\mathcal{NP}$-hard in general [Bra14], for directed graphs without sources and sink, we develop a linear-time recognition algorithm that is based on our dual graph characterization of **RUP** graphs.

Preface

Writing such a thesis is the task of one, and yet it involves many more. I would like to express my gratitude to those who supported me in this task.

First of all, I would like to thank my advisor Prof. Dr. Franz Brandenburg for letting me pursue my own ideas while at the same time giving me valuable advice. The useful counsel and time Franz has shared with me during the years I have worked at his chair, finds its expression also in this thesis. I also would like to thank Prof. Dr. Ernst Mayr for co-supervising my thesis and for giving me the chance to present my ideas at his chair at the Technische Universität München.

During the last years, my colleagues have not only been appreciated discussion partners, they have also become friends. I would like to say my thanks to (in alphabetical order): Christian Bachmaier, Wolfgang Brunner, Andreas Gleißner, Kathrin Hanauer, Andreas Hofmeier, Marco Matzeder, Daniel Neuwirth, Josef Reislhuber, and Patrick Wüchner. I am in particular grateful to Andreas Hofmeier and Josef Reislhuber for their useful feedback on Chapter 2, and to Kathrin Hanauer for carefully proof-reading Chapters 3 and 4. A special thanks goes to Wolfgang Brunner for his helpful feedback and suggestions for improvement for the algorithm in Sect. 3.5.

To my wife, Bernadette, who rejoiced with me during the highs and lifted me up during the lows that both inevitably come when writing a thesis, I am also deeply grateful. She shared with me her viewpoint, helped me to focus on the important things, and the completion of this thesis must also be credited to her. I would also like to thank my family, especially my father Werner Auer, for always supporting me in my decisions, my parents in law, Brigitte and Jakob, my sister Bianca and her husband Wolfgang.

Christopher Auer

Remarks on the Photographs

With photography I have found an activity during the last years that is technically and, in particular, artistically demanding. As both this thesis and photography have been a central part of my life during that years, I have decided to introduce each chapter with one of my photographs.

Contents

Preface i

Contents iii

1 Introduction 3

1.1 Preliminaries . . . 4
1.1.1 Directed Graphs . . . 5
1.1.2 Undirected Graphs . . . 5
1.1.3 Multigraphs . . . 5
1.1.4 Isomorphism . . . 6
1.1.5 Subgraphs . . . 6
1.1.6 Connectivity . . . 6
1.1.7 Planar Graphs . . . 7
1.1.8 Dual Graphs . . . 9
1.1.9 Cylinder Surfaces . . . 11

2 Deque Layouts and Linear Cylindric Drawings 15

2.1 Introduction . . . 15
2.1.1 Graph Layouts . . . 16
2.1.1.1 Stack Graphs . . . 16
2.1.1.2 Queue Graphs . . . 19
2.1.1.3 Contributions of this Thesis . . . 20
2.1.2 Other Approaches to Study Fundamental Data Structures . . . 22
2.1.2.1 Permutation Networks . . . 22
2.1.2.2 Automata and Formal Languages . . . 25
2.2 Deque Graphs and Linear Cylindric Drawings . . . 26
2.2.1 Linear Cylindric Drawings . . . 27
2.2.2 Deque Layouts . . . 30
2.2.2.1 The Deque Data Structure . . . 30
2.2.2.2 Graph Layouts in the Deque . . . 31
2.2.3 Linear Cylindric Planar Graphs and Deque Graphs . . . 32
2.2.4 Deque Graphs and Hamiltonian Paths . . . 41
2.2.4.1 Comparing Two Stacks with a Single Deque . . . 44
2.2.4.2 Duals of Deque Graphs . . . 44
2.2.5 Deciding whether a Graph is a Deque Graph is $\mathcal{NP}$-Complete . . . 48
2.3 Linear Cylindric Drawings of Deque-Reducible Data Structure and Mixed Layouts 52

2.4 Queue Graphs . . . 53
2.4.1 Linear Cylindric Drawings of Queue Graphs . . . 54
2.4.2 Proper Leveled Planar Graphs and Bipartite Queue Graphs . . . 55
2.4.3 Duals of Queue Graphs . . . 57
2.5 Characterizing Planarity by the Splittable Deque . . . 65
2.5.1 Depth-First Search Trees . . . 65
2.5.2 Splittable Deque Layouts . . . 67
2.5.2.1 The Splittable Deque Data Structure . . . 67
2.5.2.2 Graph Layouts in the Splittable Deque . . . 68
2.5.3 Testing Planarity of a Rotation System by the Splittable Deque . . . 70
2.5.4 Remarks on the Running Time of `IsSDLayout` . . . 81
2.5.5 Relating Splittable Deque Layouts to the de Fraysseix-Rosenstiehl Planarity Criterion . . . 82
2.5.6 Testing Planarity by Switching Trains . . . 84
2.6 Summary, Further Remarks, and Future Work . . . 87
2.6.1 Further Considerations of Complexity Issues . . . 88
2.6.1.1 Restricted Problems . . . 89
2.6.1.2 Fixed-Parameter Tractability . . . 89
2.6.2 Open Gauß Codes and Deque Layouts . . . 91
2.6.3 Dual Layouts . . . 93
2.6.4 General Planarity Testing by the Splittable Deque . . . 95

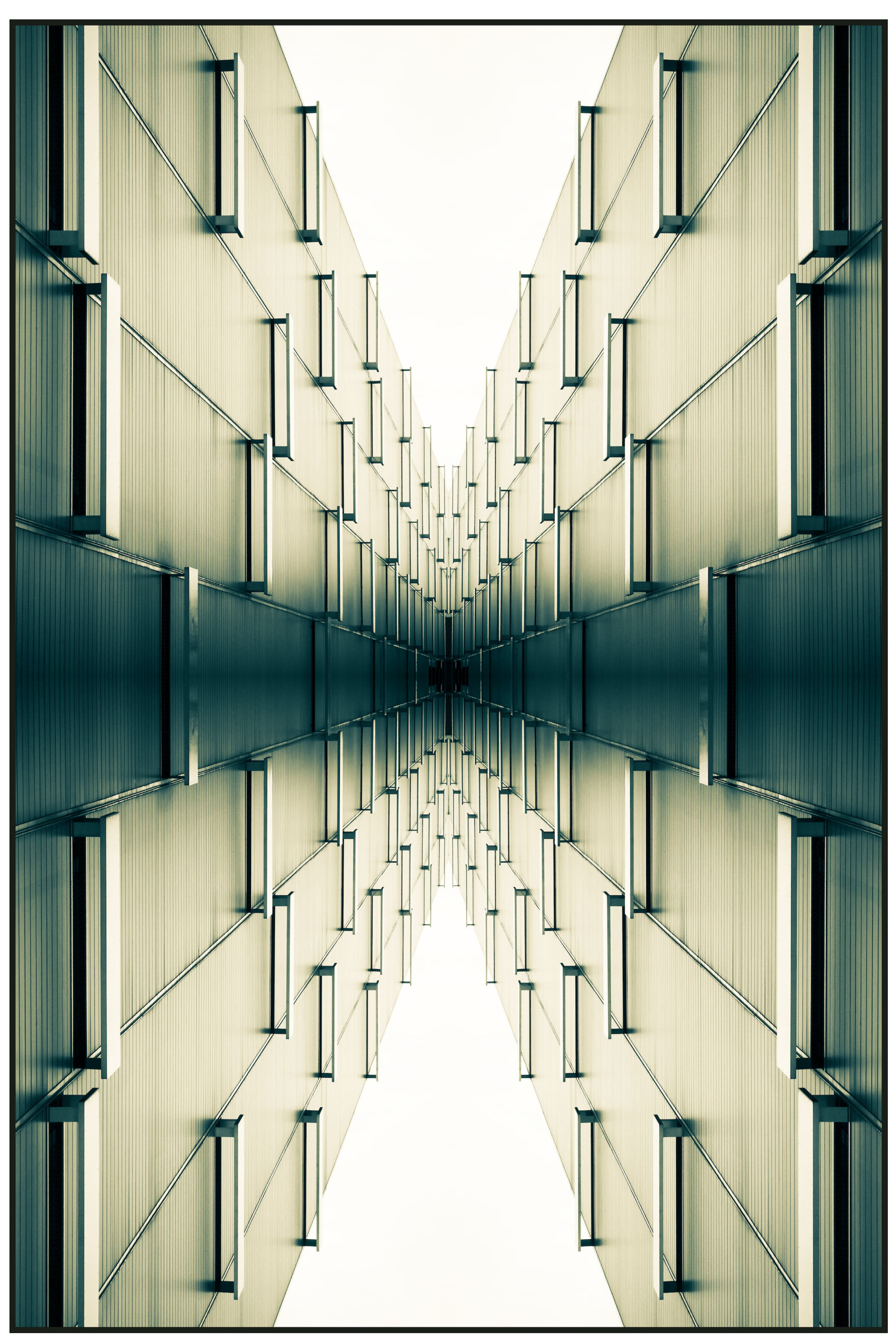
3 Rolling Upward Planar Digraphs **97**
3.1 Introduction . . . 97
3.2 Classification of Upward Planarity . . . 99
3.2.1 Upward Planar Digraphs . . . 100
3.2.2 Standing Upward Planar Digraphs . . . 101
3.2.3 Rolling Upward Planar Digraphs . . . 102
3.2.4 Other Cases of Upward Planarity . . . 103
3.3 Contributions of this Thesis . . . 105
3.4 Characterizing **RUP** Digraphs by their Duals . . . 107
3.4.1 The Compound Digraph . . . 107
3.4.2 Directed Duals and Dicuts . . . 109
3.4.3 **RUP** Digraphs and their Duals . . . 110
3.5 Efficient Rolling Upward Planarity Testing of Closed Digraphs . . . 123
3.5.1 Closed Digraphs . . . 123
3.5.1.1 Transits and Compounds of Closed **RUP** Digraphs . . . 124
3.5.1.2 The Algorithm for Closed Digraphs . . . 131
3.5.2 Compounds . . . 137
3.5.2.1 Cut Vertices and Cut Faces . . . 137
3.5.2.2 Block-Cut Trees of Acyclic Dipoles . . . 142
3.5.2.3 Characterizing **RUP** Compounds by their (Directed) Block-Cut Trees . . . 144
3.5.2.4 The Algorithm for Compounds . . . 152
3.5.3 Biconnected Compounds . . . 158
3.5.3.1 A Primer on SPQR Trees . . . 159
3.5.3.2 Dual SPQR Trees . . . 164

3.5.3.3 dSPQR Trees of Acylic Digraphs 168
3.5.3.4 dSPQR Trees of Acyclic Dipoles 171
3.5.3.5 dSPQR Trees of Compounds 176
3.5.3.6 Dual dSPQR Trees of Compounds 179
3.5.3.7 dSPQR Trees of **RUP** Compounds 181
3.5.3.8 The Algorithm for Biconnected Compounds 187
3.6 **wSUP** Digraphs 218
3.7 Summary, Further Remarks, and Future Work 222
3.7.1 Minors of Non-**RUP** Compounds 224
3.7.2 Upward Toroidal Digraphs 227
3.7.3 Drawing **RUP** Digraphs 229
3.7.4 Further Complexity Considerations 230
3.7.4.1 Other Digraph Classes 230
3.7.4.2 Fixed-Parameter Tractability 231
3.7.5 Acyclic **RUP** Digraphs 232

4 Bringing Chapters 2 and 3 Together **235**

Bibliography **241**

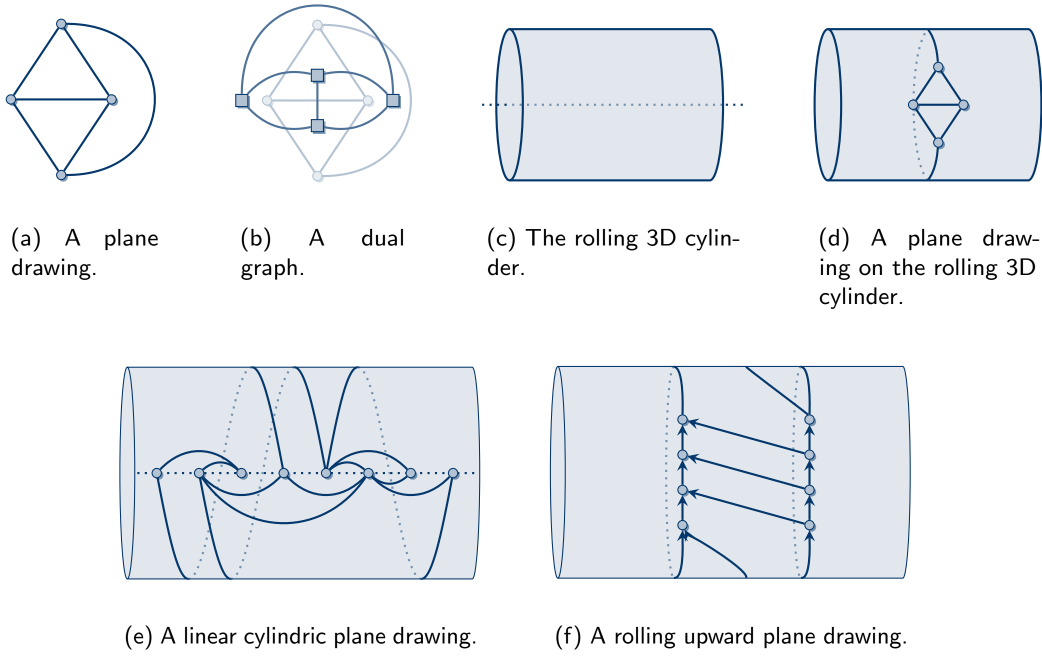
Index **253**



Chapter 1

Introduction

A graph consists of a finite set of entities, the vertices, and a binary relation on the entities, the edges. A graph is undirected if the relation is symmetric. Otherwise, we have a digraph, i. e., a directed graph, in which each edge points from its source to its target. As a general purpose-tool, graphs have a wide range of applications, in both theoretical and practical scenarios.



(a) A plane drawing.

(b) A dual graph.

(c) The rolling 3D cylinder.

(d) A plane drawing on the rolling 3D cylinder.

(e) A linear cylindric plane drawing.

(f) A rolling upward plane drawing.

Figure 1.1: Plane drawings on the Euclidean plane and the surface of a 3D cylinder.

A particularly interesting class of graphs are the planar graphs: A graph is called planar if it has a plane drawing which maps the vertices to distinct points in the Euclidean plane and the edges to simple Jordan arcs that connect the edges' endpoints such that no two edge curves share a point except for common endpoints. Fig. 1.1(a) shows an example of a plane drawing. Planar graphs have been studied extensively in the past and still play an important role, for instance, in the research field of graph drawing that deals with the graphical representation of

graphs. With every plane drawing comes a dual graph which can be obtained by placing (face) vertices into the regions that are bounded by the drawing and connecting two face vertices by a (dual) edge if they are separated by an original edge (Fig. 1.1(b)).

In this thesis, we investigate plane drawings on the rolling cylinder (cf. Fig. 1.1(c)): Instead of the Euclidean plane, the vertices and edges are drawn on the surface of a three-dimensional cylinder whose axis (dotted) is parallel to the x-axis. An example is shown in Fig. 1.1(d), where one edge winds around the cylinder (dotted). Every planar graph has a plane drawing on the rolling cylinder and vice versa and, thus, plane cylinder drawings are no extension of planarity. Still, cylinder drawings become interesting when we introduce restrictions of which we consider two types:

- **Linear Cylindric Drawings of Undirected Graphs**
  The vertices must be placed on a horizontal line parallel to the cylinder's axis and no edge curve must cross this line (Fig. 1.1(e)).

- **Rolling Upward Plane Drawings of Digraphs**
  Each edge curve, from its source to its target, must wind around the cylinder's axis in a certain direction (Fig. 1.1(f)).

These two types of drawings correspond to the two parts of this thesis, which is organized as follows: We give basic definitions that are relevant for all chapters in Sect. 1.1. In Chapter 2, we introduce linear cylindric drawings (Fig. 1.1(e)) and investigate their relationship to graph layouts in fundamental data structures as, for instance, the stack, the queue, and, in particular, the double-ended queue (deque). It turns out that the structure of dual graphs obtained from plane linear cylindric drawings is closely related to the working principle of the deque and the queue. Moreover, we will characterize all planar graphs by using a slightly modified version of the deque, the splittable deque. Rolling upward plane drawings (Fig. 1.1(f)) are the topic of Chapter 3. We give a combinatorial characterization of digraphs that admit such a drawing by means of their (directed) dual graphs. Based on this characterization, we develop an algorithm that decides whether a digraph without sources and sinks admits a rolling upward plane drawing. In Chapter 4, we bring Chapters 2 and 3 together.

1.1 Preliminaries

This section contains the basic definitions we use throughout the whole thesis. The definitions are made along the line with standard text books about graph theory [Eve12] and graph drawing [DBETT99, KW01].

First, we fix some notation. For two real numbers $a < b$, we denote by $[a, b] := \{x \in \mathbb{R} \mid a \leq x \leq b\}$ the closed interval and by $(a, b) := \{x \in \mathbb{R} \mid a < x < b\}$ the open interval from a to b. The cardinality, i. e., number of elements, of a finite set A is denoted by $|A|$. $\mathcal{P}_k(\mathbf{A}) := \{A \subseteq \mathbf{A} \mid |A| = k\}$ denotes the set of subsets of $\mathbf{A}$ with cardinality k where $k \leq |\mathbf{A}|$. For any function $f : \mathbf{A} \to \mathbf{B}$ and set $A \subseteq \mathbf{A}$, we define by $f[A] := \{f(a)|a \in \mathbf{A}\}$ the image of A under f. For a tuple $\mathbf{a} = (a_1, a_2, \dots, a_k)$, we define by $f(\mathbf{a}) := (f(a_1), f(a_2), \dots, f(a_k))$ the coordinate-wise application of f. For any sequence, $a_1, a_2, a_3, \dots, a_k$ we will use the shortened notation $a_1, \dots, a_k$ whenever the index between two successive elements increases by one.

1.1.1 Directed Graphs

A *digraph* or *directed graph* G is a tuple $G = (V, E)$ with a finite set V of vertices and a set E of edges. The set of edges is a binary relation $E \subseteq V \times V$ consisting of directed edges $(u, v) \in E$. For each edge $e = (u, v) \in E$, u and v are called *endpoints* of e, where u is the *source* and v is the *target* of e. We call $\mathrm{E}^+(v) := \{e \in E \mid v \text{ is source of } e\}$ the set of *outgoing edges of* v and $\mathrm{E}^-(v) := \{e \in E \mid v \text{ is target of } e\}$ the set of *incoming edges of* v. The *outdegree* $\mathrm{d}^+(v)$ *of* v is the cardinality of $\mathrm{E}^+(v)$ and the *indegree* $\mathrm{d}^-(v)$ *of* v is the cardinality of $\mathrm{E}^-(v)$. A vertex v with $\mathrm{d}^-(v) = 0$ is called *source* and if $\mathrm{d}^+(v) = 0$, v is called *sink*.

A pair of vertices $u, v \in V$ is said to be *adjacent* if they are connected by an edge, i. e., $(u, v) \in E \vee (v, u) \in E$. Then, e is said to be *incident to* u *and* v. An edge $(v, v) \in E$ is called a *directed loop* and G is called *simple digraph* if it contains no directed loops.

A *dipath* p *starting at vertex* v_0 *and ending at vertex* v_k is a sequence $p = (v_0, e_1, v_1, e_2, \dots, e_k, v_k)$ of vertices and edges with $e_i = (v_{i-1}, v_i) \in E$ for all $1 \leq i \leq k$, where $k \geq 0$. If $k = 0$, p stays at vertex v_0. For convenience, we only list the vertices of a dipath and include the edges only when necessary, e. g., in the context of multigraphs and duals as defined later. As a shorthand, we write $p = v_0 \rightsquigarrow v_k$ to denote a dipath from vertex v_0 to vertex v_k. The number k of edges involved in p is the *length of* p. Each vertex v_i with $0 \leq i \leq k$ is said to *be visited by* p or, simply, to *be on* p. Likewise, each edge e_i $(1 \leq i \leq k)$ is *visited by* p or, simply, *is on* p. A dipath is called *simple* if $v_i \neq v_j$ for all $0 \leq i < j \leq k$. A dipath $p = v_0 \rightsquigarrow v_k$ is called *cycle* if $v_0 = v_k$ and it is called *simple cycle* if all vertices on p are distinct except for $v_0 = v_k$. A digraph is called *acyclic* if it contains no cycles.

A simple dipath p is called *Hamiltonian* if each vertex $v \in V$ is visited by p. A dipath p is called *Eulerian* if each edge $e \in E$ is visited exactly once by p. Analogously, a simple cycle is called *Hamiltonian* if it visits each vertex and it is called *Eulerian* if it visits each edge exactly once.

1.1.2 Undirected Graphs

In contrast to digraphs, the edges in an *undirected graph* or, simply, *graph* $G = (V, E)$ have no designated direction. Each edge $e \in E$ is an unordered pair $\{u, v\}$ with $u \neq v$ and, thus, $E \subseteq \mathcal{P}_2(V)$. The definition of endpoints, adjacency, and incidence are defined for the undirected case as for the directed case. The set of incident edges of a vertex $v \in V$ is denoted by $\mathrm{E}(v)$ and the cardinality of $\mathrm{E}(v)$ is the *degree* $\mathrm{d}(v)$ *of* v. The definitions of (simple) dipath and (simple) cycles are adopted for the undirected case, where a dipath is called *path* and a cycle is called *circle*, both possibly with the attributes of being simple, Hamiltonian, or Eulerian. An *undirected loop* from v onto itself is a unary set $\{v\}$ and a graph is called *simple*, if it contains no undirected loops. We call a simple graph $G = (V, E)$ *complete* if there is an edge between every pair of distinct vertices, i. e., $E = \mathcal{P}_2(V)$.

Given a digraph $G = (V, E)$, the *underlying undirected graph* $G' = (V, E')$ is obtained from G be discarding the edge directions. More formally, $E' = \{\{u, v\} \mid u \text{ and } v \text{ are adjacent in } G\}$. Note that two distinct edges $(u, v), (v, u) \in E$ are mapped to a single undirected edge in G'.

1.1.3 Multigraphs

From time to time, we need to model that two vertices are connected by more than one edge and that a single vertex has more than one loop. In this case, we obtain a *multigraph*

$G = (V, E, \epsilon)$, which again can be directed or undirected. In either case, we have a set of edges $E = \{e_1, e_2, \ldots, e_m\}$ and a function $\epsilon : E \to V \times V$ for the directed case and a function $\epsilon : E \to \mathcal{P}_1(V) \cup \mathcal{P}_2(V)$ for the undirected case. In both cases, ϵ maps an edge to its endpoints, i. e., an (un-)ordered pair or a (un-)directed loop. All other aforementioned definitions are adopted for multigraphs as well. Referring to an edge e of a multigraph by using its endpoints $\{u, v\}$ (or (u, v)) is ambiguous in general. Nevertheless, we use the latter notation whenever it leads to a conciser description without causing confusion.

1.1.4 Isomorphism

Two graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there is a bijection $\pi : V \to V'$ with $\{u, v\} \in E$ if and only if $\{\pi(u), \pi(v)\} \in E'$. Analogously, two digraphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there is a bijection $\pi : V \to V'$ with $(u, v) \in E$ if and only if $(\pi(u), \pi(v)) \in E'$, and two multigraphs $G = (V, E, \epsilon)$ and $G' = (V', E', \epsilon')$ are isomorphic if there are bijections $\pi_V : V \to V'$ and $\pi_E : E \to E'$ such that for all $e \in E$ $\pi_V[\epsilon(e)] = \epsilon'(\pi_E(e))$ in the undirected case, and $\pi_V(\epsilon(e)) = \epsilon'(\pi_E(e))$ in the directed case. Although, isomorphism is a weaker condition that equality, we will denote by $G = G'$ that G is isomorphic to G' unless stated otherwise.

1.1.5 Subgraphs

The following definitions equally apply to all previously defined types of graphs. We, therefore, simply speak of "graphs" for now. For multigraphs we assume, with a slight abuse of notation, that the mapping between the edges and its endpoints is equal for graph G and its subgraphs.

Given two graphs $G = (V, E)$ and $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$, we say that *G' is a subgraph of G* and *G is a supergraph of G'*, denoted by $G' \subseteq G$. If $V = V'$, then *G' is a spanning subgraph of G* and *G a spanning supergraph of G'*. For $E' \subseteq E$, the graph $G_{E'} = (V, E')$ is the subgraph *induced by the edges E'*. The graph $G \setminus E' = (V, E \setminus E')$ is the subgraph *obtained by removing the edges E' (from G)*. For $U \subseteq V$, the graph $G' = (U, E_U)$ with $E_U = \{e \in E \mid \text{both endpoints of } e \text{ are in } U\}$ is the subgraph *induced by the vertices U*. Analogously, $G \setminus U = (V \setminus U, E_{\setminus U})$ is the subgraph *obtained by removing the vertices U (from G)* with $E_{\setminus U} = \{e \in E \mid \text{neither endpoint of } e \text{ is in } U\}$.

1.1.6 Connectivity

There are two types of connectivity: In case of digraphs, we speak of *strongly connected components* and, in the case of undirected graphs, we have *connected, biconnected and triconnected components*.

Strongly Connected Components Let $G = (V, E)$ be a digraph. As a reminder, $u \rightsquigarrow v$ indicates the existence of a dipath from u to v, where $u = v$ is possible. If $u \rightsquigarrow v$ and $v \rightsquigarrow w$, then also $u \rightsquigarrow w$. Hence, $\rightsquigarrow$ is a transitive and reflexive binary relation on V that defines equivalence classes on V. We denote by $\sigma(v)$ the equivalence class to which $v \in V$ belongs, i. e., $\sigma(v) = \{w \in V \mid v \rightsquigarrow w \wedge w \rightsquigarrow v\}$. Denote by $\mathcal{V} := \{\sigma(v) \subseteq V \mid v \in V\}$ the set of equivalence classes. For any $v \in V$, the subgraph $G' = (\sigma(v), E')$ of G induced by $\sigma(v)$ is called the *strongly connected component* or, simply, *component* of G. A component either contains at least one edge or consists of a single vertex with no edge. In the latter case, we call

the component *trivial*. Note that by this definition a component consisting of a single vertex with a directed loop is not trivial.

Let $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ be the digraph, where the vertices are the components of a digraph $G = (V, E)$. For each edge $(u, v) \in E$ from one component $\sigma(u)$ to another component $\sigma(v)$, there is a unique edge $(\sigma(u), \sigma(v))$ in $\mathbb{E}$. Note that in this case, there is a dipath from all vertices in $\sigma(u)$ to all vertices in $\sigma(v)$. $\mathbb{G}$ is a multigraph and called the *digraph of strongly connected components* or simply *component digraph*. Note that $\mathbb{G}$ is acyclic.

If a digraph G itself is a strongly connected component, then G is called *strongly connected*. Note that a strongly connected digraph has neither sources nor sinks. However, the converse is not true as there are digraphs without sources and sinks that are not strongly connected.

Connected, Biconnected and Triconnected Components The following definitions apply to undirected graphs and multigraphs, and they canonically extend to digraphs by considering their underlying undirected graphs. A graph G is called *(vertex-)connected* if there is a path between every pair of distinct vertices. Otherwise, G is called *unconnected*. If G contains no or only one vertex, we define G to be connected. For a connected graph, a vertex $v \in V$ is called a *cut vertex* if $G \setminus \{v\}$ is unconnected. A connected graph is called *biconnected* if it has no cut vertex. If G is biconnected, a pair of distinct vertices $u, v \in V$ is called *split pair* if $G \setminus \{u, v\}$ is unconnected. If G has no split pair, it is called *triconnected*.

The subgraph G' of a graph G is called *connected component of* G if G' is connected and maximal with respect to $\subseteq$, i. e., for any connected subgraph $G'' \subseteq G$, $G' \subseteq G''$ implies that $G'' = G'$. A biconnected subgraph which is maximal with respect to $\subseteq$ is called *block*.

Let $\mathbf{B} = \{B_1, \ldots, B_k\}$ be the set of blocks of a connected graph $G = (V, E)$ and denote by $\mathbf{C}$ the set of cut vertices of G. The *block-cut tree* $\mathcal{T}_B = (\mathbf{B}, \mathbf{C}, \mathbf{E}_B)$ of G is a graph whose vertices are the blocks and cut vertices. There is an edge $\{B_i, c\} \in \mathbf{E}_B$ if block $B_i = (V_i, E_i)$ contains c, i. e., $c \in V_i$. A block-cut tree is connected as G is connected and it does not contain a circle, i. e., it is a *tree*.

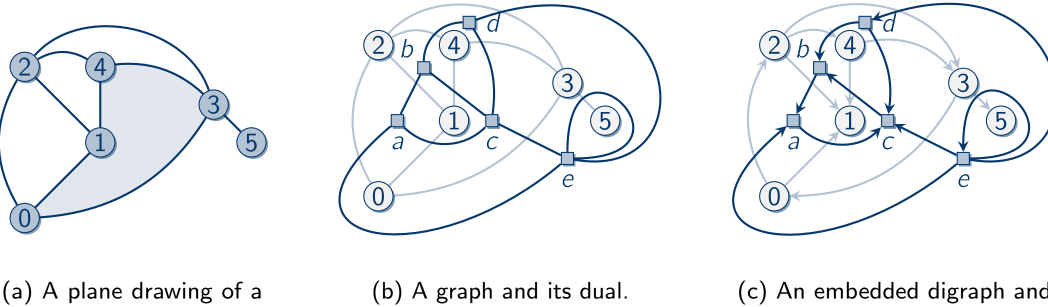
1.1.7 Planar Graphs

The following definitions apply to all cases of directed/undirected graphs/multigraphs similarly, where we apply the definitions to an undirected graph $G = (V, E)$.

Plane Drawings A *drawing* Γ *of a graph* G defines a representation of G in the Euclidean plane. Γ is a function which maps vertices $v \in V$ to points $\Gamma(v)$ in $\mathbb{R}^2$, the *vertex positions*, and each edge to an *edge curve*. An edge curve $\Gamma(e) : [0, 1] \to \mathbb{R}^2$ of an edge $e = \{u, v\}$ is a Jordan arc, i. e., a continuous, non-self-intersecting open curve, which connects the endpoints of the edge with $\Gamma(e)(0) = \Gamma(u)$ and $\Gamma(e)(1) = \Gamma(v)$. Note that this definition of edge curves also accounts for loops, where the Jordan arc is closed. Given a drawing, we identify a vertex (edge) with its position (curve) whenever this ambiguity leads to no confusion. The set $\Gamma(e)[(0, 1)]$ is the *inner part of the edge curve*.

A drawing is called *plane* if the vertex positions are distinct, no inner parts of two edge curves share a point, and no vertex lies on the inner part of an edge curve. More formally, we have:

$$\begin{aligned} \forall u, v \in V : u \neq v \Rightarrow \Gamma(v) \neq \Gamma(u)\,, \\ \Gamma[V] \cap \textstyle\bigcup_{e \in E} \Gamma(e)[(0, 1)] = \emptyset\,, \\ \forall e, e' \in E : e \neq e \Rightarrow \Gamma(e)[(0, 1)] \cap \Gamma(e)[(0, 1)] = \emptyset\,. \end{aligned} \tag{1.1}$$



(a) A plane drawing of a graph.

(b) A graph and its dual.

(c) An embedded digraph and its (directed) dual.

Figure 1.2: An embedded planar graph with its dual.

A graph admitting a plane drawing is called *planar*. A plane drawing of a graph is shown in Fig. 1.2(a). The vertices are drawn as shaded circles, labeled with the name of the vertex, and the edges are drawn as curved and straight lines.

Jordan's Curve Theorem An important theorem, not only in the context of plane drawings, is Jordan's curve theorem [Jor87, pp. 587–594] which states that every Jordan curve $\mathcal{C} = [0,1] \to \mathbb{R}^2$, i. e., a non-self-intersecting, continuous and closed ($\mathcal{C}(0) = \mathcal{C}(1)$) curve, divides $\mathbb{R}^2$ into two connected regions of which one, denoted by R, is bounded and the other, $\overline{R} = \mathbb{R}^2 \setminus R$, is unbounded. "Connected" in this context means that R ($\overline{R}$) cannot be represented as the union of two or more disjoint non-empty open subsets of $\mathbb{R}^2$. Any continuous curve in $\mathbb{R}^2$ that has points in R as well as $\overline{R}$ inevitably also contains at least one point on $\mathcal{C}$. We use the following special case of Jordan's curve theorem for plane drawings:

Proposition 1.1. *Let Γ be a plane drawing of a graph $G = (V, E)$ and let C be a simple circle in G. By following the edge curves on C, we obtain a Jordan curve $\mathcal{C}$. Denote by R the bounded region enclosed by $\mathcal{C}$ and by $\overline{R} = \mathbb{R}^2 \setminus R$ the unbounded region. Let e be an edge that is not in C. Then, every inner part of the edge curve of e is either completely within R or within $\overline{R}$:*

$$\forall e \in E : \Gamma(e)[(0,1)] \subsetneq R \vee \Gamma(e)[(0,1)] \subsetneq \overline{R} .$$

In particular, no inner part of an edge curve has points in R and $\overline{R}$ as this would lead to a crossing with $\mathcal{C}$ and a non-plane drawing.

For an example, consider the shaded region R in Fig. 1.2(a) which is enclosed by a simple circle of the graph. The inner part of the edge curve of $\{1,2\}$ lies completely outside of R.

Rotation Systems and Embeddings A *rotation system* $\mathcal{R}$ assigns to each vertex v a cyclic order of its incident edges $\mathrm{E}(v)$. A cyclic order on $\mathrm{E}(v)$ is a ternary relation $R \subseteq \mathrm{E}(V)^3$ that is cyclic, asymmetric, transitive, and total [Hun16]. An element $(e_1, e_2, e_3) \in R$ means that "on the way, starting at e_1 and going to e_3, one always passes e_2". Intuitively, a cyclic order arranges elements on a cycle, which is in contrast to a total order that arranges the elements on a dipath. In a rotation system of a vertex v, each edge e has an immediate successor $\mathrm{Succ}_v(e)$

and an immediate predecessor $\mathrm{Pred}_v(e)$. To conveniently define a rotation system of a vertex v, we define a total order $e_1, e_2, \dots, e_{\mathrm{d}(v)}$ on $\mathrm{E}(v)$ that is implicitly extended to a cyclic order by defining $\mathrm{Pred}_v(e_1) := e_{\mathrm{d}(v)}$ and $\mathrm{Succ}_v(e_{\mathrm{d}(v)}) = e_1$.

A plane drawing, as in Fig. 1.2(a), *implies* a rotation system. The rotation system at a vertex v is the counterclockwise order in which the edges $\mathrm{E}(v)$ enter v in the plane drawing. For instance, in Fig. 1.2(a), the rotation system at vertex 3 is $\{3,5\}, \{3,2\}, \{3,4\}, \{3,0\}$. Note that each plane drawing uniquely implies a rotation system while not for every rotation system there is a plane drawing that implies it. We call a rotation system $\mathcal{R}$ planar if there is a plane drawing that implies $\mathcal{R}$.

The following definitions apply to connected graphs. With some technical effort they can be extended to unconnected graphs as well. However, as most of the results presented in this thesis apply to connected graphs w. l. o. g., we assume that each graph in connected in the following.

A drawing of a graph subdivides the Euclidean plane into plane regions each of which is enclosed by a circle of the graph, e. g., in Fig. 1.2(a), the shaded region is bounded by the circle $0, \{0,1\}, 1, \{1,4\}, 4, \{4,3\}, 3, \{3,0\}, 0$ when traversing the region's boundary in clockwise direction. Such a circle defines a *face*, where the circle enclosing the unbounded region is called *outer face*. In Fig. 1.2(a), the outer face is enclosed by the circle $0, \{0,3\}, 3, \{3,5\}, 5, \{5,3\}, 3, \{3,2\}, 2, \{2,0\}, 0$. Note that moving vertex 5 into the shaded region would result in a different set of faces.

We call two plane drawings of a graph G *equivalent* if they define the same set of faces. Note that the outer face plays no special role in this definition. In particular, two plane drawings may be equivalent even if they have different outer faces. The so obtained equivalence classes on the set of plane drawings are called *embeddings* of G and a single embedding is denoted by $\mathcal{E}$. A planar rotation system defines an embedding and vice versa and, therefore, we identify them in the following. From the rotation system as implied by the drawing in Fig. 1.2(a), we obtain the shaded face by starting at vertex 0 with edge $\{0,1\}$ and then proceed to the other vertices by following the successor edges according to the rotation system. For instance, after following edge $\{0,1\}$ to vertex 1 we follow edge $\mathrm{Succ}_1(\{0,1\}) = \{1,4\}$ to get to vertex 4, and so forth, until we return to vertex 0. Analogously, a planar rotation system can be obtained from an embedding. By using the successor of an edge in the rotation system of each vertex, we traverse the faces clockwise. Note that by this definition the outer face in Fig. 1.2(a) is traversed clockwise, although it is traversed counterclockwise in the geometrical sense.

We say that two embeddings $\mathcal{E}$ and $\mathcal{E}^{\mathrm{R}}$ are equal *up to inversion* if the rotation system of vertex v according to $\mathcal{E}$, say $e_1, e_2, \dots, e_{d(v)}$, is the reversed version of the rotation system of v according to $\mathcal{E}^{\mathrm{R}}$, i. e., $e_{d(v)}, e_{d(v)-1}, \dots, e_1$. Intuitively, if Γ is a drawing which implies $\mathcal{E}$, then the drawing which is obtained by looking on Γ "from behind", e. g., by turning the page and holding it against the light, implies $\mathcal{E}^{\mathrm{R}}$. An important result due to Whitney [Whi33] is that the embedding of a triconnected and planar graph is unique up to inversion.

1.1.8 Dual Graphs

Let $\mathcal{E}$ be an embedding of a connected and planar graph $G = (V, E)$ and F be the set of faces. Remember, a face is defined by a circle of G. A vertex $v \in V$ is *incident* to a face $f \in F$ if f visits v, i. e., v is at a "corner" of f. Analogously, we say that an edge $e \in E$ is *incident* to a face $f \in F$, if f visits e, i. e., e lies at the boundary of f. Each edge is either incident to two different faces $f, g \in F$ or it is incident to a single face $f \in F$ if f visits e twice. The dual

graph $G^* = (F, E^*)$ of the embedded *primal graph* G is a multigraph where the faces are the vertices. For each *primal edge* $e \in E$ that is incident to faces $f, g \in F$, there is exactly one *dual edge* $e^* = \{f, g\} \in E^*$. Note that $f = g$ is possible. The primal and dual graph can always be simultaneously displayed in one drawing as shown in Fig. 1.2(b). The faces (rectangular shapes) are placed within the corresponding plane regions and each primal edge crosses its dual. Such a drawing of a dual graph implies a planar rotation system and an embedding of $\mathcal{E}^*$ of G^*, where the rotation system of a face is obtained by traversing the face in counterclockwise order.

Assuming that G is connected, the faces of G^* correspond to the vertices of G. In fact, the dual of G^* with embedding $\mathcal{E}^*$ is isomorphic to G with embedding $\mathcal{E}$. A terminological problem arises when a phrase like "face of G^*" is used as it may refer to a face $f \in F$ or to a face in the embedding of G^*, which is in turn a vertex $v \in V$. To avoid this ambiguity, we consistently refer to elements in F as "faces" and to elements in V as "vertices".

Doubly-Connected Edge Lists For an efficient implementation, an embedding or a rotation system is usually modelled by a doubly-connected edge list (DCEL) [dBCvKO08]. In this data structure, each edge at a vertex has a pointer to its predecessor and its successor in the rotation system. Moreover, in case of planar rotation systems, DCELs can be augmented by further information to store faces and dual edges and, thereby, the dual graph, or the information to which faces a vertex or an edge is incident. In this thesis, whenever we algorithmically deal with rotation systems and embeddings, we assume a DCEL implementation and refer the reader to, e. g., [dBCvKO08] for more information.

Cut and Cutset We use the following definitions and results at several occasions in the thesis. Let $G = (V, E)$ be an embedded graph and $G^* = (F, E^*)$ be its dual. A *cut* is a partition of V into proper subsets $V_C \subsetneq V$ and $\overline{V}_C := V \setminus V_C$. For instance, $V_C = \{0, 1, 2\}$ and $\overline{V}_C = \{3, 4, 5\}$ is a cut of the graph in Fig. 1.2(b). The set of all edges $E_C := \{\{u, v\} \in E \mid u \in V_C \wedge v \in \overline{V}_C\}$ that connect vertices in V_C with vertices in $\overline{V}_C$ is called *cut-set*, e. g., $E_C = \{\{0, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}\}$ in Fig. 1.2(b). E_C^* denotes the duals of E_C, e. g., $E_C^* = \{\{c, e\}, \{b, c\}, \{d, e\}, \{b, d\}\}$. Then, the edges E_C^* always form a simple circle in G^* [Eve12, pp. 149]. Conversely, the primal edges of a cut-set in G^* form a simple circle in G.

Directed Duals Dual graphs are also defined for planar digraphs. Let $G = (V, E)$ be a planar digraph with embedding $\mathcal{E}$ and faces F. The faces, which are circles, are defined on the underlying undirected graph of G. Consider a face $f \in F$ and an edge $e = (u, v)$ incident to f. When traversing f clockwise, edge e is either traversed in its direction from u to v, against its direction from v to u, or in both directions if f visits e twice. We say that f *is right of* e if e is traversed in its direction and f *is left of* e if e is traversed against its direction. The directed dual $G^* = (F, E^*)$ of an embedded primal digraph G consists of the faces F and of directed dual edges E^* where the dual e^* of a primal edge e is directed from the face left of e to the face right of e. In case a face is both to the left and to the right of e, e^* is a directed loop.

Fig. 1.2(c) shows the directed version of the graph from Fig. 1.2(b) along with its directed dual. In a clockwise traversal of face c, primal edge $(4, 1)$ is traversed against its direction and, hence, face c is left of $(4, 1)$. Analogously, face b is right of $(4, 1)$. Therefore, the dual of edge $(4, 1)$ points from c to b. Intuitively, when "looking" into the direction of the primal edge, its dual crosses the primal edge from the left to the right side. Note that for the loop from

the outer face e onto itself in the rotation system of face e, the portion of the loop that is outgoing is the predecessor of the loop's incoming portion.

1.1.9 Cylinder Surfaces

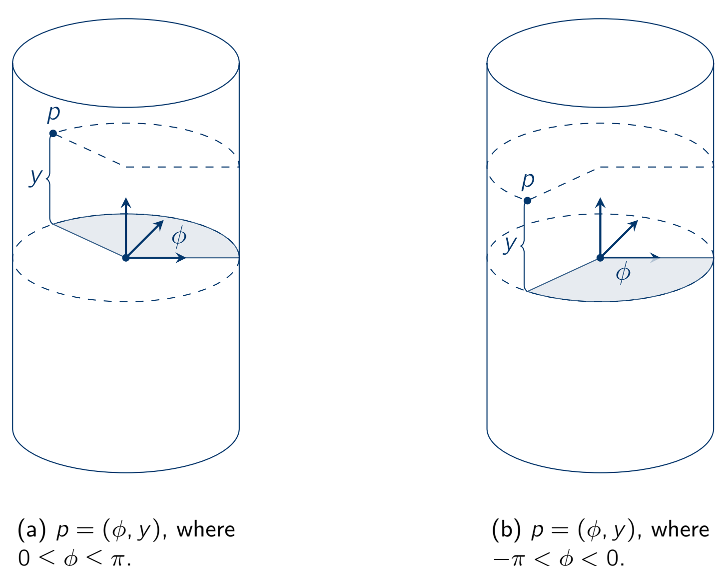
In this thesis, we consider drawings of graphs on cylinder surfaces. We distinguish between two cases: the *standing* and the *rolling cylinder*. In the three-dimensional space $\mathbb{R}^3$, the surface of the standing cylinder is defined by the set of points:

$$\mathbf{C}_s^3 = \{(x, y, z) \in \mathbb{R}^3 \mid x^2 + z^2 = 1 \wedge -1 < y < 1\},$$

where the coordinate system is defined according to the right-hand rule with the index finger pointing upwards (see Fig. 1.3(a)). A point $p = (x, y, z) \in \mathbf{C}_s^3$ is uniquely be defined by its cylindrical coordinates (ϕ, y) [Haz87], where $-\pi < \phi < \pi$ is the azimuth, i. e., the acute angle between the x-axis and the line from the origin to $(x, 0, z)$ on the xz-plane ($y = 0$). See Fig. 1.3(a) for an illustration. The azimuth is calculated by:

$$\phi = \begin{cases} \frac{\pi}{2} - \arcsin x\,, & \text{if } z \geq 0\,, \\ -\frac{\pi}{2} + \arcsin x\,, & \text{if } z < 0\,. \end{cases}$$

Note that the azimuth "wraps" to $-\pi$ when ϕ approaches π from below. In general, the azimuth is in $[0, \pi]$ if $z \geq 0$ (Fig. 1.3(a)) and it is in $(-\pi, 0)$ if $z < 0$ (Fig. 1.3(b)). Altogether, the azimuth ϕ is in the half-open interval $(-\pi, \pi]$. When ϕ approaches 0, whether from above or below, its limiting value is always 0, i. e., no "wrapping" occurs.



(a) $p = (\phi, y)$, where $0 \leq \phi \leq \pi$.

(b) $p = (\phi, y)$, where $-\pi < \phi < 0$.

Figure 1.3: Cylindrical coordinates.

The rolling cylinder is defined by:

$$\mathbf{C}_r^3 = \{(x, y, z) \in \mathbb{R}^3 \mid y^2 + z^2 = 1 \wedge -1 < x < 1\}.$$

It is obtained from the standing cylinder by swapping the x- with the y-coordinate which effectively rotates the cylinder 90° around the z-axis. Similar to the standing cylinder, a point in $\mathbf{C}_r^3$ is uniquely determined by its x-coordinate and by its azimuth which is the acute angle ϕ between the y-axis and the line from the origin to $(0, y, z)$ on the yz-plane $(x = 0)$.

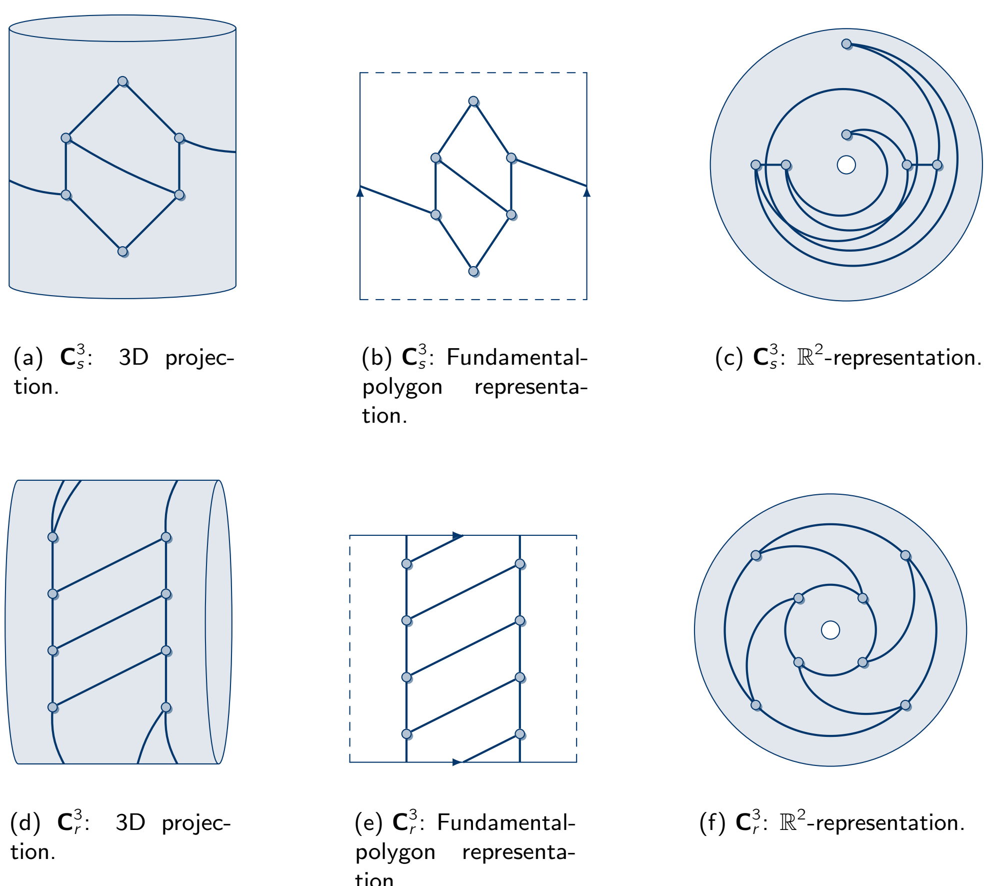
Let $\mathbf{C}^3$ be the point set of either the standing or rolling cylinder. Similar to the plane, a drawing Γ of a graph in $\mathbf{C}^3$ maps each vertex $v \in V$ to its position $\Gamma(v) \in \mathbf{C}^3$ and each edge $e = \{u, v\} \in E$ to a continuous and injective, i. e., non-self-intersecting, function $\Gamma(e) : [0, 1] \to \mathbf{C}^3$ with $\Gamma(e)(0) = \Gamma(u)$ and $\Gamma(e)(1) = \Gamma(v)$. A drawing Γ of a graph on a cylinder surface is called plane if Eq. (1.1) holds. As defined in Sect. 1.1.7, a plane drawing of a graph on a cylinder surface defines a planar rotation system and, thus, an embedding, and a dual graph. Observe, in this case, we have no outer face as there is no unbounded region. Plane drawings on $\mathbf{C}_s^3$ and $\mathbf{C}_r^3$ are shown in Figs. 1.4(a) and 1.4(d), respectively. Both figures show 3D projections of the respective cylinder surfaces and the drawings.

A disadvantage of this representation is that certain parts of the drawings — the ones on the backside of the cylinders — are hidden. To circumvent this drawback, we use the *fundamental polygon* which is a general principle to represent 2D surfaces of higher genus in the plane [Mas67]. For cylinder surfaces, we use the most basic case. Let $I = (-1, 1) \subseteq \mathbb{R}^2$ be the open unit interval and derive $I_\circ$ from I by "identifying" the boundaries -1 and 1 of I. In order to rigorously define what "identifying" means, some mathematical and technical hurdles have to be overcome. We refrain from these rigorous treatment the sake of conciseness and refer the interested reader to [Mas67]. The fundamental polygon of the standing cylinder is defined by $\mathbf{C}_s = I_\circ \times I$. $\mathbf{C}_s$ can be drawn as a square of side length 2 (see Fig. 1.4(b)) in which the arrows on the left and right sides indicate their identification. The top and bottom sides, which do not belong to $\mathbf{C}_s$, are drawn as dashed lines. Intuitively, when reaching either the left or right side of $\mathbf{C}_s$, a "wrapping" to the other side occurs. The x-coordinate of a point $(x, y) \in \mathbf{C}_s$ corresponds to the azimuth on the standing cylinder. Any drawing in $\mathbf{C}_s^3$ can be mapped to a drawing in $\mathbf{C}_s$ by the transformation $(\phi, y) \mapsto (\phi/\pi, y)$, where (ϕ, y) are cylindrical coordinates in $\mathbf{C}_s^3$. By the inverse transformation, we obtain a drawing in $\mathbf{C}_s^3$ from a drawing in $\mathbf{C}_s$. Both transformations preserve planarity. From Fig. 1.4(a), we obtain the fundamental polygon representation in Fig. 1.4(b).

The fundamental polygon of the rolling cylinder is defined by $\mathbf{C}_r = I \times I_\circ$. In this case, a point in $\mathbf{C}_r^3$ with cylindrical coordinates (x, ϕ) is mapped to $(x, \phi/\pi) \in \mathbf{C}_r$. For a point $(x, y) \in \mathbf{C}_r$ the y-coordinate corresponds to the azimuth and, therefore, the bottom and top sides of square are identified (see Fig. 1.4(e)). For the drawing in Fig. 1.4(d), we obtain the fundamental polygon representation in Fig. 1.4(e).

The third type of representation we use in this thesis is the $\mathbb{R}^2$, which is a reinterpretation of the cylindrical coordinates as polar coordinates in the plane. Let (ϕ, y) be the cylindrical coordinates of a point in $\mathbf{C}_s^3$. As polar coordinates, we use ϕ as the azimuth and $\varepsilon + y + 1$ as radius for some $\varepsilon > 0$. Thereby, any drawing on $\mathbf{C}_s^3$ is mapped to a drawing on a ring in the plane with inner radius ε and outer radius $\varepsilon + 2$. For Fig. 1.4(a), we obtain the drawing in Fig. 1.4(c). Again, this mapping and its inverse preserve planarity. For the rolling cylinder, we use a similar transformation, where the x-coordinate is mapped to $\varepsilon + x + 1$ for some $\varepsilon > 0$. Fig. 1.4(f) shows the result for the drawing in Fig. 1.4(d).

Throughout this thesis, we will use all three types of representation, though, we will scale their width and height to obtain comprehensible drawings. These transformations do not affect the planarity of the drawings.



(a) $\mathbf{C}_s^3$: 3D projection.

(b) $\mathbf{C}_s^3$: Fundamental-polygon representation.

(c) $\mathbf{C}_s^3$: $\mathbb{R}^2$-representation.

(d) $\mathbf{C}_r^3$: 3D projection.

(e) $\mathbf{C}_r^3$: Fundamental-polygon representation.

(f) $\mathbf{C}_r^3$: $\mathbb{R}^2$-representation.

Figure 1.4: The three types of representations of drawings on the standing and rolling cylinder.

21 80 0720 175-7
RIV D-DB Tds

Chapter 2

Deque Layouts and Linear Cylindric Drawings

2.1 Introduction

The stack, the queue, and the deque (double-ended queue) are the most fundamental data structures in computer science [Knu97, pp. 238–242]. They appear in almost any algorithm, sometimes directly and sometimes in a disguised form. The stack and its last in, first out (LIFO) principle (left side of Fig. 2.1) appear in situations involving recursion or nesting structures, like arithmetic expressions. The dual of the stack is the queue (middle of Fig. 2.1) and its working principle is first in, first out (FIFO). It finds its applications, for instance, when a buffer is needed, such as when tasks arrive that have to be subsequently processed, or in data streams. The deque generalizes and, at the same time, combines the stack and the queue: It has two sides, head and tail, to insert and remove items (right side of Fig. 2.1). As such, it can emulate two stacks (in a single data structure) and additionally allows "queue items" inserted and removed at different sides. A real-world example which can be modelled by a deque is a railway that passes a train station where trains can arrive from and depart to either side [CDS07].

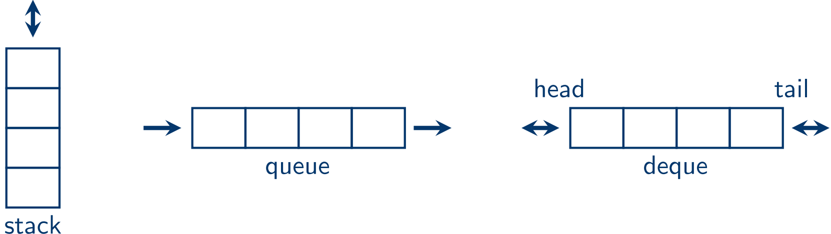


Figure 2.1: The stack, the queue, and the deque.

Although, these data structures are rather simple to use and implement — after all, a typical assignment for undergraduate students in a programing course is to implement them — their behavior and capabilities in various theoretical scenarios are far from trivial and the topic of ongoing research. This research not only sheds new light on the data structures themselves but also yields interesting insights to other mathematical objects, resulting in a two-way gain in

knowledge. Here, we study graph layouts in the stack, the queue, and, especially, the deque that characterize important classes of planar graphs. Even more, it turns out that planarity is the key to understand these data structures and, vice versa, understanding these data structure yields new insights to planarity.

2.1.1 Graph Layouts

In a graph layout, the vertices are processed according to a total order, which is called *linear layout*. The edges correspond to data items that are inserted to and removed from a data structure. Each edge is inserted at the end vertex that occurs first according to the linear layout and is removed at its other end vertex. Alg. 2.1 shows the general procedure of a graph layout: Starting with an empty data structure $\mathcal{D}$, e. g., a stack, a queue, or a deque, the vertices are processed in order of the linear layout. At each vertex v, each incident edge is removed from $\mathcal{D}$ if it points to a predecessor of v in the linear layout and it is inserted if it points to a successor. Whereas inserting an edge to $\mathcal{D}$ is always possible, removing it might not be possible and in this case `false` in returned in Alg. 2.1. For instance, if $\mathcal{D}$ is a stack, the edge may not be on top or, in case of a queue, an edge cannot be removed as another edge must be removed before. A graph is a $\mathcal{D}$-graph if it has a $\mathcal{D}$-layout which is a linear layout such all edges can be *processed* in $\mathcal{D}$, i. e., Alg. 2.1 returns `true`.

Algorithm 2.1. A graph layout in a data structure $\mathcal{D}$.

Input: a graph $G = (V, E)$, a linear layout, and a data structure $\mathcal{D}$
Output: `true` if all edges can be processed in $\mathcal{D}$; `false` otherwise

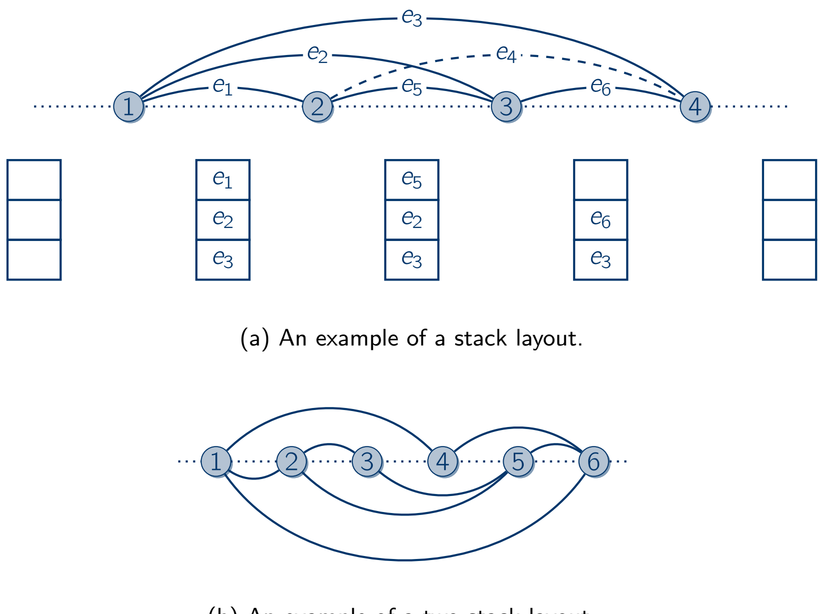
1 $\mathcal{D} \leftarrow$ empty data structure
2 **foreach** vertex v in order of the linear layout **do**
3   **foreach** edge $e = \{u, v\}$ incident to v **do**
4   **if** v comes before u in linear layout **then** insert e to $\mathcal{D}$
5   **if** u comes before v in linear layout **then** remove e from $\mathcal{D}$ or return `false` if not possible
6 **return** `true`

2.1.1.1 Stack Graphs

Fig. 2.2(a) illustrates a (one-)stack layout. The vertices are placed on a horizontal line (dotted) in order of the linear layout and the edges are drawn as arches above the horizontal line. We ignore edge e_4 (dashed) for the moment. Below, the content of the stack before and after each vertex is shown. We start with an empty stack at vertex 1 and all edges incident to vertex 1 are pushed onto the stack. The order of insertions is determined by the reversed order of vertex 1's neighbors according to the linear layout. At vertex 2, edge e_1 is removed from the top of the stack and e_5 is inserted. By proceeding in this way, all edges can be processed in the stack and, hence, the depicted graph is a stack graph. An important observation is that all edges that were processed in the stack do not cross in the drawing. Now, suppose that we introduce edge e_4 that is inserted at vertex 2. After vertex 2, the stack's content is (e_5, e_4, e_2, e_3) (from top to bottom). Although, e_5 can be removed at vertex 3, edge e_2 cannot be removed because e_4, removed at 4, is on top of the stack. At the same time, e_2 and e_4 are

crossing. This observation holds true in general: A graph admits a plane drawing of the type shown in Fig. 2.2(a) if and only if it is a stack graph.

This observation already indicates that there is a relationship between graph layouts and planarity. Bernhart and Kainen have shown that a graph is a stack graph if and only if it is outerplanar [BK79]. A graph is called outerplanar if it has a plane drawing such that all vertices are incident to the outer face. Indeed, all vertices in Fig. 2.2(a) are incident to the outer face and the drawing is plane without e_4. In fact, introducing edge e_4 results in the complete graph with four vertices K_4 which is known to be not outerplanar. Bernhard and Kainen have also characterized the two-stack graphs as the spanning subgraphs of planar graphs with a Hamiltonian circle. In a two-stack layout, the set of edges is partitioned into two subsets such that both induced subgraphs have a stack layout with the same linear layout. In our example, edge e_4 can be processed in a second stack without interfering with the other edges and, indeed, the K_4 is planar and contains a Hamiltonian circle. A more complex example of a two-stack layout is shown in Fig. 2.2(b), where the edges above the horizontal line are processed in the first stack and the edges below in the second stack. An interesting feature of Figs. 2.2(a) and 2.2(b) is how they reflect the working principle of the stack: In the example in Fig. 2.2(a), edge e_3 is always at the bottom of the stack and, at the same time, all other edges are nested within the e_3. These nesting edge curves reflect the LIFO principle of the stack.



(a) An example of a stack layout.

(b) An example of a two-stack layout.

Figure 2.2: A one- and a two-stack layout.

Stack layouts are also known as book embeddings which have been studied first by Atneosen [Atn68], Ollman [Oll73] and Kainen [Kai74]. The idea of a book embedding is to place the vertices on the spine of a book, i. e., the dotted line in Fig. 2.2(a), and to draw the edges plane on k pages for some $k \geq 1$. This is equivalent to a k-stack layout, where the graph is partitioned into k spanning subgraphs such that each subgraph has a stack layout and all layouts use the same linear layout. Thus, a page in a k-page book embedding

corresponds to a stack in a k-stack layout. The page number, also called book thickness, of a graph G is the minimum k such that G has a k-page book embedding and it is equal to the stack number, i. e., the minimum k such that G has a k-stack layout. For instance, the page number of the graph in Fig. 2.2(a) without e_4 is 1, whereas with e_4 it is 2. The study of book embeddings and stack layouts has produced a vast amount of publications, e. g., [HLR92, BK79, DW05, HPT99a, HPT99b, BS84, Yan89, ENO97, Hea84, HI87, Mal94b, Mal94a] among many others, and more recent publications show that book embeddings are still a topic of active research [FFRV11, Miy06, BES13]. Practical applications of stack layouts include VLSI design [CLR87], folding of RNA [GS99], and fault-tolerant processing [Ros83]. A slightly dated and yet useful survey about book embeddings is [Bil92].

The relationship between stack layouts and planarity has been the topic of many research papers. As mentioned before, the one-stack graphs are exactly the outerplanar graphs and the two-stack graphs are the spanning subgraphs of planar graphs with a Hamiltonian circle [BK79]. Series-parallel graphs, which are also planar, have a layout in two stacks [CLR87]. As not every maximal planar graph contains a Hamiltonian circle, there are planar graphs that need at least three stacks. Bernhard and Kainen conjectured that the stack number of planar graphs is unbounded [BK79]. This conjecture was refuted by Buss and Shor [BS84] by giving an algorithm to compute a nine-stack layout for any planar graph. Heath improved this bound to seven [Hea84] and, eventually, Yannakakis settled the case by showing that four pages are sufficient and necessary [Yan86, Yan89]. However, as of today and to the best of our knowledge, no planar graph that actually needs four stacks has been constructed explicitly and the construction of such a graph was postponed by Yannakakis to a long version of [Yan86] which has yet to appear. Genus g graphs have also been studied, where a graph is of genus g if it can be embedded on a genus g surface or, equivalently, the surface of a sphere with g handles [GT01b]. Heath and Istrail showed that the stack number of a genus g graph is in $\mathcal{O}(g)$ [HI87], and conjectured that the stack number is in $\mathcal{O}(\sqrt{g})$ which was proved by Malitz in [Mal94a].

Deciding whether a graph is a one-stack graph can be done in linear time as it is equivalent to testing outerplanarity [BK79, Wie87]. In contrast, the decision problem for two stacks is $\mathcal{NP}$-complete as deciding whether a maximal planar graph contains a Hamiltonian circle is $\mathcal{NP}$-complete [Chv85, Wig82]. Therefore, testing for a k-stack layout is also $\mathcal{NP}$-complete. For planar graphs, the proof in [Yan89] yields a linear-time algorithm by which a four-stack layout can be obtained.

Whereas generally testing for a k-stack layout is $\mathcal{NP}$-complete, special cases may be tractable. Unfortunately, even finding the minimum number of stacks for a given fixed linear layout is also $\mathcal{NP}$-complete [DW04a]. For the cases of two or three stacks, efficient algorithms exist by reducing the problem to the coloring problem of circle graphs [DW04a, Ung92]. The complementary problem is to ask for a suitable linear layout for a given assignment of the edges to k stacks, which can be solved in linear time [HN09].

Generalizations of book embeddings include k-stack subdivisions which are also called topological book embeddings: A graph has a k-stack subdivision if its edges can be replaced by paths such that the obtained graph has a k-stack layout. Atneosen has shown that every graph has a three-stack subdivision [Atn68] and Dujmovic and Wood have shown that a graph is planar if and only if has a two-stack subdivision [DW05].

Stack layouts have also been studied for acyclic digraphs by Heath et al. in [HPT99a, HPT99b] where the linear layout must be a topological ordering of the digraph. The combination of directed stack layouts and upward planarity has recently been studied by Frati et al. [FFRV11].

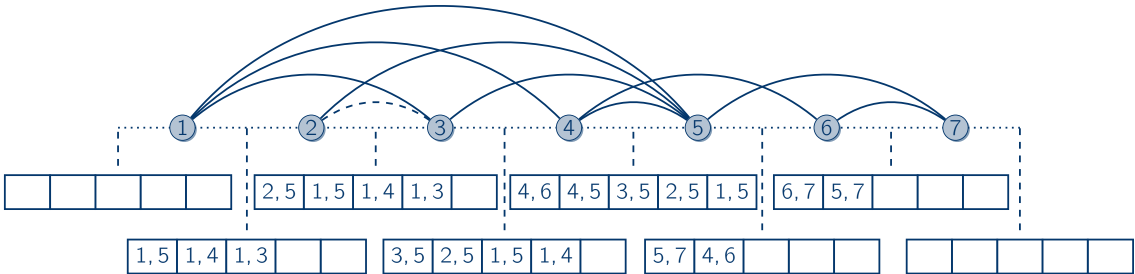
2.1.1.2 Queue Graphs

Queue layouts have also received their fair share of attention [DBFP10, DW05, HLR92, HPT99a, HPT99b, HR92, Pem92, Woo02], and they find their application in, for instance, 3D drawings of graphs [Woo02, DW04b] and VLSI design [HR92]. Fig. 2.3(a) (without the dashed edge) depicts a (one-)queue layout. Again, the vertices are placed on a horizontal line and the edges are drawn as arches above. The content of the queue between the vertices is shown below the horizontal line. For instance, at vertex 1 the edges $\{1,3\}$, $\{1,4\}$, and $\{1,5\}$ are inserted at the head, i. e., the left side, of the queue in order of their endpoints such that they can be removed at the tail of the queue at vertices 3, 4, and 5, respectively. An edge e *properly nests* another edge e' if e and e' have no common endpoints and e must be inserted before e', and e' must be removed before e. Such configurations are also called *rainbows* [HR92] and they are not allowed in queue layouts. For instance, edge $\{1,5\}$ properly nests edge $\{2,3\}$ (dashed) in Fig. 2.3(a). Note that $\{2,3\}$ can be inserted at vertex 2 but not removed at vertex 3 because $\{1,5\}$, which is removed at vertex 5, has to be removed from the queue before $\{2,3\}$ can be removed. Representing a queue layout as in Fig. 2.3(a), naturally leads to many crossings, also called twists [HR92]. In fact, a twist, as with edges $\{1,4\}$ and $\{2,5\}$, reflects the FIFO principle of the queue.

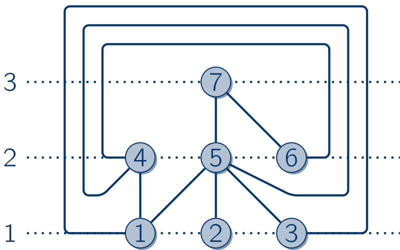
Still, queue graphs are planar which is one of the many fundamental results on queue layouts proved by Heath and Rosenberg in [HR92] which, together with [HLR92], started the study of queue layouts. In [HR92], Heath and Rosenberg have characterized the queue graphs as the arched leveled-planar graphs that have a plane drawing in which the vertices are placed on horizontal levels. Inter-level edges connect vertices between two adjacent levels and intra-level edges, called arches, connect the leftmost vertex on a level to accessible vertices on the right side of the same level. The arched leveled-plane drawing of the graph from Fig. 2.3(a) (without the dashed edge) is shown in Fig. 2.3(b). Note that the linear layout in Fig. 2.3(a) is the order of the vertices within the levels from left to right and, between the levels, from bottom to top in Fig. 2.3(b).

Heath and Rosenberg have also shown in [HR92] that deciding whether a graph is a queue graph is $\mathcal{NP}$-complete. For a fixed linear layout, they devised an algorithm that computes the minimum number of required queues in time $\mathcal{O}(|E| \log |V|)$ for a graph $G = (V, E)$. In addition, they found out that there are stack graphs that are no queue graphs and queue graphs that are no stack graphs, thereby, the stack and the queue are not comparable in the context of graph layouts. In addition, Heath and Rosenberg have shown that every queue graph is a two-stack graph and every stack graph is a two-queue graph. They also investigated the queue number, which is defined analogously to the stack number, for special graph classes, e. g., for the complete graph K_n with n vertices it is $\lceil \frac{n}{2} \rceil$.

For planar graphs, Heath and Rosenberg conjecture in [HR92] that the queue number is bounded by a constant. Pemmaraju, in contrast, conjectures in [Pem92] that certain planar graphs, planar 3-trees, with n vertices have an unbounded queue number in $\mathcal{O}(\log n)$. Dujmović et al. disproved this by showing that graph classes with a bounded treewidth [RS84], e. g., planar 3-trees, have bounded queue number [DMW05]. More recently, Di Battista et al. have shown that every planar graph with n vertices has a queue layout in $\mathcal{O}(\log^4 n)$ many queues [DBFP10]. The original conjecture of a bounded queue number for planar graphs in [HR92] is still open. Queue subdivisions have also been studied, where every graph has a two-queue subdivision and every planar graph has a one-queue subdivision [DW05].



(a) A linear representation of a queue layout (without the dashed edge). The content of the queue is shown below. The dashed edge is nested and destroys the queue layout.



(b) An arched leveled-plane drawing of the queue graph in Fig. 2.3(a) (without edge $\{2, 3\}$).

Figure 2.3: Examples of queue layouts.

2.1.1.3 Contributions of this Thesis

Tab. 2.1 shows an overview of the characterizations and the decision problems for stack, two-stack, and queue layouts. In this thesis, we extend the study of graph layouts to the deque. The deque generalizes the stack and the queue and, at the same time, combines them. As suggested by the study of stack and queue layouts, there is a profound relationship between planarity and graph layouts. For deque layouts, we introduce a novel type of drawing on the surface of a rolling cylinder named *linear cylindric drawing* (Sect. 2.2.1). In these drawings, the vertices are placed on a straight line that is parallel to the cylinder's axis, the *front line*, and the edge curves share no point with the front line except for their endpoints. An example of such a drawing is shown in Fig. 2.5(b) on page 28. We call graphs permitting a plane linear cylindric drawing *linear cylindric planar*. It turns out that the linear cylindric planar graphs characterize the deque graphs (Sect. 2.2.3). In comparison to the previous study of stack and queue layouts [BK79, HR92], we use a low-level approach based on the rotation system: The rotation system obtained from a linear cylindric plane drawing defines the order in which the edges are processed in a deque and vice versa.

With the help of linear cylindric drawings, we obtain another characterization of deque graphs in Sect. 2.2.4. A graph is a deque graph if and only if it is the spanning subgraph of a planar graph with a Hamiltonian path. This characterization is of special interest when

Data Structure	Characterization	Decision Problem
stack	outerplanar [BK79]	linear time [BK79, Wie87]
queue	arched leveled-planar [HR92]	$\mathcal{NP}$-complete [HR92]
two stacks	subgraph of planar graph with Hamiltonian circle [BK79]	$\mathcal{NP}$-complete [Chv85, Wig82]

Table 2.1: Overview: stack, two-stack, and queue graphs.

compared to two-stack graphs. By [BK79], we know that a graph is a two-stack graph if and only if it is the spanning subgraph of a planar graph with a Hamiltonian circle. Also, a deque can emulate two stacks (in a single data structure) and additionally allows for queue edges. Hence, the ability to process queue edges in a deque in comparison to two stacks exactly corresponds to the difference between Hamiltonian paths and circles in planar graphs. This observation also leads to intriguing insights into the duals of planar graphs with a Hamiltonian path (Sect. 2.2.4.2). In particular, we see how the dual reflects which edges use the "queue mode" and sheds new light on the working principle of the deque as well as the queue. We also consider the complexity of deciding whether a graph is a deque graph and give the (negative) answer that it is $\mathcal{NP}$-complete (Sect. 2.2.5) by showing that finding a Hamiltonian path in a maximal planar graph is $\mathcal{NP}$-complete. In Sect. 2.3, we see how linear cylindric drawings help to concisely visualize layouts in *deque-reducible* data structures, i. e., data structures that permit only a subset of the deque operations, and mixed layouts, e. g., stack-queue layouts.

Linear cylindric drawings are not only a neat tool to study deque layouts but also yield a new characterization of queue graphs, which are the topic of Sect. 2.4. It turns out that the FIFO principle of the queue is concisely displayed by linear cylindric drawings that can also be used to decide whether a linear layout allows for a queue layout by simply checking for crossings (Sect. 2.4.1). In addition, we will see in Sect. 2.4.3 that the duals of queue graphs exhibit a very linear structure just as linear cylindric planar graphs themselves. This observation leads to a novel characterization of queue graphs by means of their duals. In a nutshell, the duals of (slightly modified) queue graphs are also queue graphs.

By Yannakakis [Yan86, Yan89], we know that every planar graph has a layout in four stacks. In consequence, every planar graph has a layout in two deques; not a very satisfactory insight as the queue mode of the deque is not exploited in such a layout. Further, there are non-planar graphs that have a layout in three or four stacks, e. g., the complete graph with five vertices, and, thus, four-stack layouts do not characterize planarity. However, neither do deque layouts as there are maximal planar graphs that lack a Hamiltonian path [Hel07]. What is the additional operation a deque must support in order to characterize planarity? In Sect. 2.5, we find the answer to this question: The deque must support a split operation that divides it into smaller deques. The modified deque is called *splittable deque* and we show that a graph is a splittable deque graph if and only if it is planar.

For the proof, we devise an algorithm that uses the splittable deque to test whether a rotation system is planar. The algorithm is a means to an end for the characterization of planarity by the splittable deque, and there are other algorithms especially designed for solving the same problem [DM03]. Nevertheless, the algorithm has the benefit that it operates on an elementary data structure, i. e., a deque, which is simple in comparison to other ones used for

general planarity testing [SH99]. At a first glance, it may seem that testing the planarity of a rotation system is significantly simpler than general planarity testing. However, our algorithm has to tackle issues that are ignored in the general case, namely, crossings between edges incident to the same vertex.

In Sect. 2.6, we summarize this chapter, give further remarks, and discuss future work.

2.1.2 Other Approaches to Study Fundamental Data Structures

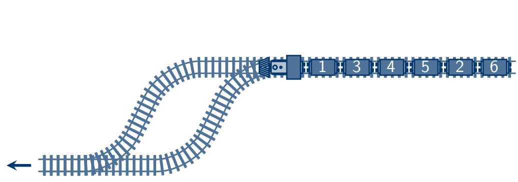
Before we start with linear cylindric drawings and deque layouts, we roam among two other mathematical disciplines that shed new light on the power of fundamental data structures and their differences. We start with permutation networks which are related to graph layouts.

2.1.2.1 Permutation Networks

In an exercise in [Knu97], Knuth asks the reader the following question: assume we are given a switchyard network with an entry and an exit, and a train consisting of cars numbered from 1 to n in a certain order. Is it possible to shunt the cars within the switchyard such that the train exits with the cars in ascending order? Two instances of the switchyard problem are shown in Figs. 2.4(a) and 2.4(b). The switchyard problem is formalized as follows: a permutation network is an acyclic digraph $G = (V, E)$ with exactly one source s, the entry of the switchyard, and exactly one sink t, the exit of the switchyard, and all other vertices $V \setminus \{s, t\}$ are data structures, e. g., stacks, queues, or deques. The permutation network belonging to Fig. 2.4(a) is displayed in Fig. 2.4(c) and it consists of two parallel queues, and for Fig. 2.4(b), the permutation network in Fig. 2.4(d) consists of two parallel stacks. Let π be a permutation of the numbers $1, \dots, n$, which is the order in which the numbers arrive at s. Is it possible to process the numbers in the data structures, following the edges in their direction, such that the numbers arrive in order $1, \dots, n$ at t? For Figs. 2.4(c) and 2.4(d), the answer is yes.

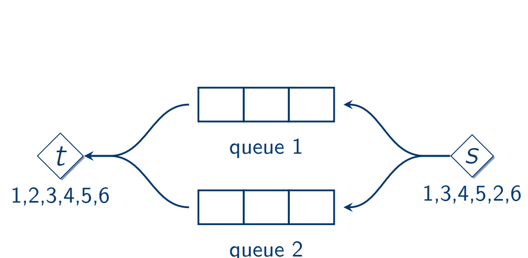
One of the reasons why we discuss permutation networks is their relationship with graph layouts: Assume we are given a permutation network of k parallel stacks, queues, or deques, e. g., two parallel queues or two parallel stacks as in Figs. 2.4(c) and 2.4(d), respectively. For a given permutation π of the numbers from 1 to n, let $G_\pi = (V_\pi, E_\pi)$ be a graph with vertices I_i and O_i for all $1 \leq i \leq n$, where I_i and O_i are connected by an edge. π is sortable in k parallel stacks, queues, or deques if and only if G_π has a k-stack, queue, or deque layout, respectively, where for the linear layout $\prec$, $I_i \prec O_i$ for all $1 \leq i \leq n$, $I_i \prec I_j \Leftrightarrow i <_\pi j$, and $O_i \prec O_j \Leftrightarrow i < j$ for all $1 \leq i, j \leq n$. The relation $<_\pi$ is the order defined by π. By this transformation, each number i in π becomes an edge which is inserted at vertex I_i and removed at vertex O_i. The restriction on the order of the vertices I_i guarantees that the input permutation is π and the restriction on the order of the vertices O_i guarantees that the edges are sorted in increasing order. For Figs. 2.4(c) and 2.4(d), the corresponding two-queue and two-stack layouts are shown in Figs. 2.4(e) and 2.4(f), respectively. Hence, the permutations are sortable in the corresponding networks. By this transformation, graph layouts can be used to concisely display how a permutation network processes the numbers. The transformation can also be adapted for general permutation networks, where for each transfer of a number i from one data structure to another we get a subdivision of the edge representing i.

Permutation networks haven been studied extensively in the past, e. g., [Knu97, Tar72b, Pra73, Eve71, Bó02] among many others. The typical questions that arise with permutation networks are: What permutations can be sorted in the network? Are there typical patterns

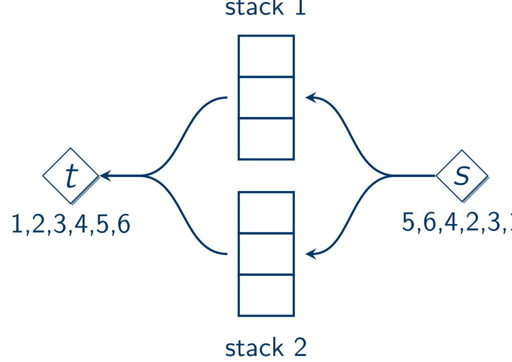


(a) A switchyard with two parallel “queues”.

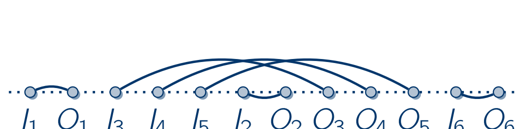
(b) A switchyard with two parallel “stacks”.



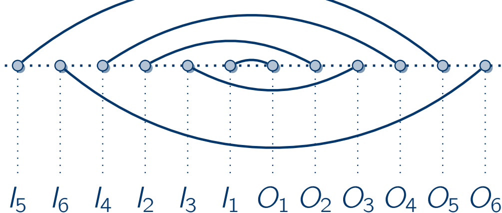
(c) The permutation network corresponding to Fig. 2.4(a).



(d) The permutation network corresponding to Fig. 2.4(b).



(e) The graph layout of the solution to Fig. 2.4(c).



(f) The graph layout of the solution to Fig. 2.4(d).

Figure 2.4: The switchyard problem.

in permutations that cannot be sorted, i. e., so called forbidden patterns, and, if so, are there finitely many of them? What is the number of permutations that can be sorted depending on the length of the input permutation? How hard is it to decide whether a given permutation can be sorted in the network? Answers to these questions shed new light on the used data structures and their differences; and indirectly on graph layouts.

In [Knu97], Knuth has studied the network consisting of only one stack. There, a permutation can be sorted if and only if it does not contain the pattern $2, 3, 1$, i. e., there are no three numbers $i < j < k$ with $j <_\pi k <_\pi i$, where $<_\pi$ is the order according to the input permutation π. The proof also yields an efficient algorithm to decide whether a permutation is stack-sortable. Further, Knuth showed that the number of sortable permutations of length n is [Knu97]:

$$C_n = \frac{1}{n+1}\binom{2n}{n}, \tag{2.1}$$

which are the Catalan numbers. Interestingly, the number of permutations sortable in two parallel queues is also given by the Catalan numbers [Pra73]. Hence, a single stack and two parallel queues are equally powerful with respect to the number of sortable permutations. With respect to the permutations themselves, however, a stack and two parallel queues are incomparable as $3, 2, 1$ is not sortable in two parallel queues but in a stack.

Since Knuth [Knu97], certain classes of permutation networks, in particular, networks of parallel stacks or queues, have been investigated. Tarjan [Tar72b] investigated the case of k parallel queues and found the forbidden pattern $k+1, k, k-1, \dots, 2, 1$. That is, a permutation is sortable with k parallel queues if and only if its longest monotonically decreasing subsequence has at most length k. Tarjan also studied k parallel stacks with the restriction that first all numbers have to be completely inserted to the stacks and, afterwards, all numbers are removed from the stacks. With this restriction, a permutation is sortable in k parallel stacks if and only if it does not contain the pattern $1, 2, 3, 4, \dots, k+1$ [Tar72b]. For instance, the permutation in Fig. 2.4(d) is also sortable with this restriction.

Without this restriction, k parallel stacks are algorithmically much harder than k parallel queues. By [Eve71], the decision problem whether a permutation is sortable in k parallel queues is reducible to the problem of k-coloring a perfect graph [Gol04] which can be done in polynomial time [EPL72]. The respective decision problem for k parallel stacks can be reduced to the k-coloring problem of a circle graph, however, this time the coloring problem is $\mathcal{NP}$-complete for $k \geq 4$ [Ung88]. For $k = 1$, $k = 2$, and $k = 3$ efficient algorithms are given in [Knu97, RT84, Ung92], respectively, where [RT84] will be discussed in more detail in Sect. 2.6.2. That k parallel stacks are more complicated than k queues is also reflected in the fact that already for $k = 2$ there is an infinite number of forbidden patterns [Pra73].

Variants of a single deque are considered in [Pra73] and [Knu97]: Pratt gives the finitely many forbidden patterns for the output-restricted (input-restricted) deque, where removals (insertions) are allowed at only one side. For the (unrestricted) deque, Pratt [Pra73] also gives a scheme for all forbidden patterns of which there infinitely many. The algorithm to decide the sortability of a permutation in two parallel stacks from [RT84] can be adapted to decide whether a permutation is sortable in a single deque.

Series compositions of stacks have also been investigated. Note that k queues in series can only "sort" the permutation $1, \dots, n$ and are, therefore, uninteresting. In stark contrast to this, the case of k stacks in series is much harder as already for the case of $k = 2$ it is still open whether the respective decision problem is $\mathcal{NP}$-complete or efficiently solvable [Bó02].

2.1.2.2 Automata and Formal Languages

Every computer science student sooner or later learns about finite automaton which characterize the regular languages [HMU03]. More powerful language classes are obtained by equipping a finite automaton with a data structure to store information: Adding a stack results in push-down automata (PDAs) that characterize the context-free languages, and with an infinite tape we get the famous Turing machine and the recursively enumerable languages. By comparing the languages recognized by machines equipped with certain data structures, we can draw conclusions about the power of the data structures themselves and their differences. Unfortunately, this approach, when applied as suggested, gives only a very coarse-grained classification: Already with two stacks, an automaton is as powerful as a Turing machine and the same holds when equipped with a queue [Vol70, Ros93] and, thus, with a deque.

Things become interesting again when restrictions are put on the computation time: Real-time automata equipped with a queue have been studied in [Vol70], where the queue is called "Pufferspeicher" (buffer), and in [Bra80], where Post machines are considered. In a real-time computation, an automaton reads a symbol from the input in each time step. Vollmar [Vol70] showed that deterministic real-time queue automata are strictly less powerful than their non-deterministic counterparts which parallels the situation for PDAs. Remember that the language $L = \{ww^r | w \in \{0,1\}^*\}$, where w^r is the reversed version of w, cannot be recognized by a deterministic PDA but by a non-deterministic one which guesses the middle of the word. The PDA uses the stack to match w with w^r by exploiting the stack's LIFO principle. Interestingly, though not surprisingly, Vollmar showed that the language $L = \{ww | w \in \{0,1\}^*\}$ cannot be recognized by a deterministic real-time queue automaton but by a non-deterministic one, where w is matched with its second occurrence using the queue's FIFO principle. Let DQL be the class of languages accepted by a deterministic real-time queue automaton. Vollmar classifies DQL into the Chomsky hierarchy: every regular language is in DQL and every language in DQL is context-sensitive. With respect to context-free languages, DQL lies askew: there are context-free languages that are not in DQL, and there are languages in DQL that are context-sensitive but not context-free [Vol70]. Thus, with respect to the Chomsky-hierarchy, a queue is incomparable to a stack.

Ayers considered real-time deque automata in [Aye85] which recognize a proper superset of the context-free languages. Altogether, we have that a deque is strictly more powerful than two stacks, which are strictly more powerful than a queue and a stack, and the latter of which are incomparable. Note that the language $L = \{a^n b^n c^n | n \leq 0\}$ is recognizable in real-time by an automaton with two stacks.

Another possibility to compare different data structures is simulation: If an automaton with one data structure $\mathcal{D}_1$ can simulate another automaton with a different data structure $\mathcal{D}_2$ with no time overhead or only a time overhead that is linear, then $\mathcal{D}_1$ is considered at least as powerful as $\mathcal{D}_2$. Brandenburg commented on [Aye85] in [Bra87] and raised the question whether there is a strict hierarchy between the languages accepted in real-time/linear time by non-deterministic machines equipped with a queue, an output-restricted deque, a deque, or a finite number of tapes. A deque is output-restricted if elements can only be removed at one side. For the deterministic case, it turns out that a queue machine cannot simulate a stack machine in linear and, thus, also real-time [Li88]. Further, two stacks are at least as powerful as an output-restricted deque as the first can simulate the latter in linear time. For insertions, each side of the deque is modeled by one stack, the head stack and the tail stack. Assume that items can only be removed from the head of the deque. In case of a removal, the

item is removed from the head stack if not empty. Otherwise, all elements of the tail stack are removed and pushed onto the head stack. By this mechanism each item is inserted and removed at most twice which results in a linear total running time.

Petersen [Pet01] compared two stacks with the deque and its variants. He showed that deterministic machines with an output-restricted deque cannot simulate deterministic machines with two stacks (or one deque) in real-time. He further points out that there are languages accepted in real-time by a deterministic queue machine while there is no such deterministic two-stack machine. Hence, two stacks are incomparable to a single queue or output-restricted deque with respect to real-time computations. As shown by Rosenberg [Ros93], three stacks suffice to deterministically simulate a deque machine in linear time. Whether two stacks can simulate a deque, and two queues can simulate a stack, both deterministically and in linear time, is still open. Interestingly, in the non-deterministic setting, two stacks are equivalent to a deque using a linear-time simulation [Pet01].

Kosaraju considered a curious variant of the deque [Kos79]: The concatenable deque extends the deque by allowing the concatenation of two deques. He shows that every machine equipped with k concatenable deques can be real-time simulated by an $\mathcal{O}(k)$ deque machine. In the context of graph layouts, we define the time-inverted variant of the concatenable deque in Sect. 2.5: the splittable deque can be split into deques and we use it to characterize planarity.

Languages of machines equipped with a certain data structure can be characterized by an archetype language that reflects the working principle of the data structure. The famous Chomsky-Schützenberger theorem [CS63] states that a language L is context-free if and only if it can be expressed as $L = g(h^{-1}(L_{\text{Dyck}}) \cap L_R)$ where g is a homomorphism, h^{-1} is an inverse homomorphism, L_{Dyck} is the Dyck-language, i. e., the balanced strings of parentheses/brackets on the alphabet $\{(,),[,]\}$, and L_R is a regular language. From the perspective of this theorem, the Dyck-language is the archetype of a context-free language and, thus, language-theoretically characterizes the stack and LIFO principle. The queue or FIFO pendant of the Dyck-language, denoted by L_{FIFO}, is the set of words on $\{(,),[,]\}$, where no pair of parenthesis or brackets nests another pair of brackets or parenthesis, respectively, e. g., "([ )]". Book and Brandenburg proved a Chomsky-Schützenberger-like theorem for real-time queue automatons in [BB80]: A language is a queue language if and only if it can be expressed as $L = g(h^{-1}(L_{\text{FIFO}}) \cap L_R)$ where g is a homomorphism, h^{-1} is an inverse homomorphism, and L_R is a regular language. Again, the FIFO principle of the queue is directly reflected in the characteristic queue language L_{FIFO}. To the best of our knowledge, no Chomsky-Schützerberger-like theorem for deque languages has been published.

2.2 Deque Graphs and Linear Cylindric Drawings

For a graph $G = (V, E)$, we call a total order $\prec$ on the set of vertices V *linear layout* (of G). The reflexive closure of $\prec$ is denoted by $\preceq$, i. e., $u \preceq v \Leftrightarrow u \prec v \vee u = v$. A linear layout imposes a direction on each edge $\{u, v\} \in E$ such that $\{u, v\}$ is directed from u to v if $u \prec v$. In the following, we identify each edge's undirected with its directed version. If $u \prec v$, then u is said to be a *predecessor* of v or u *precedes* v. Conversely, v is a successor of u or v *succeeds* u. If there is no vertex w with $u \prec w \prec v$, then u is the *immediate predecessor* of v and v the *immediate successor* of u. The vertex with no predecessor is called *first vertex* and the vertex with no successor is called *last vertex*.

2.2.1 Linear Cylindric Drawings

A linear cylindric (LC) drawing is a drawing of a graph on the surface of the rolling cylinder with certain properties. As defined in Sect. 1.1.9, $\mathbf{C}_r^3 \subsetneq \mathbb{R}^3$ is the set of points of the rolling cylinder's surface. Each point p in $\mathbf{C}_r^3$ is uniquely defined by its cylindrical coordinates consisting of its x-coordinate and its azimuth ϕ. Let $h = \{(x, \phi) \in \mathbf{C}_r^3 \mid \phi = 0\}$ be an open straight line on the cylinder's surfaces which is called the *front line*. In an LC drawing, the vertices are placed on the front line; hence, the name *linear* cylindric. The edge curves share no points with the front line except for their endpoints.

Definition 2.1. *A drawing* Γ *on* $\mathbf{C}_r^3$ *is called* linear cylindric *if all vertices lie on the front line:*

$$\forall v \in V : \Gamma(v) \in h\,,$$

and no inner part of an edge curve shares a point with the front line:

$$\forall e \in E : \Gamma(e)[(0,1)] \cap h = \emptyset\,.$$

Every graph has an LC drawing by arbitrarily placing the vertices on distinct points on the front line and connecting them by edge curves fulfilling the properties of Def. 2.1. However, only certain graphs allow for a plane LC drawing.

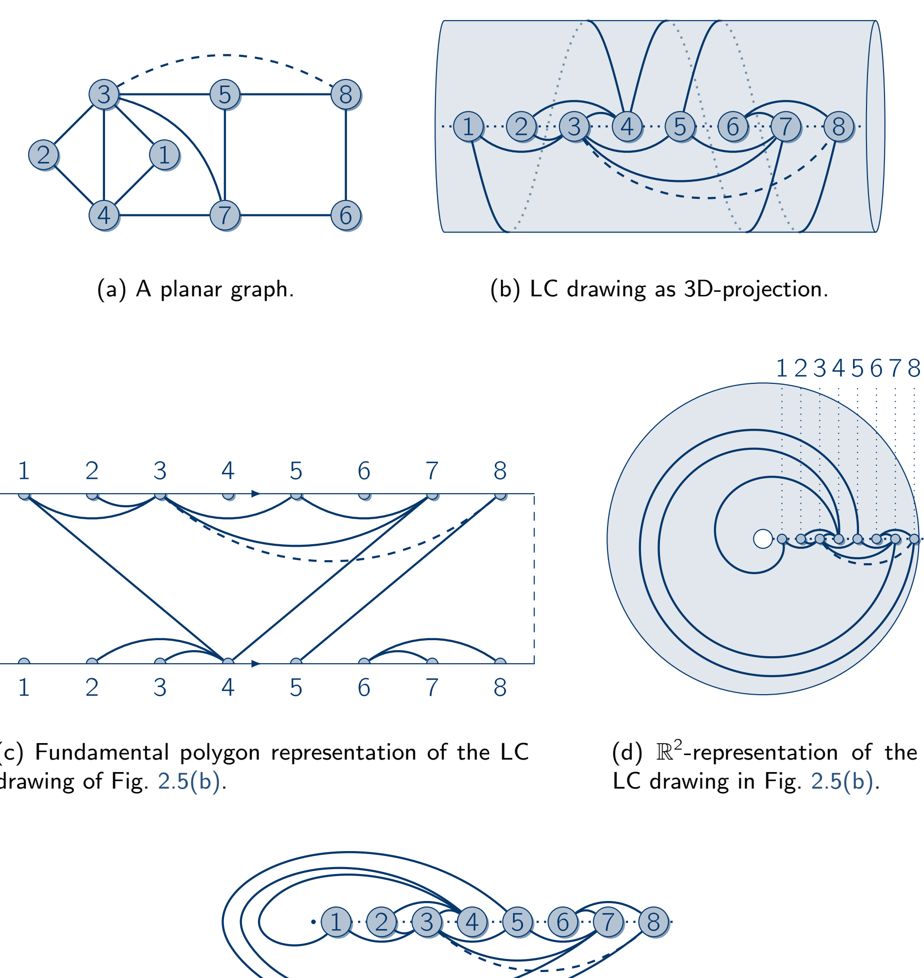
Definition 2.2. *An LC drawing* Γ *is called* plane *if Eq.* (1.1) *holds for* Γ*, i. e.,* Γ *is plane. A graph which has a plane LC drawing is called linear cylindric planar (*LC planar*).*

Consider the planar graph from Fig. 2.5(a) and its LC drawing in Fig. 2.5(b) as a 3D projection. The vertices are labeled according to their positions on the front line from the left to the right. The LC drawing is plane without edge $(3, 8)$ (dashed). Fig. 2.5(c) shows the fundamental polygon representation, where the front line coincides with the top and bottom of the fundamental polygon. The labels of the vertices are shown above and below the fundamental polygon. The $\mathbb{R}^2$-representation is shown in Fig. 2.5(d). As the edges that wind around the cylinder consume much space, we opt for a more condensed and equivalent $\mathbb{R}^2$-representation as shown in Fig. 2.5(e), where the inner part of the disc in Fig. 2.5(d) is symbolized by a black dot. Every plane LC drawing has a plane $\mathbb{R}^2$-representation and we get:

Proposition 2.1. *An LC planar graph is also planar.*

Given a plane drawing of a graph, we are usually not interested in the exact positions of the vertices or the curvatures of the edge curves but rather in a discrete description of the drawing. In the case of plane drawings, this is the embedding. A discrete description of a plane LC drawing Γ of a graph $G = (V, E)$ can be obtained by the following observations: First, since the vertices are positioned on distinct points on the front line, the vertices' x-coordinates induce a linear layout $\prec$ on V, i. e., $u \prec v$ if $x_u < x_v$, where x_u and x_v are the x-coordinates of u and v, respectively. Second, the edge curve of an edge e, which is incident to vertex v, enters v from either above or below the front line. This follows from the continuity of the edge curves and from the fact that each edge curve shares only its endpoints with the front line. Denote by $\phi(x)$ the azimuth of a point $x \in \mathbf{C}_r^3$. Assume that $\Gamma(e)(0) = \Gamma(v)$. Then, we say that e *enters* v *from above* the front line if:

$$\exists t_0 \in (0,1) : \forall t \in (0, t_0) : \phi(\Gamma(e)(t)) > 0\,. \tag{2.2}$$



(a) A planar graph.

(b) LC drawing as 3D-projection.

(c) Fundamental polygon representation of the LC drawing of Fig. 2.5(b).

(d) $\mathbb{R}^2$-representation of the LC drawing in Fig. 2.5(b).

(e) "Condensed" version of Fig. 2.5(d).

Figure 2.5: A graph and a possible LC drawing.

If $\Gamma(e)(1) = \Gamma(v)$, we get:

$$\exists t_0 \in (0,1) : \forall t \in (t_0, 1) : \phi(\Gamma(e)(t)) > 0 \,. \tag{2.3}$$

This means that the azimuth of e's edge curve in the vicinity of v is positive and, hence, the edge curve is above the front line. For instance, in Fig. 2.5(c), edge $(2,4)$ enters both of its endpoints from above. We say that e *enters* v *from below* if Eq. (2.2) (resp. Eq. (2.3)) holds with inverted relation sign. In Fig. 2.5(c), edge $(3,7)$ enters both of its endpoint from below. We partition the set $E(v)$ of edges incident to a vertex v into two sets $E^{\perp}(v)$ and $E^{\top}(v)$ with:

$$E^{\perp}(v) := \{e \in E(v) \mid e \text{ enters } v \text{ from above}\}\,,$$
$$E^{\top}(v) := \{e \in E(v) \mid e \text{ enters } v \text{ from below}\}\,.$$

The superscripts are chosen to resemble the way an edge enters the vertex.

Assume we are given a plane LC drawing. From this drawing we obtain a planar rotation system, i. e., the counterclockwise cyclic order of the edges around each vertex. We use this rotation system to totally order the sets $E^{\perp}(v)$ and $E^{\top}(v)$ for each vertex v. For this, consider Fig. 2.6: For $E^{\perp}(v)$, we obtain the total order $e_1^{\perp}, \dots, e_k^{\perp}$ $(k \geq 0)$ by following the rotation system in clockwise order. Symmetrically, for $E^{\top}(v)$, we obtain the total order $e_1^{\top}, \dots, e_{\ell}^{\top}$ $(\ell \geq 0)$ by following the rotation system in counterclockwise order. In order to indicate that $E^{\perp}(v)$ and $E^{\top}(v)$ are totally ordered, we denote them by tuples $E^{\perp}(v) = (e_1^{\perp}, \dots, e_k^{\perp})$ and $E^{\top}(v) = (e_1^{\top}, \dots, e_{\ell}^{\top})$. The reason why we define the order of $E^{\perp}(v)$ and $E^{\top}(v)$ in this way will become clear in Sect. 2.2.3 when study the relationship between LC drawings and deque layouts. Note that by totally ordering the sets $E^{\perp}(v)$ and $E^{\top}(v)$, we obtain a rotation system of vertex v as shown in Fig. 2.6.

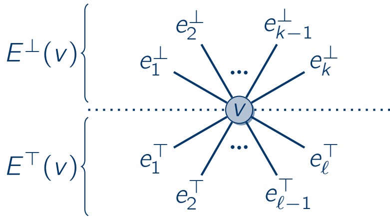


Figure 2.6: The rotation system at a vertex v defines total orders of the sets $E^{\perp}(v)$ and $E^{\top}(v)$.

Definition 2.3 (Linear Cylindric Rotation System (LC Rotation System)). *We call the tuple $\Lambda = (\prec, E^{\perp}, E^{\top})$ consisting of a linear layout and of totally ordered sets $E^{\perp}(v)$ and $E^{\top}(v)$ for all vertices v a* linear cylindric rotation system *(*LC rotation system*).*

Remember that in the case of planar graphs, a rotation system is called planar if there is a plane drawing which implies the rotation system (cf. Sect. 1.1.7). For LC rotation systems, we make an analogous definition.

Definition 2.4 (Planar LC Rotation System). *We call an LC rotation system* planar *if there is a plane LC drawing that implies the LC rotation system.*

A planar rotation system defines an embedding consisting of vertices, edges and faces (cf. Sect. 1.1.7). Note that a planar LC rotation system also defines a rotation system which is planar in the usual sense and, hence, we obtain an embedding. We call this embedding *linear cylindric embedding* (*LC embedding*).

2.2.2 Deque Layouts

In this section, we introduce layouts of graphs in the deque. Since the stack, two stacks, and the queue are just special cases of the deque, all of the following definitions also apply to them by using adequate restrictions.

2.2.2.1 The Deque Data Structure

A deque is a data structure that is usually implemented as a (doubly connected) list with the possibility to insert and remove data items at the deque's two sides, called head `h` and tail `t` (see top of Fig. 2.7). In the case of graph layouts, these data items are the edges of a graph. At each time instant, the current state of the deque is defined by its *content* $\mathcal{C} = (e_1, \dots, e_k)$ ($k \geq 0$), where e_1 is at the deque's head and e_k at its tail. For instance, $\mathcal{C} = (e_1, e_2, e_3, e_4, e_5)$ at the top of Fig. 2.7. If $k = 0$, the deque is empty which is denoted by the empty content $()$. With a slight abuse of notation, we denote by $e \in \mathcal{C}$ that e appears in $\mathcal{C}$. A content $\mathcal{C} = (e_1, e_2, \dots, e_k)$ induces a total order $\ll_{\mathcal{C}}$ on the edges in $\mathcal{C}$ with $e_1 \ll_{\mathcal{C}} e_2 \ll_{\mathcal{C}} \dots \ll_{\mathcal{C}} e_k$. As long as no ambiguity concerning the content can arise, we drop the subscript $\mathcal{C}$ in $\ll_{\mathcal{C}}$, i. e., $e_1 \ll e_2 \ll \dots \ll e_k$. For example, in Fig. 2.7, $e_3 \ll e_4$ for all contents.

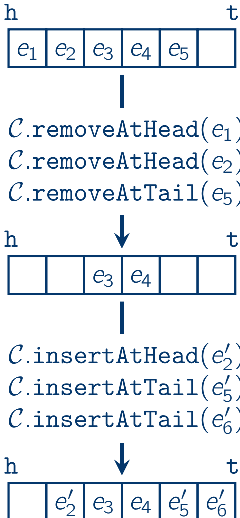


Figure 2.7: A deque in which edges are processed.

Let $\mathcal{C} = (e_1, \dots, e_k)$ ($k \geq 1$) be a non-empty content. With the deque, either edge e_1 can be removed at the head or e_k can be removed at the tail. For convenience, we adopt the object-oriented point of view, where $\mathcal{C}$ is an object with methods to manipulate its content: $\mathcal{C}$.`removeAtHead`(e_1) removes e_1 at the head and $\mathcal{C}$.`removeAtTail`(e_k) removes e_k at the tail with resulting content $(e_2, \dots, e_k)$ and $(e_1, \dots, e_{k-1})$, respectively. If $k = 1$, the resulting content is empty. As return values, both methods return `true` if the removal is successful,

i. e., the item was accessible at the corresponding side, or `false` otherwise. In the example in Fig. 2.7, edges e_1 and e_2 are removed at the head in this order and, afterwards, edge e_5 is removed at the tail.

To insert edges, the deque provides the methods `insertAtHead` and `insertAtTail`. Both have no return value since an insertion is always successful. As the names suggest, $\mathcal{C}$.`insertAtHead`(e) inserts e at the head and, hence, transforms a deque's content $\mathcal{C} = (e_1, \dots, e_k)$ $(k \geq 0)$ to $\mathcal{C} = (e, e_1, \dots, e_k)$. Likewise, $\mathcal{C}$.`insertAtTail`(e) results in $\mathcal{C} = (e_1, \dots, e_k, e)$. Note that if $\mathcal{C} = ()$ beforehand, we obtain $\mathcal{C} = (e)$ in both cases. For our example in Fig. 2.7, edge e'_2 is inserted at the head, and edges e'_5 and e'_6 at the tail in this order.

2.2.2.2 Graph Layouts in the Deque

Assume, we are given a linear layout of a graph. In a deque layout, the vertices can be seen as processing units chained together in a processing pipeline in order of the given linear layout. Each vertex v receives as input a deque with a certain content as produced by its immediate predecessor; or an empty deque if v is the first vertex. All incoming edges (u, v), i. e., edges from predecessors, are removed and all outgoing edges (v, w), i. e., edges to successors, are inserted. Afterwards, the resulting content is passed to the immediate successor of v. If v is the last vertex, its output must be empty.

We further need to specify how the edges are processed at each vertex: At each vertex v, each of its incident edges is either processed at the head or the tail. This partitions the set of v's incident edges into two sets $E^{\mathtt{h}}(v)$ and $E^{\mathtt{t}}(v)$. Each incoming edge $e \in E^{\mathtt{h}}(v)$ $(e \in E^{\mathtt{t}}(v))$ is removed at the head (tail) and each outgoing edge $e \in E^{\mathtt{h}}(v)$ $(e \in E^{\mathtt{t}}(v))$ is inserted at the head (tail). Any edge $e = (u, v)$ with $e \in E^{\mathtt{h}}(u) \wedge e \in E^{\mathtt{h}}(v)$ or $e \in E^{\mathtt{t}}(u) \wedge e \in E^{\mathtt{t}}(v)$ is processed in the deque like in a stack and so we call e *stack edge*. Likewise, we call any edge $e = (u, v)$ with $e \in E^{\mathtt{h}}(u) \wedge e \in E^{\mathtt{t}}(v)$ or $e \in E^{\mathtt{t}}(u) \wedge e \in E^{\mathtt{h}}(v)$ *queue edge*.

We assume that both sets $E^{\mathtt{h}}(v)$ and $E^{\mathtt{t}}(v)$ are totally ordered, i. e., $E^{\mathtt{h}}(v) = (e^{\mathtt{h}}_1, \dots, e^{\mathtt{h}}_k)$ $(k \geq 0)$ and $E^{\mathtt{t}}(v) = (e^{\mathtt{t}}_1, \dots, e^{\mathtt{t}}_\ell)$ $(\ell \geq 0)$. Then, for any two edges $e^{\mathtt{h}}_i$, $e^{\mathtt{h}}_j$ with $1 \leq i < j \leq k$, edge $e^{\mathtt{h}}_i$ is processed before $e^{\mathtt{h}}_j$ at vertex v. For instance, if both $e^{\mathtt{h}}_i$ and $e^{\mathtt{h}}_j$ are to be removed at v, then $e^{\mathtt{h}}_i$ is removed at the head before $e^{\mathtt{h}}_j$ is removed from the same side. Likewise, the edges in $E^{\mathtt{t}}(v)$ are processed in order.

Definition 2.5. *We call a tuple $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ consisting of a linear layout and of totally ordered sets $E^{\mathtt{h}}(v)$ and $E^{\mathtt{t}}(v)$ for each vertex v a* deque schedule.

Assuming that all edges can be processed in the deque, a deque schedule uniquely defines the content of each input and output of all vertices: Let $e^{\mathtt{h}}$ and $e^{\mathtt{t}}$ be two edges of which each is either inserted or removed at vertex v such that $e^{\mathtt{h}} \in E^{\mathtt{h}}(v)$ and $e^{\mathtt{t}} \in E^{\mathtt{t}}(v)$. For the output of vertex v, it makes no difference whether $e^{\mathtt{h}}$ is processed before $e^{\mathtt{t}}$ or the other way around. The result is always the same as both are processed at different sides. As a canonical order, we assume that all edges in $E^{\mathtt{h}}(v)$ are processed before the edges in $E^{\mathtt{t}}(v)$.

Alg. 2.2 shows `IsDequeLayout` which takes as input a graph and a deque schedule. `IsDequeLayout` starts with an empty deque $\mathcal{C} = ()$ and processes the vertices in order of the linear layout. For each vertex, `IsDequeLayout` calls `ProcessVertex` (Alg. 2.3). The parameters are the current content $\mathcal{C}$ of the deque, the linear layout $\prec$, and the deque schedule $E^{\mathtt{h}}(v)$ and $E^{\mathtt{t}}(v)$ of v. In `ProcessVertex`, at first all edges in $E^{\mathtt{h}}(v) = (e^{\mathtt{h}}_1, \dots, e^{\mathtt{h}}_k)$ are processed in order, followed by all edges in $E^{\mathtt{t}}(v)$. Each edge whose other endpoint is a predecessor is

removed and edges to successors are inserted. If all operations succeed, `ProcessVertex` returns the resulting content of the deque. Otherwise, a removal failed and `ProcessVertex` returns $\perp$, and `IsDequeLayout` returns `false`. If all edges can be processed, then `IsDequeLayout` returns true.

Algorithm 2.2. `IsDequeLayout`

Input: graph $G = (V, E)$ and deque schedule $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$
Output: `true` if deque schedule is a deque layout and `false` otherwise

1 $\mathcal{C} \leftarrow ()$
2 **foreach** $v_i = v_1, \dots, v_n$ in order of $\prec$ **do**
3 $\quad\mathcal{C} \leftarrow$ `ProcessVertex`$(\mathcal{C}, \prec, E^{\mathtt{h}}(v_i), E^{\mathtt{t}}(v_i))$
4 $\quad$**if** $\mathcal{C} = \perp$ **then return** `false`
5 **return** `true`

Algorithm 2.3. `ProcessVertex`

Input: deque with content $\mathcal{C}$, linear layout $\prec$, and deque schedule $E^{\mathtt{h}}(v)$ and $E^{\mathtt{t}}(v)$ of vertex v
Output: content of deque after processing the edges incident to v, or $\perp$ if an edge could not be removed

1 **foreach** $e_i^{\mathtt{h}} = \{u, v\}$ in $E^{\mathtt{h}}(v) = (e_1^{\mathtt{h}}, \dots, e_k^{\mathtt{h}})$ in order **do**
2 $\quad$**if** $u \prec v$ **then** $e_i^{\mathtt{h}}$ is an incoming edge from a predecessor
3 $\quad\quad$**if** $\mathcal{C}$`.removeAtHead`$(e_i^{\mathtt{h}}) =$ `false` **then return** $\perp$
4 $\quad$**else** $\mathcal{C}$`.insertAtHead`$(e_i^{\mathtt{h}})$, where $e_i^{\mathtt{h}}$ is an outgoing edge to a successor, i. e., $v \prec u$
5 **foreach** $e_j^{\mathtt{t}} = \{u, v\}$ in $E^{\mathtt{t}}(v) = (e_1^{\mathtt{t}}, \dots, e_\ell^{\mathtt{t}})$ in order **do**
6 $\quad$**if** $u \prec v$ **then** $e_j^{\mathtt{t}}$ is an incoming edge from a predecessor
7 $\quad\quad$**if** $\mathcal{C}$`.removeAtTail`$(e_j^{\mathtt{t}}) =$ `false` **then return** $\perp$
8 $\quad$**else** $\mathcal{C}$`.insertAtHead`$(e_j^{\mathtt{t}})$, where $e_j^{\mathtt{t}}$ is an outgoing edge to a successor, i. e., $v \prec u$
9 **return** $\mathcal{C}$

Definition 2.6 (Deque Layout and Deque Graph). *A deque schedule of a graph G is called* deque layout *if all edges can be processed accordingly, i. e.,* `IsDequeLayout` *in Alg. 2.2 returns* `true`*. G is called* deque graph *if it has a deque layout.*

Fig. 2.8 illustrates a deque layout of the K_4, the complete graph with four vertices. The vertices are placed on the dotted line in order of the linear layout. Each of the vertices is connected to a shaded box which contains a snippet of pseudo-code that shows how the edges are inserted and removed. The content of the deque is shown below the dotted line.

2.2.3 Linear Cylindric Planar Graphs and Deque Graphs

As already suggested in Sect. 2.1, there is a strong relationship between planarity and the ability to layout a graph in a data structure. The aim of this section is to prove the following theorem:

Theorem 2.1. *A graph is a deque graph if and only if it is linear cylindric planar.*

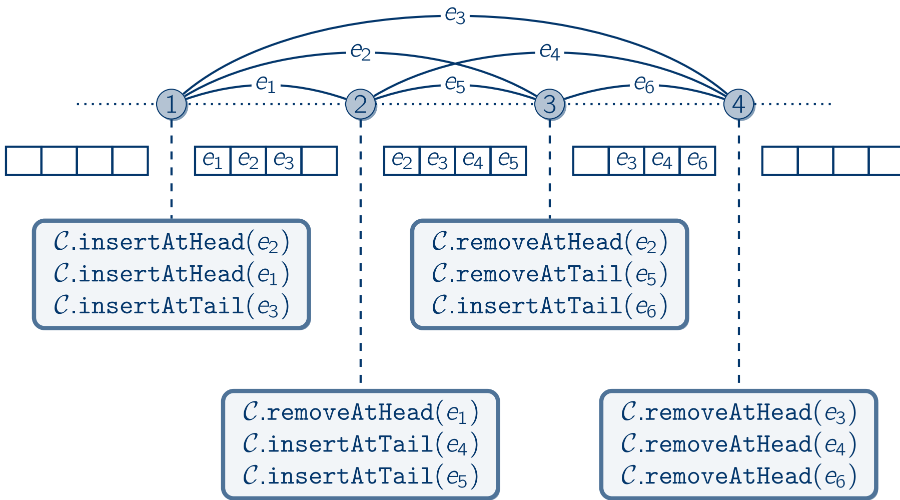


Figure 2.8: A deque layout of the K_4.

In order to combinatorially describe LC drawings, we have introduced LC rotation systems which are tuples $\Lambda = (\prec, E^\perp, E^\top)$ (Def. 2.3) consisting of a linear layout, i. e., the order of vertices on the front line, and of totally ordered sets $E^\perp(v)$ for $E^\top(v)$ for each vertex v. The latter define a rotation system of the graph. In order to describe how edges are processed in the deque, we have introduced deque schedules which are tuples $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ (Def. 2.5) also consisting of a linear layout, i. e., the order in which the vertices are processed, and of totally ordered sets $E^{\mathtt{h}}(v)$ for $E^{\mathtt{t}}(v)$ for each vertex v, which define at which side and in which order the edges are processed in the deque. Note that we use the same mathematical objects for LC rotation systems and deque schedules. Even more, these two concepts are also equal with respect to planarity and deque layouts:

Lemma 2.1. *Let Σ be a deque schedule and $\Lambda := \Sigma$ be the corresponding LC rotation system. Σ is a deque layout if and only if Λ is planar.*

Before we start proving Lem. 2.1, we explore the intuition behind the statement. Consider Fig. 2.9 which shows a plane LC drawing. From the drawing, we obtain a planar LC rotation system $\Lambda = (\prec, E^\perp, E^\top)$. By Lem. 2.1, the corresponding deque schedule $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ is a deque layout. In Fig. 2.9, the operations corresponding to the deque layout are shown in the shaded rectangles above and below the fundamental polygon, connected by dashed lines to the respective vertices. The input contents $\mathcal{C}_i$ for each vertex i are:

$$\mathcal{C}_1 = () , \qquad \mathcal{C}_2 = ((1,4),(1,3)) ,$$
$$\mathcal{C}_3 = ((2,4),(1,4),(1,3),(2,3)) , \qquad \mathcal{C}_4 = ((3,4),(2,4),(1,4),(3,7),(3,5)) ,$$
$$\mathcal{C}_5 = ((4,7),(3,7),(3,5)) , \qquad \mathcal{C}_6 = ((5,8),(4,7),(3,7),(5,7)) ,$$
$$\mathcal{C}_7 = ((6,7),(6,8),(5,8),(4,7),(3,7),(5,7)) , \qquad \mathcal{C}_8 = ((6,8),(5,8)) ,$$
$$\mathcal{C}_9 = () ,$$

where $\mathcal{C}_9$ is the output of vertex 9. The correspondence between Λ and Σ implies that the order in which the vertices are processed is equal to the order of the vertices on the front line

of the LC drawing. Further, let $e = (v, w)$ be an edge with $e \in E^{\perp}(v)$, e. g., edge $(2, 4)$ in Fig. 2.9. In the deque layout, $e \in E^{\mathtt{h}}(v)$ and, hence, e is inserted at the head of the deque. Likewise, any edge in $E^{\top}(v)$ is processed at the tail of the deque. In Fig. 2.9, the area that corresponds to tail is shaded and any edge e that enters one of its endpoints v within the shaded area is processed at v at the tail. Likewise, any edge that enters one of its endpoints within the white region is processed at the head. In general, any stack edge, like $(1, 3)$ and $(2, 4)$, enters both of its endpoints within either the shaded or the white area, whereas any queue edge, like $(1, 4)$ and $(4, 7)$, changes sides. As for the order in which the edges need to be inserted and removed, the LC rotation system obtained from Fig. 2.9 is of help: Consider the rotation system of vertex 4 with $E^{\perp}(4) = ((3, 4), (2, 4), (1, 4), (4, 7))$ which is the order in which the edges are processed at the head of the deque.

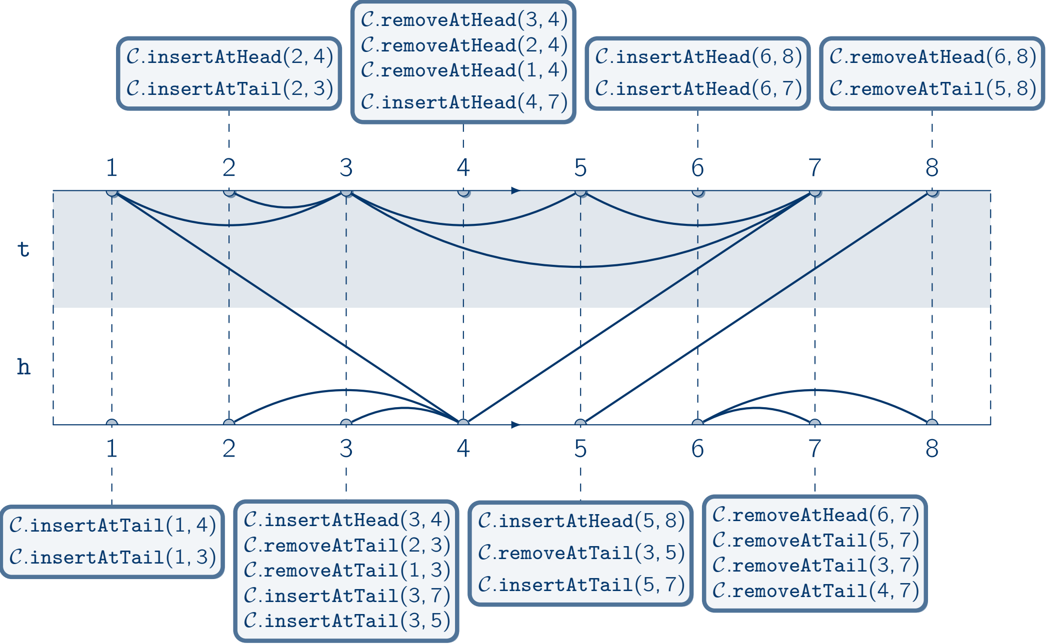


Figure 2.9: An LC embedding and its corresponding deque layout.

We readily obtain an observation from Fig. 2.9: At vertex 4, at first edges $(3, 4)$, $(2, 4)$, and $(1, 4)$ are removed at the head before edge $(4, 7)$ is inserted at the head. This is true in general. Suppose that two edges e and e' are processed at the head such that e is removed and e' is inserted. Then inserting e' first would imply that $e' \ll e$, hence, making it impossible to remove e at the head.

Proposition 2.2. *In a deque layout, at each vertex v all edges in $E^{\mathtt{h}}(v)$ ($E^{\mathtt{t}}(v)$) pointing to predecessors are removed before the edges in $E^{\mathtt{h}}(v)$ ($E^{\mathtt{t}}(v)$) pointing to successors are inserted.*

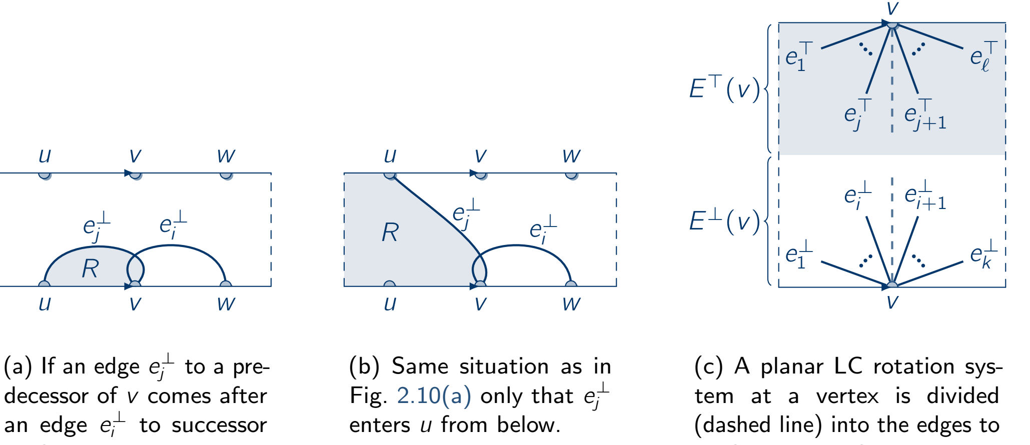
Proposition 2.3. *Let $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ be a deque layout. For each vertex v, $E^{\mathtt{h}}(v) = (e^{\mathtt{h}}_1, \dots, e^{\mathtt{h}}_k)$ can be divided into two halves $E^{\mathtt{h}}_{\text{Pred}}(v) = (e^{\mathtt{h}}_1, \dots, e^{\mathtt{h}}_i)$ and $E^{\mathtt{h}}_{\text{Succ}}(v) = (e^{\mathtt{h}}_{i+1}, \dots, e^{\mathtt{h}}_k)$ for some $0 \le i \le k$ such that all edges in $E^{\mathtt{h}}_{\text{Pred}}(v)$ are removed at the head at v, and all edges in $E^{\mathtt{h}}_{\text{Succ}}(v)$ are inserted at the head at v. If $i = 0$, then $E^{\mathtt{h}}_{\text{Pred}}(v) = \emptyset$ and, if $i = k$, then $E^{\mathtt{h}}_{\text{Succ}}(v) = \emptyset$. The same holds true for $E^{\mathtt{t}}(v)$.*

The analogous observation for plane LC rotation systems is as follows: Consider again the total order of edges in $E^{\perp}(4) = ((3,4),(2,4),(1,4),(4,7))$ as shown in Fig. 2.9. There, all edges to predecessors of 4, i. e., $(3,4)$, $(2,4)$, $(1,4)$, come before edge $(4,7)$, which points to a successor of 4. If $(4,7)$ would be situated before any of the other edges in the rotation system, we would obtain a situation as depicted in Fig. 2.10(a) or (b) and a crossing would be inevitable.

Lemma 2.2. *Let $\Lambda = (\prec, E^{\perp}, E^{\top})$ be a plane LC rotation system. For each vertex v, the total order of the edges in $E^{\perp}(v) = (e_1^{\perp}, \dots, e_k^{\perp})$ can be divided into two halves $E^{\perp}_{\text{Pred}}(v) = (e_1^{\perp}, \dots, e_i^{\perp})$ and $E^{\perp}_{\text{Succ}}(v) = (e_{i+1}^{\perp}, \dots, e_k^{\perp})$ for some $0 \le i \le k$ such that all edges in $E^{\perp}_{\text{Pred}}(v)$ are incident to predecessors of v, and all edges in $E^{\perp}_{\text{Succ}}(v)$ are incident to successors. If $i = 0$, then $E^{\perp}_{\text{Pred}}(v) = \emptyset$ and, if $i = k$, then $E^{\perp}_{\text{Succ}}(v) = \emptyset$. The same holds true for $E^{\top}(v)$.*

Proof. For contradiction, suppose that there is a vertex v and edges $e_i^{\perp}$ and $e_j^{\perp}$ in $E^{\perp}(v) = (e_1^{\perp}, \dots, e_k^{\perp})$ such that $i < j$ and $e_i^{\perp}$ is incident to a successor w of v and $e_j^{\perp}$ is incident to a predecessor u of v. The situation is depicted in Fig. 2.10(a). As the inner part of the edge curve of $e_j^{\perp}$ has no point in common with the front line, $e_j^{\perp}$ and the front line enclose a region R (shaded in Fig. 2.10(a)). Edge $e_i^{\perp}$ enters v from above and also has no point in common with the front line except for its endpoints. Hence, $e_i^{\perp}$ leaves v within R. The other endpoint w of $e_i^{\perp}$ lies outside of R as if w would lie within, an inner point of $e_j^{\perp}$'s edge curve would lie on the front line. Consequently, the inner part of $e_i^{\perp}$'s edge curve has points inside and outside of R which inevitably leads to a crossing by Jordan's curve theorem (Prop. 1.1); a contradiction to the planarity of the LC rotation system. The proof for $E^{\top}(v)$ is analogous.

Note that we assumed that $e_j^{\perp}$ enters u from above which in general must not be the case. However, even if $e_j^{\perp}$ enters u from below, it encloses a region R together with the front line as illustrated in Fig. 2.10(b) and the proof is analogous. □



(a) If an edge $e_j^{\perp}$ to a predecessor of v comes after an edge $e_i^{\perp}$ to successor in the rotation system, a crossing is inevitable.

(b) Same situation as in Fig. 2.10(a) only that $e_j^{\perp}$ enters u from below.

(c) A planar LC rotation system at a vertex is divided (dashed line) into the edges to predecessors and successors.

Figure 2.10: LC rotation systems of a vertex.

Fig. 2.10(c) shows how the rotation system in a planar LC rotation system is split according to Lem. 2.2. We combine Prop. 2.3 and Lem. 2.2 as follows:

Corollary 2.1. *Let $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ be a deque schedule with corresponding LC rotation system $\Lambda = (\prec, E^{\perp}, E^{\top}) := \Sigma$. If Σ is a deque layout and Λ is planar, then, for each vertex v, $E^{\mathtt{h}}(v) = E^{\perp}(v) = (e_1, \dots, e_k)$ can be split into two halves $E^{\mathtt{h}}_{\text{Pred}}(v) = E^{\perp}_{\text{Pred}}(v) = (e_1, \dots, e_i)$ and $E^{\mathtt{h}}_{\text{Succ}}(v) = E^{\perp}_{\text{Succ}}(v) = (e_{i+1}, \dots, e_k)$ with $0 \leq i \leq k$ such that all $e_1, \dots, e_i$ are incident to a predecessor of v and all $e_{i+1}, \dots, e_k$ are incident to a successor of v. This holds also true for $E^{\mathtt{t}}(v) = E^{\top}(v)$.*

We are now able to prove Lem. 2.1 which states that a deque schedule is a deque layout if and only if the corresponding LC rotation system is planar. The proof is a complete case differentiation in which we show that two edges e and e' cross if and only if e cannot be removed from the deque because e' is blocking its way or vice versa. The cases correspond to the different sides at which the edges are inserted and removed in the deque layout and, respectively, the sides from which the edges enter the front line.

Proof. [Lem. 2.1] Let $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ be a deque schedule and $\Lambda = (\prec, E^{\perp}, E^{\top}) := \Sigma$ be the corresponding LC rotation system.

$\Rightarrow$: By assumption Σ is a deque layout and we have to show that Λ is planar. Assume for contradiction that there are two distinct edges $e = (u, v)$ and $e' = (u', v')$, with $u \prec v$ and $u' \prec v'$, for which a crossing is inevitable. W. l. o. g., we assume that according to the deque schedule e is inserted to the deque before e'. This implies that $u \preceq u'$. If $u = u'$, then either $e, e' \in E^{\mathtt{h}}(u)$, $e, e' \in E^{\mathtt{t}}(u)$, or $e \in E^{\mathtt{h}}(u)$ and $e' \in E^{\mathtt{t}}(u)$. In the first two cases, e comes before e' in $E^{\mathtt{h}}(u)$ and $E^{\mathtt{t}}(u)$, and for the latter case remember that all edges in $E^{\mathtt{h}}(u)$ are processed before the ones in $E^{\mathtt{t}}(u)$ (cf. Alg. 2.3).

If $u \prec v \prec u' \prec v'$, then e and e' are never in the deque at the same time and, hence, both are processed completely independently from each other. Likewise, in the LC drawing, a crossing between e and e' is always avoidable since their endpoints lie on completely disjoint intervals on the front line. If $u \prec v = u' \prec v'$, i. e., e and e' share $v = u'$ as common endpoint, we can apply Cor. 2.1: If Σ is a deque layout, then the rotation system at v has the structure as depicted in Fig. 2.10(c) and e and e' can be drawn without a crossing.

In the following, we assume that $u \preceq u' \prec v$ and distinguish five cases:

- **Both e and e' are stack edges**

  If both edges are stack edges processed at different sides of the deque, then e enters both its endpoints from, say, below, whereas e' enters its endpoints from above. Then, a crossing between e and e' is always avoidable in an LC drawing. Hence, we assume that e and e' are processed at the head; the reasoning is similar if they are inserted and removed at the tail. Similar to the proof of Lem. 2.2, the edge curve of e and the front line enclose a region R in an LC drawing. A crossing between e and e' is inevitable if and only if one of the inner points of the edge curve of e' is within R while another inner point is outside of R (cf. Prop. 1.1). This is the case if and only if one of the following cases applies:

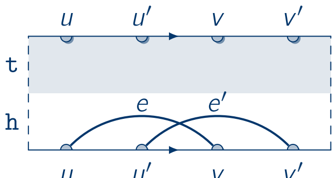
  (i) $u \prec u' \prec v \prec v'$ (cf. Fig. 2.11(a))

  (ii) $u = u' \prec v \prec v'$ and e comes before e' in $E^{\perp}(u)$ (cf. Fig. 2.11(b))

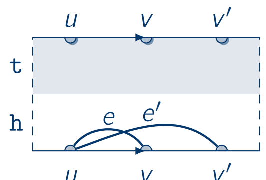
  (iii) $u \prec u' \prec v = v'$ and e comes before e' in $E^{\perp}(v)$

  By assumption, e is inserted at the head before e' and, thus, $e' \ll e$ in the deque. Moreover, in all cases, e must be removed from the head before e': For (i) and (ii), we

have that $v \prec v'$ and for (iii) e comes before e' in $E^{\perp}(v)$. However, removing e is not possible since e' is blocking its way; a contradiction to the assumption that Σ is a deque layout.



(a) Both edges are stack edges processed at the head.



(b) Both edges are stack edges incident to u and processed at the head.

Figure 2.11: Both edges are stack edges.

▶ **Both e and e' are queue edges inserted at the same side**
We assume that both e and e' are inserted at the head and removed at the tail. A crossing is inevitable if and only if one of the following cases applies:

(i) $u \prec u' \prec v' \prec v$ (cf. Fig. 2.12)
(ii) $u = u' \prec v' \prec v$ and e comes before e' in $E^{\perp}(u)$
(iii) $u \prec u' \prec v' = v$ and e comes after e' in $E^{\top}(v)$

Since e is inserted at the head before e', we get $e' \ll e$ in the deque. Note that in (i), edge e' is properly nested in edge e which already suggests that this configuration is incompatible with queue edges. In any of the cases, e' is removed before e: For (i) and (ii), $v' \prec v$ and, for case (iii), e comes after e' in $E^{\top}(v)$. However, e' cannot be removed from the tail since $e' \ll e$; again a contradiction.

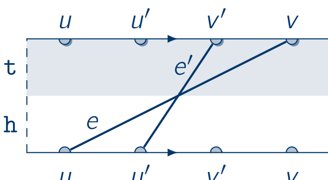


Figure 2.12: Both edges are queue edges inserted at the same side.

▶ **Both e and e' are queue edges inserted at different sides**
We assume that e is inserted at the head and e' at the tail. The situation we obtain is depicted in Fig. 2.13. By assumption, $u \preceq u' \prec v$, and since $e \in E^{\perp}(u) \cap E^{\top}(v)$ and $e' \in E^{\top}(u') \cap E^{\perp}(v')$, a crossing between e and e' is always inevitable. Also, in the deque $e \ll e'$, where e' is removed from the head and e from the tail which is never possible regardless of when each of the edges is removed.

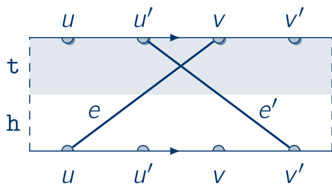


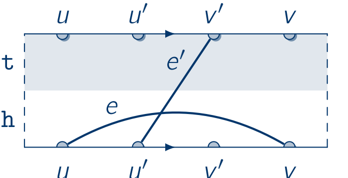
Figure 2.13: Both edges are queue edges inserted at different sides.

- **e is a stack and e' a queue edge**
  Let e be a stack edge inserted and removed at the head (cf. Fig. 2.14(a)). Edge e enters both of its endpoints from above the front line and, thereby, encloses a region R. There is a crossing between e and e' if and only if any inner point of the edge curve of e' is within R since e' is a queue edge. For instance, in Fig. 2.14(a), $u \prec u' \prec v' \prec v$ and u' is inserted at the head and removed from the tail.

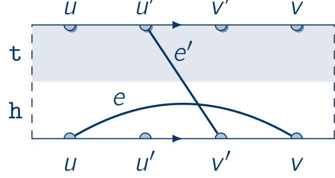
  In general, e and e' cross if and only if one of the following conditions holds true:

  (i) Edge e' enters u' from above. Then, e' necessarily enters u' within R as $u' \prec v$ and either $u \prec u'$, or $u = u'$ and e comes before e' in $E^{\perp}(u)$. Further, e' is a queue edge and so it enters v' from below the front line and, thus, e' leaves R which leads to a crossing. In this case, we obtain $e' \ll e$ for the deque, where e is removed from the head and e' from the tail which is not possible and, thus, a contradiction.

  (ii) Edge e' enters u' from below and $v' \preceq v$, where for $v' = v$ edge e' comes before e in $E^{\perp}(v)$ (cf. Fig. 2.14(b)). In this case, e' enters u' outside of R and v' within, which causes a crossing. In the deque, $e \ll e'$ and e' is removed from the head before e is removed which is not possible; again a contradiction.



(a) e is a stack and e' a queue edge inserted at the head.



(b) e is a stack and e' a queue edge inserted at the tail.

Figure 2.14: e is a stack and e' a queue edge.

- **e is a queue and e' a stack edge**
  This is symmetric to the prior case where e is a stack and e' and queue edge, where the roles of e and e', and of head and tail are swapped, and the linear layout is reversed (Fig. 2.15). The proof is thus similar.

Altogether, we can conclude that Λ is a planar LC rotation system.

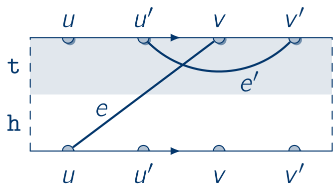


Figure 2.15: e is a queue and e' a stack edge.

$\Leftarrow$: In the following, we assume that Λ is a planar LC rotation system but, towards a contradiction, Σ is no deque layout. As insertions to the deque never cause trouble, there must be an edge which cannot be removed from the deque because another edge is blocking its way. Let $e = (u, v)$ and $e' = (u', v')$ with $u \prec v$ and $u' \prec v'$ be two such edges. As before, we assume w. l. o. g. that e is inserted to the deque before e'. Further, by the very reasoning from the "$\Rightarrow$"-direction, we can assume that $u \preceq u' \prec v$. Again, we distinguish the same five cases as before and, as the reasoning is very similar, we keep it brief here:

- **Both e and e' are stack edges**
  We assume that both edges are inserted and removed at the head. As e is inserted before e', we get $e' \ll e$. Removing e is not possible if and only if e is removed before e' and, thus, $v \preceq v'$, where for $v = v'$ edge e comes before e' in $E^{\mathtt{h}}(v)$. If $v \prec v'$, then $u \preceq u' \prec v \prec v'$ and we obtain the situation depicted in Fig. 2.11(a) and there is an inevitable crossing between e and e'. If $v = v'$ and e comes before e' in $E^{\mathtt{h}}(v)$, we also obtain an inevitable crossing as shown in Fig. 2.11(b). Hence, Λ cannot be planar; a contradiction.

- **Both e and e' are queue edges inserted at the same side**
  If both edges are inserted at the head, then $e' \ll e$ and we have no deque layout if and only if e' has to be removed before e. Hence, $u \preceq u' \prec v' \preceq v$ and we obtain the situation in Fig. 2.12; again a contradiction to the planarity of Λ. The reasoning is analogous if e and e' are inserted at the tail.

- **Both e and e' are queue edges inserted at different sides**
  If e is inserted at the head and e' at the tail, we get $e \ll e'$ and neither can be removed regardless of when any of e and e' has to be removed. Also, in the LC drawing, e and e' always cross as shown in Fig. 2.13.

- **e is a stack and e' a queue edge**
  We assume that e is inserted at the head and distinguish two cases: Either e' is inserted at the head or at the tail. In the first case, $e' \ll e$ and e neither can be removed from the head nor e' from the tail. We obtain the situation in Fig. 2.14(a) and Λ cannot be planar. If e' is inserted at the tail, then $e \ll e'$. Now, removing e from the head is always possible. However, removing e' before e is not possible. This is the case if and only if $v' \prec v$, or $v' = v$ and e' comes before e in $E^{\mathtt{h}}(v)$. For the LC drawing, we obtain the situation in Fig. 2.14(b) and an inevitable crossing which implies that Λ cannot be planar.

- **e is a queue and e' a stack edge**
  By swapping the role of e with e', and $\mathtt{h}$ with $\mathtt{t}$, and reversing the linear layout, this case is proved analogously to the previous case.

Altogether, we have obtained a contradiction in all cases and can conclude that Σ is a deque layout. □

By Lem. 2.1, Thm. 2.1 follows. As every LC planar graph is planar (cf. Prop. 2.1), we get:

Corollary 2.2. *All deque graphs are planar.*

From the proof of Lem. 2.1, we can derive the following interesting interpretation of crossings in LC drawings and invalid deque operations. A crossing between two edges e and e' is inevitable if and only if one of them cannot be removed from the deque because the other is blocking its way. For instance, if $e \ll e'$ then it is not possible to remove e' at the head before e. Assume that e and e' are directly adjacent in the deque. Then, swapping their positions such that $e' \ll e$, which is no valid deque operation, enables us to remove e' at the head. This position swap has a neat graphical interpretation: Consider Fig. 2.16 which shows an LC drawing that is not plane. Nevertheless, we canonically follow the deque layout: At vertex 1 we insert $(1, 4)$ and $(1, 3)$ at the tail and obtain $\mathcal{C}_2 = ((1, 4), (1, 3))$ as input for 2. There is a vertical line with label a between 1 and 2 in Fig. 2.16. By following this line from bottom to top, we encounter all edges in $\mathcal{C}_2$ in order. This observation is always true if the edge curves are x-monotone. By gradually moving this line from left to right, we obtain the content of the deque at each time instant. Now, the interesting thing happens when the line reaches the crossing: Consider the vertical line with label b just before the crossing of edges $(3, 8)$ and $(4, 7)$. There, the content of the deque is $\mathcal{C}_6 = ((5, 8), (4, 7), (3, 8), (3, 7), (5, 7))$. At vertex 7, edge $(4, 7)$ cannot be removed at the tail as $(3, 8)$ is blocking its way; note that in contrast edges $(5, 7)$ and $(3, 7)$ can be removed. Swapping the positions of $(4, 7)$ and $(3, 8)$ in the deque would resolve the issue, which is reflected by the crossing in the drawing: Edge $(4, 7)$ is below $(3, 8)$ before the crossing, whereas afterwards their positions with respect to the vertical ordering are swapped. Thus, an (invalid) swap operation in the deque implies a crossing of the swapped edges and vice versa.

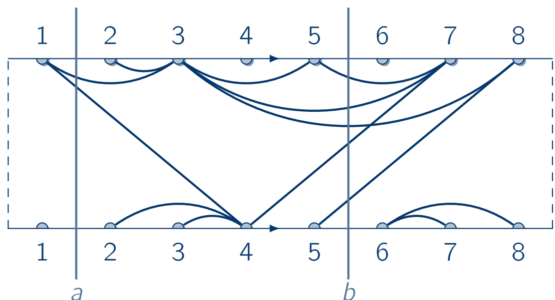


Figure 2.16: A inevitable crossing of two edges e and e' in an LC drawing corresponds to a swap of e and e' in the deque.

2.2.4 Deque Graphs and Hamiltonian Paths

In this section, we further study deque graphs by means of LC drawings. In particular, we will see that deque graphs and LC planar graphs are subgraphs of planar graphs with a Hamiltonian path. As it turns out, the converse is also true. This characterization further illustrates the linear structure of deque graphs and LC planar graphs.

We start with a definition. First, remember that for a linear layout $\prec$ of a graph $G = (V, E)$, a vertex w is the immediate successor of an another vertex v in $\prec$, if $v \prec w$ and $\nexists v' \in V : v \prec v' \prec w$.

Definition 2.7. *For a linear layout $\prec$ of a graph $G = (V, E)$, the* $\prec$-augmentation $G_\prec = (V, E_\prec)$ *is a spanning supergraph of G that contains all edges of G and all edges between a vertex and its immediate successor if not already existent in E, i. e.:*

$$E_\prec = E \cup \{\{u, v\} \mid u, v \in V \land v \text{ is the immediate successor of } v \text{ in } \prec\}\,.$$

Proposition 2.4. *The $\prec$-augmentation of a graph contains a Hamiltonian path.*

$\prec$-augmentations play an important role for deque layouts: Let $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ be a deque schedule of a graph $G = (V, E)$ and let $G_\prec = (V, E_\prec)$ be the $\prec$-augmentation of G. The *$\prec$-augmentation* $\Sigma_\prec = (\prec, E^{\mathtt{h}}_\prec, E^{\mathtt{t}}_\prec)$ of Σ is a deque schedule of $G_\prec$ which equals Σ for all edges in E, whereas all edges in $E_\prec \setminus E$ are processed as follows: Let $e = \{v, w\} \in E_\prec \setminus E$ be an edge where w is the immediate successor of v. At vertex v, edge e is inserted at the head after all edges in $E^{\mathtt{h}}(v)$ have been processed, i. e., e is last in $E^{\mathtt{h}}(v)$. At vertex w, edge e is removed at the head of the deque before all edges in $E^{\mathtt{h}}(w)$ are processed, i. e., e is first in $E^{\mathtt{h}}(w)$. In other words, e is a stack edge processed at the head. Note that $E^{\mathtt{t}}_\prec = E^{\mathtt{t}}$, i. e., $E^{\mathtt{t}}$ stays unchanged when deriving the $\prec$-augmentation; we use the superscript for uniformity. In fact, the choice of inserting and removing the edges in $E_\prec \setminus E$ at the head is arbitrary as processing all or only some of them at the tail equally works. The important property is that these edges are processed as stack edges. This canonical way of processing the edges in $E_\prec \setminus E$ as stack edges is always possible. In particular, we get:

Proposition 2.5. *A deque schedule $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ is a deque layout of a graph G if and only if the $\prec$-augmentation of Σ is a deque layout of $G_\prec$.*

By combining Propositions 2.4 and 2.5 and Thm. 2.1, we get:

Corollary 2.3. *A graph G is a deque (LC planar) graph if and only if it is a spanning subgraph of a graph $G_\prec$ which contains a Hamiltonian path and which is a deque (LC planar) graph.*

Proof. $\Rightarrow$: This follows from Propositions 2.4 and 2.5 and Thm. 2.1.

$\Leftarrow$: If $G_\prec$ is a deque (LC planar) graph, then so is any subgraph of $G_\prec$. In particular G is a deque (LC planar) graph. □

It is important to note that a deque graph G cannot be arbitrarily augmented to a deque graph with a Hamiltonian path because the Hamiltonian path must correspond to a linear layout of G that belongs to a deque layout.

Fig. 2.17 shows the LC embedding of the $\prec$-augmentation of the graph in Fig. 2.5(c). All newly introduced edges (dashed) are edges that enter their endpoints from above the front line as in the deque layout they are stack edges processed at the head.

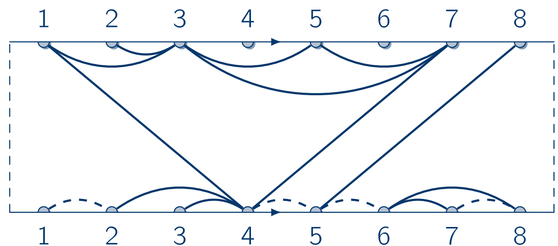


Figure 2.17: $\prec$-augmentation of the LC embedding in Fig. 2.5(c).

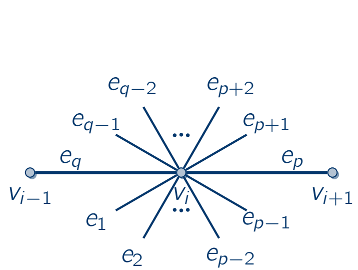
Next, we generalize Cor. 2.3. First remember that a deque graph is also planar (see Cor. 2.2). Thus, every deque graph is a spanning subgraph of a planar graph with a Hamiltonian path. With the help of the following lemma, we can also prove the converse:

Lemma 2.3. *A planar graph with a Hamiltonian path is a deque graph.*

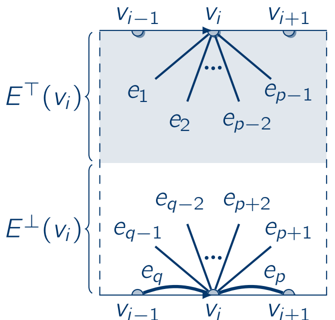
Proof. Let $G = (V, E)$ be an embedded graph with Hamiltonian path $P = (v_1, \dots, v_n)$. We assume that $n \geq 2$ as otherwise G consists of a single vertex which is always a deque graph. G is embedded and, thus, endowed with a planar rotation system $\mathcal{R}$. The idea of the proof is to derive a planar LC rotation system $\Lambda_P = (\prec, E^\perp, E^\top)$ from G's rotation system with the help of the Hamiltonian path. Remember that an LC rotation system induces a rotation system in the usual sense, i. e., a cyclic ordering of the incident edges at each vertex (see Sect. 2.2.1). We define Λ_P such that it induces the same rotation system as $\mathcal{R}$.

First, for the linear layout $\prec$, we choose the order of the vertices on P, i. e., $v_i \prec v_j$ if and only if $i < j$ for all $1 \leq i, j \leq n$. Second, let v_i be an inner vertex on P, i. e., $0 < i < n$, and let $e_1, \dots, e_{p-1}, e_p, e_{p+1}, \dots, e_q$ be the rotation system of v_i according to $\mathcal{R}$ with $1 \leq p < q$, where e_p is the edge to the immediate successor v_{i+1} of v_i and e_q is the edge to the immediate predecessor v_{i-1} of v_i on the Hamiltonian path. Fig. 2.18(a) illustrates the rotation system of v_i, where the edges e_p and e_q are drawn bold. The Hamiltonian path divides the rotation system of v_i into two halves, i. e., an upper and a lower half. We define $E^\perp(v_i) = (e_q, e_{q-1}, \dots, e_{p+1}, e_p)$ and $E^\top(v_i) = (e_1, e_2, \dots, e_{p-2}, e_{p-1})$. Note the reverse order in $E^\perp(v_i)$. The so obtained LC rotation system is illustrated in Fig. 2.18(b). As only one edge from P is incident to v_1 and v_n, their rotation system can be split at an arbitrary point. Here, we define the LC rotation system of v_1 (v_n) such that all edges enter v_1 (v_n) from above. That is, let $e_1, \dots, e_p$ be the rotation system of v_1 according to $\mathcal{R}$ with $p \geq 1$, where e_p is the edge to v_2. Then, $E^\perp(v_1) = (e_{p-1}, e_{p-2}, \dots, e_2, e_1, e_p)$ and $E^\top(v_1)$ is empty. Likewise, let $e_1, \dots, e_q$ with $q \geq 1$ be the rotation system of v_n, where e_q is the edge to v_{n-1}. Then, we define $E^\perp(v_n) = (e_q, e_{q-1}, \dots, e_2, e_1)$ and $E^\top(v_n)$ is empty.

In the following, we consider LC drawings of G that respect the constructed LC rotation system $\Lambda_P = (\prec, E^\perp, E^\top)$, i. e., the vertices are placed on the front line according to $\prec$ and the edge curves enter their endpoints according to $E^\perp$ and $E^\top$. The crucial observation is that by the way we constructed Λ_P, the Hamiltonian path is "aligned" with the front line and planarity ensures that no edge curve has as a point in common with the front line. In the remainder of the proof, we show that there is a plane LC drawing of G that respects Λ_P. For contradiction, assume that every such LC drawing has at least one crossing. Among those drawings, let Γ_P be one with a minimum number of crossings and, thereby, any crossing in Γ_P is inevitable. Let e and e' be two crossing edges, where $e = \{v_i, v_j\}$. There is a circle C formed by the



(a) Rotation system of an inner vertex v_i of a graph with a Hamiltonian path.



(b) From the rotation system in Fig. 2.18(a), we obtain an LC rotation system of v_i.

Figure 2.18: The rotation system of vertex v_i of a graph with a Hamiltonian path defines an LC rotation system of v_i.

path $v_i \rightsquigarrow v_j$ which is part of the Hamiltonian path and edge e. Circle C encloses a region R such that at least one inner point of the edge curve of e' lies within R and one inner point lies outside of R. Remember that G is endowed with a planar rotation system $\mathcal{R}$ and, hence, it has a plane drawing Γ in $\mathbb{R}^2$ which implies $\mathcal{R}$. By construction, the rotation system as induced by the LC drawing Γ_P is the same as $\mathcal{R}$. Thus, circle C encloses a region R' in Γ such that at least one inner point of the edge curve of e' must lie within R' and one inner point must lie outside, which causes a crossing in Γ by Jordan's curve theorem (Prop. 1.1). This contradicts the planarity of Γ. Hence, there is no crossing in Γ_P and so Λ_P is a planar LC rotation system. □

We arrive at the following characterization of deque graphs:

Theorem 2.2. *A graph is a deque graph if and only if it is a spanning subgraph of a planar graph with a Hamiltonian path.*

Proof. $\Rightarrow$: Follows from Corollaries 2.2 and 2.3.

$\Leftarrow$: Assume that G is a spanning subgraph of a planar graph G' with a Hamiltonian path. By Lem. 2.3, G' is a deque graph and so must be G. □

We extract another neat observation from the proof of Lem. 2.3: From the planar rotation system $\mathcal{R}$ of a graph G with a Hamiltonian path P, we obtain a planar LC rotation system $\Lambda_P = (\prec, E^\perp, E^\top)$, where $\prec$ is the order of the vertices on P and $E^\perp$ and $E^\top$ are defined according to the proof of Lem. 2.3 (see also Fig. 2.18). We generalize this idea: Let G be a graph endowed with a (not necessarily planar) rotation system $\mathcal{R}$ and assume that G contains a Hamiltonian path P. Let $\Lambda_P = (\prec, E^\perp, E^\top)$ be the LC rotation system obtained from $\mathcal{R}$ and P as in the proof of Lem. 2.3. We say that Λ_P is *induced by* P. Moreover, we obtain a deque schedule $\Sigma_P := \Lambda_P$ which we also call *induced by* P. By combining Lem. 2.3 and Thm. 2.1, we get:

Corollary 2.4. *Let G be a graph endowed with a rotation system and assume that G contains a Hamiltonian path P. Further, let Λ_P and Σ_P be the LC rotation system and deque schedule induced by P, respectively. The following statements are equivalent:*

(i) The rotation system of G is planar.

(ii) Λ_P is a planar LC rotation system.

(iii) Σ_P is a deque layout.

Remember that, by definition, a deque schedule is a deque layout if and only if `IsDeque-Layout` in Alg. 2.2 returns `true`. Hence, from Cor. 2.4 we obtain a straightfoward algorithm to test whether the rotation system of a graph with a Hamiltonian path is planar. All these observations will be particularly useful when we study graph layouts of planar graphs in the splittable deque in Sect. 2.5.2.

2.2.4.1 Comparing Two Stacks with a Single Deque

Before we turn our attention to other aspects of deque graphs, let us compare deque graphs with two-stack graphs with the help of Thm. 2.2: A deque can emulate two stacks by allowing only stack edges which implies that every two-stack graph is also a deque graph. In addition, queue edges are allowed in the deque. However, these queue edges cannot "move independently" of the stack edges as, for instance, no stack edge must be at the head when a queue edge is inserted at the same side. Now, the question is: What is the additional power these queue edges yield in comparison to two stacks? There is a short and a longer answer to this question. The short answer is given by the following (informal) equation:

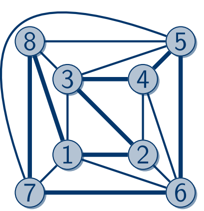
$$\frac{\text{deque layout}}{\text{two-stack layout}} = \frac{\text{planar \& Hamiltonian path}}{\text{planar \& Hamiltonian circle}}.$$

A graph is a two-stack graph if and only if it is the spanning subgraph of a planar graph with a Hamiltonian *circle* by [BK79]. We have found out that a graph is a deque graph if and only if it is the spanning subgraph of a planar graph with a Hamiltonian *path* (Thm. 2.2). In other words, the ability to process queue edges in addition to stack edges exactly corresponds to the difference between Hamiltonian paths and Hamiltonian circles in planar graphs. This observation, in turn, raises another question: How is this difference between Hamiltonian paths and circles reflected in deque and two-stack layouts? The (longer) answer is given in the following section.

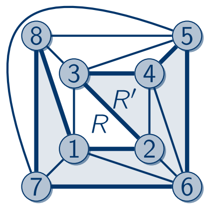
2.2.4.2 Duals of Deque Graphs

Consider the maximal planar graph depicted in Fig. 2.19(a) which contains the Hamiltonian circle $1, \ldots, 8, 1$. The edges of the Hamiltonian circle are drawn bold. By [BK79], this graph has a two-stack layout. In the following, we use the deque with no queue edges to emulate the two stacks and construct the corresponding layout as follows: The Hamiltonian circle defines a closed curve C that divides the plane into regions R and $R' := \mathbb{R}^2 \setminus R$ as shown in Fig. 2.19(b), where R' is shaded. Now, the inner part of each edge not on the Hamiltonian circle either lies completely within R or within R'. The idea is to identify R with the head and R' with the tail of the deque. The respective two-stack layout is shown in Fig. 2.19(c) as an LC drawing with no queue edges, where the linear layout is obtained by splitting the Hamiltonian circle between

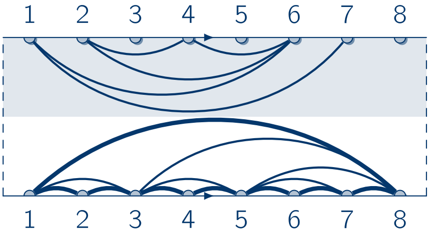
vertices 1 and 8. For instance, edge $(5, 7)$ lies within R and is, thus, a stack edge processed at the head, whereas edge $(1, 6)$ is in R' and processed at the tail. All edges on the Hamiltonian path are stack edges processed at the head, which is, again, an arbitrary decision as they can equally be processed as stack edges at the tail.



(a) A plane drawing of a graph with a Hamiltonian circle (edges drawn bold).



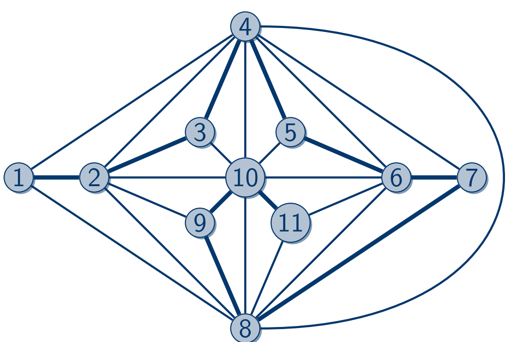
(b) The Hamiltonian circle partitions the drawing into two regions (shaded and white).



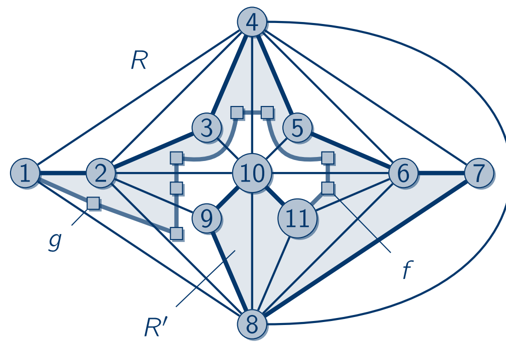
(c) In the two-stack layout, the edges in the white region in Fig. 2.19(b) are stack edges processed at the head and the ones in the shaded regions are processed at the tail.

Figure 2.19: A Hamiltonian circle in a plane drawing partitions the drawing into two regions, each corresponding to a stack in the two-stack layout.

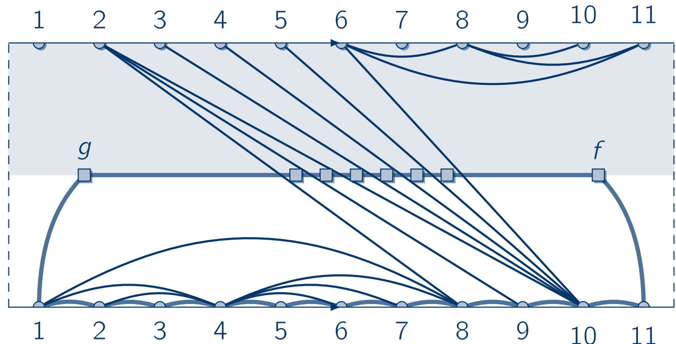
We can apply the same line of arguments to deque layouts, however, with an intriguing twist. Fig. 2.20(a) shows the Goldner-Harary graph G_{GH} [GH75] which is the smallest maximal planar graph with no Hamiltonian circle. The edges drawn bold belong to the Hamiltonian path $1, \ldots, 11$. Therefore, G_{GH} is no two-stack graph but a deque graph. To construct the deque layout, we use the same idea as before, i. e., we divide the plane into two regions and identify one region with the head and one region with the tail. For this, consider the (open) curve defined by the Hamiltonian path in Fig. 2.20(b). We close the curve by proceeding on the dual graph (see Sect. 1.1.8): First, we connect vertex 11 with face f to which 11 is incident. The choice of face f is arbitrary as long as vertex 11 is incident to f. Then, we find a path in the dual graph from face f to face g, where vertex 1 is incident to g, such that we never use any dual edge of an edge on the Hamiltonian path. Finally, we connect g to 1 and close the curve. Thereby, we obtain a closed and non-selfintersecting curve C that partitions the plane into regions R and R' (shaded). For the deque layout, we identify R with the head and R' with the tail, and the linear layout is the order of the vertices on the Hamiltonian path. The LC drawing corresponding to the deque layout is shown in Fig. 2.20(c). As with the two-stack graph from before, we have edge curves whose inner parts are completely within either R or R'. For instance, the inner part of $(1, 4)$ is in R and, indeed, is a stack edge processed at the head. Also as before, the edges on the Hamiltonian path are stack edges processed at the head. An interesting thing happens to the edges that cross C. For instance, edge $(3, 10)$ starts within R' and ends within R which means that it is inserted at the tail and removed at the head of the deque. In general, all queue edges cross C and they are exactly the primals of the edges on the path from f to g in the dual. For instance, in the LC drawing in Fig. 2.20(c), the path from f to g in the dual is opposite of the front line and it is crossed by all queue edges as they "change sides". We summarize these observations by the following theorem:



(a) A plane drawing of the Goldner-Harary graph [GH75] which contains no Hamiltonian circle but a Hamiltonian path (edges drawn bold).



(b) By connecting the endpoints 1 and 11 of the Hamiltonian path, we obtain a circle that partitions the drawing into two regions (shaded and white).



(c) In the deque layout, an edge e that enters its endpoint v within the white area in Fig. 2.20(b) is processed at v at the head of the deque; the shaded area corresponds to the tail. The edges of the Hamiltonian path can all be processed as stack edges at the head.

Figure 2.20: The endpoints of a Hamiltonian path in a plane drawing can be connected via a path in the dual and the resulting circle partitions the drawing into two regions. One of these region corresponds to the head and the other one corresponds to the tail of the deque.

Theorem 2.3. *Let $G = (V, E)$ be an embedded graph with Hamiltonian path $P = (v_1, \dots, v_n)$ and E_P be the edges on P. Let $G^* = (F, E^*)$ be the dual of G and denote by E_P^* the dual edges of E_P. Further, let $f_1, f_2 \in F$ be two faces such that v_1 is incident to f_p and v_n is incident to f_1. Then, the following two statements hold true:*

(i) There is a simple path $Q^* = f_1 \rightsquigarrow f_p$ in $G^* \setminus E_P^*$.

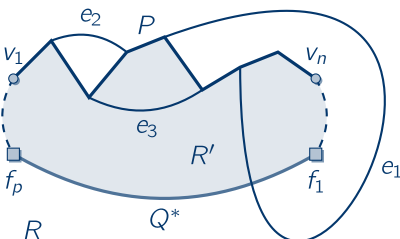
(ii) Let E_Q^* be the set of edges traversed by Q^* and let E_Q be the primal edges of E_Q^*. G has a deque layout $\Sigma = (\prec, E^{\mathrm{h}}, E^{\mathrm{t}})$ such that $\prec$ is the order of the vertices on P, and an edge $e \in E$ is a queue edge if and only if $e \in E_Q$.

Before we prove this theorem, recall the definition of cut and cut-set from Sect. 1.1.8: Let $G = (V, E)$ be an embedded graph with dual $G^* = (F, E^*)$. A cut partitions V into proper subsets $V_C \subsetneq V$ and $\overline{V}_C := V \setminus V_C$ and the set of edges $E_C := \{\{u, v\} \in E \mid u \in V_C \wedge v \in \overline{V}_C\}$ between V_C and $\overline{V}_C$ is the cut-set. Then, edges E_C^* form a simple circle in G^*, where E_C^* denotes the duals of E_C.

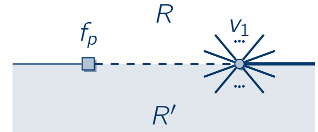
Proof. (i): Suppose for contradiction that there is no path $f_1 \rightsquigarrow f_p$ in $G^*_{\not P} := G^* \setminus E_P^*$, where the subscript reminds us that $G^*_{\not P}$ contains no dual edges from the Hamiltonian path. Since the dual G^* is connected, there is a path from f_1 to any other face f in G^*. In particular $f_1 \rightsquigarrow f_p$ in G^*. Hence, removing the dual edges of the Hamiltonian path disconnects f_1 from f_p. Denote by F_C the set of faces f for which there is a path $f_1 \rightsquigarrow f$ in $G^*_{\not P}$. By construction, $f_1 \in F_C$ and $f_p \in \overline{F}_C := F \setminus F_C$. Then, F_C and $\overline{F}_C$ is a cut with cut-set $E_C^* \subseteq E^*$. Let $e^* = \{f, g\}$ be an edge from E_C^* with $f \in F_C$ and $g \in \overline{F}_C$. Dual edge e^* must be in E_P^* as otherwise there would be a path from f_1 to g. Hence, $E_C^* \subseteq E_P^*$ and $E_C \subseteq E_P$, where E_C are the primal edges of E_C^*. As E_C^* is a cut-set, E_C forms a circle in G. However, then the Hamiltonian path contains a circle which is a contradiction. Therefore, there is a path from f_1 to f_p and also a simple path from f_1 to f_p.

(ii): In his seminal paper [Tut63], Tutte has shown that an embedded graph $G = (V, E)$ and its dual $G = (F, E^*)$ can be drawn simultaneously such that each of G and G^* is drawn plane, all faces are placed within the regions to which they correspond, and such that each primal edge e crosses its dual e^* exactly once. This is the drawing style of graphs and their duals that we usually take for granted (cf. Fig. 1.2(b) on page 8). Let Γ be such a simultaneous drawing of G and G^* in $\mathbb{R}^2$. The situation we obtain is sketched in Fig. 2.21(a). Denote by $\Gamma[P]$ the set of points in Γ of the simple curve defined by the Hamiltonian path P in G (drawn bold in Fig. 2.21(a)). By (i), we know that there is a simple path $Q^* = f_1 \rightsquigarrow f_p$ in G^* (shaded line between f_1 and f_p). $\Gamma[Q^*]$ denotes the set of points in Γ that belong to Q^*. Since Q^* contains no edge of E_P^*, $\Gamma[P]$ and $\Gamma[Q^*]$ are two disjoint simple curves. We connect these two curves as follows: Let $\Gamma(v_1)$ be the position of v_1 and $\Gamma(f_p)$ be the position of f_p. Since v_1 is incident to f_p, there is a continuous, simple, and open curve from v_1 to f_p that shares no point with Γ except for its endpoints (dashed line between v_1 and f_p). Likewise, there is a continuous, simple, and open curve from v_n to f_1 (dashed line between v_n and f_1). Putting all things together, we obtain the simple and closed curve C that partitions $\mathbb{R}^2$ into regions R and R' (shaded).

Let $\Lambda = (\prec, E^{\perp}, E^{\top})$ be the LC rotation system induced by the Hamiltonian path. Remember that the Hamiltonian path P splits the rotation system of each vertex v into the edges that enter v from above and those that enter v from below. Also remember that the rotation systems of v_1 and v_n can be split arbitrarily as only one edge of P is incident to each of them.



(a) Connecting the endpoints of the Hamiltonian path P and Q^* encloses the region R'.



(b) The rotation system of v_1 is split by the Hamiltonian path and by the curve connecting v_1 with f_1.

Figure 2.21: Situation obtained in the proof of Thm. 2.3.

In the construction of C, we have connected v_1 with f_p by a curve which can be seen as an edge curve. Together with the edge to vertex v_2, the rotation system of v_1 is split as shown in Fig. 2.21(b), where the curve between v_1 and f_p is dashed. The edges that enter v_1 within R (R'), enter v_1 from above (below) in the LC rotation system Λ. Likewise, the rotation system of v_n is split. The so obtained LC rotation system is planar as shown in the proof of Lem. 2.3 and the corresponding deque schedule $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ is a deque layout by Cor. 2.4.

Now, let e^* be an edge on Q^* and let $e = \{u, v\}$ be its primal. By construction, the edge curve of e crosses C exactly once, e. g., e_1 in Fig. 2.21(a). This implies that e either enters u within R and v within R' or vice versa. Hence, e enters u from above and v from below or vice versa. In both cases, e is a queue edge in Σ. Further, let e be an edge such that the inner part of its edge curve lies completely within R, e. g., edge e_2. Then, e enters both its endpoints from above and is, thus, a stack edge inserted at the head in Σ. Likewise, an edge that lies within R' is a stack edge inserted at the tail in Σ. This proves (ii). □

In conclusion, the additional power a deque has in comparison to two stacks is to connect the endpoints of the Hamiltonian path via the dual, which in turn produces queue edges. We will use this insight in Sect. 2.4 and discuss its implications on queue graphs.

2.2.5 Deciding whether a Graph is a Deque Graph is $\mathcal{NP}$-Complete

We now turn to computational complexity issues. How hard is it to decide whether a given graph is a deque graph? The problem of deciding whether a maximal planar graph has a Hamiltonian circle is $\mathcal{NP}$-complete [Chv85, Wig82]. Consequently, the decision problem "Is a given graph a two-stack graph?" is $\mathcal{NP}$-complete. Also, deciding whether a graph is a queue graph is $\mathcal{NP}$-complete [HR92]. In contrast, deciding whether a graph permits a one-stack layout can be done in linear time as it is equivalent to deciding whether the graph is outerplanar [BK79, Wie87]. For the deque, the respective decision problem is also $\mathcal{NP}$-complete. To show this, we prove the $\mathcal{NP}$-completeness of the following problem:

Theorem 2.4. *The decision problem "Given a maximal planar graph, does it contain a Hamiltonian path?" is $\mathcal{NP}$-complete.*

In [Wig82], Wigderson uses an elegant reduction from the $\mathcal{NP}$-complete Hamiltonian circle problem for triconnected, cubic planar graphs [GJ90] to prove that the same problem is also $\mathcal{NP}$-complete for maximal planar graphs. Our proof of Thm. 2.4 heavily relies on this reduction: Given a triconnected, cubic planar graph G, we apply the reduction from [Wig82] to obtain a maximal planar graph G'. By [Wig82], G' contains a Hamiltonian circle if and only if G does. In the reduction, the vertices of G are replaced by certain widgets. We locally modify one of these widgets in G' to obtain another graph G''. Our construction is such that G'' definitely has no Hamiltonian circle, however, it has a Hamiltonian path if and only if G' has a Hamiltonian circle. Before we delve into the technicalities of the proof, it should be noted that we only introduce those widgets from [Wig82] that we need in our construction. For all widgets and proofs of their properties, the reader is advised to consult [Wig82].

Proof. The widget of *type A* is the base building block of the construction in [Wig82] and it is depicted in Fig. 2.22(a) at the top. Wigderson has shown that if a Hamiltonian circle enters and leaves, i. e., *traverses*, a type-A widget A, then all vertices of A must be visited before A can be left. He shows this by enumerating all possibilities of traversing a type-A widget, where all possibilities use edge e^* (bold in Fig. 2.22(a)); a fact which we will use later in our construction. In the following, a type-A widget is symbolized by a shaded triangle with contains a line (see bottom of Fig. 2.22(a)).

The *type-B* widget, also from [Wig82], shown in Fig. 2.22(b), is composed of two type-A widgets A_1 and A_2. In A_1, the vertices b_1 and b_3 are identified with the vertices a_1 and a_3 of the type-A widget, respectively, and, in A_2, the vertices b_2 and b_3 are identified with a_1 and a_3, respectively. Wigderson has shown that a Hamiltonian circle traversing a type-B widget must enter and leave at vertices b_1 and b_2 and, again, all vertices in A_1 and A_2 have to be visited during this traversal. Hence, it is not possible to visit, for instance, b_3 and then visit the remaining vertices of the widget later. A type-B widget is symbolized by a shaded triangle, where the corners symbolizing b_1 and b_2 are connected by an arc.

We additionally introduce type-C and -D widgets. A *type-C* widget is depicted in Fig. 2.22(c) and it consists of three type-B widgets. A type-C widget C has the property that any Hamiltonian path in a graph that contains C must have at least one of its end points in C, i. e., C cannot be traversed completely: Suppose that a Hamiltonian path enters at c_1, then it must visit all vertices of B_1 until it reaches *center vertex* c. Then either B_2 or B_3 have to be traversed entirely. W. l. o. g. it traverses B_2 and reaches c_2. Still, the vertices of B_3 have to be visited. The only possibility (by the properties of type-A and -B widgets) of doing this is to reenter C at c_3 and to visit all vertices of B_3 just before center vertex c is reached. At the vertex that is adjacent to c, the Hamiltonian path ends. The same holds if the Hamiltonian path enters at c_2 or c_3. A type-C widget is denoted by a triangle with a dot symbolizing the end point of a Hamiltonian path.

Two type-C widgets and one type-B widget are arranged to a *type-D* widget (Fig. 2.22(d)). We show that both ends of a Hamiltonian path must lie within a type-D widget. The relevant parts of the widget are C_1, C_2, and B and the solid edges. We additionally need to introduce the shaded, dashed edges to obtain a maximal planar graph, where all faces are triangles. This triangulation step can be done arbitrarily as long as the "solid backbone" of the type-D widgets stays as shown in Fig. 2.22(d). Let D be a type-D widget. A Hamiltonian path of a graph containing D must have its one end in C_1 and its other end in C_2. The type-B widget B ensures that any Hamiltonian path enters and leaves D at d_1 and d_2: Since any Hamiltonian path has to end in C_1 and C_2, B has to be traversed by a Hamiltonian path, i. e., no endpoint

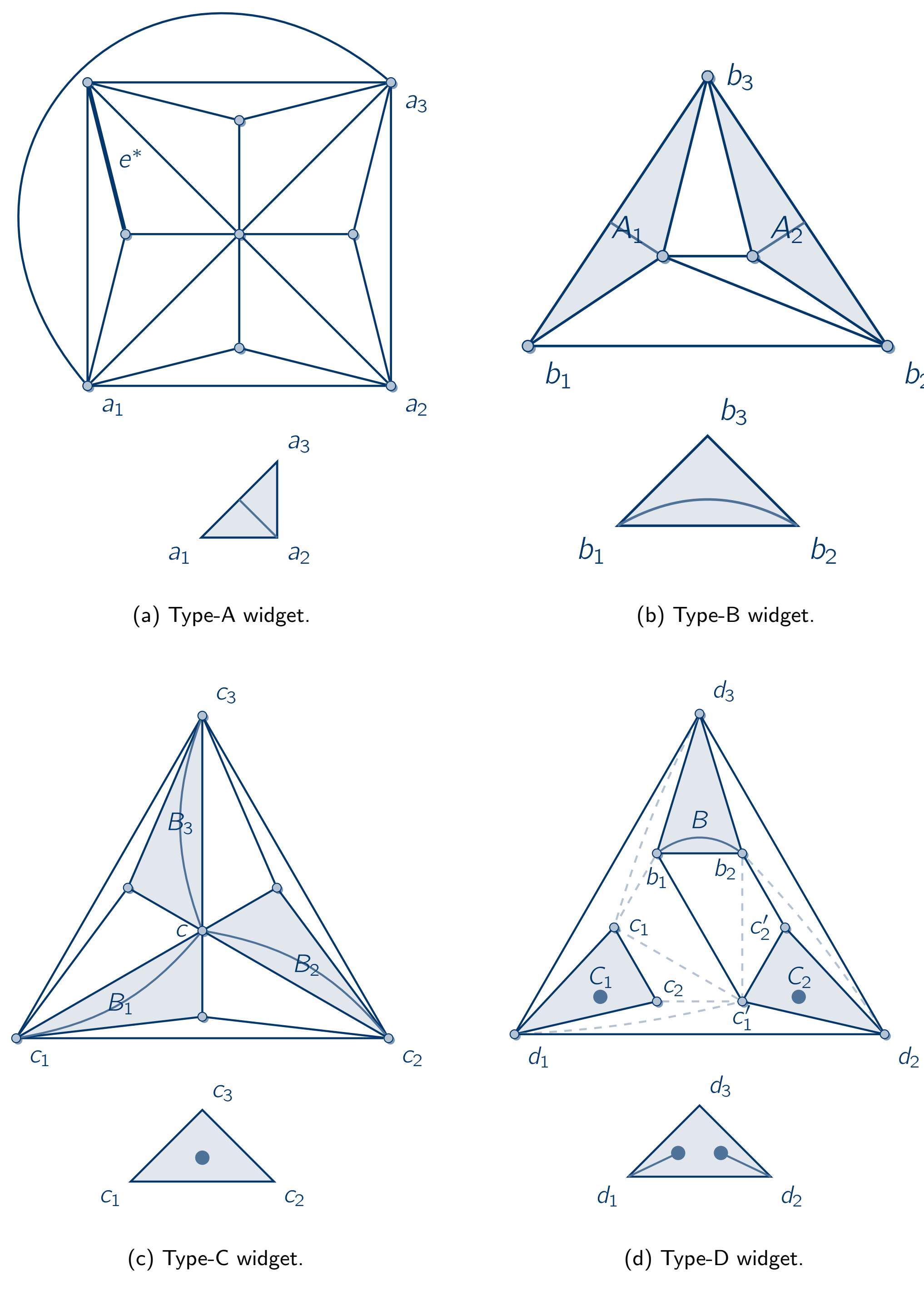


Figure 2.22: Widgets needed for the $\mathcal{NP}$-completeness reduction in Thm. 2.4.

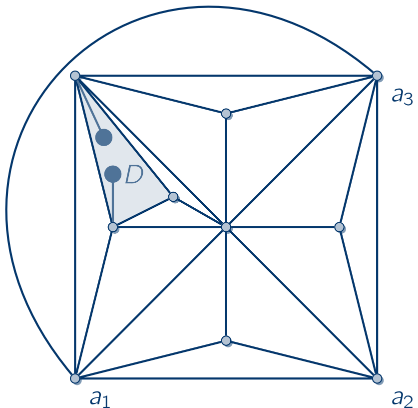


Figure 2.23: Type-A widget with e^* replaced by a type-D widget.

can lie within B. By the property of type-B widgets, the Hamiltonian path has to enter and leave B at b_1 and b_2, and during the traversal d_3 is visited. Hence, D cannot be entered or left via vertex d_3. The symbol for a type-D widget (bottom of Fig. 2.22(d)) is a shaded triangle with two pins connected to d_1 and d_2 that indicate that a Hamiltonian path must have both its endpoints within the widget.

Let G be a triconnected, cubic graph. The reduction in [Wig82] replaces each vertex of G by a widget composed of three type-B widgets, where each consists of two type-A widgets. Afterwards, edges are adequately introduced to obtain the maximal planar graph G'. Let A be an arbitrary type-A widget in G'. We insert a type-D widget D into A as displayed in Fig. 2.23: Edge e^* is replaced by D where the endpoints of e^* are the vertices d_1 and d_2 of D and vertex d_3 is connected to the central vertex of A. We denote the so obtained graph by G''. Remember that e^* is the edge that always has to be used when traversing a type-A widget.

Suppose that G contains a Hamiltonian circle. Then, by the construction in [Wig82], G' also contains a Hamiltonian circle. This Hamiltonian circle must use edge e^* in the type-A widget A of G'. In G'' and the corresponding type-D widget (cf. Figs. 2.22(c) and 2.22(d)), there is a path p that visits the vertices d_2, c_2', b_2, b_1, c_1', and the vertex adjacent to the center vertex of C_2 in order such that p visits all vertices of C_2 and B exactly once and ends within C_2. Similarly, there is a path p' that visits d_1, c_1, c_2, and the vertex adjacent to center vertex of C_1 in order such that p' visits all vertices of C_1 and ends within C_1. In G'', we split the Hamiltonian circle at e^* and obtain a Hamiltonian path which ends in C_1 and C_2, and visits all vertices of D exactly once.

Conversely, assume that G'' contains a Hamiltonian path. Then, by construction, the Hamiltonian path must end in C_1 and C_2. Remember that the Hamiltonian path must leave D at d_1 and d_2. Thus, there is a Hamiltonian path in G' with endpoints d_1 and d_2. By adding edge e^* to the Hamiltonian path we get a Hamiltonian circle. Altogether, G' contains a Hamiltonian circle and so does G by [Wig82].

We can conclude that finding a Hamiltonian path in a maximal planar graph is $\mathcal{NP}$-hard. The problem is also in $\mathcal{NP}$: First, we non-deterministically guess a permutation $v_1, \ldots, v_n$ of the vertices and check if v_i and v_{i+1} are adjacent for all $1 \leq i < n$. □

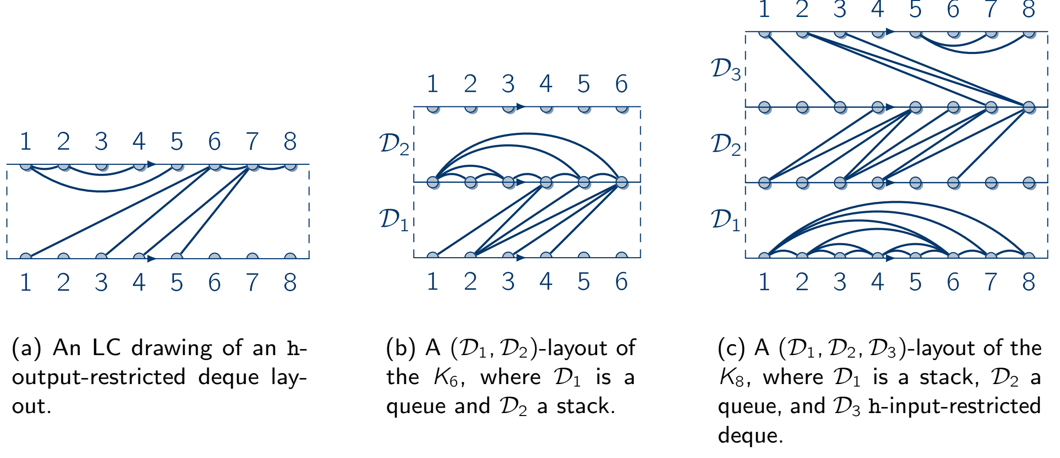
Theorem 2.5. *The decision problem "Is a given graph a deque graph?" is $\mathcal{NP}$-complete.*

Proof. First, we non-deterministically guess a deque schedule $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ for the given graph $G = (V, E)$. This takes $\mathcal{O}(|V| + |E|)$ time steps. Then, we use `IsDequeLayout` in Alg. 2.2, which runs in time $\mathcal{O}(|V| + |E|)$, to test whether Σ is a deque layout. Hence, the decision problem is in $\mathcal{NP}$.

For the $\mathcal{NP}$-hardness, let $G = (V, E)$ be a maximal planar graph. G is a deque graph if and only if it contains a Hamiltonian path by Thm. 2.2. Thus, the decision problem "Does a maximal planar graph contain a Hamiltonian path?" reduces to "Is a graph a deque graph?", which is then also $\mathcal{NP}$-hard by Thm. 2.4. □

2.3 Linear Cylindric Drawings of Deque-Reducible Data Structure and Mixed Layouts

LC drawings also help to better understand graph layouts using data structures that support only a limited set of the deque operations. We call a data structure *deque-reducible* if it supports a "sensible" subset of the deque operations. "Sensible" means that at least one insertion and one removal operation is possible. For instance, by disallowing queue edges, we obtain two stacks (in a single data structure). Likewise, if only queue edges inserted at the head (tail) are allowed, we obtain a queue. Forbidding that edges are inserted at the head (tail) leads to an `h`-*input-restricted* (`t`-*input-restricted) deque*, and, analogously, forbidding edges removed at the head (tail) leads to an `h`-*output-restricted* (`t`-*output-restricted) deque*.



(a) An LC drawing of an `h`-output-restricted deque layout.

(b) A $(\mathcal{D}_1, \mathcal{D}_2)$-layout of the K_6, where $\mathcal{D}_1$ is a queue and $\mathcal{D}_2$ a stack.

(c) A $(\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3)$-layout of the K_8, where $\mathcal{D}_1$ is a stack, $\mathcal{D}_2$ a queue, and $\mathcal{D}_3$ `h`-input-restricted deque.

Figure 2.24: LC drawings layouts in deque-reducible data structure and of mixed layouts.

For any deque-reducible data structure $\mathcal{D}$, we call a deque schedule $\Sigma_{\mathcal{D}} = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ *$\mathcal{D}$-schedule* if $\Sigma_{\mathcal{D}}$ only uses the operations allowed by $\mathcal{D}$. If $\Sigma_{\mathcal{D}}$ is additionally a deque layout, it is called *$\mathcal{D}$-layout*, and a graph that has a $\mathcal{D}$-layout is called *$\mathcal{D}$-graph*. We generalize Lem. 2.1 to deque-reducible data structures:

Corollary 2.5. *Let $\mathcal{D}$ be a deque-reducible data structure and $\Sigma_{\mathcal{D}}$ be a $\mathcal{D}$-schedule with corresponding LC rotation system $\Lambda_{\mathcal{D}} := \Sigma_{\mathcal{D}}$. $\Sigma_{\mathcal{D}}$ is a $\mathcal{D}$-layout if and only if $\Lambda_{\mathcal{D}}$ is planar.*

Fig. 2.24(a) shows a plane LC drawing that corresponds to an $\mathtt{h}$-output-restricted deque layout. There, no edge (u, v) enters v from above the front line as no edge is removed at the head of the deque. Note that the LC drawing can be flipped horizontally and vertically without destroying planarity. The first corresponds to reversing the linear layout, the latter to swapping the roles of head and tail; and with both operations the rotation systems are inverted (cf. Sect. 1.1.7). From this observation, we obtain:

Corollary 2.6. *For any graph G, the following statements are equivalent:*

(i) G is an $\mathtt{h}$-input-restricted deque graph.

(ii) G is a $\mathtt{t}$-input-restricted deque graph.

(iii) G is an $\mathtt{h}$-output-restricted deque graph.

(iv) G is a $\mathtt{t}$-output-restricted deque graph.

Suppose we are given a set $\mathcal{D}_1, \dots, \mathcal{D}_k$ of $k \geq 1$ deque-reducible data structures and a graph $G = (V, E)$. A partition of the edges $E = E_1 \dot\cup E_2 \dot\cup \dots \dot\cup E_k$ together with a sequence $\Sigma_i = (\prec, E_i^{\mathtt{h}}, E_i^{\mathtt{t}})$ for $i = 1, \dots, k$, where Σ_i is a $\mathcal{D}_i$-schedule of $G = (V, E_i)$, is called $(\mathcal{D}_1, \dots, \mathcal{D}_k)$*-layout* or, generally, *mixed layout*, if each Σ_i is a $\mathcal{D}_i$-layout. In other words, G is partitioned into k layers such that layer i allows for a $\mathcal{D}_i$-layout. Note that we assume the same linear layout for all Σ_i.

Corollary 2.7. *Let $\mathcal{D}_1, \dots, \mathcal{D}_k$ be $k \geq 1$ deque-reducible data structures and let $G = (V, E)$ be a graph. $E = E_1 \dot\cup \dots \dot\cup E_k$ together with $\Sigma_1, \dots, \Sigma_k$ is a $(\mathcal{D}_1, \dots, \mathcal{D}_k)$-layout of G if and only if each LC rotation system $\Lambda_i := \Sigma_i$ is planar.*

Hence, a graph has a $(\mathcal{D}_1, \dots, \mathcal{D}_k)$-layout if and only if we can partition its edge set such that each resulting subgraph has a plane LC drawing with the respective restrictions. We can conveniently draw such a mixed layout by stacking the fundamental polygon representations of the corresponding LC drawings on top of each other. Fig. 2.24(b) shows two plane LC drawings, where the lower corresponds to a queue layout ($\mathcal{D}_1$) and the upper to a stack layout ($\mathcal{D}_2$). The displayed graph is the complete graph K_6 with six vertices, where each edge is either processed in $\mathcal{D}_1$ or in $\mathcal{D}_2$. As this drawing is plane, we can conclude that the K_6 has a stack-queue layout. Another example is the $(\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3)$-layout of the complete graph K_8 with eight vertices in Fig. 2.24(c), where $\mathcal{D}_1$ is a stack, $\mathcal{D}_2$ a queue and $\mathcal{D}_3$ an $\mathtt{h}$-input-restricted deque.

2.4 Queue Graphs

In this section, we apply our findings from Sect. 2.2 to queue graphs. As discussed in Sect. 2.3, the queue can be seen as a restricted deque, i. e., only queue edges inserted at a particular side, say, the head are allowed.

Definition 2.8 (Queue Schedule, Layout and Graph). *A deque schedule $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ is called* queue schedule *if for all edges $e = (u, v)$:*

$$e \in E^{\mathtt{h}}(u) \wedge e \in E^{\mathtt{t}}(v)\,.$$

We call a queue schedule queue layout *if* `IsDequeLayout` *in Alg. 2.2 returns* `true`*, i. e., it is a deque layout. A graph is called* queue graph *if it has a queue layout.*

All of the results in this section equally apply to the symmetric case of the queue where the edges are inserted at the tail and removed at the head. Recall, a deque schedule is a deque layout if and only if the corresponding LC rotation system is planar (Lem. 2.1). From this, we obtain the following two results:

Corollary 2.8. *A queue schedule Σ is a queue layout if and only if the corresponding LC rotation system $\Lambda := \Sigma$ is planar.*

Corollary 2.9. *A graph is a queue graph if and only if it has a plane LC drawing such that all edges $e = (u, v)$ enter u from above and v from the below the front line.*

We start with a comparison of different ways to visualize queue layouts.

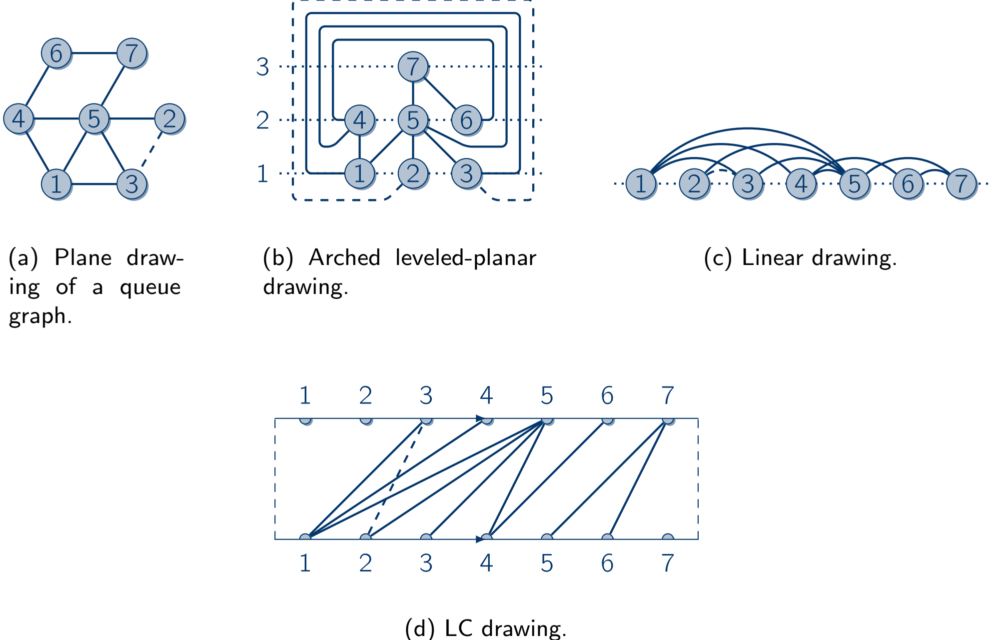
2.4.1 Linear Cylindric Drawings of Queue Graphs

Assume we are given a linear layout $\prec$ of a graph $G = (V, E)$ and we want to find out if G has a queue layout according to $\prec$, i.e., there is a queue layout $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ for some $E^{\mathtt{h}}$ and $E^{\mathtt{t}}$. Note that, given the linear layout, the order in which the edges must be inserted and removed at each vertex v, i.e., the total orders of $E^{\mathtt{h}}(v)$ and $E^{\mathtt{t}}(v)$, is uniquely determined in a queue layout. For instance, if $e_1 = (u, v_1)$ and $e_2 = (u, v_2)$ are inserted at u with $v_1 \prec v_2$, then e_1 must be inserted before e_2.

In the following, we consider the graph displayed in Fig. 2.25(a) where the vertices are labeled according to the linear layout. To give away the answer, the linear layout allows for a queue layout if we remove the dashed edge. However, this is not directly deducible from Fig. 2.25(a). To see that the dashed edge is the culprit, we could, in principle, try to process the edges in the queue and see if this works; a rather tedious and mechanical, not to say, error-prone task. Or, we follow the spirit of this section and appropriately draw the graph to see if the linear layout works for the queue layout.

Heath and Rosenberg have characterized queue graphs as the arched leveled-planar graphs [HR92]. These graphs permit an arched leveled-plane drawing as the one shown in Fig. 2.25(b), where we ignore the dashed edge for the moment. There, the vertices are placed on horizontal levels, numbered from 1 to 3 in the example. The order of the vertices on each level and from bottom to top is the linear layout. Edge curves are only allowed between adjacent levels, e.g., edge $(1, 4)$, or between the leftmost vertex on a level and vertices to the right on the same level, e.g., edge $(4, 6)$. The latter edge curves are called arches. Note that all edges that are drawn solid in Fig. 2.25(b) respect these rules. The dashed edge connects vertex 2 in the middle of level 1 with vertex 3 which is rightmost on the same level and this is not allowed. Though, introducing an invalid arch, an edge that overleaps a level, or an edge that causes a crossing in an arched leveled-planar graph does not necessarily imply that the linear layout does not allow for a queue layout. For instance, by redistributing the vertices among the levels, a valid arched leveled-plane drawing is in principle still possible. This, however, is not directly visible in Fig. 2.25(b).

A drawing where it is, at least in principle, directly possible to see whether an edge destroys a queue layout is a linear drawing as shown in Fig. 2.25(c). As with LC drawings, the vertices are placed on a horizontal line (dotted) in order of the linear layout and all edge curves enter their endpoints from above this line. For a queue layout, no two edges must properly nest as those cannot be processed in the queue. Generally, two edges $e = (u, v)$ and $e' = (u', v')$ properly nest if $u \prec u' \prec v' \prec v$. For instance, edge $(2, 3)$ is properly nested in $(1, 4)$ and, thus,



(a) Plane drawing of a queue graph.

(b) Arched leveled-planar drawing.

(c) Linear drawing.

(d) LC drawing.

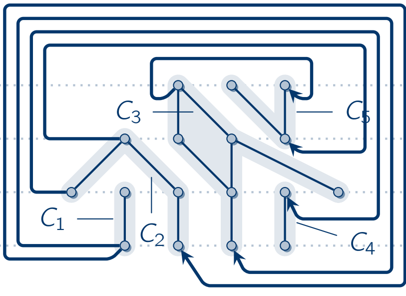
Figure 2.25: Several ways of displaying the same queue graph.

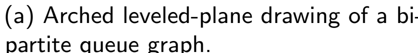
$(2, 3)$ destroys the queue layout. A linear drawing has the drawback of many edge crossings: In general, two edges (u, v) and (u', v') must cross if and only if $u \prec u' \prec v \prec v'$, which is called *twist* [HR92]. A twist reflects first in, first out principle of the queue and, thus, a linear drawing of a queue layout intrinsically must contain many crossings. This makes it a cumbersome task to find properly nesting edges that destroy the queue layout.

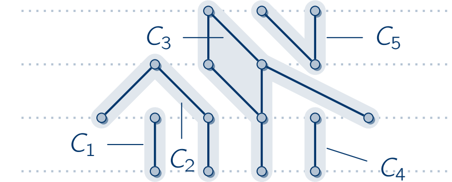
The spirit of this chapter is that the ability to layout a graph in a data structure corresponds to planarity. By applying Cor. 2.8, we obtain the LC drawing in Fig. 2.25(d). Neither of the solid edges cross and, consequently, these edges can be processed in the queue. In contrast, the dashed edge $(2, 3)$ crosses edges $(1, 4)$ and $(1, 5)$, and we have found our culprits. As two edges cannot be processed in the queue if and only if they cross, we can conclude that removing either edge $(2, 3)$, or both edges $(1, 4)$ and $(1, 5)$ yields a plane LC drawing and, thus, a queue layout. Note that this information is also not directly deducible from the arched leveled-plane drawing in Fig. 2.25(b). An LC drawing, thus, concisely incorporates all information to decide whether a linear layout allows for a queue layout.

2.4.2 Proper Leveled Planar Graphs and Bipartite Queue Graphs

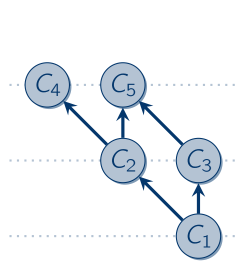
In [AG11], we have studied the relationship of queue graphs with a close relative of arched leveled-planar graphs, namely, the proper leveled-planar graphs. A graph is proper leveled-planar if it has an arched leveled-plane drawing without arches. Hence, Heath and Rosenberg called the arched leveled-planar graphs also "almost [proper] leveled-planar" [HR92]. Proper leveled-planar graphs naturally arise and play an important role in graph drawing [DBN88, STT81]. An example of a proper leveled-plane drawing is shown in Fig. 2.26(d).



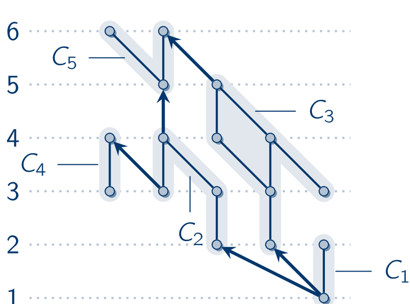

(a) Arched leveled-plane drawing of a bipartite queue graph.



(b) Removing the arches yields a proper leveled-plane drawing.



(c) The connected components.



(d) Resulting proper leveled-plane drawing.

Figure 2.26: Construction to obtain a proper leveled-plane drawing from an arched leveled-plane drawing of a bipartite graph

A proper leveled-planar graph is always bipartite and, thereby, every proper leveled-planar graph is a bipartite queue graph. In fact, the converse is also true as we have shown in [AG11]. The idea of the proof is as follows: Let $G = (V, E)$ be a bipartite, connected queue graph with the arched leveled-plane drawing shown in Fig. 2.26(a). We assume that the arches are directed away from the leftmost vertex. First, we remove all arches (Fig. 2.26(b)) which partitions G into (connected) components C_1, C_2, C_3, C_4, and C_5 (on shaded background). Each component is drawn proper leveled-plane and we reuse these drawings in our construction. The idea is to rearrange the relative positions of the components such that the arches point from one level to an adjacent level. This, however, is only possible if there is no arch with both endpoints in the same component. Note that such an arch, which connects two vertices on the same level, would imply a circle of odd length in G which is a contradiction to G being bipartite. Next we obtain a proper leveled-plane drawing of the component graph (Fig. 2.26(c)), where the vertices are the components interconnected by the arches. In [AG11], we have shown that such a drawing always exists and we use it to obtain the desired relative positions of the components: We start with C_1 and place its lowest vertex rightmost and on level 1 (cf. Fig. 2.26(d)). The lowest vertices of C_2 and C_3 are positioned on level 2 in order and sufficiently wide apart from C_1 and from each other to ensure planarity. Then, the arches from C_1 to C_2 and C_3 point

from level 1 to 2 and cause no crossings. With the help of Fig. 2.26(c), we proceed in this manner to obtain a proper leveled-plane drawing of G.

Proposition 2.6 ([AG11]). *A graph is proper leveled-planar if and only if it is a bipartite queue graph.*

Heath and Rosenberg have also shown that deciding whether a graph is a proper leveled-planar graph is $\mathcal{NP}$-complete [HR92].

Proposition 2.7 ([AG11]). *Deciding whether a graph is a queue graph is $\mathcal{NP}$-complete even for bipartite graphs.*

2.4.3 Duals of Queue Graphs

In Sect. 2.2.4.2, we have studied the duals of $\prec$-augmented deque graphs, i. e., planar graphs with a Hamiltonian path according to the linear layout $\prec$. We have found out that if we connect the endpoints of the Hamiltonian path via its dual, we "cross" exactly the queue edges (Thm. 2.3). By applying these ideas to queue graphs, it turns out that $\prec$-augmented queue graphs are "selfdual", i. e., their duals are also $\prec$-augmented queue graphs. In the following, we use a slightly modified version of Def. 2.7.

Definition 2.9. *Given a linear layout $\prec$ and a graph $G = (V, E_Q)$, the* exuberant $\prec$-augmentation *$G_\prec = (V, E_Q \dot\cup E_P)$ of G is a multigraph that is obtained by introducing edges E_P with:*

$$E_P := \{\{u, v\} \mid u, v \in V \wedge v \text{ is the immediate successor of } u \text{ in } \prec\}.$$

Note that the edge set of $G_\prec$ is the disjoint union of E_Q and E_P, i. e., we introduce an edge between a vertex and its immediate successor regardless of whether it is already in E_Q. Hence, in contrast to $\prec$-augmentations, the exuberant $\prec$-augmentation is a multigraph. Since we only deal with exuberant $\prec$-augmentations in the following, we simply speak of $\prec$-augmentations for convenience. Also for convenience and as we deal with queue graphs, we say that E_P is the Hamiltonian path and E_Q are the queue edges of $G_\prec$. By Thm. 2.3, in a $\prec$-augmentation of a deque graph, we can connect the endpoints of the resulting Hamiltonian path via the dual and, thereby, cross exactly the queue edges. In a queue graph $G = (V, E_Q)$, all edges are queue edges, hence, in its $\prec$-augmentation, all edges E_Q are crossed when connecting the endpoints of the Hamiltonian path. We specialize Thm. 2.3 for queue graphs as follows.

Lemma 2.4. *Let $G = (V, E_Q)$ be a queue graph with queue layout $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ and $\prec$-augmentation $G_\prec = (V, E_Q \dot\cup E_P)$. Denote by v_1 and v_n the first and last vertex of the Hamiltonian path E_P, respectively. Then, $G_\prec$ has an embedding with dual $G^*_\prec = (F_\prec, E^*_Q \dot\cup E^*_P)$ with the following properties: There exist faces $f_1, f_p \in F_\prec$, where v_n is incident to f_1 and v_1 is incident to f_p, such that there is a simple path $f_1 \rightsquigarrow f_p$ in $G^*_\prec \setminus E^*_P$ that contains all edges E^*_Q.*

Proof. Let $\Sigma_\prec$ be the $\prec$-augmentation of Σ, i. e., $\Sigma_\prec$ is a deque schedule of $G_\prec$ that equals Σ for all edges in E_Q and all edges in E_P are stack edges processed at the head. Let $\Lambda_\prec = (\prec, E^\perp, E^\top)$ be the LC rotation system corresponding to $\Sigma_\prec$. By Lem. 2.1, we know that $\Lambda_\prec$ is planar, hence, we obtain an embedding of $G_\prec$ with dual $G^*_\prec = (F_\prec, E^*_Q \dot\cup E^*_P)$. Further, $G_\prec$ contains a Hamiltonian path E_P with first vertex v_1 and last vertex v_n. We apply Thm. 2.3, where we choose f_1 and f_p such that all edges in E_Q that are incident to v_1 enter v_1

from above and all edges in E_Q incident to v_n enter v_n from below. Hence, there is a simple path $Q^* = f_1 \rightsquigarrow f_p$ in $G^*_\prec \setminus E^*_P$ that contains each edge in E^*_Q. □

Given a $\prec$-augmentation $G_\prec = (V, E_Q \dot\cup E_P)$ of a queue graph, we say that an embedding of $G_\prec$ is an *augmented queue embedding* (with Hamiltonian path E_P and queue edges E_Q) if it fulfills the properties of Lem. 2.4. Note that we can construct an augmented queue embedding by following the proof of Lem. 2.4.

Corollary 2.10. *A graph $G = (V, E_Q)$ is a queue graph if and only if it is a spanning subgraph of a $\prec$-augmentation $G_\prec = (V, E_Q \dot\cup E_P)$ such that $G_\prec$ has an augmented queue embedding.*

Proof. $\Rightarrow$: Follows from Lem. 2.4.

$\Leftarrow$: The augmented queue embedding of $G_\prec$ and the Hamiltonian path induce a deque layout $\Sigma_\prec = (\prec, E^{\mathrm{h}}, E^{\mathrm{t}})$ where all edges in E_P are stack and all edges in E_Q are queue edges. Removing all edges E_P from $G_\prec$ yields G, which is then a queue graph. □

Figs. 2.27 and 2.28 show the augmented queue embedding of the $\prec$-augmentation $G_\prec = (V, E_Q \dot\cup E_P)$ (solid) and its dual $G^*_\prec = (F, E^*_Q \dot\cup E^*_P)$ (dashed) of the queue graph in Fig. 2.25(d) on page 55 (without the dashed edge). Note that all edges of E_P enter their endpoints from above, whereas all edges in E_Q change sides. Also observe that $G_\prec$ is a multigraph as it contains the edge $(6, 7)$ twice. Interestingly, the dual has the same structure as the primal: The duals of E_Q, denoted by E^*_Q, are a Hamiltonian path $Q^* = f_1, \dots, f_{10}$ in $G^*_\prec$. Let $\prec^*$ be the linear layout of $G^*_\prec$ in order of Q^*, e. g., $f_i \prec^* f_j \Leftrightarrow i < j$ for all $1 \le i, j \le 10$ in Figs. 2.27 and 2.28. The duals of E_P, denoted by E^*_P, behave like queue edges: For instance, the dual of $(4, 5) \in E_P$ is edge (f_3, f_8) which enters f_3 from below and f_8 from above. Thus, the embedding of $G^*_\prec$ is an augmented queue embedding with Hamiltonian path E^*_Q and queue edges E^*_P. Put differently, Hamiltonian path edges and queue edges swap their roles when going from the primal to the dual. In fact, this characterizes augmented queue embeddings:

Theorem 2.6. *Let $G = (V, E_Q \dot\cup E_P)$ be an embedded graph such that E_P is a Hamiltonian path. Further, let $G^* = (F, E^*_Q \dot\cup E^*_P)$ be the dual of G. The following two statements are equivalent:*

(i) The embedding of G is an augmented queue embedding, where E_P is the Hamiltonian path and E_Q are the queue edges.

(ii) The embedding of G^* is an augmented queue embedding, where E^*_Q is the Hamiltonian path and E^*_P are the queue edges.

Proof. (i) $\Rightarrow$ (ii): Let $v_1, \dots, v_n$ be the Hamiltonian path E_P and $\prec$ be the respective linear layout. As the embedding of G is an augmented queue embedding, there exist two faces f_1 and f_p such that v_n is incident to f_1, v_1 is incident to f_p, and there is a simple path $Q^* = f_1 \rightsquigarrow f_p$ in $G^* \setminus E^*_P$ that contains all edges E^*_Q (Lem. 2.4).

First, we show that Q^* is a Hamiltonian path of G^*. To every face $f \in F$ there is at least one edge of E^*_Q incident. For contradiction, suppose that this is not the case. Then, there is a face f with incident edges E^*_f such that $E^*_f \subseteq E^*_P$. As the dual is connected, $E^*_f \neq \emptyset$. E^*_f is a cut-set and, hence, the primal edges of E^*_f form a simple circle in G that consists of Hamiltonian path edges, which is a contradiction. As Q^* contains all edges in E^*_Q, it visits all faces at least once. Further, Q^* is also simple and, thus, a Hamiltonian path of G^* from f_1 to f_p.

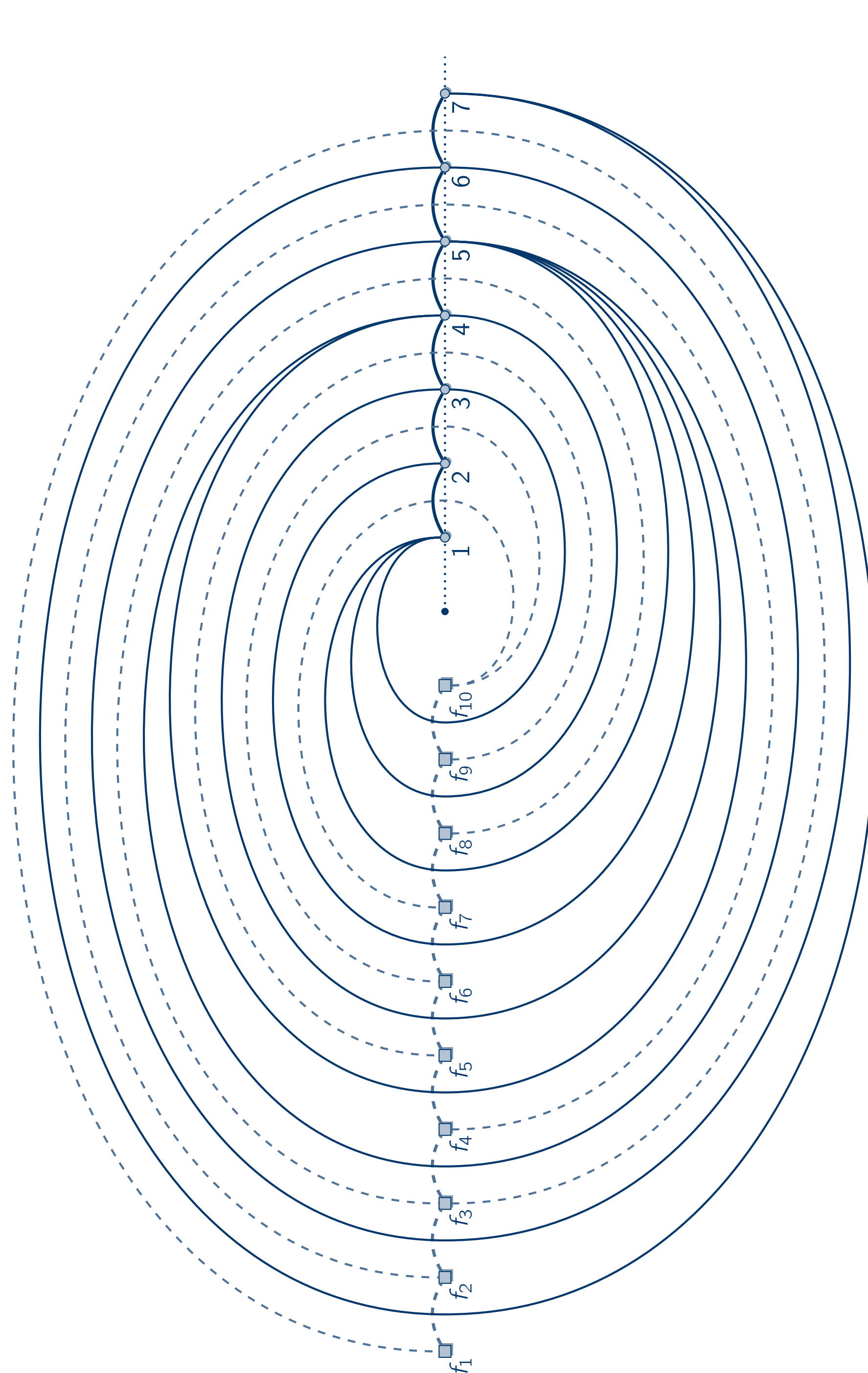


Figure 2.27: Plane representation of the augmented queue embedding and its dual (dashed) of the queue graph in Fig. 2.25(d) (without the dashed edge).

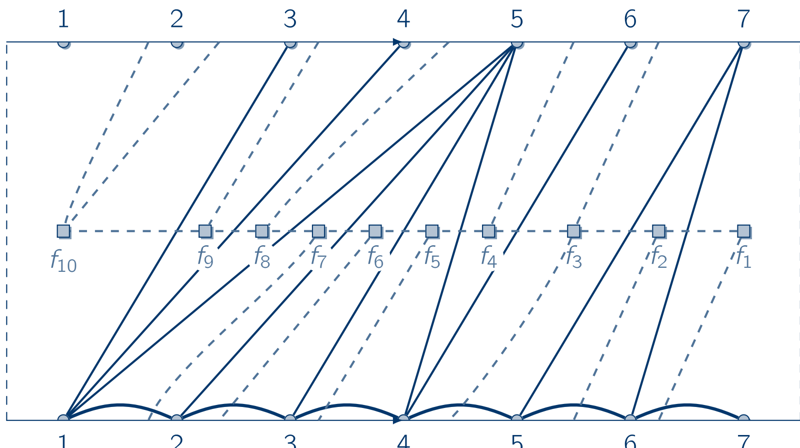


Figure 2.28: Fundamental polygon representation of the augmented queue embedding and its dual (dashed) of the queue graph in Fig. 2.25(d) (without the dashed edge).

Now, let $\Lambda^* = (\prec^*, E^{*\bot}, E^{*\top})$ be the LC rotation system of G^* as induced by the Hamiltonian path Q^*. As the rotation system of f_1 and f_p can be split at an arbitrary point, we assume that all edges incident to f_1 enter f_1 from above and all edges incident to f_p enter f_p from below. Let Σ^* be the deque schedule that corresponds to Λ^*. Due to planarity and Thm. 2.3, Λ^* is planar and Σ^* is a deque layout. What is left to show is that all edges in E_P^* are queue edges in Σ^* that are inserted at the head and removed at the tail.

As in the proof of Thm. 2.3, we assume that we are given a simultaneous drawing of G and G^*, i. e., each of G and G^* is drawn plane according to the given LC rotation systems, all faces are placed within the regions to which they correspond, and such that each primal edge e crosses its dual e^* exactly once. Further, for every edge $e \in E_P \cup E_Q$, we denote by $l(e)$ and $r(e)$ the endpoints of e with $v_1 \preceq l(e) \prec r(e) \preceq v_n$. That is, if $e \in E_P$, then $r(e)$ is the immediate successor of $l(e)$ on the Hamiltonian path E_P, and if $e \in E_Q$, then e is inserted at $l(e)$ and removed at $r(e)$ in the queue layout of G. Analogously, for every edge $e^* \in E_P^* \cup E_Q^*$, $l(e^*)$ and $r(e^*)$ are the faces to which e is incident with $f_1 \preceq^* l(e^*) \prec^* r(e^*) \preceq^* f_p$. By definition, all edges incident to f_1 enter f_1 from above and, hence, are inserted at the head in Σ^*. Likewise, all edges incident to f_p are removed at the tail.

Let $e^* \in E_P^*$ be an edge such that $f_1 \prec^* l(e^*) \prec^* r(e^*) \prec^* f_p$. In the remainder of the proof, we show that e^* enters $l(e^*)$ from above and $r(e^*)$ from below the Hamiltonian path E_Q^*. The situation we obtain is sketched in Fig. 2.29, where again the dual edges are drawn dashed for clarity. The strategy of the proof is as follows: Consider the shaded region in Fig. 2.29. This region forms a "tube" in which e^* must lie and it guarantees that e^* enters its endpoints from the correct sides. We start by defining the elements that form the border of the "tube".

As $l(e^*) \prec^* r(e^*)$ there is an edge $e_1^* \in E_Q^*$ on the Hamiltonian path of the dual such that $l(e^*) \preceq^* l(e_1^*) \prec^* r(e_1^*) \preceq^* r(e^*)$. The primal of e_1^* is denoted by e_1, where, by assumption, e_1 enters $l(e_1)$ from above and $r(e_1)$ from below. Note that e_1 is a queue edge in G. For the endpoints of e_1, we show that:

$$v_1 \preceq l(e_1) \preceq l(e) \prec r(e) \preceq r(e_1) \preceq v_n\,. \tag{2.4}$$

First, e_1 crosses its dual e_1^* between $l(e^*)$ and $r(e^*)$ on the Hamiltonian path of G^* since $l(e^*) \preceq^* l(e_1^*) \prec^* r(e_1^*) \preceq^* r(e^*)$. Second, e_1 enters $l(e_1)$ from above and $r(e_1)$ from below by assumption. Hence, if $l(e) \prec l(e_1)$ or $r(e_1) \prec r(e)$, then e_1 would cross e^* which is not possible as e_1 is not the primal of e^*. Note that this crossing exists independently of the sides from which e^* enters its endpoints.

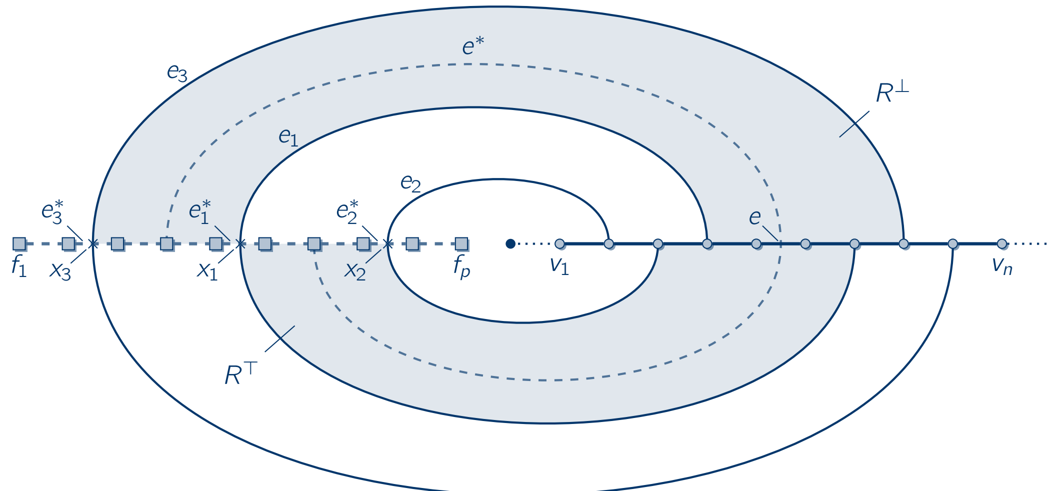


Figure 2.29: Situation obtained in the proof of Thm. 2.6.

Further, as $r(e^*) \prec^* f_p$, there is an edge $e_2^* \in E_Q^*$ such that $r(e^*) \preceq^* l(e_2^*) \prec^* r(e_2^*) \preceq^* f_p$. For the endpoints of the primal e_2 of e_2^*, we show that:

$$v_1 \preceq l(e_2) \preceq l(e_1)\,, \qquad\qquad l(e_2) \prec r(e_2) \preceq l(e)\,. \tag{2.5}$$

The edge curves of e_1 and the edges on the Hamiltonian path between $l(e_1)$ and $r(e_1)$ enclose a region R in the drawing, where, w. l. o. g., f_p is situated within R. By definition, the endpoints of e_2^* lie between $r(e^*)$ and f_p on the Hamiltonian path of the dual. Thus, the position of the crossing of e_2 with e_2^* is between $r(e^*)$ and f_p and, in particular, also between the crossing of e_1^* with e_1 and the position of f_p. Also remember that vertex v_1 is incident to f_p. For these reasons and due to the planarity of the drawing of G, the inner part of the edge curve of e_2 lies completely within R. Further, e_2 enters $l(e_2)$ from above and $r(e_2)$ from below and, hence, $v_1 \preceq l(e_2) \preceq l(e_1)$ and $l(e_2) \prec r(e_2) \preceq r(e_1)$. What is left to show is $r(e_2) \preceq l(e)$. For this, consider the crossing of e^* with e. If $l(e) \prec r(e_2)$, then e_2 would cross e^* which is not possible as e^* is not the dual of e_2. Hence, Eq. (2.5) follows.

By symmetric arguments, there is an edge $e_3^* \in E_Q^*$ such that $f_1 \preceq^* l(e_3^*) \prec^* r(e_3^*) \preceq^* l(e^*)$, where the primal e_3 of e_3^* has the properties:

$$r(e) \preceq l(e_3) \prec r(e_3)\,, \qquad\qquad r(e_1) \preceq r(e_3) \preceq v_n\,. \tag{2.6}$$

Consider the bounded region $R^\perp$ enclosed by the curve that consists of the following elements of the drawing (shaded and above the front line in Fig. 2.29):

- The edge curves on the Hamiltonian path E_P of G from $l(e_1)$ to $l(e_3)$, where $l(e_1) \preceq l(e) \prec r(e) \preceq l(e_3)$ by Eqs. (2.4) and (2.6).

- The part of the edge curve of e_3 from $l(e_3)$ to the crossing of e_3 with e_3^*. We denote this crossing point by x_3.
- The edge curves of the Hamiltonian path E_Q^* of G^* from x_3 to the crossing point x_1 between e_1 and e_1^*.
- The part of the edge curve of e_1 from x_1 to $l(e_1)$, which closes the curve.

Additionally, we define the bounded region $R^\top$ which is enclosed by the following elements of the drawing (shaded and below the front line in Fig. 2.29):

- The edge curves on the Hamiltonian path E_P of G from $r(e_2)$ to $r(e_1)$, where $r(e_2) \preceq l(e) \prec r(e) \preceq r(e_1)$ by Eqs. (2.4) and (2.5).
- The part of the edge curve of e_1 from $r(e_1)$ to the crossing point x_1 of e_1 with e_1^*.
- The edge curves of the Hamiltonian path E_Q^* of G^* from x_1 to the crossing point x_2 of e_2 with e_2^*.
- The part of the edge curve of e_2 from x_2 to $r(e_2)$, which closes the curve.

Note that due to the properties of e_1, e_2, and e_3 both curves we have just constructed are non-selfintersecting and, thus, $R^\perp$ and $R^\top$ are well defined.

After these prearrangements, we are finally ready to show that e^* must enter $l(e^*)$ from above and $r(e^*)$ from below. Remember, the idea of the construction is that the regions $R^\perp$ and $R^\top$ form a "tube" which forces the edge curve of e^* to enter its endpoints from the correct sides. First note that the crossing point of e with e^* lies at the boundaries of both $R^\perp$ and $R^\top$ as $l(e_1) \preceq l(e) \prec r(e) \preceq l(e_3)$ and $r(e_2) \prec l(e) \prec r(e) \preceq r(e_1)$, respectively. Also note that $l(e^*)$ is at the boundary of $R^\perp$ and $r(e^*)$ at the boundary of $R^\top$. Hence, all inner points of the edge curve of e^* are within $R^\perp \cup R^\top$ as otherwise e^* cannot cross its primal e. Consequently, e^* enters $l(e^*)$ from within $R^\perp$ and, by the construction of $R^\perp$, e^* must enter $l(e^*)$ from above. Similarly, e^* enters $r(e^*)$ from within $R^\top$ and, thus, from below. We can conclude that e^* is a queue edge in the deque schedule Σ^* inserted at the head and removed at the tail.

For an edge e^* with endpoint f_1, an analogous reasoning shows that e^* is removed at the tail, and if e^* has endpoint f_p, it is inserted at the head.

(ii) $\Rightarrow$ (i): The proof is equal to before with swapped roles of primal and dual. □

From Cor. 2.10 and Thm. 2.6, we obtain:

Corollary 2.11. *For any graph $G = (V, E_Q)$, the following statements are equivalent.*

(i) G is a queue graph.

(ii) G is a spanning subgraph of an exuberant $\prec$-augmentation $G_\prec = (V, E_Q \dot\cup E_P)$ such that $G_\prec$ has an augmented queue embedding with Hamiltonian path E_P and queue edges E_Q.

(iii) G is a spanning subgraph of an exuberant $\prec$-augmentation $G_\prec = (V, E_Q \dot\cup E_P)$ where $G_\prec$ has an embedding such that the embedding of its dual $G_\prec^* = (F, E_Q^* \dot\cup E_P^*)$ is an augmented queue embedding with Hamiltonian path E_Q^* and queue edges E_P^*.

Suppose we are given a graph $G = (V, E_Q \dot{\cup} E_P)$ endowed with an augmented queue embedding. By Thm. 2.6, we know that $G^* = (F, E_Q^* \dot{\cup} E_P^*)$ is also endowed with an augmented queue embedding with Hamiltonian path Q^*. Not only is Q^* Hamiltonian, it is also Eulerian on E_Q^* as each edge of E_Q^* is traversed exactly once. Now, suppose we remove an edge $e \in E_P$ from G. In the dual G^*, this is equivalent to contracting the dual edge e^* of e, i. e., the endpoints of e^* are identified [Eve12, pp. 149]. In the obtained dual graph, Q^* is not Hamiltonian anymore as one face is visited twice. However, Q^* is still Eulerian and it stays so after removing all edges of E_P from G. The obtained graph $G \setminus E_P$ is a queue graph and is endowed with a planar LC rotation system that corresponds to a queue layout.

Corollary 2.12. *If G is a queue graph with queue layout Σ such that G is embedded according to the corresponding LC rotation system $\Lambda := \Sigma$, then G^* contains a Eulerian path.*

The converse of Cor. 2.12 is not true for the following reasons: Deciding whether a bipartite graph is a queue graph is $\mathcal{NP}$-complete by Prop. 2.7, and so is deciding whether a bipartite and planar graph is a queue graph. Hence, there is at least one planar bipartite graph G which is no queue graph. As G is bipartite, the dual of any of its embeddings contains a Eulerian cycle and, hence, a Eulerian path.

We wrap this section up with an interpretation of Thm. 2.6: Consider again the graph $G = (V, E_Q \dot{\cup} E_P)$ and its augmented queue embedding in Fig. 2.28. In the corresponding queue layout, only the edges in E_Q are processed. Denote by $\mathcal{C}_i$ the input of vertex i and by $\mathcal{C}_8$ the output of vertex 7. For the example, we obtain:

$$
\begin{aligned}
\mathcal{C}_1 &= () , & \mathcal{C}_2 &= ((1,5),(1,4),(1,3)) , \\
\mathcal{C}_3 &= ((2,5),(1,5),(1,4),(1,3)) , & \mathcal{C}_4 &= ((3,5),(2,5),(1,5),(1,4)) , \\
\mathcal{C}_5 &= ((4,6),(4,5),(3,5),(2,5),(1,5)) , & \mathcal{C}_6 &= ((5,7),(4,6)) , \\
\mathcal{C}_7 &= ((6,7),(5,7)) , & \mathcal{C}_8 &= () ,
\end{aligned}
$$

By Thm. 2.6, we know that the dual of every queue edge in the primal belongs to the Hamiltonian path of the dual. For instance, the dual of $(1,4)$ connects f_9 with f_8, and the dual of $(1,5)$ connects f_8 with f_7. Hence, edges $(1,4)$ and $(1,5)$ "frame" f_8 in the embedding. Intriguingly, edge $(1,4)$ and $(1,5)$ are also adjacent in the queue, e. g., in $\mathcal{C}_2$. We rewrite the contents of the queue such that they incorporates the faces as follows:

$$
\begin{aligned}
\mathcal{C}'_1 &= (f_{10}) , \\
\mathcal{C}'_2 &= (f_7,(1,5),f_8,(1,4),f_9,(1,3),f_{10}) , \\
\mathcal{C}'_3 &= (f_6,(2,5),f_7,(1,5),f_8,(1,4),f_9,(1,3),f_{10}) , \\
\mathcal{C}'_4 &= (f_5,(3,5),f_6,(2,5),f_7,(1,5),f_8,(1,4),f_9) , \\
\mathcal{C}'_5 &= (f_3,(4,6),f_4,(4,5),f_5,(3,5),f_6,(2,5),f_7,(1,5),f_8) , \\
\mathcal{C}'_6 &= (f_2,(5,7),f_3,(4,6),f_4) , \\
\mathcal{C}'_7 &= (f_1,(6,7),f_2,(5,7),f_6) , \\
\mathcal{C}'_8 &= (f_1) .
\end{aligned}
$$

Now, every edge in the queue has two faces to both of its sides which are exactly the same as in the embedding. We further modify the contents by removing all primal edges:

$$\mathcal{C}_1^* = (f_{10}), \qquad \mathcal{C}_2^* = (f_7, f_8, f_9, f_{10}), \qquad \mathcal{C}_3^* = (f_6, f_7, f_8, f_9, f_{10}),$$
$$\mathcal{C}_4^* = (f_5, f_6, f_7, f_8, f_9), \qquad \mathcal{C}_5^* = (f_3, f_4, f_5, f_6, f_7, f_8), \qquad \mathcal{C}_6^* = (f_2, f_3, f_4),$$
$$\mathcal{C}_7^* = (f_1, f_2, f_6), \qquad \mathcal{C}_8^* = (f_1).$$

Now, the data items in the queue are the faces. What we have now obtained is what we coin *dual queue layout*, which is the dual variant of a queue layout. In a dual queue layout, instead of the edges, the faces are processed and there is an edge between two faces if they are adjacent in the queue. For the example, this works as follows: Initially, we start with face f_{10} in the queue. Remember that in the primal, edges $(1, 3)$, $(1, 4)$, and $(1, 5)$ are inserted at vertex 1 to obtain $\mathcal{C}_2$. In the dual queue layout, this means that we insert faces f_9, f_8, and f_7 at the head in order which yields $\mathcal{C}_2^*$. Inserting $(2, 5)$ in the primal corresponds to inserting f_6 in the dual. Removing edge $(1, 3)$ at vertex 3, corresponds to removing face f_{10} at the tail in the dual queue layout. Note that edge $(1, 3)$ is at the boundary of faces f_9 and f_{10} and now, in $\mathcal{C}_4^*$, f_9 is at the tail of the queue. We can proceed in this manner until we obtain the queue with content $\mathcal{C}_8^* = (f_1)$. Note that in contrast to (primal) queue layouts, we start and finish with a queue that contains a single face. These faces are the left- and rightmost faces in an LC embedding.

By using dual queue layouts, we can immediately conclude that the dual graph must contain a Hamiltonian path: First, every pair of faces that is adjacent in the queue is connected by an edge. Second, all faces are processed in the queue in a first in, first out manner which yields the Hamiltonian path. In our outlook to possible future work, we generalize dual queue layouts to dual deque layouts (Sect. 2.6.3).

2.5 Characterizing Planarity by the Splittable Deque

Fig. 2.30 shows a maximal planar graph with no Hamiltonian path [Hel07] and, therefore, it is no deque graph. This raises the following question: How can we extend the deque such that it characterizes all planar graphs? In the following, we investigate this question and find a remarkably simple answer: We need to split the deque into smaller deques. For this, we define the splittable deque and prove the following:

Theorem 2.7. *A graph is planar if and only if it is a splittable deque graph.*

Remember, we only deal with connected graphs and Thm. 2.7 canonically extends to unconnected graphs.

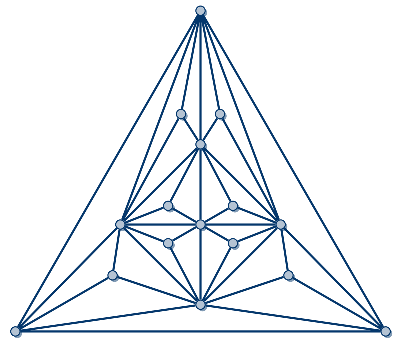


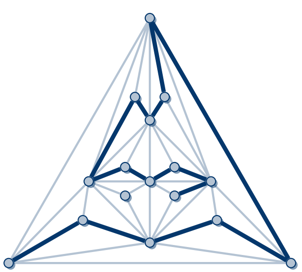
Figure 2.30: A maximal planar graph with no Hamiltonian path [Hel07].

The graph $G = (V, E)$ from Fig. 2.30 may contain no Hamiltonian path itself, however, it contains an "almost-Hamiltonian" path p (drawn bold in Fig. 2.31(a)) that falls short of only one vertex. Path p induces a subgraph $G_p = (V_p, E_p)$, where V_p contains all vertices on p and E_p are all edges from E for which both endpoints are on p. G_p is depicted in Fig. 2.31(b). Path p is Hamiltonian in G_p and, hence, G_p is a deque graph by Thm. 2.2. In general, any path in an embedded graph induces a deque graph. This insight is one idea behind the splittable deque: As long as we follow a path, we can process all edges in the deque according to the planar rotation system.

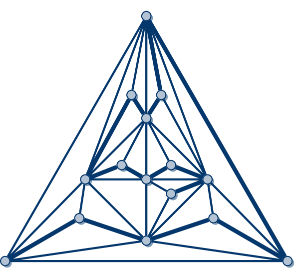
Consider Fig. 2.31(c), which again shows path p from vertex v_s to v_{e_1} (bold). Additionally, at vertex v_c, path p branches (dashed) to vertex v_{e_2} which is not visited by p. Starting with an empty deque at vertex v_s, we process all edges in the deque, even the ones incident to v_{e_2}, until we branch at v_c. This leads to the second idea behind the splittable deque: At such a branching point, the deque is split into two pieces such that one piece contains all edges removed on the path from v_c to v_{e_1}, and the other piece contains all edges removed at v_{e_2}. Generally, we have to allow any number of such branching points, and the structure underlying these paths and branching points is a special kind of spanning tree, namely, a depth-first search tree.

2.5.1 Depth-First Search Trees

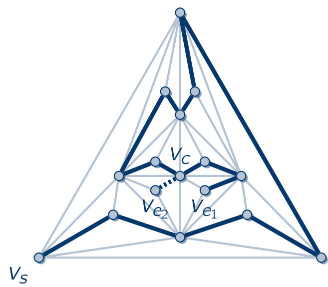
For a connected graph $G = (V, E)$, a *depth-first search tree* or *DFS tree* $\mathcal{T} = (V, E_\mathcal{T})$ is a rooted, directed spanning tree of G obtained from a DFS traversal starting at a root vertex $r \in V$ [Eve12]. We assume that the *tree edges* $E_\mathcal{T}$ are directed from the parent to its



(a) An "almost-Hamiltonian" path p in the graph from Fig. 2.30.



(b) The subgraph as induced by path p, which is a deque graph by Thm. 2.2.



(c) All edges can be processed in the deque along the path starting at v_s until a branch (v_c) is reached. There the deque must be split.

Figure 2.31: A path in the planar graph from Fig. 2.30 induces a deque layout. At a branching point, the deque must be split.

children. By $u \rightarrow v$ we denote that $(u, v) \in E_\mathcal{T}$, and by $u \stackrel{+}{\rightarrow} v$, we denote a dipath of tree edges (at least one) from u to v. Vertex u is an *ancestor* of v and v is a *descendant* of u. By $u \stackrel{*}{\rightarrow} v$, we denote that $u = v$ or $u \stackrel{+}{\rightarrow} v$. For convenience, we combine "$\rightarrow$", "$\stackrel{+}{\rightarrow}$", and "$\stackrel{*}{\rightarrow}$" for compound dipaths in $\mathcal{T}$, e. g., $p = r \stackrel{+}{\rightarrow} v \rightarrow w \stackrel{*}{\rightarrow} x$ is obtained from the successive traversal of dipath $r \stackrel{+}{\rightarrow} v$ with $r \neq v$, edge $v \rightarrow w$, and dipath $w \stackrel{*}{\rightarrow} x$. A vertex with no children is called *leaf*. For a vertex v, we call the subgraph of $\mathcal{T}$ that is induced by v and all dipaths to descendants of v the *subtree of* v. A dipath $r \stackrel{*}{\rightarrow} v$ from the root r to a leaf v is called *root-to-leaf dipath*. $\mathcal{T}$ partitions E into tree edges $E_\mathcal{T}$ and *forward edges* F. For each forward edge $\{u, v\} \in F$, there is a dipath $u \stackrel{+}{\rightarrow} v$ where u is an ancestor of v.

For our purposes, we use the DFS algorithm shown in Alg. 2.4 which computes a DFS tree along with the *DFS numbers* [Eve12]. `DFS` uses the recursive procedure `DFSVisit` for the DFS traversal. The DFS number $D(v)$ is the time instant, starting with 1, when a vertex v is encountered first by `DFSVisit`. $D(v)$ is initialized to 0 for each vertex v (line 3) which indicates that v has not been visisted by `DFSVisit`. Additionally, and for reasons that become clear later, `DFS` computes $D^+(v)$ which is the next DFS number after all vertices in the subtree of v have been visited (cf. line 14). The running time of Alg. 2.4 is in $\mathcal{O}(|V| + |E|)$.

A topological sorting of a DFS tree $\mathcal{T}$ yields a linear layout $\prec_\mathcal{T}$. We call $\prec_\mathcal{T}$ *linear layout induced by* $\mathcal{T}$. For instance, the DFS numbering D corresponds to a topological sorting of $\mathcal{T}$ with $u \prec_\mathcal{T} v$ if and only if $D(u) < D(v)$.

Fig. 2.33(a) on page 71 shows a graph along with a possible DFS tree, where the tree edges are drawn bold and vertex 1 is the root. The vertices are labeled with their DFS number. A more convenient drawing of the graph and its DFS tree is shown in Fig. 2.33(c) on page 71. There, the DFS tree is drawn hierarchically from top to bottom with the root at the top and the leaves at the bottom.

Algorithm 2.4. DFS

```
    Input: graph G = (V, E)
    Output: DFS tree T = (V, E_T) and DFS numbers D and D+
 1  d ← 1
 2  global d
 3  foreach v ∈ V do D(v) ← 0 and D+(v) ← 0
 4  r ← any vertex v ∈ V
 5  E_T, D, D+ ← DFSVisit(r, ∅, D, D+)
 6  return T = (V, E_T), D, D+

 7  Procedure DFSVisit(v, E_T, D, D+)
 8  |   D(v) ← d
 9  |   d ← d + 1
10  |   foreach w ∈ {w ∈ V | w is adjacent to v} do
11  |   |   if D(w) = 0 then w has not yet been visited
12  |   |   |   E_T ← E_T ∪ {(v, w)}
13  |   |   |   E_T, D, D+ ← DFSVisit(w, E_T, D, D+)
14  |   D+(v) ← d
15  |   return E_T, D, D+
```

2.5.2 Splittable Deque Layouts

Next, we introduce a split operation to the deque and, thereby, obtain the splittable deque as an (abstract) data structure. We then define how a layout of a graph in a splittable deque looks like.

2.5.2.1 The Splittable Deque Data Structure

The splittable deque (*SD*) supports all deque operations and additionally can be split. As with the deque, the content of an SD is given by a tuple $\mathcal{C} = (e_1, \dots, e_k)$, where e_1 is at the head and e_k at the tail. Again, $()$ denotes the empty content. Data items can be inserted and removed at the head and the tail with the deque operations as defined in Sect. 2.2.2.1.

Given two SD contents $\mathcal{C} = (e_1, \dots, e_k)$ and $\mathcal{C}' = (e'_1, \dots, e'_{k'})$ with $k, k' \geq 0$, we denote by $\mathcal{C} \circ \mathcal{C}'$ the *concatenation* of $\mathcal{C}$ with $\mathcal{C}'$, where $\mathcal{C} \circ \mathcal{C}' := (e_1, \dots, e_k, e'_1, \dots, e'_{k'})$. Note that $\circ$ is associative. The split operation divides $\mathcal{C}$ into ℓ pieces $\mathcal{C}_1, \dots, \mathcal{C}_\ell$. Each of $\mathcal{C}_i$ is again the (possibly empty) content of an SD where:

$$\mathcal{C} = \mathcal{C}_1 \circ \mathcal{C}_2 \circ \dots \circ \mathcal{C}_\ell \,.$$

Each of the obtained pieces supports all operations of the SD, i. e., items can be inserted and removed, and it can be further split.

Defined as such, the SD is too powerful to characterize planarity as it allows (almost) direct access to each element: Let $\mathcal{C} = (e_1, \dots, e_i, \dots, e_k)$ with $1 < i < k$. If we want to access e_i, we split $\mathcal{C}$ into pieces $\mathcal{C}_1 = (e_1, \dots, e_i)$ and $\mathcal{C}_2 = (e_{i+1}, \dots, e_k)$ and remove e_i at the tail of $\mathcal{C}_1$. With the deque, we either have to remove all data items $e_1, \dots, e_{i-1}$ at the head in order, or $e_k, e_{k-1}, \dots, e_{i+1}$ at the tail in order, to gain access to e_i. Thus, using the SD without any

restrictions for graph layouts is of little interest as all edges can be removed at any time from the SD by applying the appropriate split operation. This would allow any graph to have an SD layout. In the next section, we see how a graph layout with the SD must be restricted in order to be “interesting”, especially, in order to characterize planarity.

2.5.2.2 Graph Layouts in the Splittable Deque

Remember that in a deque schedule (Def. 2.5) the order in which the vertices are processed is defined by a linear layout, which can be seen as a processing pipeline, i. e., if u is the immediate predecessor of v, then the output of u is the input of v. For SD schedules, we generalize linear layouts to tree layouts. A *tree layout* of a graph $G = (V, E)$ is a DFS tree $\mathcal{T} = (V, E_{\mathcal{T}})$ of G. Just like a linear layout, a tree layout can be seen as a processing pipeline in which a vertex v can have multiple immediate successors, namely, its children in $\mathcal{T}$. If v is a vertex with $\ell \geq 1$ children, then the input SD of vertex v with content $\mathcal{C}_v$ is split into ℓ pieces $\mathcal{C}_1, \dots, \mathcal{C}_\ell$. For instance in Fig. 2.32, v has three children w_1, w_2 and w_3 and, hence, its input $\mathcal{C}_v = (e_1, \dots, e_9)$ is split into three pieces $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$. Recall that a deque schedule contains the mappings $E^{\mathtt{h}}$ and $E^{\mathtt{t}}$ that define at which side and in which order the edges are processed at each vertex. Moreover, $E^{\mathtt{h}}(v)$ together with $E^{\mathtt{t}}(v)$ induces an LC rotation system at vertex v and vice versa. Using (general) rotation systems and tree layouts, we define SD schedules:

Definition 2.10 (SD Schedule). *For a graph $G = (V, E)$, we call a tuple $\Psi = (\mathcal{T}, \mathcal{R})$, consisting of a tree layout $\mathcal{T}$ and a rotation system $\mathcal{R}$ of G, an* SD schedule*.*

The symbol Ψ is chosen to resemble a branch in a tree layout where the SD is split.

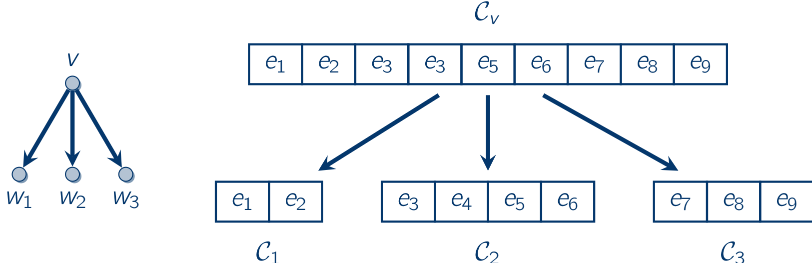


Figure 2.32: Vertex v has three children and its input $\mathcal{C}_v$ is split into three pieces $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$

A high-level description of an SD layout is as follows: Let $\mathcal{C}_v$ be the content of the input SD of a vertex v with $\ell \geq 0$ children.

1. Split $\mathcal{C}_v$ into to ℓ pieces. If v is a leaf, $\mathcal{C}_v$ is not split and is emptied in the next step.
2. Remove each edge incident to v and an ancestor of v from one of the pieces, and insert each edge incident to a descendant of v to one of the pieces.
3. Pass each piece to a distinct child of v.

In detail, for a given SD schedule $\Psi = (\mathcal{T}, \mathcal{R})$ with tree layout $\mathcal{T} = (V, E_{\mathcal{T}})$ where $r \in V$ is the root, the SD layout works as follows. The input $\mathcal{C}_r = ()$ of r is empty. Let $\mathcal{C}_v = (e_1, \dots, e_k)$ be the input of vertex $v \in V$. Further, let $\mathcal{R}_v = (e_1, \dots, e_p)$ be the rotation system of $\mathcal{R}_v$ such that, if v is not the root, then $e_1 = (u, v)$ is the edge from v's parent u to v, and if v is the

root, then e_1 is an edge from v to any child of v. $\mathcal{R}_v$ defined as such implies a total order on the incident edges of v. We denote by $e <_{\mathcal{R}_v} e'$ that e comes before e' in $\mathcal{R}_v$. Let $w_1, \cdots, w_\ell$ be the children of v in $\mathcal{T}$ in order of $\mathcal{R}_v$, i. e., $(v, w_1) <_{\mathcal{R}_v} (v, w_2) <_{\mathcal{R}_v} \ldots <_{\mathcal{R}_v} (v, w_\ell)$. Assume for now that v has at least one child. $\mathcal{C}_v$ is split into $\ell \geq 1$ pieces $c_{w_1}, \ldots, c_{w_\ell}$ (Step **1.**), where piece c_{w_i} is the SD that is transformed into the input $\mathcal{C}_{w_i}$ of child w_i after all edges incident to v have been processed. We use a lower case letter for the piece c_{w_i} to distinguish it from the input $\mathcal{C}_{w_i}$ of w_i.

Similar to deque layouts, an edge $e = \{v, w\}$ has to be removed at v if w is an ancestor of v in $\mathcal{T}$, and e has to be inserted if w is a descendant of v. Let w_i be a child of v with corresponding content c_{w_i} of the SD. Further, let E_v^i be the set of edges $e = \{v, w\}$ incident to v with the following properties: Either, $e \in c_{w_i}$, i. e., e was inserted at an ancestor of v and has to be removed from c_{w_i} at v, or there is a dipath $w_i \xrightarrow{*} w$, i. e., $w_i = w$ or w is a descendant of w_i. In the latter case, we say that e *is removed in the subtree of* w_i. By using the very same ideas as for deque layouts (Sect. 2.2.4), the rotation system of v defines how the edges in E_v^i are processed. Let $\mathcal{R}_v^i$ be the rotation system obtained from $\mathcal{R}_v$ by removing all edges not in E_v^i. Further, let p be the dipath $r \xrightarrow{*} v \rightarrow w_i$ in $\mathcal{T}$. Dipath p divides $\mathcal{R}_v^i$ at v into two regions from which we obtain an LC rotation system at v as shown in Fig. 2.18 on page 43. Further, the LC rotation system at v defines a deque schedule $E_i^{\mathtt{h}}(v)$ and $E_i^{\mathtt{t}}(v)$ for all edges in E_v^i. After processing all edges in E_v^i in c_{w_i} according to $E_i^{\mathtt{h}}(v)$ and $E_i^{\mathtt{t}}(v)$ (Step **2.**), we obtain the content $\mathcal{C}_{w_i}$ which is the input of vertex w_i (Step **3.**). The same steps are carried out for all other children of v. If v is a leaf, the SD is not split and must be emptied. Note that if $\mathcal{T}$ is a (Hamiltonian) dipath, the SD is never split and the edges are processed as with a deque schedule that is induced by the Hamiltonian dipath and the rotation system of G. In particular, the SD behaves like a deque. Given a tree layout $\mathcal{T}$ of graph G, G is an *SD graph* if all edges can be processed in the SD as described.

For an example, consider the graph in Fig. 2.33(a) whose rotation system can be obtained from the drawing. The dashed edge is ignored for the moment. All tree edges are directed from parent to children and drawn bold. We denote each tree edge by t_v, where v is the child, e. g., t_2 is the tree edge $1 \rightarrow 2$. In Fig. 2.33(c), the tree layout as defined by the DFS tree is displayed where the children are ordered from left to right according to the rotation system. In fact, the rotation system in Fig. 2.33(c) is equal to the one obtained from Fig. 2.33(a). We start with vertex 1 and an empty SD $\mathcal{C}_1 = ()$. Until we reach the first vertex with more than one child, e. g., 4, the SD behaves exactly like a deque. For instance, at vertex 1, the SD is "split" into one piece c_2 to which edges t_2 and e_1 are inserted. As defined in Sect. 2.2.4, the rotation system of 1 is split such that all edges are inserted at the head in order, i. e., at first e_1 and then t_2 is inserted, and we obtain for the input of vertex 2 the content $\mathcal{C}_2 = (t_2, e_1)$. At vertex 2, $\mathcal{C}_2$ is again "split" into one piece c_3, where t_2 is removed at the head. In the following, we assume that all edges that enter the respective dipath in $\mathcal{T}$, e. g., $1 \xrightarrow{+} 3$, from the left side, e. g., edge e_2, are processed at the head, and all other edges, e. g., e_3 and e_4, at the tail. Hence, we obtain $\mathcal{C}_3 = (t_3, e_2, e_1, e_4, e_3)$ as the input of vertex 3. Note that the tree edges are processed as stack edges at the head.

We proceed in this manner until we reach vertex 4: The input SD of vertex 4 is $\mathcal{C}_4 = (t_4, e_5, e_2, e_1, e_4, e_3, e_6)$ and the SD is split into three pieces $c_5 = (t_4, e_5, e_2)$, $c_{10} = (e_1)$, and $c_{11} = (e_4, e_3, e_6)$. Tree edge t_4 is removed at the head of c_5 and, as edge e_4 enters the dipath $1 \xrightarrow{+} 11$ from the left side, it is removed at the head of c_{11}. The tree edges t_5, t_{10}, and t_{11} are all inserted at the head of c_5, c_{10}, and c_{11}, respectively. We obtain $\mathcal{C}_5 = (t_5, e_5, e_2)$, $\mathcal{C}_{10} = (t_{10}, e_1)$, and $\mathcal{C}_{11} = (t_{11}, e_3, e_6)$ as the inputs of vertices 5, 10, and 11, respectively.

We process all edges in this way until the SDs must be emptied at the leaves. Note that, in principle, $\mathcal{C}_4$ can also be split such that $c_{10} = (e_1, e_4)$ and $c_{11} = (e_3, e_6)$ and then e_4 is removed at the tail of c_{10}. This ambiguity occurs at vertices with more than one child and does not influence the property of allowing a layout in the SD.

2.5.3 Testing Planarity of a Rotation System by the Splittable Deque

`IsSDLayout` in Alg. 2.5 implements the procedure described in Sect. 2.5.2 to determine whether an SD schedule is an SD layout. `IsSDLayout` is essentially a DFS traversal of a given tree layout $\mathcal{T} = (V, E_{\mathcal{T}})$. As input, it receives the current vertex v of the DFS, the input $\mathcal{C}_v$ of v, and the SD schedule $\Psi = (\mathcal{T}, \mathcal{R})$. The return value of `IsSDLayout` is `true`, if all edges can be processed in the SD, and `false` otherwise. The initial call is `IsSDLayout`$(r, (), \Psi)$ where r is the root of $\mathcal{T}$.

Recall, we defined a deque schedule to be a deque layout if `IsDequeLayout` in Alg. 2.2 returns `true` (Def. 2.6). We use such an "algorithmic definition" for SD layouts as well:

Definition 2.11 (SD Layout). *Let $\Psi = (\mathcal{T}, \mathcal{R})$ be an SD schedule where r is the root of $\mathcal{T}$. $\Psi = (\mathcal{T}, \mathcal{R})$ is an* SD layout *if* `IsSDLayout`$(r, (), \Psi)$ *(Alg. 2.5) returns* `true`.

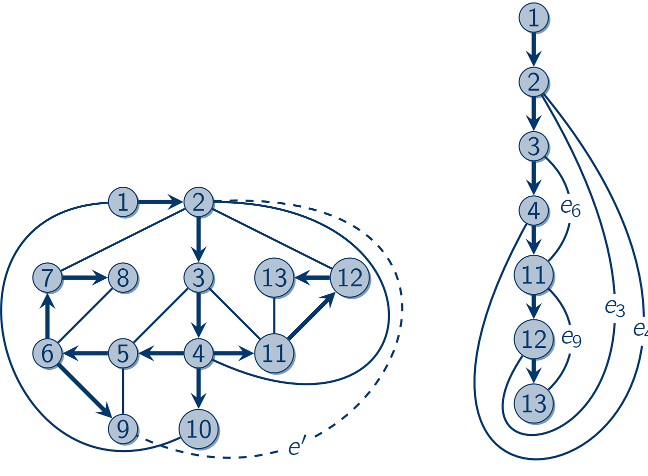
A deque schedule is a deque layout if and only if the corresponding LC rotation system is planar (Lem. 2.1). We prove the respective result for SD schedules:

Lemma 2.5. *An SD schedule $\Psi = (\mathcal{T}, \mathcal{R})$ is an SD layout if and only if $\mathcal{R}$ is planar.*

For the proof of Lem. 2.5, we analyze `IsSDLayout` step by step and show that if one of its recursive calls returns `false`, a crossing between two edges is inevitable. Conversely, if `IsSDLayout` returns `true`, we can conclude that no edges cross and the rotation system is planar. For the analysis of `IsSDLayout`, let v, $\mathcal{C}_v$, and $\Psi = (\mathcal{T}, \mathcal{R})$ be the current set of parameters. `IsSDLayout` consists of three phases: Phase 1 splits the input SD $\mathcal{C}_v$, phase 2 determines which edges incident to v have to be processed in which pieces, and phase 3 processes edges in the pieces and recursively calls `IsSDLayout` for all of v's children. We assume for convenience that the DFS numbers D and D^+ of $\mathcal{T}$ are globally accessible. In the following, let $\prec_{\mathcal{T}}$ be a linear layout as induced by $\mathcal{T}$ or, more precisely, by D (line 1). In this section, we proof the correctness of `IsSDLayout` and postpone the analysis of its running time to Sect. 2.5.4.

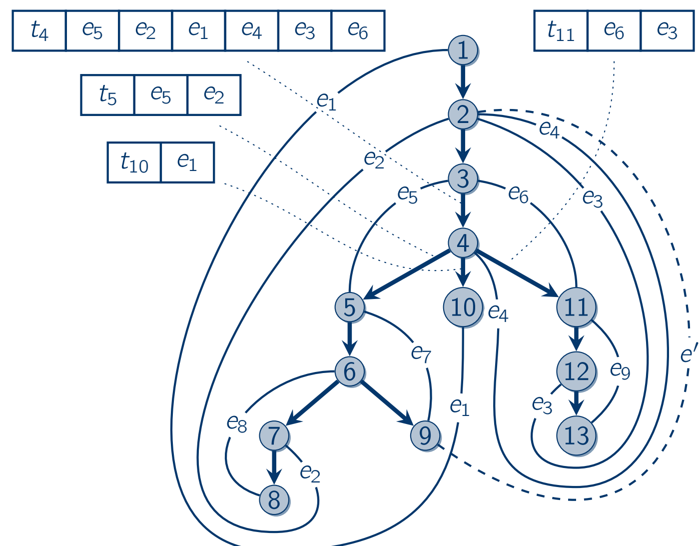
To actually insert and remove edges, `IsSDLayout` reuses `ProcessVertex` in Alg. 2.3 on page 32. Remember that the SD behaves like a deque whenever the tree layout is a dipath: Let p be a root-to-leaf dipath in $\mathcal{T}$ and denote by G_p the subgraph of G induced by p which inherits G's rotation system. For instance, the subgraph G_p for the root-to-leaf dipath $p = 1 \xrightarrow{+} 13$ in Fig. 2.33(c) is shown in Fig. 2.33(b). G_p's rotation system is planar if and only if it admits a deque layout (Cor. 2.4). In `IsSDLayout`, all edges that lie on a common root-to-leaf dipath are processed in the SD just like in a deque. This is already reflected in lines 2 to 6 in Alg. 2.5: If v is a leaf, the deque schedule $E^{\mathtt{h}}(v)$ and $E^{\mathtt{t}}(v)$ as induced by the dipath $p = r \xrightarrow{*} v$ and v's rotation system $\mathcal{R}_v$ is obtained and `true` is returned if and only if `ProcessVertex` empties $\mathcal{C}_v$.

For the remainder of this section and the proofs, we assume a drawing of the given graph that respects the given rotation system, and in which no pair of edges crosses more than once and no edge crosses any of the tree edges. As the tree layout $\mathcal{T}$ itself contains no cycles, it is always planar regardless of the rotation system. Moreover, all forward edges can be drawn such that they cause no crossing with any tree edge. This is reflected as well in the SD where all



(a) A planar graph and a tree layout (directed and bold drawn edges).

(b) A root-to-leaf dipath in the tree layout corresponds to a deque schedule.



(c) SD schedule of the graph in Fig. 2.33(a) along with the input SD for some vertices.

Figure 2.33: A graph endowed with and its SD schedule. The SD schedule is an SD layout without the dashed edge.

Algorithm 2.5. IsSDLayout

Input: vertex v, content $\mathcal{C}_v$ of SD, SD schedule $\Psi = (\mathcal{T}, \mathcal{R})$

Output: true if $\mathcal{R}$ is planar; false otherwise

```
1  ≺_T ← linear layout induced by D, i. e., u ≺_T v ⇔ D(u) < D(v)
2  if v is leaf then
3  |  E^h(v), E^t(v) ← deque schedule of v induced by p = r →* v and R_v
4  |  C_v ← ProcessVertex(C_v, ≺_T, E^h(v), E^t(v))
5  |  if C_v = () then return true
6  |  else return false
   /* Phase 1                                                     */
7  if v is root of T then
8  |  R_v = (e_1, ..., e_k) ← v's rotation system, where e_1 is tree edge to any of v's children
9  else
10 |  R_v = (e_1, ..., e_k) ← v's rotation system, where e_1 is the edge from parent of v
11 w_1, ..., w_ℓ ← children of v in order of R_v, i. e., w_1 <_{R_v} ... <_{R_v} w_ℓ
12 c_{w_1}, ..., c_{w_ℓ} ← Split(C_v, v, R_v, (w_1, ..., w_ℓ))
13 if return value of Split is ⊥ then return false
   /* Phase 2                                                     */
14 foreach w_i ∈ {w_1, ..., w_ℓ} do
15 |  ρ_i ← ()
16 |  removed(i) ← false
17 S ← stack which contains 0
18 foreach e = e_1, ..., e_k do
19 |  w_i ← child to which e is assigned, i. e., e ∈ c_{w_i} or e is removed in the subtree of w_i
20 |  ρ_i.insertAtTail(e)
21 |  u ← endpoint of e distinct from v
22 |  if D(u) < D(v) then u is ancestor of v
23 |  |  t ← 0
24 |  else t ← i
25 |  if t ∈ S then
26 |  |  while S.top() ≠ t do
27 |  |  |  t' ← S.pop()
28 |  |  |  removed(t') ← true
29 |  else if ¬removed(t) then S.push(t)
30 |  else return false
   /* Phase 3                                                     */
31 foreach w_i = w_1, ..., w_ℓ do
32 |  E_i^h(v), E_i^t(v) ← deque schedule of v as induced by p = r →* v → w_i and ρ_i
33 |  C_{w_i} ← ProcessVertex(c_{w_i}, ≺_T, E_i^h(v), E_i^t(v))
34 |  if C_{w_i} = ⊥ then return false
35 |  else if ¬IsSDLayout(w_i, C_{w_i}, Ψ) then return false
36 return true
```

tree edges can be processed canonically as stack edges at the head without interfering with any forward edge: After splitting the SD, the tree edge from the parent can be removed at the head of the first piece and, as the last step, each tree edge to child w_i can be inserted at the head of the piece for child w_i.

Phase 1 In the first phase, `IsSDLayout` splits the SD. Let $\mathcal{R}_v = (e_1, \dots, e_k)$ be the rotation system of v according to lines 7 to 10. Further, let $w_1, \dots, w_\ell$ be v's children in $\mathcal{T}$ in order of $\mathcal{R}_v$, i.e., $(v, w_1) <_{\mathcal{R}_v} \dots <_{\mathcal{R}_v} (v, w_\ell)$. In line 12 of `IsSDLayout`, the subroutine `Split` (Alg. 2.6) is called. `Split` takes as input the content $\mathcal{C}_v$ of the SD, vertex v, v's rotation system $\mathcal{R}_v$ and children $w_1, \dots, w_\ell$. $\mathcal{C}_v$ is split into pieces $c_{w_1}, \dots, c_{w_\ell}$ such that, for all $e \in c_{w_i}$, e is either removed at v or in the subtree of w_i. If successful, the pieces are returned and otherwise the return value is $\bot$.

Algorithm 2.6. `Split`

```
   Input: content C_v, vertex v, rotation system R_v = (e_1, ..., e_k) and children w_1, ..., w_ℓ
   Output: pieces c_{w_1}, ..., c_{w_ℓ}; ⊥ if split is not possible
 1 foreach w_i = w_1, ..., w_ℓ do c_{w_i} ← ()
 2 i ← 1; m_v, m_C ← −∞
 3 foreach e = e_1, ..., e_k do
 4     if e is the edge to child w_{i'} of v then
 5         i ← i'
 6         if C_v ≠ () and C_v.head() is removed in subtree of w_i then
 7             m_C ← max{i, m_C}
 8             if i < m_v then return ⊥
 9             while C_v ≠ () and ê = C_v.head() is removed in subtree of w_i do
10                 C_v.removeAtHead(ê) and c_{w_i}.insertAtTail(ê)
11     else if e ∈ C_v then
12         if C_v.removeAtHead(e) = true then c_{w_i}.insertAtTail(e)
13         else return ⊥
14     else e is inserted at v and removed in subtree of w_{i'}
15         m_v ← max{i', m_v}
16         if i' < m_C then return ⊥
17 if C_v ≠ () then return ⊥
18 else return c_{w_1}, ..., c_{w_ℓ}
```

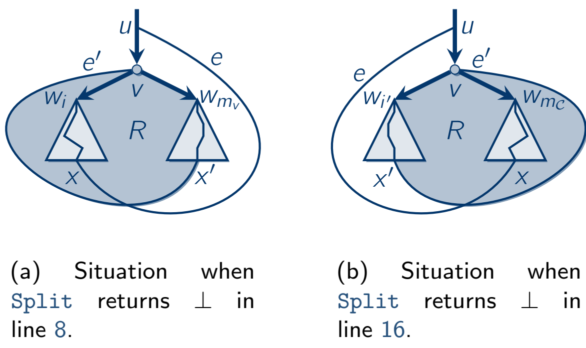
While computing the pieces, `Split` matches the content of $\mathcal{C}_v$ with the rotation system of v to detect crossings of edges that belong to distinct pieces, and crossings of edges in $\mathcal{C}_v$ that are not incident to v with edges that are inserted at v. Initially, an empty piece is created for each child (line 1). `Split` subsequently extracts edges at the head of $\mathcal{C}_v$ and appends them to the current piece c_{w_i}, starting with c_{w_1} (line 2). To avoid confusion, we use the term "extract" when `Split` removes an edge from $\mathcal{C}_v$ to distinguish it from "removing" an edge in the SD layout as done in `ProcessVertex`. The edges incident to v are processed in order of $\mathcal{R}_v$ (line 3). Let e be the edge of the current iteration. Three cases are distinguished: Either e is an edge to a child $w_{i'}$ (line 4), $e \in \mathcal{C}_v$ which implies that e is incident to an ancestor of v (line 11), or e is not in $\mathcal{C}_v$ which implies that e is incident to a descendant of v (line 14). The

content $\mathcal{C}_v$ of v's input SD gets changed by Split. For clarity, we denote by $\overline{\mathcal{C}}_v$ the unchanged content of the input SD of v as passed to Split.

First, we see how Split tests whether there are crossing edges e and e' such that $e \notin \overline{\mathcal{C}}_v$ is incident to v and $e' \in \overline{\mathcal{C}}_v$ is not incident to v, where e and e' are removed in different subtrees of v. For this, Split maintains two variables m_v and $m_{\mathcal{C}}$ which are both initialized to $-\infty$ in line 2. In the loop in line 3, let e_i be the current edge of the iteration. At the start of each iteration, m_v is the maximum so far encountered index of a child w_{m_v} such that there is an edge $e_{i'} \notin \overline{\mathcal{C}}_v$ with $i' < i$ which is incident to v and removed in the subtree of w_{m_v}. Similarly, $m_{\mathcal{C}}$ is the maximum so far encountered index of a child $w_{m_{\mathcal{C}}}$ such that there is an edge $e_{i'} \in \overline{\mathcal{C}}_v$ with $i' < i$ where $e_{i'}$ is removed in the subtree of $w_{m_{\mathcal{C}}}$ and $e_{i'}$ is not incident to v. In case $m_v = -\infty$ or $m_{\mathcal{C}} = -\infty$, the respective edge does not exist. The values of $m_{\mathcal{C}}$ and m_v are updated in lines 7 and 15, respectively. Under certain circumstances (cf. lines 8 and 16), Split returns $\bot$ when a crossing is inevitable.

Lemma 2.6. Split *(Alg. 2.6) returns $\bot$ in line 8 or line 16 if and only if there are crossing edges $e \notin \overline{\mathcal{C}}_v$ and $e' \in \overline{\mathcal{C}}_v$ such that e is incident to v and e' is not incident to v, where e and e' are removed at different subtrees of v.*

Proof. First, suppose that Split returns $\bot$ in line 8, where the current edge of the loop in line 3 points to child w_i of v. In this situation, there is an edge $e = (u, x)$ at the head of $\mathcal{C}_v$ which is removed in the subtree of child w_i and, since $i < m_v$, there is also an edge $e' = (v, x')$ which is inserted at v and removed in the subtree of child w_{m_v}. This situation is shown in Fig. 2.34(a). Consider the region R (shaded) enclosed by the path $p = v \to w_{m_v} \xrightarrow{+} x'$ and the edge curve of e'. Since $e' <_{\mathcal{R}_v} (v, w_i) <_{\mathcal{R}_v} (v, w_{m_v})$ and u is an ancestor of v, edge e enters u outside of R and enters x inside which leads to a crossing with e'.



(a) Situation when Split returns $\bot$ in line 8.

(b) Situation when Split returns $\bot$ in line 16.

Figure 2.34: Situations obtained in the proof of Lem. 2.6.

Conversely, if a crossing between e and e' is inevitable, then e must enter u outside of R and enter x inside. Since u is an ancestor of v, this implies that $e' <_{\mathcal{R}_v} (v, w_i) <_{\mathcal{R}_v} (v, w_{m_v})$ as only in this case x lies within R. This implies that $i < m_v$ in line 8 and Split returns $\bot$.

The reasoning for the case where Split returns $\bot$ in line 16 is similar (Fig. 2.34(b)). Here, let e' be the current edge of the iteration with $e' = (v, x')$ and $e \notin \overline{\mathcal{C}}_v$, where e is removed in the subtree of child $w_{i'}$. There is an edge $e = (u, x)$ such that $u \in \overline{\mathcal{C}}_v$ and u is removed in the subtree of child $w_{m_{\mathcal{C}}}$ with $i' < m_{\mathcal{C}}$. Again, there is a region R (shaded) enclosed by the path $p = v \to w_{i'} \xrightarrow{+} x'$ and edge e' such that e enters u outside of R and x inside which causes

a crossing. Conversely, if a crossing between e and e' is inevitable, then, similar to before, $i' < m_{\mathcal{C}}$ and Split returns $\bot$ in line 16. □

Next, we discuss how Split finds out whether two edges in $\overline{\mathcal{C}}_v$ which later end up in different pieces cross. Assume that e points to child $w_{i'}$ (line 4), then $c_{w_{i'}}$ is the piece to be considered next and i is set to i'. In the loop in line 9, subsequently all edges $\hat{e}$ at the head of $\mathcal{C}_v$ which are removed in the subtree of w_i are extracted from $\mathcal{C}_v$ and appended to c_{w_i}. If $e \in \mathcal{C}_v$ (line 11), then e is appended to c_{w_i} if e is at the head of $\mathcal{C}_v$. If this is not the case, $\bot$ is returned. This also happens if $\mathcal{C}_v$ is not empty after all edges are processed (line 17). Otherwise, the desired pieces are returned. Note that for all $e \in c_{w_i}$, e is either removed at v or in the subtree of w_i.

If Split returns $\bot$ in lines 13 and 17, then there are at least two crossing edges in $\mathcal{C}$, which will end up in different pieces. For an example, consider edge e' in Fig. 2.33(c) (dashed). Edge e_1 is inserted at the head at vertex 1 and e' at the tail at vertex 2, i. e., $e_1 \ll e'$. Further, in the rotation system of vertex 4, $(4,5) <_{\mathcal{R}_4} (4,10)$. At vertex 4, the SD has to be split such that $e_1 \in c_{10}$ and $e' \in c_5$, which is not possible. In Split, edge $(4,5)$ comes before edge $(4,10)$ in the loop in line 3. In the iteration of edge $(4,5)$, e' cannot be inserted to c_5 as $e_1 \ll e'$ and, therefore, edge e' is still in $\mathcal{C}_v$ in line 17 and $\bot$ is returned.

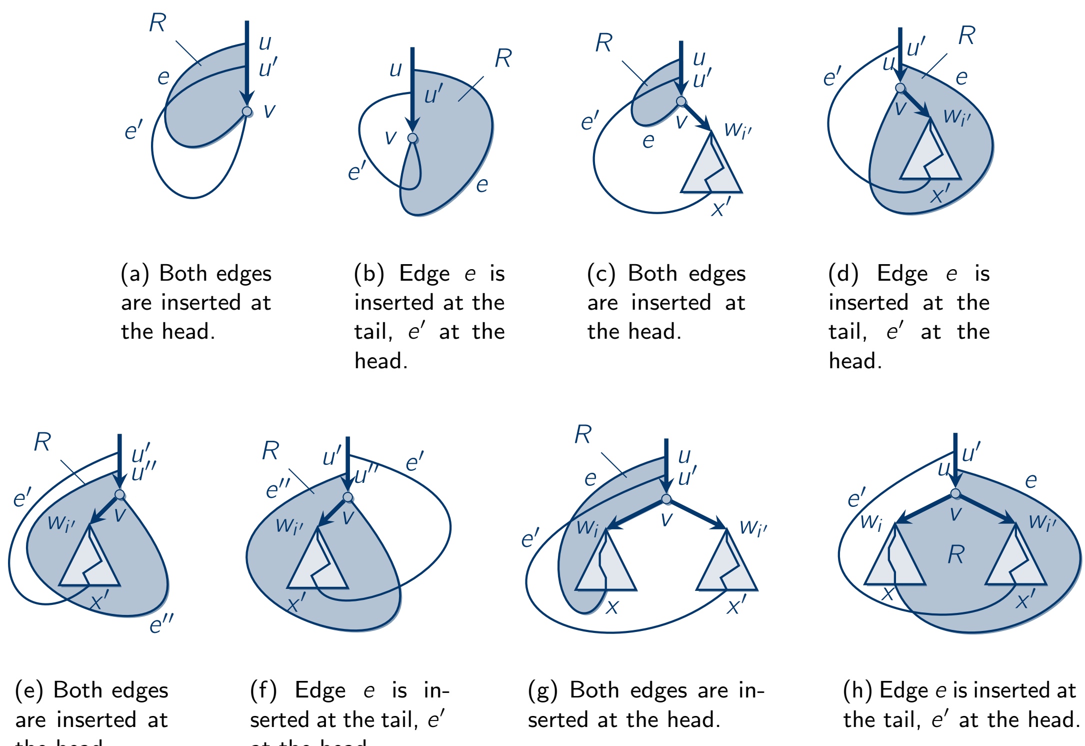
Lemma 2.7. *If* Split *(Alg. 2.6) cannot split $\mathcal{C}_v$ and returns $\bot$ in line 13 or line 17, then there are at least two crossing edges $e, e' \in \overline{\mathcal{C}}_v$ which are removed in different subtrees of v.*

Proof. We start with line 13: Edge $e \in \mathcal{C}_v$, which is incident to v, has to be appended to c_{w_i} but e is not at the head of $\mathcal{C}_v$. Instead, $e' \neq e$ is at the head with $e' \ll_{\mathcal{C}_v} e$.

Edge e' is either incident to v or not, where we assume the first for the moment. In the rotation system of v, $e <_{\mathcal{R}_v} e'$ as e is the edge of the current iteration. Since $e' \ll_{\mathcal{C}_v} e$, either both are inserted at the head, where e is inserted before e', both are inserted at the tail, where e' is inserted before e, or e' is inserted at the head and e at the tail. The first case is depicted in Fig. 2.35(a) where e and e' are inserted at u and u', respectively. Consider the bounded region R (shaded) enclosed by the dipath $p = u \xrightarrow{+} v$ and edge e. Note that both enter their endpoints u and u' from the left side as both edges are inserted at the head. As e is inserted before e', either u is an ancestor of u', or $u = u'$ and $e <_{\mathcal{R}_u} e' <_{\mathcal{R}_u} e_d$, where $\mathcal{R}_u$ is the rotation system of u and e_d is the edge from u to its child on p. In any case, e' enters u' within R. However, as $e <_{\mathcal{R}_v} e'$, e' enters v outside of R which leads to a crossing with e. The case where both e and e' are inserted at the tail is symmetric. Now, assume that e is inserted at the tail and e' at the head (Fig. 2.35(b)). Again, dipath $p = u \xrightarrow{+} v$ and e enclose the bounded region R (shaded). As e is inserted at the tail and e' at the head, e' enters u outside of R and, as $e <_{\mathcal{R}_v} e'$, e' enters v within R which causes a crossing.

Next, we assume that e' is not incident to v. In this case, e' is removed at a vertex x' in the subtree of a child $w_{i'}$ of v. Again, e and e' are inserted at u and u', respectively. Two cases have to be distinguished: Either $e <_{\mathcal{R}_v} (v, w_{i'})$ or $(v, w_{i'}) <_{\mathcal{R}_v} e$. First, we assume that $e <_{\mathcal{R}_v} (v, w_{i'})$. Remember that $e' \ll_{\mathcal{C}_v} e$. We make the same case differentiation as before: either e is inserted at the head before e' (Fig. 2.35(c)), e' is inserted at the tail before e, or e is inserted at the tail and e' at the head (Fig. 2.35(d)). As before, dipath $p = u \xrightarrow{*} v$ and edge e enclose a bounded region R such that e' enters one of its endpoints within R and one outside which leads to a crossing. Note that in this case R encloses the subtree of $w_{i'}$.

Now, we assume that $(v, w_{i'}) <_{\mathcal{R}_v} e$. Edge $(v, w_{i'})$ comes before e in line 3 of Split. Further, in the iteration of $(v, w_{i'})$, the case in line 4 applies. In the loop in line 9, subsequently



(a) Both edges are inserted at the head.

(b) Edge e is inserted at the tail, e' at the head.

(c) Both edges are inserted at the head.

(d) Edge e is inserted at the tail, e' at the head.

(e) Both edges are inserted at the head.

(f) Edge e is inserted at the tail, e' at the head.

(g) Both edges are inserted at the head.

(h) Edge e is inserted at the tail, e' at the head.

Figure 2.35: The situations obtained in the proof of Lem. 2.7, where the SD cannot be split adequately and `Split` returns $\perp$.

all edges at the head of $\mathcal{C}_v$ are extracted until there is an edge $e'' = \mathcal{C}_v.\texttt{head}() \neq e'$ with no endpoint in the subtree of $w_{i'}$. Therefore, $e'' \ll_{\mathcal{C}_v} e' \ll_{\mathcal{C}_v} e$. Moreover, in a subsequent iteration, e'' is successfully extracted from $\mathcal{C}_v$. If e'' is incident to v, then $(v, w_{i'}) <_{\mathcal{R}_v} e''$. Let u'' be the vertex at which e'' is inserted. We obtain the cases depicted in Figs. 2.35(e) and 2.35(f), where in both e'' and the dipath $u'' \xrightarrow{+} v$ enclose a region R (shaded) such that one endpoint of e' lies inside of R and the other one outside. Thus, e'' and e' cross. If e'' is not incident to v, then there is a child $w_{i''}$ such that e'' is extracted from $\mathcal{C}_v$ in line 10 when $(v, w_{i''})$ is the current edge in line 3. We get that $(v, w_{i'}) <_{\mathcal{R}_v} (v, w_{i''})$ and, remember, $e'' \ll_{\mathcal{C}_v} e'$. Then, the cases in Figs. 2.35(g) and 2.35(h) apply, where $w_{i'}$ and e' assume the roles of w_i and e, and $w_{i''}$ and e'' assume the roles of $w_{i'}$ and e'. Again, e'' and e' must cross.

For the remainder of the proof, suppose that `Split` returns $\perp$ in line 17 and denote by e the edge at the head of $\mathcal{C}_v$ at this time instant. Edge e cannot be incident to v as otherwise e would either have been extracted from $\mathcal{C}_v$ in line 12 or $\perp$ would have been returned in line 13. Let u and x be the endpoints of e with dipath $p = u \xrightarrow{+} v \rightarrow w_i \xrightarrow{*} x$. As e is still in $\mathcal{C}_v$, e could not be extracted and appended in line 10 when it was the turn of edge (v, w_i) in line 3. For this iteration, let $e' \neq e$ be the edge at the head of $\mathcal{C}_v$ when the loop in line 9 aborted. We get $e' \ll_{\mathcal{C}_v} e$. If e' is incident to v, then $(v, w_i) <_{\mathcal{R}_v} e'$. Similar to before, there is a bounded region R enclosed by the dipath $p = u \xrightarrow{+} v \rightarrow w_i \xrightarrow{*} x$ and e such that e' enters one of its endpoint within R and one outside of R which leads to a crossing. Finally, assume

that e' is not incident to v. Edge e' is extracted from $\mathcal{C}_v$ and appended to the piece $c_{w_{i'}}$ belonging to child $w_{i'}$ in line 10. We assume that u' and x' are the endpoints of e' with dipath $p = u' \xrightarrow{+} v \rightarrow w_{i'} \xrightarrow{*} x'$. Note that $w_i \neq w_{i'}$ as, by assumption, e' has not been extracted when (v, w_i) was the current edge of the iteration. Hence, $(v, w_i) <_{\mathcal{R}_v} (v, w_{i'})$ and we obtain the cases as shown in Figs. 2.35(g) and 2.35(h). By the same line of arguments as before, dipath p and edge e enclose a bounded region R such that e' enters one of its endpoints within R and one outside of R. Again, e and e' must cross. □

Lemma 2.8. *If* `Split` *returns pieces $c_{w_1}, \dots, c_{w_\ell}$, then no edges from distinct pieces cross.*

Proof. Let e and e' be two forward edges with $e \in c_{w_i}$ and $e' \in c_{w_{i'}}$ such that $w_i \neq w_{i'}$. Further, let u and u' be the endpoints of e and e', respectively, such that u and u' are ancestors of v. We prove the contrapositive and assume that a crossing between e and e' is inevitable. There are three cases:

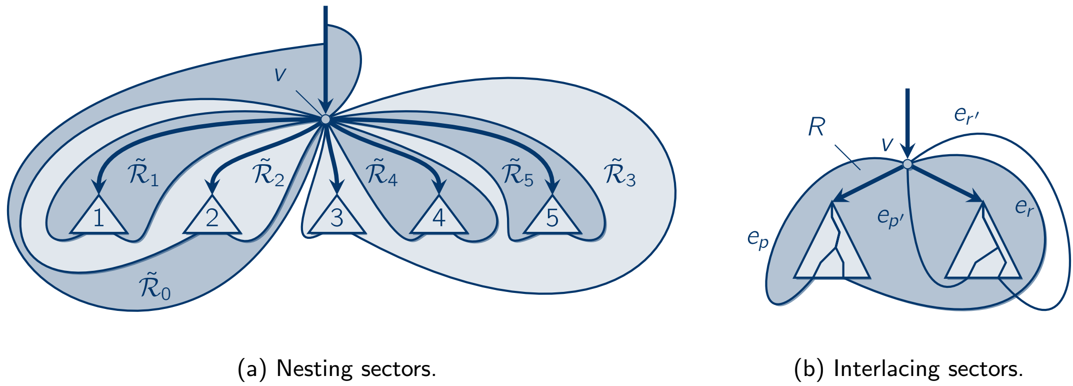
- **Both e and e' are incident to v**
  Let R be the bounded region enclosed by the dipath $p = u \xrightarrow{+} v$ and edge e. If a crossing between e and e' is inevitable, then e' enters u within R and v outside of R, or vice versa. This corresponds to the cases depicted in Figs. 2.35(a) and 2.35(b), and the version of Fig. 2.35(a) where both e and e' enter u and u' from the right side, respectively, and e' is inserted before e. In all cases, we obtain $e' \ll_{\mathcal{C}_v} e$ and $e <_{\mathcal{R}_v} e'$. Hence, when it is the turn of e in line 3, e cannot be extracted from $\mathcal{C}_v$ in line 12 as $e' \ll_{\mathcal{C}_v} e$, and $\perp$ is returned.

- **Exactly one of e or e' is incident to v**
  W. l. o. g., we assume that e is incident to v and e' is removed at vertex x' with dipath $p = u' \xrightarrow{+} v \rightarrow w_{i'} \xrightarrow{*} x'$. As in the previous case, e together with $p = u \xrightarrow{+} v$ encloses the bounded region R. First, we assume that $e <_{\mathcal{R}_v} (v, w_{i'})$. If a crossing between e and e' is inevitable, then we obtain one of the situations in Figs. 2.35(c) and 2.35(d), and the version of Fig. 2.35(c) where both e and e' enter u and u' from the right side, respectively, and e' is inserted before e. In all cases, $e' \ll_{\mathcal{C}_v} e$. When e is the current edge in line 3, e cannot be extracted in line 12 as $e' \ll_{\mathcal{C}_v} e$ and as e' can only be extracted in a subsequent iteration. Hence, $\perp$ is returned.

  The case $(v, w_{i'}) <_{\mathcal{R}_v} e$ corresponds to Figs. 2.35(e) and 2.35(f), where e assume the role of e'' in Figs. 2.35(e) and 2.35(f). For these cases, $e \ll_{\mathcal{C}_v} e'$ and e' cannot be extracted in line 10 when it is the turn of $(v, w_{i'})$ in line 3. Hence, e' cannot be extracted at all and $\perp$ is returned at the latest in line 17.

- **Neither e nor e' is incident to v**
  Then, e is removed in the subtree of w_i and e' in the subtree of $w_{i'}$. Similar to before, if a crossing is inevitable, we obtain one of the situations in Figs. 2.35(g) and 2.35(h) or the version of Fig. 2.35(h), where e and e' enter u and u' from the right side, respectively, and e' is inserted before e. Again, in all cases, $e' \ll_{\mathcal{C}_v} e$ and $(v, w_i) <_{\mathcal{R}_v} (v, w_{i'})$. Edge e' can only be extracted from $\mathcal{C}_v$ in line 10 when $(v, w_{i'})$ is the current edge in line 3. The same holds for e and (v, w_i). When it is the turn of (v, w_i), then e cannot be extracted from $\mathcal{C}_v$ as $e' \ll_{\mathcal{C}_v} e$. Later, when $(v, w_{i'})$ is the edge of the iteration, e' may be extracted from $\mathcal{C}_v$. However, after all edges are processed, $e \in \mathcal{C}_v$ in line 17 and, hence, $\perp$ is returned.

□



(a) Nesting sectors.

(b) Interlacing sectors.

Figure 2.36: Nesting and interlacing sectors.

Phase 2 In the second phase, `IsSDLayout` determines the edges that have to be processed in each piece c_i as obtained by `Split` and, en route, `IsSDLayout` tests whether edges incident to v cross. Suppose that `Split` successfully returned pieces $c_1, \dots, c_\ell$. Further, assume for the moment that v has parent u in $\mathcal{T}$. Then, $\mathcal{R}_v = (e_1, \dots, e_k)$ is the rotation system of v according to line 10 in Alg. 2.5 with $e_1 = (u, v)$. For any $0 \leq i \leq \ell$, a *sector* $\tilde{\mathcal{R}}_i$ is the subsequence $\tilde{\mathcal{R}}_i = (e_p, \dots, e_q)$ $(p \leq q)$ of $\mathcal{R}_v$ where, for $i \geq 1$, $\tilde{\mathcal{R}}_i$ contains all edges $e \in \mathcal{R}_v$ which are either incident to w_i or a descendant of w_i, and for $i = 0$, $\tilde{\mathcal{R}}_0$ contains all edges $e \in \mathcal{R}_v$ which are incident to an ancestor of v. Remember, in contrast to a substring, the elements in a sector, which is a subsequence of $\mathcal{R}_v$, must not be directly consecutive in $\mathcal{R}_v$. If v is the root, then $\tilde{\mathcal{R}}_0$ is empty and e_1 is the edge from v to w_1 (line 8). In a planar rotation system, the sectors of $\mathcal{R}_v$ properly nest or are disjoint. Fig. 2.36(a) illustrates this where vertex v has five children corresponding to five subtrees. The sectors are shaded regions, where $\tilde{\mathcal{R}}_2$ encloses $\tilde{\mathcal{R}}_1$, and $\tilde{\mathcal{R}}_3$ encloses $\tilde{\mathcal{R}}_4$ and $\tilde{\mathcal{R}}_5$, whereas $\tilde{\mathcal{R}}_4$ and $\tilde{\mathcal{R}}_5$ are disjoint. $\tilde{\mathcal{R}}_0$ plays a special role as it encloses $\tilde{\mathcal{R}}_1$ and $\tilde{\mathcal{R}}_2$ (dark shaded) as well as $\tilde{\mathcal{R}}_3$, $\tilde{\mathcal{R}}_4$, and $\tilde{\mathcal{R}}_5$ (outer face). Let $\tilde{\mathcal{R}}_i = (e_p, \dots, e_q)$ and $\tilde{\mathcal{R}}_{i'} = (e_{p'}, \dots, e_{q'})$ be two sectors. We say that $\tilde{\mathcal{R}}_i$ and $\tilde{\mathcal{R}}_{i'}$ *interlace* if there exist edges $e_r \in \tilde{\mathcal{R}}_i$ and $e_{r'} \in \tilde{\mathcal{R}}_{i'}$ with $p < p' < r < r'$ (see Fig. 2.36(b)) or $r < r' < q < q'$.

Lemma 2.9. *No pair of edges from distinct sectors cross if and only if no sectors interlace.*

Proof. $\Rightarrow$: Assume for contradiction that the rotation system of the graph is planar but the sectors $\tilde{\mathcal{R}}_i = (e_p, \dots, e_q)$ and $\tilde{\mathcal{R}}_{i'} = (e_{p'}, \dots, e_{q'})$ interlace. Let $e_r \in \tilde{\mathcal{R}}_i$ and $e_{r'} \in \tilde{\mathcal{R}}_{i'}$ be edges with $p < p' < r < r'$. The reasoning for the case $r < r' < q < q'$ is similar. For the moment, we assume that both e_r and $e_{r'}$ are forward edges and $i, i' > 0$, i. e., none of e or e' points to an ancestor of v. The situation is illustrated in Fig. 2.36(b). There is a circle formed by v, e_p, e_r and a simple path between the endpoints of e_p and e_r distinct from v. In the drawing of G, this circle encloses a region R (shaded) which contains the endpoints of $e_{p'}$ and $e_{r'}$ that are distinct from v. As $p < p' < r < r'$, the edge curve of $e_{r'}$ enters v outside of R and enters its other endpoint inside which leads to a crossing; a contradiction. If e_r is a tree edge, then the reasoning is similar with the only difference that R is enclosed by e_p and dipath $v \xrightarrow{+} u$ where u is the endpoint of e_p distinct from v. If $i = 0$, then R is enclosed by e_p and dipath $u \xrightarrow{+} v$ where, again, u is the endpoint of e_p distinct from v.

$\Leftarrow$: If no sectors interlace, then all pairs of sectors $\tilde{\mathcal{R}}_i = (e_p, \dots, e_q)$, $\tilde{\mathcal{R}}_{i'} = (e_{p'}, \dots, e_{q'})$ with $p < p'$, are either disjoint, i. e., $p \leq q < p' \leq q'$, or nesting, i. e., $p < p' < q' < q$. In this case, we can always avoid crossings between the edges belonging to distinct sectors of v. □

In the following, we call the subscript i of $\tilde{\mathcal{R}}_i$ the *index* of $\tilde{\mathcal{R}}_i$. Remember that two sectors do not interlace if they are either disjoint or nesting. `IsSDLayout` exploits this by processing the indices of the sectors in a stack to test for interlacing sectors. The index i that is on top of the stack is the currently "active" sector $\tilde{\mathcal{R}}_i$, where initially the stack contains index 0 (line 17). Additionally, `IsSDLayout` maintains the boolean variable removed(i), initialized to `false` in line 16, which stores if i has already been removed from the stack.

En route, `IsSDLayout` computes a *partial rotation system* ρ_i for each child w_i $(1 \leq i \leq \ell)$, where ρ_i contains all edges incident to v that are later inserted to and removed from c_{w_i}. Equal to sectors, a partial rotation system ρ_i is a subsequence of $\mathcal{R}_v$ which contains all edges $e \in \mathcal{R}_v$ that are *assigned* to w_i, i. e., either $e \in c_{w_i}$ and, thus, e is removed at v from c_{w_i}, or e is removed in the subtree of w_i and, therefore, must be inserted to c_{w_i}. Note that for any $1 \leq i \leq \ell$ the partial rotation system ρ_i may contain edges to ancestors of v whereas the sector $\tilde{\mathcal{R}}_i$ does not. Each ρ_i is initialized to $()$ in line 15.

`IsSDLayout` subsequently processes all edges incident to v in order of v's rotation system (line 18). In line 19, the child w_i to which the current edge e is assigned, is determined and e is appended to ρ_i. Next, the index of the currently "active" sector t that must be on top of the stack is determined. Let u be the endpoint of e distinct from v. If u is an ancestor of v, then t is set to 0 as $e \in \tilde{\mathcal{R}}_0$. Otherwise, t is set to i, i. e., $e \in \tilde{\mathcal{R}}_i$. If $t \in \mathcal{S}$, all indices are removed from $\mathcal{S}$ until t is on top. Note if $t = 0$, the stack is emptied except for 0. For all removed indices i', removed(i') is set to `true`. If $t \notin \mathcal{S}$ and removed$(i') =$ `false`, t is pushed onto the stack (line 29), and if i' has been previously removed, `false` is returned (line 30).

Lemma 2.10. `IsSDLayout` *returns* `false` *in line 30 if and only if at least two sectors interlace.*

Proof. $\Rightarrow$: Assume that `IsSDLayout` returns `false` in line 30 and let i be the index of the currently active sector. Hence, $i \notin \mathcal{S}$ and removed$(i) =$ `true`. Index i had been removed in a previous iteration when another index i' further below in the stack needed to be on top. Let $\tilde{\mathcal{R}}_i = (e_p, \dots, e_q)$ and $\tilde{\mathcal{R}}_{i'} = (e_{p'}, \dots, e_{q'})$ be the sectors corresponding to i and i', respectively. Index i' had been inserted to $\mathcal{S}$ before i and, thus, $p' < p$. In the iteration when i is removed from $\mathcal{S}$, the edge of the iteration is $e_{r'} \in \tilde{\mathcal{R}}_{i'}$. Further, when `IsSDLayout` returns `false` in line 30, the current edge of the iteration is $e_r \in \tilde{\mathcal{R}}_i$ with $r' < r$. Altogether we get $p' < p < r' < r$ and, hence, $\tilde{\mathcal{R}}_i$ and $\tilde{\mathcal{R}}_{i'}$ interlace.

$\Leftarrow$: Let $\tilde{\mathcal{R}}_i = (e_p, \dots, e_q)$ and $\tilde{\mathcal{R}}_{i'} = (e_{p'}, \dots, e_{q'})$ be interlacing sectors. Then, there exist edges $e_r \in \tilde{\mathcal{R}}_i$ and $e_{r'} \in \tilde{\mathcal{R}}_{i'}$ with $p < p' < r < r'$ or $r < r' < q < q'$. We assume $p < p' < r < r'$; the case $r < r' < q < q'$ is similar. As $p < p'$, i' is above i in $\mathcal{S}$, after the iteration in which $e_{p'}$ is processed in line 18. When it is e_r's turn (or before), i' is removed from $\mathcal{S}$ and removed(i') is set to `true`. Later, when $e_{r'}$ is the current edge of the iteration, $i' \notin \mathcal{S}$ and it must be reinserted. However, as removed$(i') =$ `true`, `false` is returned in line 30. □

Phase 3 In the last phase, for each child $w_1, \dots, w_\ell$, `IsSDLayout` obtains the input content $\mathcal{C}_{w_i}$ for w_i from c_{w_i} and recursively calls itself (lines 31 to 35). For each w_i, let $E_i^{\mathtt{h}}(v)$ and $E_i^{\mathtt{t}}(v)$ be the deque schedule of v as induced by the dipath $r \xrightarrow{*} v \rightarrow w_i$ and the partial rotation system ρ_i (line 32). In line 33, `ProcessVertex` is called with c_{w_i}, the linear layout $\prec_\mathcal{T}$ induced by $\mathcal{T}$, and the deque schedule $E_i^{\mathtt{h}}(v)$ and $E_i^{\mathtt{t}}(v)$ as parameters to obtain the input $\mathcal{C}_{w_i}$ of child w_i. If

the return value $\mathcal{C}_{w_i}$ of `ProcessVertex` is $\perp$, not all edges could be processed and the rotation system is not planar by Cor. 2.4. Thus, `false` is returned in line 34. Otherwise, `IsSDLayout` is called recursively with w_i and input deque $\mathcal{C}_{w_i}$ (line 35). If all calls of `IsSDLayout` return `true`, the given SD schedule is an SD layout. We now prove Lem. 2.5.

Proof. [Lem. 2.5] $\Leftarrow$: If `IsSDLayout` returns `false`, then either `ProcessVertex` has returned $\perp$ (lines 4 and 33), `Split` has returned $\perp$ (line 13), or there are interlacing sectors (line 30). By Cor. 2.4 and Lemmas 2.6, 2.7 and 2.10, respectively, $\mathcal{R}$ is not planar.

$\Rightarrow$: We assume that `IsSDLayout` returns `true` and show that no pair of edges cross. Let $e = \{u, x\}$ and $e' = \{u', x'\}$ be two distinct forward edges, where u and u' are ancestors of x and x', respectively. Remember that tree edges are never involved in a crossing by assumption. We distinguish two cases: either there is a time instant, when e and e' are together in the SD, or e and e' are never in the same SD.

We start with the first case. W. l. o. g., suppose that e' is inserted into the SD at vertex u' such that e is already present in the SD. If e is removed from the SD at vertex u', i. e., $x = u'$, then there is a root-to-leaf dipath p which contains all endpoints of e and e'. Thus, e and e' are processed as in a deque layout induced by the dipath p and, e and e' do not cross by Cor. 2.4. Otherwise, there is a child w of u' with input $\mathcal{C}_w$ such that $e, e' \in \mathcal{C}_w$. We further distinguish two cases: In the first case, there is a vertex v with input $\mathcal{C}_v$ and $e, e' \in \mathcal{C}_v$ such that v has at least two children w_i and $w_{i'}$ and $\mathcal{C}_v$ is split into pieces $c_{w_1}, \dots, c_{w_\ell}$ with $e \in \mathcal{C}_{w_i}$ and $e' \in \mathcal{C}_{w'_i}$. By assumption, `Split` has successfully split $\mathcal{C}_v$ and e and e' do not cross by applying Lem. 2.8. In the second case, e (e') is removed in `ProcessVertex` while e' (e) is still in the SD. Then, there is a root-to-leaf dipath p such that all endpoints of e and e' are on p, and e and e' are processed as in a deque layout. By Cor. 2.4, we conclude that e and e' do not cross.

Next, suppose that e and e' are never in the same SD at any time instant. Again, there are two cases: either e and e' have no common endpoint or they have a common endpoint v. In the first case, one of the following three things can happen:

- There is a root-to-leaf dipath p with either $p = r \overset{*}{\to} u \overset{+}{\to} x \overset{+}{\to} u' \overset{+}{\to} x' \overset{*}{\to} v_\ell$ or $p = r \overset{*}{\to} u' \overset{+}{\to} x' \overset{+}{\to} u \overset{+}{\to} x \overset{*}{\to} v_\ell$, where r is the root and v_ℓ a leaf of the tree layout.
- There are two vertices w and w' such that all endpoints of e and e' are in the subtrees of w and w', respectively, while the subtrees of w and w' are completely disjoint, i. e., they share no vertex.
- There is a vertex v such that e is inserted at an ancestor u of v and removed in the subtree of child w_i of v, i. e., $u \overset{+}{\to} v \to w_i \overset{*}{\to} x$, and e' is inserted at $v = u'$ and removed in the subtree of child $w_{i'}$ with $w_i \neq w_{i'}$, i. e., $v \to w_{i'} \overset{+}{\to} x'$.

In the first two cases, let R be the bounded region enclosed by $u \overset{*}{\to} x$ and edge e. In both cases, all endpoints of e' lie outside of R and, thus, a crossing between e and e' is always evitable. In the third case, e is an edge which is in $\mathcal{C}_v$ but not incident to v and e' is an edge with $e' \notin \mathcal{C}_v$ which is incident to v, where e and e' are removed in different subtrees of v. Here, Lem. 2.6 applies and `Split` guarantees that e and e' do not cross.

Finally, assume that e and e' are never in the same SD and have a common endpoint v. As e and e' are never in an SD at the same time, e and e' must lie in different sectors of v: Towards a contradiction, suppose that e and e' are in the same sector. If $e, e' \in \tilde{\mathcal{R}}_0$, then both have been inserted at an ancestor of v and $e, e' \in \mathcal{C}_v$; a contradiction. Analogously, if

$e, e' \in \tilde{\mathcal{R}}_i$ for some $i \geq 1$, then both are inserted to piece c_{w_i} for child w_i of v and, thereby, $e, e' \in \mathcal{C}_{w_i}$; again, a contradiction. Hence, e and e' lie in different sectors of v. By assumption these sectors do not interlace and e and e' do not cross by Lem. 2.10. □

IsSDLayout obeys the modus operandi of the SD as defined in Sect. 2.5.2. Therefore, a graph G is an SD graph if and only if G has an SD layout. This holds true if and only if G is planar and Thm. 2.7 follows.

Whenever IsSDLayout returns false, we can find the culprit edges: If one of the calls of ProcessVertex returns $\perp$, then an edge could not be removed from the deque since at least one other edge is blocking its way, and these edges must cross. If the SD cannot be split adequately (line 12), then we obtain one of the situations in Figs. 2.34 and 2.35 and we identify the crossing edges as in the proofs of Lemmas 2.6 and 2.7. Last, if a previously removed index i must be reinserted to $\mathcal{S}$ (line 30), there are interlacing sectors $\tilde{\mathcal{R}}_i$ and $\tilde{\mathcal{R}}_{i'}$ and crossing edges $e \in \tilde{\mathcal{R}}_i$ and $e' \in \tilde{\mathcal{R}}_{i'}$ according to the proof of Lemmas 2.9 and 2.10.

2.5.4 Remarks on the Running Time of IsSDLayout

IsSDLayout is a means for proving the characterization of planarity by the SD. As such, we have not lost a word on the running time of IsSDLayout. We make good for this now.

Let $G = (V, E)$ be a graph endowed with a rotation system $\mathcal{R}$. We assume that $|E| \leq 3|V| - 6$ as otherwise G is not planar by Euler's formula. Prior to calling IsSDLayout, DFS in Alg. 2.4 is used to determine a tree layout $\mathcal{T}$ in time $\mathcal{O}(|V|)$. First, we show that the running time of IsSDLayout is in $\mathcal{O}(|V|)$ if we ignore the running time of Split called in line 12. Afterwards, we discuss the running time of Split.

Assume that IsSDLayout is called for vertex v and let $E_v := E(v)$ be the set of v's incident edges. Determining the deque schedule and processing all edges in case v is a leaf (lines 2 to 6) runs in time $\mathcal{O}(|E_v|)$ as each edge in E_v is processed at most a constant amount of times. The same running time applies in lines 7 to 10 and line 11 when "splitting" v's rotation system $\mathcal{R}_v$ and ordering v's children according to $\mathcal{R}_v$.

The loop starting in line 18 needs a more thorough investigation. In line 19, the child w_i to which e is assigned is determined. By introducing a pointer from each edge e to the SD where it currently resides, we find out whether there is a piece c_{w_i} with $e \in c_{w_i}$ in time $\mathcal{O}(1)$ and, if so, we obtain w_i. If e currently resides in no SD, then e is inserted at v and removed in the subtree of some child w_i. Let x be the vertex at which e is removed. We introduce some preprocessing to DFS (Alg. 2.4): Edge e is encountered by DFS at vertex x. At this point, the recursion step of DFS at vertex v has recursively called itself with parameter w_i, where w_i is the child of v in whose subtree e is removed. By introducing a pointer from e to w_i during the DFS traversal, we can determine w_i in time $\mathcal{O}(1)$ in line 19. Note that this also incorporates the case where e is the tree edge from v to w_i. Further, the index of each child of v is inserted exactly once to $\mathcal{S}$ and removed at most once. Hence, the number of push and pop operations on $\mathcal{S}$ is bounded by $|E_v|$. We can conclude that the running time of lines 18 to 30 is in $\mathcal{O}(|E_v|)$.

Let w_i be the child of the current iteration in line 31 with partial rotation system ρ_i. Line 32 and ProcessVertex in line 33 both run in time $\mathcal{O}(|\rho_i|)$, where $|\rho_i|$ is the number of edges in ρ_i. Ignoring the recursive call of IsSDLayout for the moment, the running time of lines 31 to 35 is in $\mathcal{O}(\sum_{i=1}^{\ell} |\rho_i|) \subseteq \mathcal{O}(|E_v|)$. Altogether, the running time of IsSDLayout without Split and the recursion is $\mathcal{O}(|E_v|)$ and, hence, the total running time of IsSDLayout is $\mathcal{O}(|E|) \subseteq \mathcal{O}(|V|)$.

Unfortunately, Split in Alg. 2.6 increases the running time of IsSDLayout to $\mathcal{O}(|V|^2)$: First observe that the running time of Split is $\mathcal{O}(|\mathcal{C}_v| + |E_v|)$ as all edges in E_v are processed

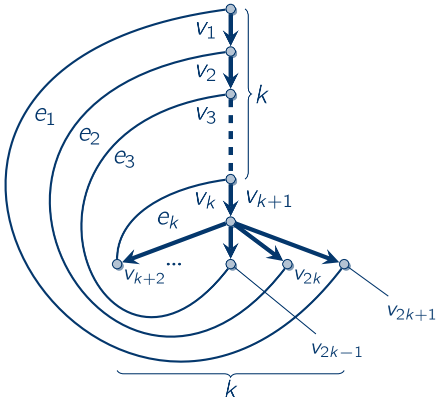


Figure 2.37: An example of a graph which leads to a quadratic running time of `IsSDLayout`.

in line 3 and all edges in $\mathcal{C}_v$ are extracted and inserted to the pieces of v's children. For each natural number $k \geq 0$, we construct a planar graph G_k with $2k+1$ vertices as shown in Fig. 2.37. The tree layout consists of a dipath $v_1, v_2, \ldots, v_{k+1}$, where vertex v_{k+1} has k children $v_{k+2}, v_{k+3}, \ldots, v_{2k+1}$. There are k forward edges and each forward edge connects vertex v_i with vertex v_{2k+2-i} for all $1 \leq i \leq k$. The rotation system is chosen according to Fig. 2.37 and it is planar. Let $\mathcal{C}_{v_i}$ be the input SD of vertex v_i with $1 < i \leq k$. $\mathcal{C}_{v_i}$ contains $i-1$ forward edges, which leads to a running time $\Theta(i)$ of `Split`. Thus, the running time of `Split` for all vertices $v_1, \ldots, v_k$ is in $\Theta(\sum_{i=1}^{k} i) = \Theta(k^2) \in \mathcal{O}(|V|^2)$. `IsSDLayout` has therefore a running time of $\mathcal{O}(|V|^2)$.

We can achieve a linear running time of `IsSDLayout` by using a slightly different approach which we briefly sketch now. Every data node representing an edge realizes the SD data structure by having pointers to its direct predecessor and successor in the SD. Hence, adding or removing an edge takes $\mathcal{O}(1)$ time. We tweak our approach by performing the splits lazily when an edge is to be extracted at the tail of the SD. For the case that the SD schedule is actually an SD layout, we lift the precondition that the SD content $\mathcal{C}_v$ passed to `IsSDLayout` consists exclusively of edges that are removed in the subtree of v. Instead, we only require that these edges form a contiguous prefix of $\mathcal{C}_v$. Before an edge is to be removed at the tail of the SD in line 8 of `ProcessVertex`, it checks whether it has a successor in the SD. If this is the case, the SD is split to the right of said edge. If multiple splits occur during a single run of `ProcessVertex`, the SD schedule is rejected by returning $\perp$. This modification does not change the behavior of the algorithm: If the SD schedule is an SD layout, each split occurs at the same location and only has been deferred. If the SD schedule is no SD layout, it may be the case that an edge e is successfully extracted which would otherwise (in the unmodified algorithm) have been in the middle of the SD. But then there are edges succeeding e in the SD which now end up in the wrong piece such that this instance will surely be rejected.

2.5.5 Relating Splittable Deque Layouts to the de Fraysseix-Rosenstiehl Planarity Criterion

The characterization of planarity by the SD is a close relative to the de Fraysseix-Rosenstiehl planarity criterion or LR planarity criterion [dFR82, dFdMR06] as both are based on the DFS

tree of a graph. A comprehensive treatment of the LR planarity criterion and the related planarity test is [Bra09] from which we also borrow some terminology and definitions.

For completeness, we state the LR planarity criterion before we relate it to SD layouts. For this, we need a batch of definitions. Let $\mathcal{T} = (V, E_\mathcal{T})$ be a DFS tree of a graph $G = (V, E)$ with forward edges F. What we call a forward edge in the context of SD layouts is called backward edge in [dFR82, dFdMR06, Bra09]. For consistency with this section, we stick to the term "forward edge", however, adopt the orientation from the descendant to the ancestor from [dFR82, dFdMR06, Bra09] as it allows for concise definitions. We denote by $v \hookrightarrow u$ a forward edge $e = \{u, v\}$ where u is the ancestor of v and we assume that e points from v to u. As an example, we consider Fig. 2.33(c) on page 71 again without the dashed edge. Each forward edge $u \hookrightarrow v$ defines a *fundamental circle* $u \xrightarrow{+} v \hookrightarrow u$ in G. Two distinct fundamental circles are either disjoint, e. g., the ones of e_8 and e_9 in Fig. 2.33(c), or their intersection is a dipath of tree edges, e. g., for e_1 and e_3, with $2 \xrightarrow{+} 4$. In the latter case, the last common tree edge on this dipath is called *fork*, e. g., tree edge $t_4 = (3, 4)$ is the fork of e_1 and e_3. For any tree edge $u \to v$, its *return edges* are all forward edges $x \hookrightarrow w$ with $w \xrightarrow{+} u \to v \xrightarrow{*} x$. For instance, the return edges of $t_4 = (3, 4)$ are e_1, e_2, e_3, and e_4. A forward edge is the single return edge of itself. Let $\prec_\mathcal{T}$ be linear layout induced by $\mathcal{T}$, e. g., according to a DFS numbering. For any forward edge $e = v \hookrightarrow u$, $\text{low}(e) := u$ is called *low point of* e. For any tree edge $e \in E_\mathcal{T}$ the *low point* $\text{low}(e)$ *of* e is the minimum low point of e's return edges with respect to the linear layout $\prec_\mathcal{T}$, e. g., $\text{low}(3, 4) = 1$ and $\text{low}(4, 11) = 2$. Remember that all tree edges point from the parents to their children and all forward edges from descendant to ancestor.

Definition 2.12 (LR Partition [dFR82, dFdMR06]). *Let $L \dot\cup R$ be a partition of F. (L, R) is called* left-right partition (LR partition) *if for every fork $(u, v) \in E_\mathcal{T}$ and all pairs of outgoing edges e_1, e_2 of v, the following conditions hold:*

(i) All return edges $e \in F$ of e_1 with $\text{low}(e_2) \prec_\mathcal{T} \text{low}(e)$ belong to L.

(ii) All return edges $e \in F$ of e_2 with $\text{low}(e_1) \prec_\mathcal{T} \text{low}(e)$ belong to R.

Proposition 2.8 ([dFR82, dFdMR06]). *A graph is planar if and only if it has an LR partition.*

The idea behind Def. 2.12 and Prop. 2.8 is that in an embedding all forward edges $v \hookrightarrow u$ in L enter u from the left side of the dipath $p = r \xrightarrow{*} u \xrightarrow{+} v$, where r is the root of $\mathcal{T}$. Likewise, all forward edges $v \hookrightarrow u$ in R enter u from the right side. As an example, consider fork $t_4 = (3, 4)$ and its outgoing edges $t_5 = (4, 5)$ and $t_{11} = (4, 11)$ in Fig. 2.33(c). The return edges of t_5 are e_2 and e_5, and the return edges of t_{11} are e_3 and e_6. For the low points, we get $\text{low}(t_5) = \text{low}(e_2) = 2$, $\text{low}(e_5) = 3$, $\text{low}(t_{11}) = \text{low}(e_3) = 2$, and $\text{low}(e_6) = 3$. Edge e_5 is a return edge of t_5 with $\text{low}(t_{11}) \prec_\mathcal{T} \text{low}(e_5)$, and e_6 is a return edge of t_{11} with $\text{low}(t_5) \prec_\mathcal{T} \text{low}(e_6)$. Hence, (i) and (ii) of Def. 2.12 apply for e_5 and e_6, and in any LR partition (L, R), e_5 must be in L and e_6 in R (or vice versa). Moreover, in Fig. 2.33(c), e_5 enters 3 from the left and e_6 enters 3 from the right side. Conversely, if both e_5 and e_6 would enter 3 from the same side, a crossing between e_5 and e_6 with other edges, e. g., e_2 and e_3, would be inevitable. An LR partition can directly be obtained from Fig. 2.33(c):

$$L = \{e_1, e_2, e_5, e_8\}, \qquad R = \{e_4, e_3, e_6, e_7, e_9\}.$$

Note that e_1 can alternatively also be in R.

Remember that in an SD layout, a forward edge $e = v \hookrightarrow u$ is inserted at the head (tail) if and only if e enters u from the left (right) side.

Definition 2.13. *Assume that G is a planar graph and let $\mathcal{T} = (V, E_{\mathcal{T}})$ be a tree layout of G with forward edges F. By Thm. 2.7, G has an SD layout $\Psi = (\mathcal{T}, \mathcal{R})$. Let $F_{\mathtt{h}} \dot\cup F_{\mathtt{t}}$ be a partition of the forward edges such that:*

$$e \in F_{\mathtt{h}} \Leftrightarrow e \text{ is inserted at the head in } \Psi\,, \qquad e \in F_{\mathtt{t}} \Leftrightarrow e \text{ is inserted at the tail in } \Psi\,.$$

$(F_{\mathtt{h}}, F_{\mathtt{t}})$ is called `ht` *partition induced by $\Psi = (\mathcal{T}, RS)$.*

We can relate the LR planarity criterion to SD layouts as follows:

Corollary 2.13. *An* `ht` *partition $(F_{\mathtt{h}}, F_{\mathtt{t}})$ induced by an SD layout $\Psi = (\mathcal{T}, \mathcal{R})$ of a graph G is an LR partition of G and $\mathcal{T}$.*

Conversely, from an LR partition, we obtain at which sides the edges must be inserted.

The major differences between the LR planarity criterion and SD layouts are that the latter directly characterize planar rotation systems and, hence, must also incorporate crossings between edges that are incident to the same vertex. These types of crossings are neglected in Def. 2.12 and Prop. 2.8 also because LR partitions are intended to be used for general planarity testing (without a given rotation system). Also note that an LR partition defines the side at which an edge is inserted to the SD but not at which side it is removed. Again, the reason is that the LR planarity criterion does not characterize the planarity of rotation systems: As soon as the side at which a forward edge $e = v \hookrightarrow u$ enters u is specified, i. e., it either belongs to L or R, the side at which e enters v automatically follows or does simply make no difference for planarity. This is not the case if the planarity of a rotation system has to be tested.

2.5.6 Testing Planarity by Switching Trains

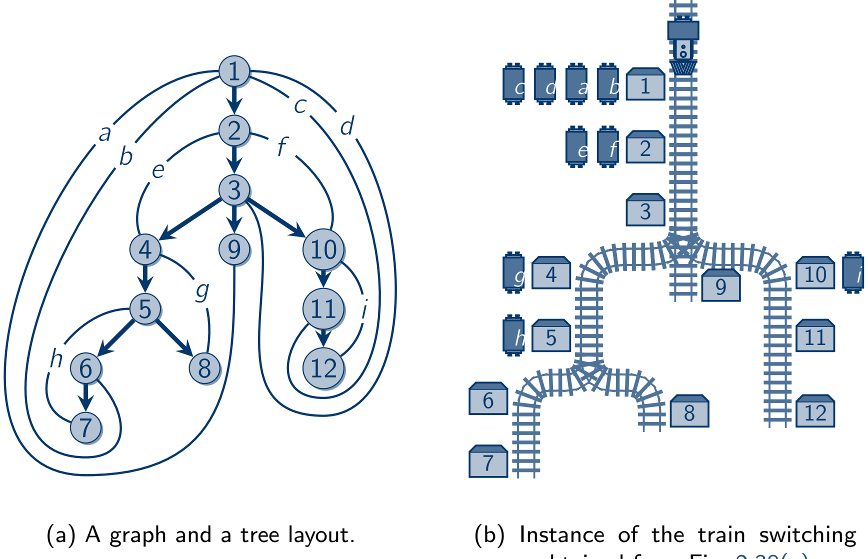
We wrap this section up with a playful insight to planarity: Fig. 2.38(a) shows a graph G endowed with a tree layout $\mathcal{T}$. G and $\mathcal{T}$ can be transformed into a train switching problem as shown in Fig. 2.38(b): The SD becomes a train (locomotive at top) that follows a railway which is obtained from the tree layout. At each train station (the vertices), cars (the edges) are appended and prepended to the train. At a junction, e. g., train station 3, the train is split into parts and each part follows one of the branches. The aim of the game is to transport all cars from their source train station to their destination train station using the modus operandi of the SD. If this is possible, we say that the train switching problem is *solvable*.

Corollary 2.14. *Let G be a graph with tree layout $\mathcal{T}$. G is planar if and only if the train switching problem obtained from G and $\mathcal{T}$ is solvable.*

In other words, the train switching problem is equivalent to planarity testing. We have presented this idea in [AGHV12].

Besides [AGHV12], we have also implemented the game *Derail* which was our contribution to the graph drawing game contest at the graph drawing symposium in 2012 in Redmond, Seattle, USA. For Derail, we have used the time-inverted variant of the SD, namely, the *mergeable deque* (MD). An MD layout starts with empty MDs, i. e., trains, at the leaves and follows the tree layout towards the root. Instead of splitting, the trains have to be merged at junctions. Derail is available for download[1] and screenshots are shown in Fig. 2.39. In Derail, a level is generated from graph and a tree layout. If it is not possible to remove a car from either

[1] `http://www.infosun.fim.uni-passau.de/br/games/derail.jar` (last accessed 2013-12-02)



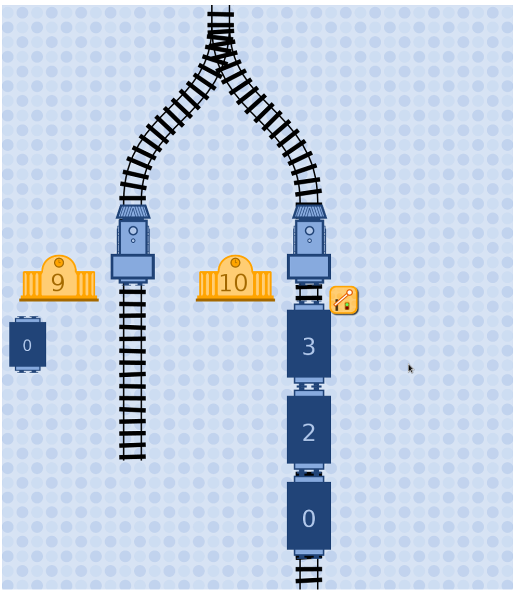
(a) A graph and a tree layout.

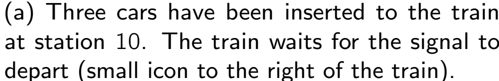
(b) Instance of the train switching game obtained from Fig. 2.38(a).

Figure 2.38: A graph and a tree layout can be transformed into a train switching game.

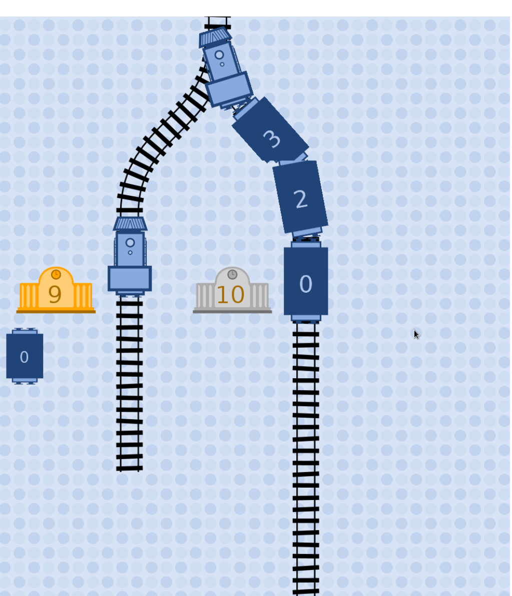
side of a train, then Derail swaps the car's position with its neighboring cars until it is at the head or the tail (Fig. 2.39(d)). Each such swap results in a penalty score for the player. The aim is to solve the game with as few swaps and as quickly as possible.

At this point it is worthwhile to point out a difference between the train switching problem here and the switchyard problem as defined by Knuth [Knu97] and as discussed in Sect. 2.1.2.1 in the context of permutation networks. In our train switching problem, the deque models the train whereas in the switchyard problem the used data structure, e. g., stack, queue, or deque, models the train tracks in the switchyard.

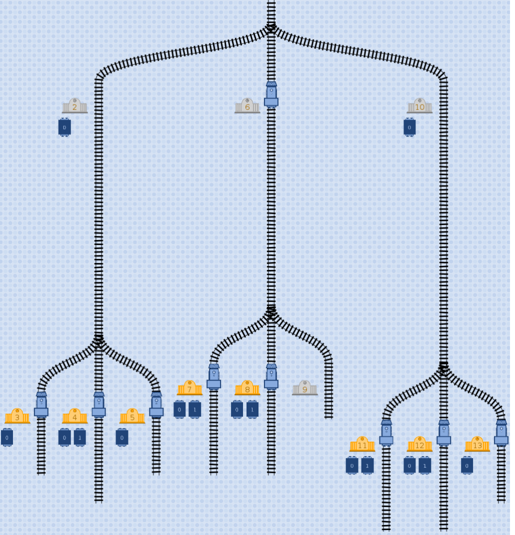




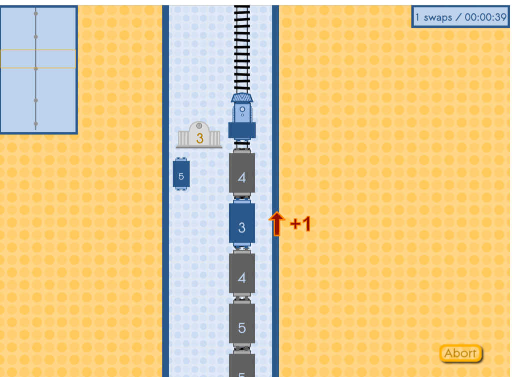
(a) Three cars have been inserted to the train at station 10. The train waits for the signal to depart (small icon to the right of the train).



(b) The train has departed from station 10. Further upwards, it is merged with the train from station 9.



(c) A more complex instance of the train switching problem.



(d) The car with destination 3 has to be swapped with its neighbors in order to remove it.

Figure 2.39: The computer game Derail implements the (time-inverted version of the) train switching problem.

2.6 Summary, Further Remarks, and Future Work

At the end of the first part, we take the opportunity to look back and see what we have achieved and in which direction future research can go.

Data Structure	Characterizations	Complexity of Decision Problem
stack	outerplanar [BK79]	linear time [BK79, Wie87]
queue	▶ arched leveled-planar [HR92] ▶ **spanning subgraphs of self-dual exuberant $\prec$-augmentation (Cor. 2.11)**	$\mathcal{NP}$-complete [HR92]
two stacks	subgraph of a planar graph with a Hamiltonian circle [BK79]	$\mathcal{NP}$-complete [Chv85, Wig82]
deque	▶ **linear cylindric planar (Thm. 2.1)** ▶ **subgraph of a planar graph with a Hamiltonian path (Thm. 2.2)**	**$\mathcal{NP}$-complete (Thm. 2.5)**
splittable deque	**planar (Thm. 2.7)**	linear time [KW01]

Table 2.2: Updated overview.

We have extended the study of stack, queue and two-stack layouts to deque layouts and splittable deque layouts. Tab. 2.2 shows an updated version of Tab. 2.1 on page 21, where the new results are emphasized (bold). To study deque graphs, we have introduced linear cylindric drawings and characterized the deque graphs as the linear cylindric planar graphs. For this, we have used a low-level approach that relates the rotation system in a linear cylindric drawing to the order in which the edges are processed in the deque. By this characterization, we have also found out that the deque graphs are exactly the subgraphs of planar graphs with a Hamiltonian path. Hence, the ability to process queue items in the deque corresponds to the difference between Hamiltonian paths and circles in planar graphs. We have also studied the duals of planar graphs with a Hamiltonian path. The queue edges are exactly those that are "crossed" when we connected the endpoints of the Hamiltonian path. As for the complexity of deciding whether a graph is a deque graph, we proved that it is $\mathcal{NP}$-complete by showing that finding a Hamiltonian path in a maximal planar graph is $\mathcal{NP}$-complete. We have also seen that linear cylindric drawings are not only useful to study deque graphs but also yield concise visualizations of layouts in deque-reducible data structures and mixed layouts with multiple deque-reducible data structures.

Our findings on deque layouts have interesting implications on queue layouts. First, we have seen the benefits of linear cylindric drawings over other approaches to visualize queue layouts. Second, our study of the duals of deque graphs paved the way for a novel characterization of queue graphs: a graph is a queue graph if and only if it is the spanning subgraph of a graph H which has an augmented queue embedding, and this holds true if and only if H^* has an augmented queue embedding. In other words, the duals of augmented queue graphs are also augmented queue graphs and, thus, selfdual.

As there are planar graphs lacking a Hamiltonian path, we asked the question how to extend the deque in order to characterize planarity. We have found the answer by introducing a split operation, and showed that a graph is planar if and only if it is a splittable deque graph. For the proof, we devised an algorithm that uses the splittable deque to test the planarity of a rotation system.

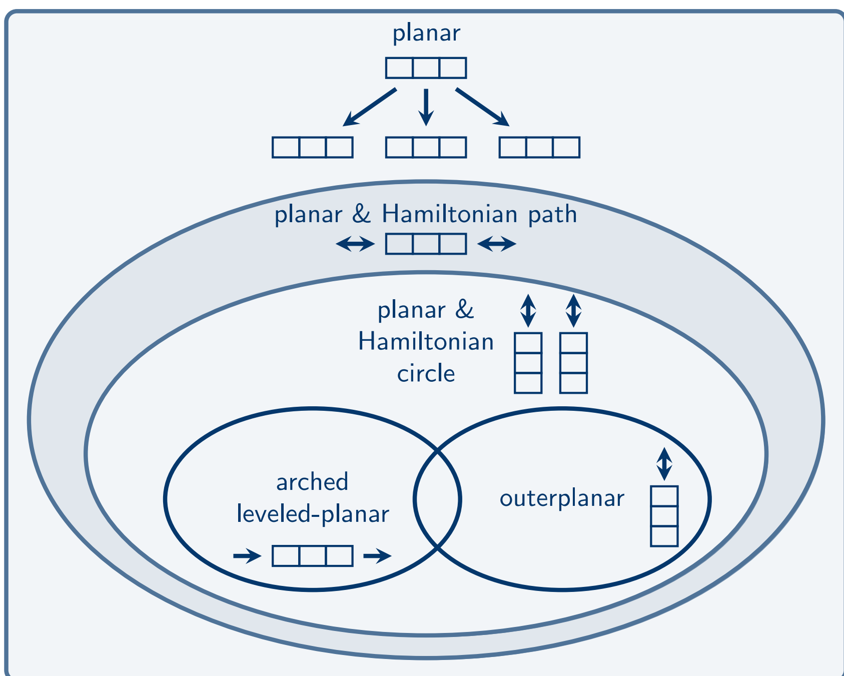


Figure 2.40: The classification of planar graphs by data structures, and the classification of data structures by classes of planar graphs.

By extending graph layouts to the deque, we have not only gained new insights to the working principle of the deque itself but also to two-stack and queue layouts, and to interesting classes of planar graphs. It has turned out that the queue-mode of the deque is reflected in its dual graph; a result with implications to queue graphs that, to the best of our knowledge, have not been considered yet. Further, by introducing the splittable deque, we have closed the gap between deque graphs and planar graphs. Our work, therefore, completes the "taxonomy of planar graphs by fundamental data structures" and, vice versa, the "taxonomy of data structures by classes of planar graphs" (see Fig. 2.40). We close this first part of the thesis with some further remarks and give pointers to possible future work.

2.6.1 Further Considerations of Complexity Issues

In Sect. 2.2.5, we have seen that deciding whether a graph is a deque graph is $\mathcal{NP}$-hard. Therefore, deciding whether a graph has a layout in k deques is also $\mathcal{NP}$-hard. Future research can go into two directions: Either restricted and special cases are considered, or parameters suitable for fixed-parameter algorithms are identified.

2.6.1.1 Restricted Problems

Given a $k \geq 3$ and a fixed linear layout, deciding whether a graph has a k-stack layout remains $\mathcal{NP}$-hard [DW04a]. The cases $k = 1$ and $k = 2$ are solvable in polynomial time [RT84]. For the deque, we know:

Corollary 2.15. *Given a graph $G = (V, E)$ and a linear layout $\prec$, deciding whether G has a deque layout $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ can be done in time $\mathcal{O}(|V|)$.*

Proof. First, we reject if $|E| > 3|V| - 6$, as G must be planar to have a deque layout. Then, we compute the $\prec$-augmentation $G_\prec$ of G and run a planarity test that runs in time $\mathcal{O}(|V|)$ [KW01]. G is a deque graph if and only if $G_\prec$ is planar by Cor. 2.4. □

It should come to no surprises for k-deque layouts:

Open Problem 2.1. *Prove the following: for a graph $G = (V, E)$, a linear layout $\prec$, and an integer $k \geq 2$, it is $\mathcal{NP}$-hard to decide whether G has a k-deque layout with linear layout $\prec$.*

The complementary problem is to ask for a suitable linear layout for a given assignment of the edges to k deques. The corresponding problem for k stacks is known to be solvable in linear time [HN09], which suggests that it might also be efficiently solvable for the deque.

Open Problem 2.2. *Is there an efficient algorithm that, given an assignment of the edges to k deques, computes a suitable linear layout?*

Special graph classes can also be considered. As deque graphs are planar, the usual suspects are outerplanar graphs and series-parallel graphs. However, the first are stack graphs [BK79] and latter are two-stack graphs [CLR87] and, hence, both are deque graphs. Another candidate is the class of planar 3-trees, also called Apollonian networks, which are obtained by starting with a triangle and subsequently subdividing a triangle into three smaller triangles. An example of a planar 3-tree is the Goldner-Harary graph in Fig. 2.20(a) on page 46 which contains no Hamiltonian circle but a Hamiltonian path. Planar 3-trees are maximal planar and deciding whether a planar 3-tree contains a Hamiltonian path can be done in polynomial time [CR01, EGW01a, EGW01b]. Hence, deciding if a planar 3-tree is a deque graph can also be done in polynomial time. The planar partial 3-trees are the subgraphs of planar 3-trees. Although, testing for a Hamiltonian path in a partial 3-tree also takes polynomial time [CR01, EGW01a, EGW01b], to the best of our knowledge, deciding if a planar partial 3-tree is the subgraph of a planar graph with a Hamiltonian path has not yet been tackled.

Open Problem 2.3. *Is there an efficient algorithm that tests whether a partial 3-tree is a deque graph?*

2.6.1.2 Fixed-Parameter Tractability

Another possibility to approach an $\mathcal{NP}$-hard problem P is to consider its fixed-parameter tractability (FPT) [Nie06, DF99]. The idea behind fixed-parameter tractability is to find a certain parameter k, which is assumed to be small, and to consider the parametrized version of the original $\mathcal{NP}$-hard problem P. Let $(\mathcal{I}, k)$ be the input to the parametrized instance of P, where $\mathcal{I}$ is an instance for the original problem P and k the parameter under consideration. P is said to be *fixed-parameter tractable* (*FPT*) with parameter k if there is an algorithm that

decides the problem in time $\mathcal{O}(f(k) \cdot |\mathcal{I}|^p)$ for some $p \in \mathbb{N}$, where $|\mathcal{I}|$ is the length of $\mathcal{I}$. Note that $f(k)$ only depends on k and not on the length of the input. In particular, the running time of an FPT algorithm can be exponential in the parameter k and it is polynomial in the length of the input, e. g., $\mathcal{O}(2^k \cdot |\mathcal{I}|)$. Intuitively, an FPT algorithm of an $\mathcal{NP}$-hard problem "extracts" the parameter k from the problem that causes the non-polynomial running time and that is small in many application scenarios.

The canonical parameter in "Does a graph has a k-deque layout?" is k. However, as already deciding whether a graph has a layout in one deque is $\mathcal{NP}$-hard, the respective decision problem is not FPT with parameter k (unless $\mathcal{NP} = \mathcal{P}$). For a given fixed linear layout, the respective decision problem is also not FPT if the statement in Problem 2.1 holds true. Hence, we have to look for other parameters.

Another promising candidate for a parameter is the *treewidth*: Let $G = (V, E)$ be a graph. A *tree decomposition* [RS84] of G is a tree T, whose nodes $B_1, \dots, B_k$ are subsets of V, with the following properties:

- $\bigcup_{i=1}^{k} B_i = V$.
- If $v \in B_i \cap B_j$, then $v \in B_\ell$ for all nodes B_ℓ on the simple path between B_i and B_j in T.
- If $\{u, v\} \in E$, then there is a node B_i with $u, v \in B_i$.

The *width* of a tree decomposition is $(\max_{i=1,\dots,k} |B_i|) - 1$ and the *treewidth* of G is the minimum width among all of G's tree decompositions. The treewidth of a graph is a measure of how "tree-like" it is and, in fact, a tree has treewidth 1. Many $\mathcal{NP}$-hard problems become efficiently solvable for graph classes with a treewidth that is bounded by a constant [Nie06]. Note that the treewidth of deque graphs is not bounded as the grid graph[2] of size $n \times n$ is a planar graph with a Hamiltonian path and treewidth n.

The famous algorithm meta-theorem by Courcelle [Cou90] states that deciding a graph property that is expressible in *monadic second-order logic* (*MSOL*) is FPT where the parameter is the treewidth (plus the length of the MSOL formula). We wont go into the details of MSOL but reuse known MSOL expressions as "black boxes": Two of the characterizing properties of deque graphs, namely, planarity [Cou00] and the existence of a Hamiltonian path [Kre12], are expressible in MSOL and so is their conjunction. We, therefore, obtain:

Proposition 2.9. *Deciding whether a graph $G = (V, E)$ is planar and contains a Hamiltonian path can be done in $\mathcal{O}(f(k) \cdot (|V| + |E|))$, where k is the treewidth of G and f is a function that depends only on k.*

For the special case of maximal planar graphs, we get:

Proposition 2.10. *Deciding whether a maximal planar graph $G = (V, E)$ is a deque graph can be done in $\mathcal{O}(f(k) \cdot |V|)$, where k is the treewidth of G.*

MSOL in its original form allows only for quantifications over the set of vertices and edges but not over arbitrary sets or binary relations. In particular, it is not possible to express the third characterizing property of deque graphs, that is, they are the *subgraphs* of planar graphs with a Hamiltonian path. At least, Prop. 2.9 indicates that the treewidth is a promising parameter for FPT algorithms in the context of deque graphs. Whether Courcelle's theorem is the right approach or a different ansatz must be used, is the topic of possible future work.

[2] A grid graph of size $n \times n$ contains n^2 vertices arranged on a quadratic grid of width n and height n interconnected by horizontal and vertical edges.

Open Problem 2.4. *Is there an FPT algorithm, with the treewidth as parameter, which decides whether a graph is a deque graph? Are there other suitable parameters for FPT algorithms that solve this problem?*

Recently, Bannister et al. [BES13] have shown that the minimization of crossings in one-page or two-page embeddings is FPT. As parameter, they use the cyclomatic number, which is the difference between the number of edges of the graph and the number of edges of one of its maximal spanning forests. Like the treewidth, the cyclomatic number can also be seen as an indicator of how tree-like a graph is. Bannister et al. have shown that computing the minimum number of crossings in one-page (two-page) embeddings can be done in time $\mathcal{O}(f(k) \cdot |V|)$, where k is the cyclomatic number and $|V|$ is the number of vertices. As two-page embeddings are close relatives to LC drawings, where additionally queue edges are allowed, it might be a fruitful endeavour to adapt the approach from [BES13] to minimize the number of crossings in LC drawings.

Open Problem 2.5. *Can the approach from [BES13] be adapted for LC drawings?*

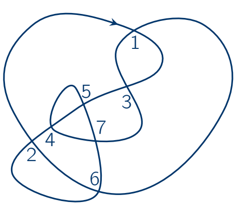
2.6.2 Open Gauß Codes and Deque Layouts

Deque layouts have interesting implications on Gauß codes as studied by Rosenstiehl and Tarjan in [RT84]. We discuss these implications only briefly here and the interested reader is referred to [RT84] and [Cet13] for details.

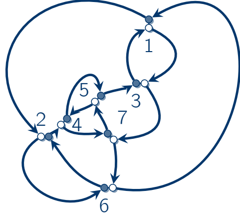
Consider the closed and self-intersecting Jordan curve in Fig. 2.41(a). The crossing points are numbered from 1 to 7. When we follow the curve, starting at the arrow, we encounter each crossing point exactly twice until we return to the starting point. The crossing points in the order of their appearance yields the *sequence* $S = (1, 3, 5, 4, 2, 6, 7, 5, 4, 7, 3, 1, 6, 2)$. A sequence S containing numbers from 1 to n where each number occurs exactly twice in S is called *closed Gauß code* if there is a closed Jordan curve whose crossing points can be numbered from 1 to n such that following the curve produces S (up to cyclic reordering and reversal). Carl Friedrich Gauß himself studied such sequences, though of course not using the name "Gauß code", and asked for a characterization [Gau00, pp. 272 and 282–286]. Such a characterization was given by Rosenstiehl in [Ros76] and, in collaboration with Tarjan, a recognition algorithm was presented in [RT84] that runs in $\mathcal{O}(n)$, where n is the number of crossings.

The idea in [RT84] is to reduce the problem to a (special case) of a planarity test using the following observations: Suppose we are given a closed curve as in Fig. 2.41(a). This curve can be transformed step-by-step as shown in Fig. 2.42: Leftmost, a crossing is depicted which is visited in order $1, 2, 3, 4$. Such a crossing point can be transformed into a tangent point as shown in the second figure from the left. Rosenstiehl and Tarjan have shown that every crossing curve can be transformed into a tangent point curve. In Fig. 2.42, the tangent point is visited in order, say, $1, 4, 3, 2$. Next, the tangent point is replaced by a vertex and the curve segments are replaced by directed edges, where the direction indicates from which sides the tangent point is entered and to which sides it is left. The vertex is split into two vertices (white and dark) which are connected by an undirected edge. The thereby obtained graph G for the crossing curve in Fig. 2.41(a) is shown in Fig. 2.41(b): The directed edges, i. e., the former segments of the closed curve in Fig. 2.41(a), form a Hamiltonian circle. Further, by construction, G is planar. Hence, G has a two-stack layout which is shown in Fig. 2.41(c).

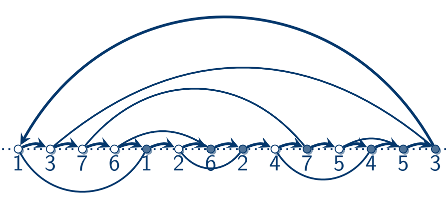
The method of Rosenstiehl and Tarjan uses these observations in order to find out whether a given sequence S is a closed Gauß code. First, S is transformed into a graph G as described



(a) The closed curve with Gauß code $S = (1, 3, 5, 4, 2, 6, 7, 5, 4, 7, 3, 1, 6, 2)$.



(b) Planar graph according to the tangent point curve from Fig. 2.41(a).



(c) Two-stack layout of the graph in Fig. 2.41(b).

Figure 2.41: A closed Gauß code.

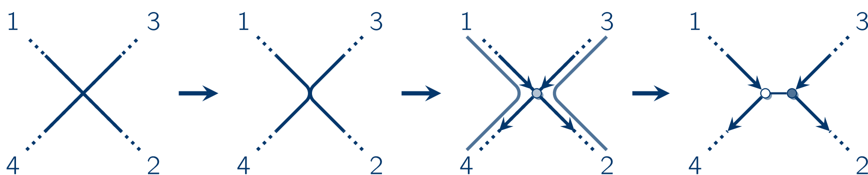
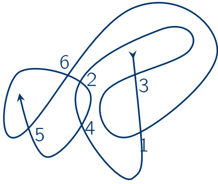


Figure 2.42: A crossing point visited in order $1, 2, 3, 4$ is transformed into a tangent point. By replacing the tangent point by a vertex and by splitting this vertex into two vertices (white and dark), we obtain a planar graph with a Hamiltonian circle for closed Gauß codes and a Hamiltonian path for open Gauß codes.

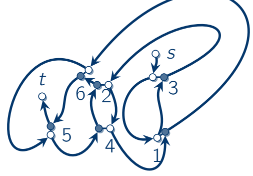
above. Then, G contains a Hamiltonian circle which corresponds to a tangent point curve. If the sequence is a closed Gauß code, then G must be planar. In order to test planarity, they exploit the facts that G contains a Hamiltonian circle and that every vertex is of degree three, i. e., two directed edges from the Hamiltonian path and one undirected edge. Thereby, they reduce the planarity test to the problem of sorting a permutation in two parallel stacks (cf. Sect. 2.1.2) for which they introduce a nifty data structure called *pile of twin-stacks*. They show that if G is planar, then S is a closed Gauß code.

Now, suppose we are given an open Jordan curve as depicted in Fig. 2.43(a). From this curve, we obtain the sequence $S = (3, 1, 4, 2, 3, 1, 6, 5, 6, 2, 4, 5)$ by following the curve from the endpoint to the right side to the endpoint to the left side. The so obtained sequence is called *open Gauß code*. By using the same transformation steps as before, we obtain a tangent point curve and the corresponding graph G as shown in Fig. 2.43(b), where s and t are the vertices representing the start and end of the open curve. Again, G is planar and, this time, the directed edges form a Hamiltonian path from s to t. By Thm. 2.2, the Hamiltonian path induces a deque layout which is shown in Fig. 2.43(c).

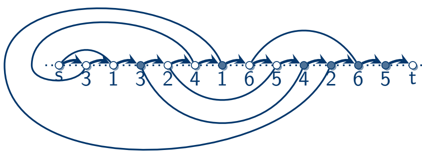
In [Cet13], co-supervised by the author of this thesis, Cetto adapts the approach of Tarjan in Rosenstiehl for open Gauß codes. Again, the input sequence is transformed into the graph that represents the corresponding tangent point curve, where this time the graph contains a Hamiltonian path. Testing the planarity of G is reduced to sorting a permutation in the deque for which Rosenstiehl and Tarjan gave an adaptation of the pile of twin-stacks in [RT84].



(a) Open curve with Gauß code $S = (3, 1, 4, 2, 3, 1, 6, 5, 6, 2, 4, 5)$.



(b) Planar graph of the tangent point curve from Fig. 2.43(a).



(c) Deque layout of the graph in Fig. 2.43(b).

Figure 2.43: An open Gauß code.

Although, it may seem straightforward to adapt the approach for closed Gauß codes to open Gauß codes, the quantity of theoretical obstacles that had to be tackled in [Cet13] is surprising. For instance, for the closed case, all undirected edges in the graph G are stack edges in the two-stack layout. In contrast, for open Gauß codes, a new algorithm had to be developed in [Cet13] that determines whether an undirected edge in G is a stack or a queue edge. The thesis also shows how deque layouts shed new light on Gauß codes and the topological structure of open Jordan curves.

2.6.3 Dual Layouts

The study of the deque has shed new light on the structure of the duals of planar graphs with a Hamiltonian path (Sect. 2.2.4 and Thm. 2.3) and the duals of queue graphs (Thm. 2.6 and Sect. 2.4.3). At the end of Sect. 2.4.3, we have hinted at how to make the dual graph more explicit in graph layouts. Consider Fig. 2.44 which shows the LC embedding of an exuberant $\prec$-augmentation along with its dual. Remember that in an exuberant $\prec$-augmentation, the (directed) edges between adjacent vertices of the linear layout are always introduced regardless of whether they already exist in the original graph.

In the following, we process only the undirected edges in the deque layout and neglect the directed edges on the Hamiltonian path. The deque has the following contents:

$$\mathcal{C}_1 = () \,, \qquad \mathcal{C}_2 = ((1,4),(1,3)) \,,$$
$$\mathcal{C}_3 = ((2,3),(2,4),(1,4),(1,3),(2,3)) \,, \qquad \mathcal{C}_4 = ((3,4),(2,4),(1,4),(3,7),(3,5)) \,,$$
$$\mathcal{C}_5 = ((4,7),(3,7),(3,5)) \,, \qquad \mathcal{C}_6 = ((5,8),(4,7),(3,7),(5,7)) \,,$$
$$\mathcal{C}_7 = ((6,7),(6,8),(5,8),(4,7),(3,7),(5,7)) \,, \qquad \mathcal{C}_8 = ((6,8),(5,8)) \,,$$
$$\mathcal{C}_9 = () \,,$$

where $\mathcal{C}_i$ is the input of vertex i and $\mathcal{C}_9$ is the output of vertex 8. Every edge lies between two faces, e. g., $(1,4)$ between f_1^q and f_2^q. By inserting the faces into, for instance, $\mathcal{C}_4'$, we see how this is reflected in the deque layout:

$$\mathcal{C}_4' = (f_3^{\mathrm{h}},(3,4),f_1^{\mathrm{h}},(2,4),f_1^q,(1,4),f_2^q,(3,7),f_3^{\mathrm{t}},(3,5),f_4^{\mathrm{t}}) \,.$$

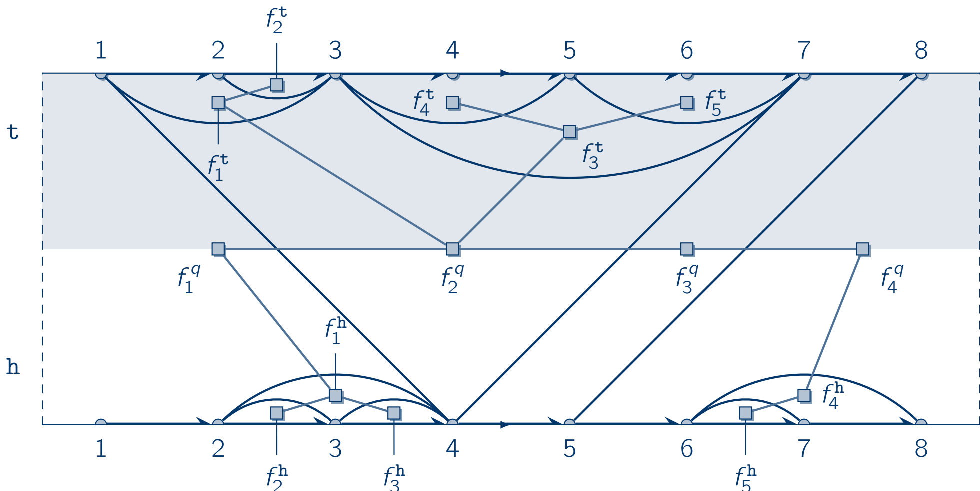


Figure 2.44: An LC embedding and its dual of an exuberant $\prec$-augmentation.

At vertex 4, edges (3, 4), (2, 4), (1, 4) are removed from $\mathcal{C}_4$ and so are the faces $f_3^{\mathtt{h}}$, $f_1^{\mathtt{h}}$, f_1^q. Then, edge (4, 7) is inserted, and with it face f_3^q. We obtain $\mathcal{C}_5'$:

$$\mathcal{C}_5' = (f_3^q, (4,7), f_2^q, (3,7), f_3^{\mathtt{t}}, (3,5), f_4^{\mathtt{t}}) .$$

The faces are processed in the deque just like the edges. Removing all edges and leaving only the faces yields:

$$\mathcal{C}_1^* = (f_1^q) , \quad \mathcal{C}_2^* = (f_1^q, f_2^q, f_1^{\mathtt{t}}) , \quad \mathcal{C}_3^* = (f_2^{\mathtt{h}}, f_1^{\mathtt{h}}, f_1^q, f_2^q, f_1^{\mathtt{t}}, f_2^{\mathtt{t}}) ,$$
$$\mathcal{C}_4^* = (f_3^{\mathtt{h}}, f_1^{\mathtt{h}}, f_1^q, f_2^q, f_3^{\mathtt{t}}, f_4^{\mathtt{t}}) , \quad \mathcal{C}_5^* = (f_3^q, f_2^q, f_3^{\mathtt{t}}, f_4^{\mathtt{t}}) , \quad \mathcal{C}_6^* = (f_4^q, f_3^q, f_2^q, f_3^{\mathtt{t}}, f_5^{\mathtt{t}}) ,$$
$$\mathcal{C}_7^* = (f_5^{\mathtt{h}}, f_4^{\mathtt{h}}, f_4^q, f_3^q, f_2^q, f_3^{\mathtt{t}}, f_5^{\mathtt{t}}) , \quad \mathcal{C}_8^* = (f_4^{\mathtt{h}}, f_4^q, f_3^q) , \quad \mathcal{C}_9^* = (f_4^q) .$$

Note that all faces f_i^q for $i = 1, 2, 3, 4$ are processed as queue elements, all faces $f_i^{\mathtt{h}}$ ($i = 1, \ldots, 5$) as stack elements at the head, and all faces $f_i^{\mathtt{t}}$ ($i = 1, \ldots, 5$) as stack elements at the tail. By construction, there is a dual edge between two faces if there is a time instant at which both are adjacent in the deque. From this, we immediately can deduce the structure of the dual:

- The faces f_i^q lie on a path as always two of them are adjacent in the deque.
- The faces f_i^q are the root of trees that contain the faces that are processed as stack elements. For instance, f_2^q is the root of the tree that contains $f_1^{\mathtt{t}}, \ldots, f_5^{\mathtt{t}}$.

In general, let $\mathcal{D}$ be a deque-reducible data structure. A *dual $\mathcal{D}$ layout* is obtained from a $\mathcal{D}$ layout by inserting the corresponding faces between the edges in $\mathcal{D}$ and removing the edges afterwards. There is an edge between two faces if and only if they are adjacent in $\mathcal{D}$ at some time instant. The so obtained graph is called *graph induced by the $\mathcal{D}$ layout*. Depending on $\mathcal{D}$, these induced graphs have a certain structure that immediately follows from the operation principle of $\mathcal{D}$:

- The graph induced by a dual (two-)stack layout is a tree, which reflects the last in, first out principle.

- The graph induced by a dual queue layout is a path, which reflects the first in, first out principle.
- The graph induced by a dual deque layout are trees whose roots are connected by a path. The deque combines last in, first out with first in, first out.

The duals (without the outer face) of stack graphs, i. e., outerplanar graphs, are known to be trees [SP83] and, yet, this observation follows directly from dual stack layouts.

Open Problem 2.6. *Are dual layouts useful to obtain further insights to stack, queue, and deque graphs, and to the working principles and differences of the stack, queue, and deque data structures?*

2.6.4 General Planarity Testing by the Splittable Deque

As byproduct of the SD characterization of planarity, we have obtained the SD-based algorithm `IsSDLayout` (Alg. 2.5 on page 72) to test the planarity of a rotation system. This raises the question for a general planarity testing algorithm that uses the SD. Remember that the order in which the edges are inserted and removed, and how the SD is split is defined by the rotation system. Conversely, if we can observe how the edges are processed and the SD is split in a deque layout, we can deduce a planar rotation system. If one of the edges cannot be processed in the SD, then the graph is not planar. Thus, in principle, planarity testing reduces to finding an SD layout.

One of the main obstacles that has to be overcome in the implementation of an SD-based planarity test is that the side at which an edge has to be inserted is unknown as no rotation system is given. A similar problem was tackled in [RT84] by Rosenstiehl and Tarjan, who presented an algorithm to test the sortability of a permutation in two parallel stacks. There, each element can either be inserted to one or the other stack. Rosenstiehl and Tarjan proposed the pile of twin-stacks for this problem, which implements a lazy approach: The decision of where an edge has to be inserted is deferred to the moment when the element itself or an element further below in the stack is removed. The pile of twin-stacks can also be adapted to test for deque sortability [RT84]. Applying such an approach to the SD is surely a first important step to an SD-based planarity testing algorithm.

Whether such a planarity test has any benefits, e. g., simplicity, over any of the other numerous existing planarity tests, e. g., [LEC67, HT73, SH99], is unsure. The hope that such an algorithm is simpler than existing approaches, may be shattered by the complexity of the suggested "lazy SD". Prototypical implementations indeed suggest that this is the case.

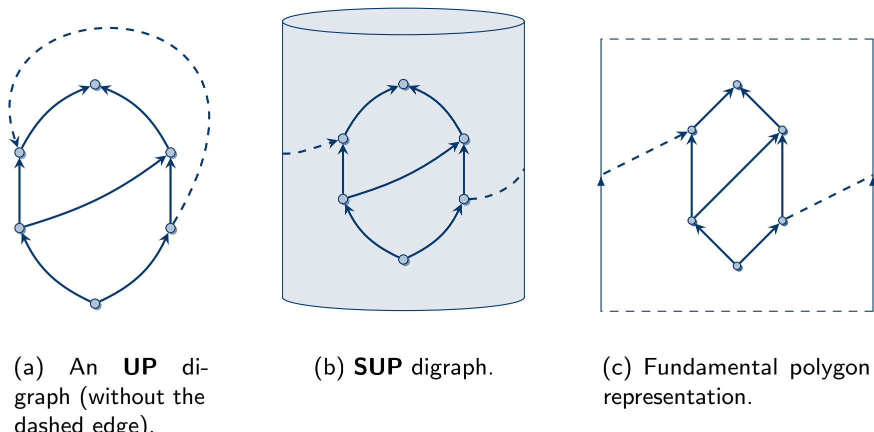
Open Problem 2.7. *Is there a planarity testing algorithm based on the SD that has benefits, e. g., simplicity, over existing planarity testing algorithms? If there is such an algorithm, can it be adapted to extract the forbidden minors $K_{3,3}$ and K_5 if the input graph is not planar?*

Chapter 3

Rolling Upward Planar Digraphs

3.1 Introduction

The de facto standard method for drawing *hierarchies*, i. e., acyclic digraphs, is the Sugiyama framework [STT81]. The idea is to transform the edge direction into a geometrical direction, thus, generating a "flow" in the diagram that displays the hierarchical dependencies between the entities, that is, the vertices. An example of a drawing generated by Sugiyama's framework is shown in Fig. 3.3(a), where the vertices are placed on horizontal levels and edges that span more than one level are subdivided by dummy vertices (dark dots). The hierarchy is displayed from bottom to top and all edge curves are monotonically increasing in y-direction from the source to the target. Such a drawing is called *upward*. One aesthetic criterion of graph drawings is the number of crossings [STT81], indeed, empirical studies suggest that it is an important one [Pur97]. In consequence, digraphs that allow a plane upward drawing are particularly suitable for the Sugiyama framework. These digraphs are called *upward planar*, where **UP** denotes the set of upward planar digraphs. An example of an upward planar digraph is shown in Fig. 3.1(a), where we ignore the dashed edge for the moment.

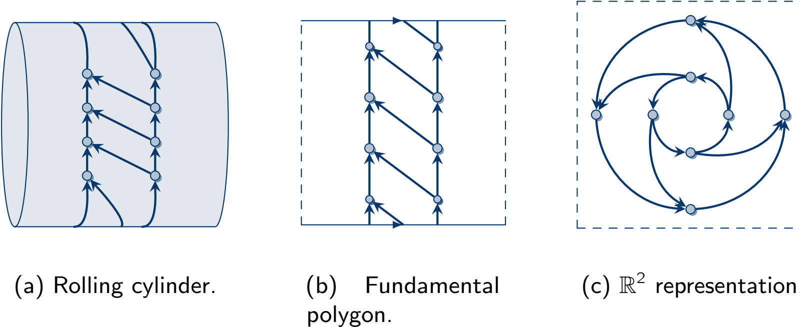


(a) An **UP** digraph (without the dashed edge).

(b) **SUP** digraph.

(c) Fundamental polygon representation.

Figure 3.1: An example of an **UP** digraph and a **SUP** digraph.

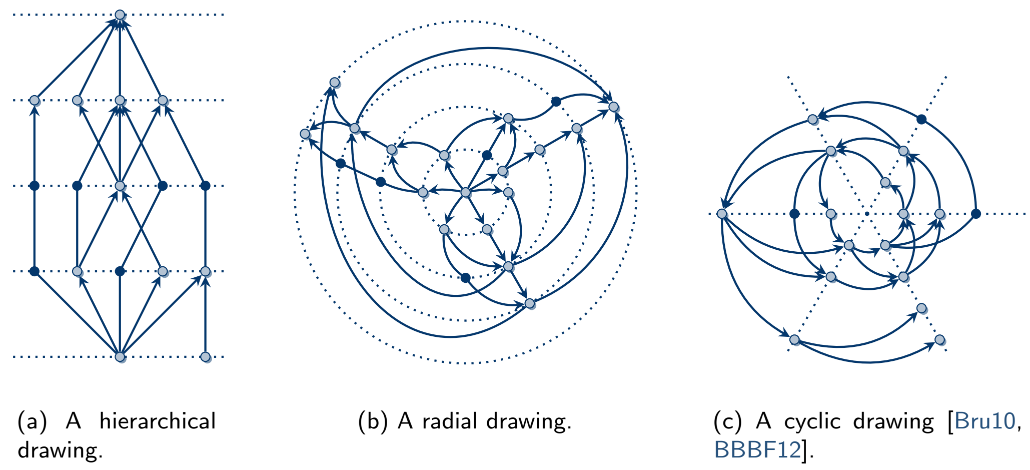


(a) Rolling cylinder.

(b) Fundamental polygon.

(c) $\mathbb{R}^2$ representation.

Figure 3.2: Different representations of a **RUP** digraph.



(a) A hierarchical drawing.

(b) A radial drawing.

(c) A cyclic drawing [Bru10, BBBF12].

Figure 3.3: Drawings generated by variants of Sugiyama's framework.

Upward planar digraphs are not only interesting in the context of graph drawing but also from a theoretical viewpoint. Whereas the plane, the surface of the cylinder, and the sphere are equivalent with respect to (undirected) planarity, the situation is different for upward planar digraphs: The digraph in Fig. 3.1(a) together with the dashed edge is not upward planar [DBETT99, p. 171], although, it is still planar. Even more, on the surface of the standing cylinder $\mathbf{C}_S^3$, the digraph with the dashed edge admits an upward plane drawing as shown in Fig. 3.1(b). The respective fundamental polygon representation is shown in Fig. 3.1(c). In both drawings, the dashed edge winds around the cylinder which is not possible in the plane. We say a digraph is *standing upward planar* if it admits an upward plane drawing on the surface of the standing cylinder. **SUP** denotes the set of standing upward planar digraphs. By the example in Fig. 3.1, we can conclude that standing upward planarity is a proper extension of upward planarity, i. e., **UP** $\subsetneq$ **SUP**.

The main focus of this chapter is rolling upward planarity, which is, in turn, a proper extension of standing upward planarity: A drawing on the surface of the rolling cylinder is

upward if all edge curves wind around the cylinder's axis in one direction. An example of such a drawing is shown in Fig. 3.2(a). In contrast to **UP** and **SUP**, which contain only acyclic digraphs, rolling upward planar digraphs may contain cycles. Whereas **UP** and **SUP** digraphs have been studied extensively in the past (as we will see in Sect. 3.2), rolling upward planarity has only recently been studied in depth in our classification scheme of upward planarity [ABBG11] and in the extension of the Sugiyama framework to recurrent hierarchies [Bru10, BBBF12, BBBL09, BBBH11]. Interestingly, Sugiyama et al. have already suggested the study of recurrent hierarchies in [STT81]. In this chapter, we extend the study of upward planarity to the rolling cylinder, thereby, introducing new tools and insights that are also useful for a better understanding of **SUP** and other cases of upward planarity.

3.2 Classification of Upward Planarity

In [ABBG11], we have presented a classification scheme that encompasses and unifies many of the generalizations of upward planarity. We adopt this classification scheme and the definitions from [ABBG11] for our discussion of related work. Consider a differentiable two-dimensional manifold $\mathbb{S}$ [MRA01, MT01], e. g., the surface of the standing cylinder, and let $F : \mathbb{S} \to \mathbb{R}^2$ be a vector field in $\mathbb{S}$. F maps each point of $\mathbb{S}$ to a two-dimensional vector that defines the direction an edge curve must "roughly" follow. More formally, let Γ be a drawing of a digraph $G = (V, E)$ in $\mathbb{S}$. We assume that each edge curve $p_e := \Gamma(e) : [0, 1] \to \mathbb{S}$ for all $e \in E$ is differentiable except for countably many critical points $\mathrm{Cr}(e) \subsetneq [0, 1]$, e. g., bends. We also assume in the following that every edge curve starts at the source and ends at the target of the edge, i. e., $p_e(0) = \Gamma(u)$ and $p_e(1) = \Gamma(v)$ with $e = (u, v)$. The edge curve p_e of e *respects* F if:

$$\forall t \in [0, 1] \setminus \mathrm{Cr}(e) : \langle p_e'(t), F(p_e(t))\rangle > 0\,,$$

where p_e' is the first order derivative of p_e and $\langle \cdot, \cdot\rangle$ is the dot product. This means that at all differentiable points, the angle between the tangent vector of p_e and the vector field is less than $\pi/2$ (see Fig. 3.4). The edge curve of e is said to *weakly respect* F if:

$$\forall t \in [0, 1] \setminus \mathrm{Cr}(e) : \langle p_e'(t), F(p_e(t))\rangle \geq 0\,.$$

In this case, the angle between the tangent vector and the vector field must not be greater than $\pi/2$. In particular, edge curves are allowed to be perpendicular to the vector field. If all edges (weakly) respect F we say that Γ is (weakly) upward with respect to $\mathbb{S}$ and F. If additionally Γ is plane (cf. Sect. 1.1.7), then Γ is (weakly) upward plane with respect to $\mathbb{S}$ and F.

For consistency, we use the following generic terminology scheme: For a given surface $\mathbb{S}$ and vector field F, a plane drawing that is upward with respect to $\mathbb{S}$ and F is called x**UP** *drawing*, e. g., **UP** or **SUP** drawing. From an x**UP** drawing, we obtain an embedding which is called x**UP** *embedding* and, likewise, a digraph which has an x**UP** embedding is called x**UP** *digraph*. For instance, Fig. 3.1(b) shows a **SUP** drawing from which we obtain a **SUP** embedding and, hence, the digraph is **SUP**. Finally, we slightly abuse the notation a bit more and let x**UP** denote the set of all x**UP** digraphs.

It is important to keep in mind that in general an arbitrary embedding of a digraph is not necessarily an x**UP** embedding. For instance, any embedding of a **UP** (**SUP**) digraph with a vertex whose incoming edges are not consecutive in its rotation system cannot be **UP** (**SUP**).

In [ABBG11], we studied several surfaces and vector fields. As in Sect. 1.1.9, let $I = (-1, 1)$ be the unit interval and derive $I_\circ$ from I by identifying its boundaries. As surfaces we consider the

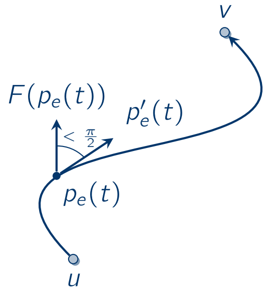
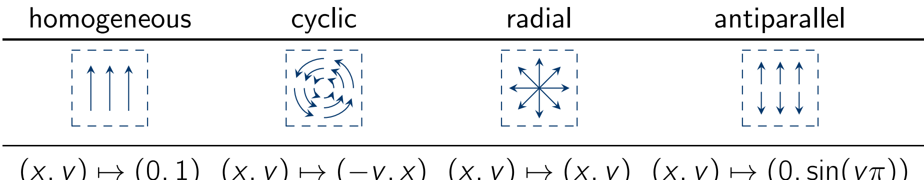


Figure 3.4: In a (weak) upward drawing, the angle between the tangent of an edge curve and the vector field must be less (not greater) than $\pi/2$.

fundamental polygon representations of the plane $\mathbf{P} = I \times I$, the standing cylinder $\mathbf{C}_s = I_\circ \times I$, the rolling cylinder $\mathbf{C}_r = I \times I_\circ$, and the torus $\mathbf{T} = I_\circ \times I_\circ$. The considered vector fields are shown in Tab. 3.1, i. e., the homogeneous, the cyclic, the radial and the antiparallel field. In the last row of the table, the vector fields are given as functions that assign each point of the respective surface a two-dimensional vector. For instance, with the homogeneous field, all edge curves must be monotonically increasing in y-direction and, with the radial field all edge curves must tend away from the origin. Note that in the cyclic field each vector is orthogonal to the respective vector in the radial field. Also note that the length of a vector can be neglected in upward plane drawings as only the angle between an edge curve and the vector plays a role.



homogeneous	cyclic	radial	antiparallel
$(x, y) \mapsto (0, 1)$	$(x, y) \mapsto (-y, x)$	$(x, y) \mapsto (x, y)$	$(x, y) \mapsto (0, \sin(y\pi))$

Table 3.1: Vector fields.

3.2.1 Upward Planar Digraphs

In our classification scheme, the **UP** digraphs are upward planar in **P** endowed with the homogeneous field. The weak case is of no interest as it coincides with the non-weak case. **UP** digraphs have been studied extensively, e. g., in [BDBLM94, BDBMT98, DGL06, FFRV11, GT01a, HL96, HL06, Pap95], and they have been characterized by [Kel87, DBT88] as the spanning subgraphs of planar st-digraphs which are acyclic digraphs with a single source s and a single sink t, and an edge from s to t. For instance, we can introduce the edge from the source to the sink in Fig. 3.1(a) without violating planarity, assuming that the digraph does not contain the dashed edge.

As the origins of upward planar digraphs lie in graph drawing, various drawing algorithms for **UP** digraphs and interesting subclasses have been proposed: A plane straight-line drawing of an **UP** digraph where the vertices are placed on an integer grid is always possible, though, it may require exponential area [DBTT92]. Note that this is in sharp contrast to the undirected case where every planar n-vertex graph admits a plane straight-line drawing with area consumption

in $\mathcal{O}(n^2)$ [dFPP90]. Exponential area for an upward drawing is needed even if the digraph is outerplanar, the underlying undirected graph is bipartite, or for directed trees where the rotation system is fixed [Fra07]. In case of *st*-digraphs with no transitive edges, a straight-line drawing with $\mathcal{O}(n^2)$ area consumption is possible [DBTT92]. The same area consumption can also be achieved if bends are allowed [DBTT92] and for series-parallel digraphs [BCDB+92]. For an extensive survey, see [DBF13].

In order to construct a polyline drawing of a planar *st*-digraph, Di Battista and Tamassia use the directed dual of embedded *st*-digraphs in [DBT88]. This is in so far interesting for us as we will see that duals play an important role for rolling upward planar digraphs as well. Di Battista and Tamassia show that the dual of an *st*-digraph is again an *st*-digraph, where the dual of the edge from s to t is reversed. This result is only one of the many intriguing properties of upward planar digraphs and their duals as we will see in Sect. 3.4.3.

As for the complexity, deciding whether a digraph is in **UP** is $\mathcal{NP}$-hard as shown by Garg and Tamassia [GT01a]. If, on the other hand, the digraph is an *st*-digraph, then an ordinary (undirected) planarity test suffices to find out whether it is upward planar. Remember that every **UP** digraph can be augmented to a planar *st*-digraph by introducing edges, which suggests that the complexity lies in finding the right edges to introduce; an observation that we will make also for other cases of upward planarity. The recognition problem is efficiently solvable for digraphs with a fixed embedding [BDBLM94], outerplanar graphs [Pap95], single-source digraphs [HL96], and if the underlying undirected graph is bipartite [DBLR90]. In the context of the Sugiyama framework, testing whether a digraph is upward planar if the vertices and dummy vertices are assigned fixed levels (see Fig. 3.3(a)) can be done in linear time [JL02].

Healy and Lynch [HL06] give two fixed-parameter tractable algorithms that decide whether a digraph $G = (V, E)$ is upward planar. One runs in time $\mathcal{O}(2^t \cdot t! \cdot |V|^2)$, where t is number of triconnected components, and the other runs in time $\mathcal{O}(|V|^2 + k^2 \cdot (2k)!)$, where $k = |E| - |V|$.

3.2.2 Standing Upward Planar Digraphs

The **SUP** digraphs are upward planar in $\mathbf{C}_s$ endowed with the homogeneous field. For the weak case, the set of digraphs is denoted by **wSUP** which is a proper superset of **SUP** as the weak case allows horizontal cycles. **SUP** digraphs are characterized by *acyclic dipoles* which are acyclic digraphs with a single source and a single sink, and the characterization is similar to that of **UP**: a digraph is **SUP** if and only if it is the spanning subgraph of a planar acyclic dipole [Han06, Has01, LMS06] (see Figs. 3.1(b) and 3.1(c)). Even more, any embedding of an acyclic dipole is also **SUP**. Note the difference between the characterizations of **UP** and **SUP** digraphs: the former must allow the edge from the source to the sink, which prevents edges from winding around the cylinder. Hashemi and Rival also proved [HRK98] that a digraph is **SUP** if and only if it has a triangulation with no saddle points, i. e., in the rotation system of each vertex, all incoming and outgoing edges must be consecutive. Hashemi et al. [HRK98, Has01, DH08] studied *spherical digraphs* which allow a plane drawing on the surface of the sphere such that the edge curves are monotonically increasing from the south to the north pole. In [ABBG11], we have shown that every **SUP** digraph is spherical and vice versa. Further, Thomassen [Tho89] studied single-source, single-sink digraphs on the standing cylinder, and Foldes et al. [FRU92] investigated ordered sets on the sphere and on the cylinder as a truncated sphere.

SUP drawings and their curve-complexity are considered in [Bra14]: for a given **SUP** drawing of an n-vertex **SUP** digraph, there is a polyline drawing on the standing cylinder with

no edge windings, i. e., no edge fully winds around the cylinder, at most two bends per edge and, all vertices and bends are placed on an integer grid of size $\mathcal{O}(n^2)$. Note that "polyline" in this context means that the edge segments, that is, the differentiable parts of an edge curve are geodesics on the cylinder surface (or straight lines on the fundamental polygon). Moreover, such a drawing can be constructed in time $\mathcal{O}(\tau n^2)$, where τ is the time to compute the intersection of an edge with a horizontal line through a vertex.

SUP digraphs also find their application in the context of circuit value problems (CVP) [Weg05, Han06, LMS09]. The CVP asks to determine the result of a Boolean circuit for a given assignment of input values, which is $\mathcal{P}$-complete in general. CVP can be solved more efficiently for monotone planar circuits endowed with a **SUP** embedding [Han06].

In [ABBG11], we have shown that **SUP** (**wSUP**) coincides with the set of digraphs that are (weakly) upward planar with respect to **P** and the radial field. In these drawings, all edge curves lead away from the origin (cf. Tab. 3.1). In the non-weak case, this type of drawing plays an important role in the adaption of the Sugiyama framework for radial drawings [Bac07]. An example of a radial drawing is shown in Fig. 3.3(b), where the levels are concentric circles instead of horizontal lines. This type of drawing has its application in, for instance, the visualization of social networks [BKW99].

As with **UP**, deciding whether a digraph is in **SUP** is $\mathcal{NP}$-hard [HRK98]. In contrast, deciding whether an acyclic dipole is **SUP** can be done efficiently as only planarity has to be tested. This parallels the situation for **UP** and *st*-digraphs, and suggests that the complexity of the **SUP** decision problem lies in the augmentation of the digraph to a planar acyclic dipole. The **SUP** decision problem becomes efficiently solvable for acyclic, triconnected, single-source digraphs [DH08]. In the context of radial drawings [Bac07], a linear-time decision algorithm exists if the vertices and dummy vertices are assigned fixed levels [BBF05].

3.2.3 Rolling Upward Planar Digraphs

UP and **SUP** digraphs are all acyclic and are, hence, suitable for modelling hierarchical structures. Sometimes, however, periodic or cyclic dependencies need to be modelled and displayed for which *rolling upward planar* digraphs can be used. A plane drawing that is upward with respect to the rolling cylinder $\mathbf{C}_r$ and the homogeneous field is called *rolling upward plane* or **RUP**. Again, **RUP** also denotes the set of digraphs that have a **RUP** drawing. The weak case is neglected as it coincides with **RUP** [Bra14].

In a **RUP** drawing, the edges wind around the axis of the cylinder in a certain direction and, thus, cycles are possible. An example of a **RUP** drawing is shown in Figs. 3.2(a) and 3.2(b), where the latter shows the fundamental polygon representation. **RUP** coincides with the set of digraphs with an upward drawing in the plane endowed with the cyclic field [ABBG11]. There, all edge curves wind around the origin in counterclockwise direction (Fig. 3.2(c)).

The relationship between **RUP**, **UP** and **SUP** is as follows [ABBG11, Bra14]:

$$\mathbf{UP} \subsetneq \mathbf{SUP} \subsetneq \mathbf{RUP} \cap \mathbf{DAG} \subsetneq \mathbf{RUP}\,,$$

where **DAG** is the set of acyclic digraphs. Hence, **RUP** is a proper extension of the other cases, even if restricted to acyclic digraphs. Fig. 3.5 shows an example from [ABBG11] of an acyclic **RUP** digraph that is not **SUP**. As for the decision problem, it should come as no surprise that it is $\mathcal{NP}$-hard in general [Bra14].

The curve complexity of **RUP** drawings was studied in [Bra14]: For every **RUP** drawing of an n-vertex **RUP** digraph, there is a polyline drawing where each edge winds at most once

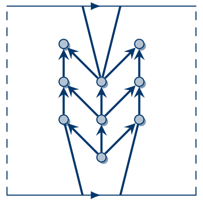


Figure 3.5: An acyclic **RUP** digraph that is not **SUP** [ABBG11].

around the cylinder and has at most two bends, and all vertices and bends are placed on integer coordinates. The area consumption of the drawing is in $\mathcal{O}(n^3)$ and the drawing can be computed in time $\mathcal{O}(\tau n^3)$, where τ is the time to compute an intersection of an edge with a horizontal line.

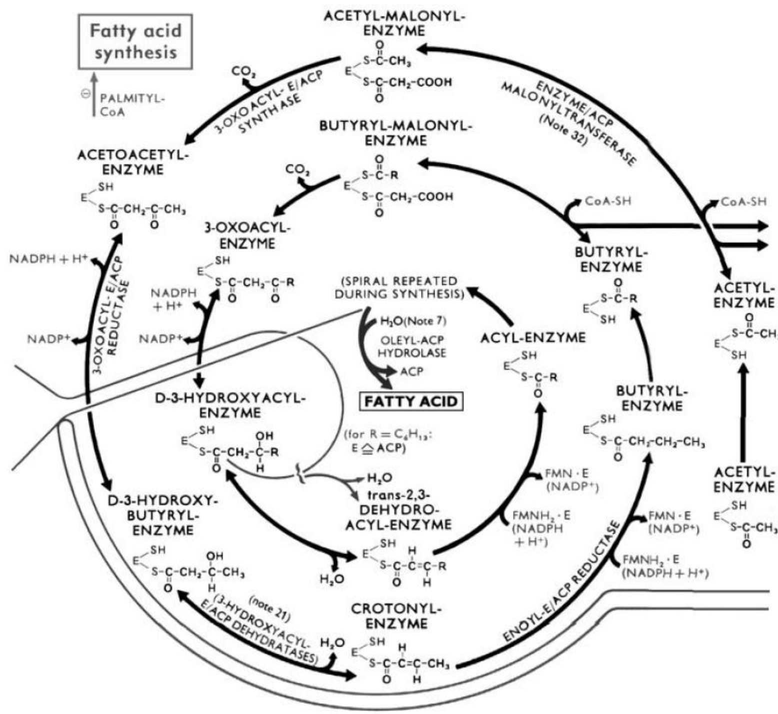
In [BBBF12], the Sugiyama framework has been extended to *recurrent hierarchies* and *cyclic drawings* which are upward in the plane with the cyclic field. An example of such a drawing is shown in Fig. 3.3(c), where the levels are half-lines extending from the origin. Similar to the hierarchical and radial case, there is a linear-time algorithm to decide whether a strongly connected digraph admits a plane cyclic drawing if the vertices and dummy vertices are assigned fixed levels [BB08, BBBF12].

Cyclic drawings and, in the planar case, **RUP** digraphs have their application in the visualization of periodic processes and dependencies. For instance, the chemical reactions in biosciences often are cyclic, e. g., the fatty acid synthesis in Fig. 3.6(a). Representing whole metabolism results involve many cyclic dependencies that have to be displayed as such. Another application arises when displaying periodic processes such as schedules in public transportation, e. g., plane, train, or bus schedules. Fig. 3.6(b) shows the daily schedule of trains operating between Paris and Lyon in the 1880s [Tuf01], where the x-axis is the time line from 6 am to 6 am on the next day and the y-axis is the position between Paris (at the top) and Lyon (at the bottom). A line in the drawing indicates the position of a train at a certain time instant. It is hard to follow the position of a train from, say, 5 a. m. to 7 a. m. as the time line is cut. Mounting the drawing on a cylinder as suggested by Tufte [Tuf01] or using a cyclic drawing is a remedy to this problem. Variants of periodic scheduling problems are studied as periodic event scheduling in [SU89] and as cyclic job-shop problems in [HLP02]. More applications of cyclic drawings can be found in [Bru10].

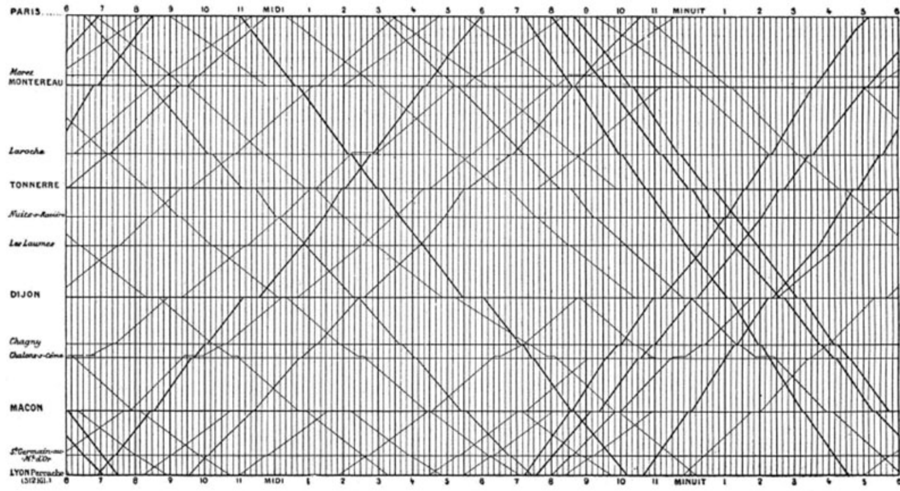
3.2.4 Other Cases of Upward Planarity

Brandenburg has defined *strict upward planarity*, where edge curves must be upward in y- as well as x-direction [Bra14]. Brandenburg showed that **SUP** = **sSUP** = **sRUP** = **wSUP** ∩ **DAG**, where **sSUP** and **sRUP** are the strict upward planar digraphs on the standing and rolling cylinder, respectively. In other words, the standing and rolling cylinder are equivalent in the strict case.

A feature that is common to all upward planar drawings of any type we have discussed is that in the rotation system at each vertex all incoming edges are consecutive and so are all outgoing edges. Bertolazzi et al. generalized this principle in [BDBD02] to define *quasi-upward plane drawings*: A plane drawing is quasi-upward if it is “locally upward” at each vertex, i. e., in a sufficiently small circular area of the plane that contains a vertex v, all incoming (outgoing)



(a) Fatty acid synthesis [Mic93].



(b) Daily train schedule between Paris and Lyon in the 1880s [Tuf01].

Figure 3.6: Application examples of cyclic and rolling upward planar drawings.

edges of v enter the horizontal line through v from below (above). Let **QUP** denote the set of digraphs that have a quasi-upward plane drawing. Bertolazzi et al. also showed that a digraph is **QUP** if and only if it has a planar *bimodal rotation system*, i. e., all incoming (outgoing) edges are consecutive in the rotation system of each vertex. Quasi-upward planarity is a proper extension of all cases of upward planarity we have discussed [Bra14].

Fig. 3.7 shows an overview from [Bra14] of the classes of upward planar digraphs we have discussed so far. The prefix **s** indicates the strict case. Coinciding sets are placed within a lightly shaded rectangle and there is an edge from class A to class B if $A \subsetneq B$. The contribution of this thesis are characterizations and recognition algorithms for the classes within the dark shaded rectangle (Sects. 3.4 to 3.6).

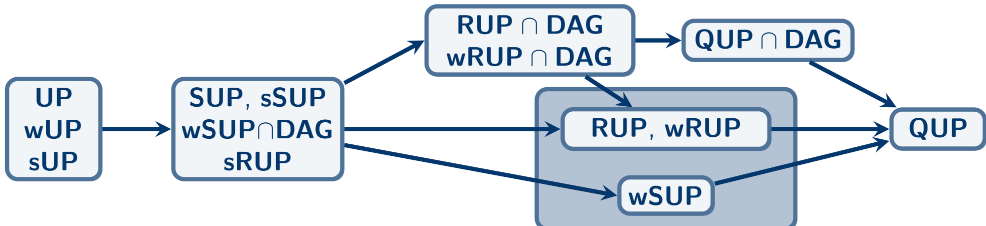


Figure 3.7: Relationship between different classes of upward planar digraphs [Bra14]. Coinciding sets are placed within a lightly shaded rectangle and there is an edge from class A to class B if $A \subsetneq B$.

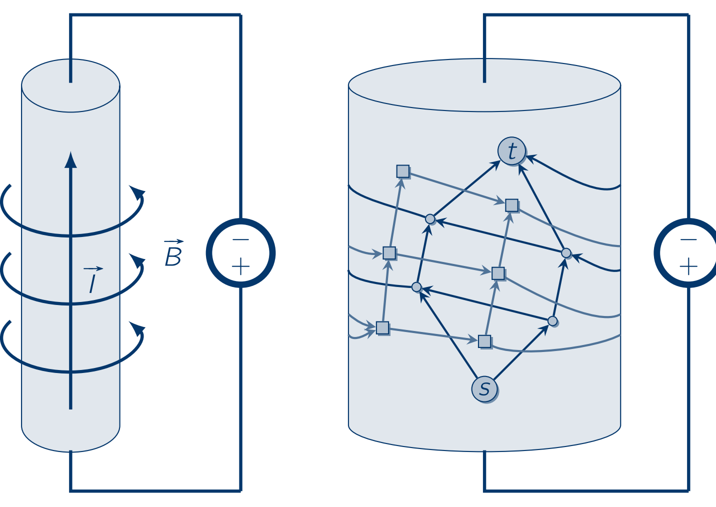
Before we close our discussion of related work, it should also be mentioned that upwardness was also defined and studied for the torus: Dolati et al. [Dol08, DHK08] studied upward toroidal drawings on the lying and the standing torus embedded in the three-dimensional space where all edge curves are monotonically increasing in y-direction. By applying our scheme, the former are upward "planar" with respect to $\mathbf{T}$ and the antiparallel field, and the latter are upward "planar" with respect to the surface of the torus $\mathbf{T}$ and the radial field [ABBG11]. Dolati et al. also showed that every digraph that is upward toroidal on the lying torus is upward toroidal on the standing torus but not vice versa.

3.3 Contributions of this Thesis

We extend the existing study of **RUP** digraphs by giving a combinatorial characterization and an algorithm that decides whether a digraph without sources and sinks is **RUP**. Remember that **UP** digraphs and **SUP** digraphs have been characterized as the subgraphs of planar st-digraphs and acyclic dipoles, respectively. Such a characterization is not possible for **RUP** digraphs as they may contain cycles. Nevertheless, it turns out that acyclic dipoles still play an important role in the characterization **RUP**, though, in an indirect way. The idea behind the characterization and, in fact, the whole chapter, is to study the directed duals of **RUP** digraphs (see Sect. 1.1.8 for the definition of directed dual digraphs). A basic property of directed duals is that the primal digraph is strongly connected if and only if its dual is acyclic, and the primal is acyclic if and only if the dual is strongly connected (Lem. 3.2).

One of the key lemmas towards our characterization has a physical interpretation: Ampère's law from electromagnetism states that electric current $\vec{I}$ flowing through a conductor generates a magnetic field $\vec{B}$ that winds around the conductor (see Fig. 3.8(a)). Now, consider Fig. 3.8(b),

which shows an acyclic dipole with "connecters" s and t upward embedded on the standing cylinder. An electric current "flowing" from s to t generates the dual digraph that winds around the cylinder, i. e., the dual is **RUP** and strongly connected as the primal is acyclic. By proving this in general in Sect. 3.4.3, we obtain a characterization of strongly connected **RUP** digraphs: a strongly connected digraph is **RUP** if and only if it has an embedding such that its dual is an acyclic dipole. To extend this characterization to closed digraphs, i. e., digraphs without sources and sinks, we generalize acyclic dipoles to (general) dipoles, which allow for cycles. For this, we introduce the compound digraph, which divides a digraph into its (non-trivial) strongly connected components, the *compounds*, and in its acyclic components, the *transits*. Based on the compound digraph, we define (general) dipoles and, following the idea behind Ampère's law, we show that a closed digraph is **RUP** if and only if it has an embedding such that its dual is a (general) dipole. Finally, we arrive at a characterization of all **RUP** digraphs by showing that each **RUP** digraph can be augmented to a closed **RUP** digraph.



(a) Ampère's law: Electric current flowing through a conductor generates a magnetic field that winds around the conductor.

(b) Ampère's law for dual graphs: The primal digraph, an acyclic dipole, with "connecters" s and t induces the dual, a strongly connected **RUP** digraph.

Figure 3.8: Ampère's law.

Remember that deciding whether a digraph is **SUP** is $\mathcal{NP}$-hard [HRK98]. In contrast, deciding whether an acyclic dipole is **SUP** is possible in linear time as it only involves a planarity test. The situation is equal for **UP** digraphs, which are the subgraphs of planar st-digraphs. As already suggested before, this indicates that the complexity lies in the augmentation of a digraph to an acyclic dipole or st-digraph. In Sect. 3.5, we will see that the situation is similar for **RUP** digraphs, for which the general decision problem is also $\mathcal{NP}$-hard. By our **RUP** characterization, we know that every **RUP** digraph can be augmented to a closed **RUP** digraph. In turn, for closed digraphs, we will derive a linear-time decision algorithm. This algorithm consist of three parts: First, it tests the **RUP**-embeddability of each compound

and transit of the compound digraph separately. Whereas testing a transit is essentially an ordinary planarity test, we have to put more effort into compounds. For the latter, we derive a characterization of **RUP** compounds by means of their block-cut trees and duals which yields a decision algorithm (Sect. 3.5.2). Finally, we tackle the blocks in Sect. 3.5.3, i. e., the biconnected components of the compounds, by using SPQR trees [DBT96]. For this, we extend SPQR trees to incorporate edge directions, cycles, and, most importantly, duals.

In Sect. 3.6, we use dipoles and the compound digraph for a characterization of **wSUP** digraphs and their duals, i. e., weakly upward planar digraphs on the standing cylinder and their duals. Finally, in Sect. 3.7, we conclude, make some further remarks, and give pointers to future work.

3.4 Characterizing RUP Digraphs by their Duals

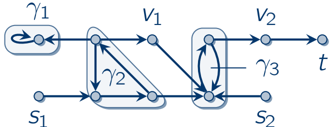
The aim of this section is to derive a characterization of **RUP** digraphs for which we use their duals. Before we can do this, we need to introduce some novel concepts: the compound digraph, introduced next, turns out to be a particularly useful tool. In the following, we only consider connected digraphs as a digraph is **RUP** if and only if each of its connected components is **RUP**. Remember, a digraph is **RUP** (**SUP**) if it has a **RUP** (**SUP**) embedding, and an embedding is **RUP** (**SUP**) if there is a **RUP** (**SUP**) drawing that is upward plane on the rolling (standing) cylinder endowed with the homogeneous field (Tab. 3.1). Also recall, an acyclic dipole is an acyclic digraph with exactly one source and exactly one sink.

3.4.1 The Compound Digraph

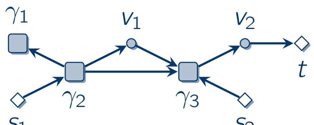
The compound digraph can be seen as a high-level description of a given digraph by means of non-trivial strongly connected and acyclic components. By the compound digraph, we eventually generalize acyclic dipoles to dipoles.

Recall the definition of the component digraph as given in Sect. 1.1.6: In the component digraph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, the vertices are the (strongly connected) components and $\sigma(v)$ denotes the component to which vertex v belongs. For each edge $(u, v) \in E$ with $\sigma(u) \neq \sigma(v)$, there is an edge $(\sigma(u), \sigma(v)) \in \mathbb{E}$ in the component digraph and, hence, the component digraph is an acyclic multigraph. If G is embedded, we assume that $\mathbb{G}$ inherits the embedding. For an example, consider the digraph shown in Fig. 3.9(a). Its component digraph is shown in Fig. 3.9(b). A component is a *compound*, denoted by γ, if it contains at least one edge, i. e., it either contains more than one vertex or a single vertex with a loop. In any case, a compound always contains a cycle. In Fig. 3.9(a), the vertices belonging to a compound are drawn on a shaded background. In the component digraph, the vertices belonging to a single compound are "shrunk" to a single vertex. We display compounds by rounded rectangles, e. g., compounds γ_1, γ_2, and γ_3 in Fig. 3.9(b). For the sake of convenience, we identify γ with the component for which it stands and call both compound. The set of all compounds is denoted by $\mathbb{V}_C$. Each component $\sigma(v)$ that is *not a compound* consists of a single vertex with no edges and is called *trivial component*, e. g., v_1, v_2, s_1, s_2, and t in Fig. 3.9(b). A trivial component which is a source (sink) in $\mathbb{G}$ is called *source (sink) terminal* and the set of all terminals is denoted by $\mathbb{T} \subseteq \mathbb{V}$. We display terminals by diamond shapes that are white.

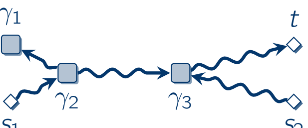
Based on the component digraph, we define the *compound digraph* $\overline{\mathbb{G}} = (\mathbb{V}_C \cup \mathbb{T}, \overline{\mathbb{E}})$, whose vertices are the compounds and terminals. Let $u, v \in \mathbb{V}_C \cup \mathbb{T}$ be two vertices of the compound



(a) A planar digraph whose compounds are enclosed by shaded rounded rectangles.



(b) The component digraph, where the compounds are displayed as rounded rectangles and the terminals by a diamond shape.



(c) The compound digraph where the transits are displayed by sinuous lines.

Figure 3.9: A digraph, its component digraph, and its compound digraph.

digraph. There is a *transit* $(u, v) \in \overline{\mathbb{E}}$ if there is a dipath $u \rightsquigarrow v$ in $\mathbb{G}$ which internally visits only trivial components. For our example, the compound digraph is shown in Fig. 3.9(c), where the transits are shown as sinuous lines. For instance, there is a transit between γ_2 and γ_3, as there is a dipath $\gamma_2 \to v_1 \to \gamma_3$ in the component digraph in Fig. 3.9(b). Note that in contrast to the component digraph, $\overline{\mathbb{G}}$ is simple. Also observe that the compound digraph is acyclic as the component digraph is acyclic.

Each edge $\tau \in \overline{\mathbb{E}}$ corresponds to a set of dipaths in $\mathbb{G}$. Hence, $\tau = (u, v)$ induces an acyclic subgraph of $\mathbb{G}$ which contains exactly one source and one sink, i. e., it is an acyclic dipole. We identify τ with its induced subgraph and call both *transit*. For example, the transit pointing from γ_2 to γ_3 corresponds to the acyclic dipole consisting of compounds γ_2 and γ_3, and trivial component v_1 in Fig. 3.9(b). Based on these definitions, we are able to define (general) dipoles:

Definition 3.1 (Dipole). *A digraph is a* dipole *if it has exactly one source s and one sink t and its compound digraph is a dipath from s to t.*

Intuitively, a dipole is a digraph with a very linear structure. In particular, the compounds and transits are totally ordered in a dipole, an observation which we will use in the following. Note that similar to the definition of *st*-digraphs [Kel87, DBT88], a dipole is not necessarily planar. Also observe that Def. 3.1 is "backwards compatible" in the following sense: if an acyclic digraph is a dipole, its compound digraph contains no compound and exactly one transit from the single source to the single sink, hence, it is an acyclic dipole. From the compound digraph in Fig. 3.9(c), we can conclude that the original digraph from Fig. 3.9(a) is not a dipole. Removing compound γ_1 and terminal s_2 yields a dipole. The following characterization of dipoles will be particularly useful:

Lemma 3.1. *Let $G = (V, E)$ be a digraph with at least one source s and at least one sink t. Then, G is a dipole if and only if the following conditions hold:*

(i) For every vertex $v \in V$, there are dipaths $s \rightsquigarrow v$ and $v \rightsquigarrow t$.

(ii) Every dipath $s \rightsquigarrow t$ contains at least one vertex of each compound.

Proof. $\Rightarrow$: We start with (ii). Since G is a dipole, its compound digraph $\overline{\mathbb{G}} = (\mathbb{V}_C \cup \mathbb{T}, \overline{\mathbb{E}})$ is a dipath $p = (v_1, v_2, \dots, v_k)$ with $s = v_1$ and $v_k = t$ and $(v_i, v_{i+1}) \in \overline{\mathbb{E}}$ for $1 \le i < k$. There is a transit $(u, v) \in \overline{\mathbb{E}}$ if and only if there is a dipath $u \rightsquigarrow v$ in the component digraph $\mathbb{G}$ which

internally visits only trivial components. For contradiction, assume that the dipath $s \rightsquigarrow t$ does not visit all the compounds of G. Then, there is a transit $(v_i, v_j) \in \overline{\mathbb{E}}$ with $j \neq i+1$ and, hence, the compound digraph is no dipath, which is a contradiction. Therefore, (ii) follows.

Let $p = s \rightsquigarrow t$ be a dipath in G. Since p contains at least one vertex of each compound and each compound is strongly connected, there are also dipaths $s \rightsquigarrow v$ and $v \rightsquigarrow t$ for each vertex v in a compound. What is left to show is that there are also dipaths $s \rightsquigarrow \hat{v}$ and $\hat{v} \rightsquigarrow t$ for each trivial component $\hat{v} \neq s, t$. Assume for contradiction that there is no dipath $s \rightsquigarrow \hat{v}$. Let $\hat{v}$ be a trivial component and $\hat{V} \subseteq V$ be the set of vertices $u \in \hat{V}$ for which there is a dipath $u \rightsquigarrow \hat{v}$. No vertex u' of a compound can be in $\hat{V}$ since, otherwise, there would be a dipath $s \rightsquigarrow u'$ and, therefore, also a dipath $s \rightsquigarrow \hat{v}$. Hence, the subgraph induced by $\hat{V}$ is an acyclic subgraph of G which does not contain s. Therefore, there must be a source $\hat{s} \in \hat{V}$ with $\hat{s} \neq s$; a contradiction. By the same reasoning it can be shown that there is a dipath from every vertex to the sink t and (i) follows.

$\Leftarrow$: Let s be a source and t be a sink of G. Since $s \rightsquigarrow v$ and $v \rightsquigarrow t$ for every $v \in V$ by (i), s is the single source and t the single sink of G. Let p be a dipath from s to t in G. By (ii), dipath p contains at least one vertex from each compound. Whenever p leaves a compound γ it does so by an edge $e = (u, v)$ that belongs to a transit. In particular, p can never return to γ as otherwise v would also belong to γ. Hence, in $\overline{\mathbb{G}}$, p corresponds to a dipath $\overline{p} = s \rightsquigarrow t$ that is Hamiltonian, i. e., $\overline{p}$ visits s, t, and each compound exactly once. This and the fact that $\overline{\mathbb{G}}$ is acyclic implies that $\overline{\mathbb{G}}$ is a dipath from s to t. □

3.4.2 Directed Duals and Dicuts

We introduce some further terminology and concepts related to (directed) duals and known results about them. In the following, we deal only with digraphs and, hence, whenever we speak of the dual we mean the directed dual. For the definition of duals along with an example, see Sect. 1.1.8. Moreover, whenever a digraph G has a dual G^*, we assume that both G and G^* are embedded. In particular, G is planar in this case.

Let G be a digraph with dual G^*. The dual of G^* is (isomorphic to) G^{-1}, where G^{-1} is the converse of G, i. e., all edge directions are inverted. Moreover, we have the following statement:

Proposition 3.1. *A digraph G is acyclic/strongly connected/an (acyclic) dipole/upward planar/***RUP***/***wSUP** *if and only if the same holds for its converse G^{-1}.*

Fig. 3.10(a) shows a **RUP** drawing of a *closed digraph*, i. e., it contains neither sources nor sinks. Its dual is shown in Fig. 3.10(b). Recall that a dual edge points from the face to the left of the primal edge to the face on its right side when "looking" into the direction of the primal edge. Consider face t, which is a sink in the dual. As t has no outgoing edges, it is *rightmost* in the sense that t is not to the left of any primal edge. The boundary of t is a directed cycle, which is at the right hand side of the fundamental polygon. In general, we call a cycle that is the boundary of a sink in the dual *rightmost*. Note that in general, an embedded digraph can have more than one rightmost cycle. Though, closed **RUP**-embedded digraphs always have only one rightmost cycle as we will see. Remember that a vertex or an edge is incident to a face, if it is part of the boundary of the face. We call a vertex or an edge of the primal *rightmost* if it is incident to a sink of the dual. Analogously, we say that a cycle of the primal is *leftmost* if it encloses a source of the dual, and a vertex or an edge of the primal is *leftmost* if it is incident to a source of the dual.

Let $G = (V, E)$ be a digraph and X be a proper and non-empty subset of V. The tuple $(X, V \setminus X)$ is called *dicut* [BJC00] if for any $x \in X$ and $y \in V \setminus X$, $(y, x) \notin E$, i. e., there are no edges pointing from $V \setminus X$ to X. We call $E_X \subseteq E$ *dicut-set* if there is a dicut $(X, V \setminus X)$ such that $E_X = \{(x, y) \in E \mid x \in X \wedge y \in V \setminus X\}$.

Proposition 3.2. *A digraph is strongly connected if and only if it has no dicut.*

Assume that G is embedded with dual G^* and let C be a simple cycle in G consisting of edges $e_1, \dots, e_k$. The set of dual edges $E^*_C = \{e^*_1, \dots, e^*_k\}$, where e^*_i is the dual edge of e_i $(1 \leq i \leq k)$, is a dicut-set [BJC00] with corresponding dicut (F_l, F_r), where $F_r := F \setminus F_l$. The faces $f \in F_l$ $(f \in F_r)$ are said *to lie to the left (right) of* C. For example, consider the rightmost cycle in Figs. 3.10(a) and 3.10(b) which defines a dicut $(F \setminus \{t\}, \{t\})$ in the dual. Although t lies geometrically to the right of the rightmost cycle in Fig. 3.10(b), this is only a topological property that must not be interpreted geometrically in general. The converse holds also true: Let (F_l, F_r) be a dicut of a dual graph $G^* = (F, E^*)$ with dicut-set E^*_C, then the primal edges E_C of E^*_C constitute a simple cycle C in the primal $G = (V, E)$ [BJC00]. Using dicuts, we prove the following lemma that we use thoroughly in the remainder of this section.

Lemma 3.2. *An embedded digraph G is acyclic if and only if its dual G^* is strongly connected, and an embedded digraph G is strongly connected if and only if its dual G^* is acyclic.*

Proof. We only prove the first part, i. e., G is acyclic if and only if G^* is strongly connected. The second part follows analogously.

$\Rightarrow$: Let $G = (V, E)$ be an embedded and acyclic digraph, and suppose for contradiction that the dual $G^* = (F, E^*)$ is not strongly connected. Then, by Prop. 3.2, G^* has a dicut (F_l, F_r) and, hence, G contains a cycle; a contradiction.

$\Leftarrow$: Suppose for contradiction that G has a cycle, then G^* contains a dicut and is not strongly connected by Prop. 3.2; again a contradiction. □

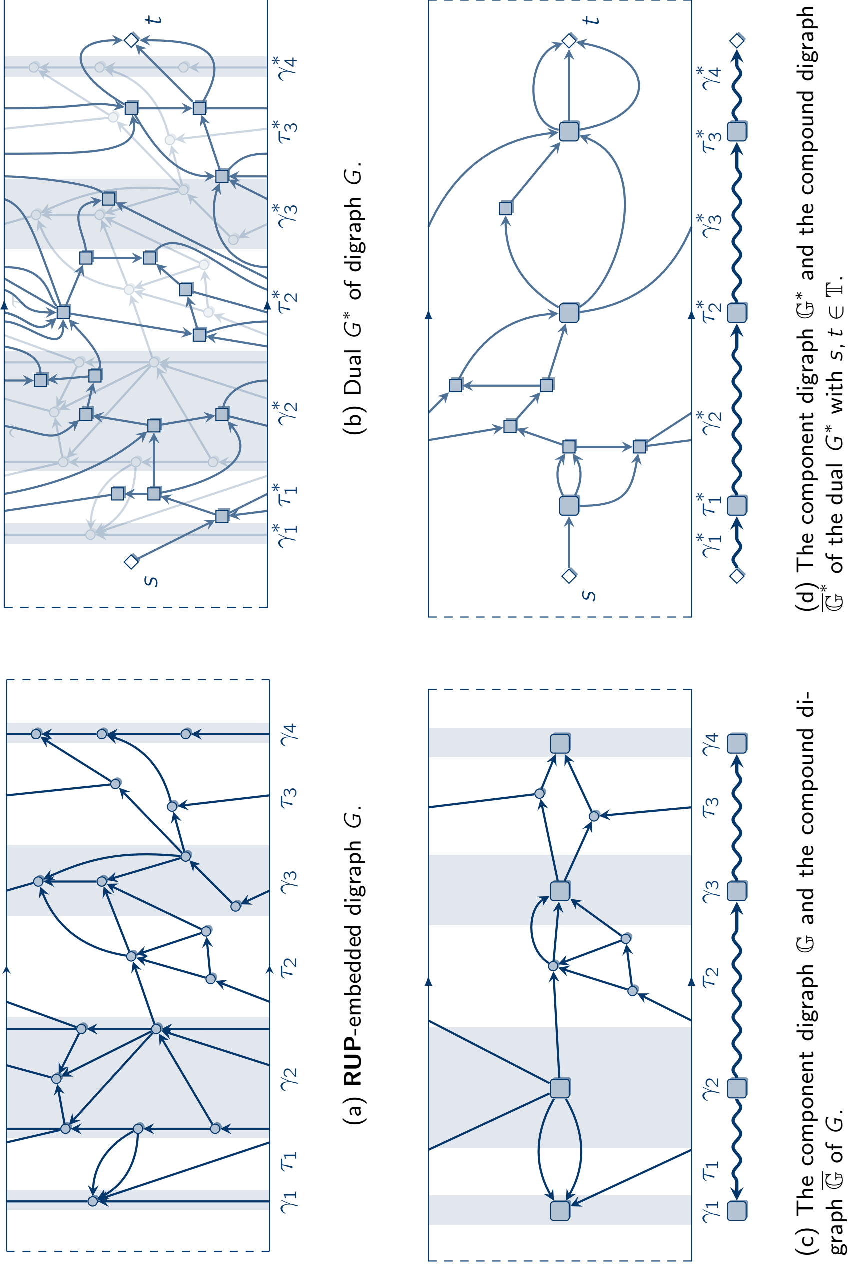
3.4.3 RUP Digraphs and their Duals

We are now equipped with the right tools to study **RUP** digraphs and their duals. Ultimately, we will arrive at the following characterization of **RUP**:

Theorem 3.1. *A digraph G is* **RUP** *if and only if G is a spanning subgraph of a closed planar digraph H whose dual is a dipole.*

The theorem is proved by a series of lemmas which are also of interest in their own. In the first part of this section, we deal with closed digraphs only and characterize closed **RUP** digraphs. In the last part, we lift this restriction by showing that all **RUP** digraphs are spanning subgraphs of closed **RUP** digraphs.

For our first observation, consider the **RUP** drawing of digraph G in Fig. 3.10(a), where all vertices within a compound are drawn on a shaded background. The component digraph $\mathbb{G}$ of G is displayed in Fig. 3.10(c) along with its compound digraph $\overline{\mathbb{G}}$ below. Consider compound γ_2 and its dual displayed in Fig. 3.11(a). As γ_2 is strongly connected, its dual is acyclic. In other words, a compound becomes a transit when going from the primal to the dual. The converse is also true: Consider transit τ_2, which is acyclic, and, thus, its dual is strongly connected as shown in Fig. 3.11(b). Therefore, τ_2 is becomes a compound of the dual. Also note that the compound digraph $\overline{\mathbb{G}}$ of G has the structure of an (undirected) path. Since compounds become



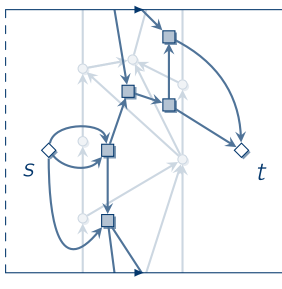
(a) **RUP**-embedded digraph G.

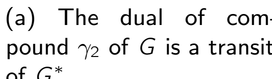
(b) Dual G^* of digraph G.

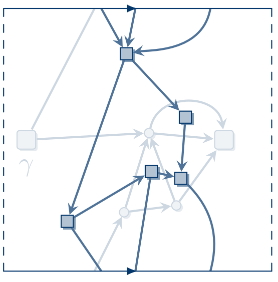
(c) The component digraph $\mathbb{G}$ and the compound digraph $\overline{\mathbb{G}}$ of G.

(d) The component digraph $\mathbb{G}^*$ and the compound digraph $\overline{\mathbb{G}}^*$ of the dual G^* with $s, t \in \mathbb{T}$.

Figure 3.10: A **RUP**-embedded digraph with its dual and their component and compound digraphs.




(a) The dual of compound γ_2 of G is a transit of G^*.



(b) The dual of transit τ_2 of G is a compound of G^*.

Figure 3.11: Transits and compounds swap their roles when going from primal to dual.

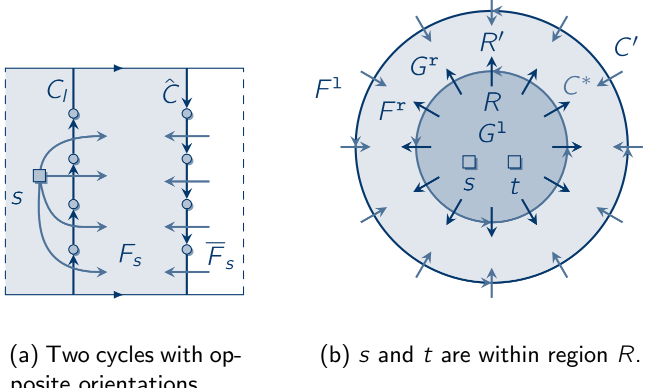
transits and transits become compounds when deriving G^* from G, the path-like structure of $\overline{\mathbb{G}}$ is also visible in the compound digraph of the dual which we denote by $\overline{\mathbb{G}}^*$ (cf. 3.10(d)). Also note that all cycles in the **RUP** drawing have the same orientation, i. e., they all wind around the cylinder in the same direction. Therefore, all transits of G^* point into the same direction. The final observation we make is that G contains neither sources nor sinks and, thereby, at the left and right border of G's **RUP** drawing is a leftmost and a rightmost cycle, respectively. In the dual, the leftmost cycle encloses a source s and the rightmost cycle a sink t. In fact, s is the single source and t the single sink of G^*. All these observations together indicate that the compound digraph of G^* is a dipath $s \rightsquigarrow t$ (see Fig. 3.10(d)), i. e., G^* is a dipole. We prove this in general.

Lemma 3.3. *The dual G^* of a closed* **RUP***-embedded digraph G is a dipole.*

Proof. First, we have to argue that $G^* = (F, E^*)$ contains at least one source and at least one sink. Let $f \in F$ be the "leftmost" face, i. e., the face that contains the border on the left hand side of the cylinder. Face f's boundary consists of two parts: one part is the left border of the cylinder and the other consists of edge curves of G's drawing. As G is closed, it contains at least one cycle that winds exactly once around the cylinder [Bra14], i. e., the left border of the cylinder is "separated" from the right border by the drawing. Hence, the part of f's boundary that belongs to G's drawing winds exactly once around the cylinder. Let v be any vertex incident to f. As G is closed and **RUP**-embedded, v has one incoming and one outgoing edge that are both incident to f. Hence, f's boundary is a cycle and f is a source in G^*. By an analogous reasoning, there is also a sink in G^* that is the rightmost face.

Let s be a source and t be a sink in G^*. We use Lem. 3.1 to show that G^* is a dipole. We start with property (i), i. e., there are dipaths $s \rightsquigarrow f$ and $f \rightsquigarrow t$ in G^* for every face $f \in F$. Let F_s be the set of faces reachable from s with $F_s = \{f \in F \mid s \rightsquigarrow f \text{ in } G^*\}$. Assume for contradiction that $F_s \subsetneq F$. F is partitioned into F_s and $\overline{F}_s = F \setminus F_s$. By the definition of F_s, any edge connecting a face in F_s with a face in $\overline{F}_s$ must point from $\overline{F}_s$ to F_s. In other words, $(\overline{F}_s, F_s)$ is a dicut of G^*. Let E_s^* be the corresponding dicut-set. The primal edges of E_s^*, denoted by E_s, form a cycle $\hat{C}$ in G. Further, denote by C_l the leftmost cycle in G that encloses source s. Fig. 3.12(a) illustrates the situation, where the shaded area covers the faces

F_s. Cycle $\hat{C}$ winds around the cylinder in the opposite direction of C_l, which contradicts our assumption of a **RUP** embedding. Analogously, it can be shown that there is a dipath from every face to sink t. This also implies that s is the single source and t the single sink of G^*.



(a) Two cycles with opposite orientations.

(b) s and t are within region R.

Figure 3.12: Situation obtained in the proof of Lem. 3.3.

Let $p^* = s \rightsquigarrow t$ be a dipath in G^* from s to t, which must exist by the reasoning from before. We now prove (ii), i. e., p^* contains a face of each compound in G^*. If G is strongly connected, then G^* is acyclic and contains no compounds at all and we are done. Hence, we assume that G^* contains at least one compound γ^* which contains cycle C^*. We show that p^* contains at least one face from C^* and, therefore, a face of γ^*. For contradiction, suppose that p^* contains no face of C^*. Consider a plane drawing in $\mathbb{R}^2$ of G^* that respects its embedding. In this drawing, C^* encloses a region R. If s and t are at opposite sides of C^*, e. g., s inside R and t outside, then path p^* needs to cross an edge of C^* to connect s and t by Jordan's curve theorem (Prop. 1.1), contradicting planarity. Hence, either both s and t are situated within R or both outside of R. First, we assume the former. The situation we obtain in depicted in Fig. 3.12(b). In the primal G, C^* defines a dicut (V^l, V^r) with $V^r = V \setminus V^l$. Let $G^r = (V^r, E^r)$ be the subgraph of G induced by V^r. Depending on the orientation of C^*, G^r lies either completely outside or inside of R. We assume that G^r lies outside, where the proof is analogous for the other case. In G^r, each vertex has at least one outgoing edge as G is closed and there is no edge that points from a vertex in V^r to a vertex in V^l. In other words, G^r contains no sink and, hence, it contains a cycle C', where C' encloses a region R' such that $R \subsetneq R'$ (Fig. 3.12(b)) or $R' \subsetneq R$. Remember that both s and t lie within R. Cycle C' defines a dicut (F^l, F^r) in G^*, where either $s, t \in F^l$ or $s, t \in F^r$ depending on the orientation of C' and on whether $R \subsetneq R'$ or $R' \subsetneq R$. If $s, t \in F^l$, there is no dipath from any face in F^r to t, and if $s, t \in F^r$ there is no dipath from s to any face in F^l (see Fig. 3.12(b)); a contradiction to (i). The case where both s and t are situated outside of R is analogous. Hence, p^* contains a vertex from C and, hence, a vertex from γ^* and (ii) follows. □

Remember Fig. 3.8 on page page 106 which illustrates the relationship between Ampère's law from physics and Ampère's for dual digraphs which states that the roles of **SUP** and **RUP** swap when going from the primal to the dual. This relationship is expressed by the following lemma.

Lemma 3.4 (Ampère's Law for Duals). *The embedding of a strongly connected digraph is* **RUP** *if and only if its dual is an acyclic dipole.*

We need the following definition for the proof of Lem. 3.4. By *rotation around the rolling cylinder* we mean the transformation:

$$T : I \times I_\circ \to I \times I_\circ : (x, y) \mapsto (x, (y + \Delta + 1) \bmod 2 - 1) \tag{3.1}$$

for some $\Delta \in \mathbb{R}$, which also can be considered as a translation in y-direction by Δ. Note that rotating a **RUP** drawing does neither affect its upwardness nor its implied embedding.

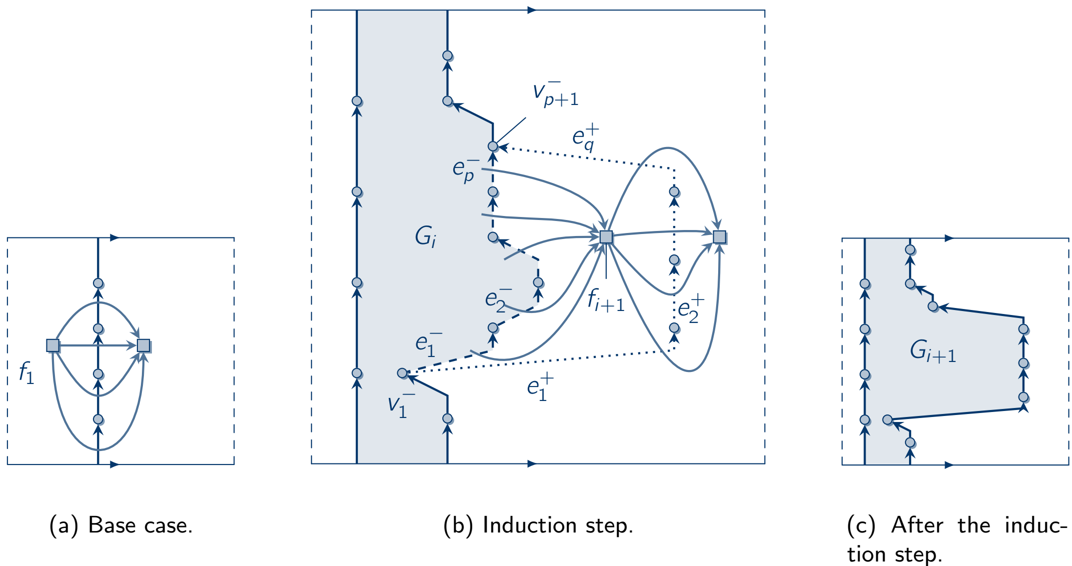
Proof. $\Rightarrow$: Follows directly from Lemmas 3.2 and 3.3.

$\Leftarrow$: To show that G is **RUP**-embedded, we inductively construct a **RUP** drawing of G on the fundamental polygon of the rolling cylinder such that the embedding of G is preserved. As G^* is an acyclic dipole, we obtain a topological ordering $f_1, \dots, f_k$ $(k \leq 1)$ of the faces, where f_1 is the single source and f_k is the single sink of G^*. Let G_i $(1 \leq i \leq k)$ be the embedded subgraph of G induced by the faces $f_1, \dots, f_i$, i. e., G_i contains exactly those edges and vertices bounding the faces $f_1, \dots, f_i$. Eventually, we obtain a **RUP** drawing of $G_k = G$.

The basic idea of the inductive proof is to add edges to G_i such that f_{i+1} is enclosed as new face and f_{i+1} lies to the left of all newly added edges. To assure a plane drawing, the x-coordinates of the newly added vertices are strictly greater than the x-coordinates of all vertices in G_i. Let $x_1, \dots, x_k$ with $x_i \in I$ for $1 \leq i \leq k$ be a sequence of strictly increasing x-coordinates, i. e., $-1 < x_i < x_{i+1} < 1$ for all $1 \leq i < k$. As induction invariant, for each G_i, we obtain a **RUP** drawing Γ_i which respects the embedding of G_i and lies within $[x_1, x_i] \times I_\circ$. Additionally, the dual G_i^* of each G_i is a planar, acyclic dipole. Especially, the right border of Γ_i is a directed cycle and all faces $f_1, \dots, f_i$ are to the left of this cycle. Moreover, the construction of the drawing is of a very special kind that assures that we can always introduce the next face without causing a crossing.

For the base case, consider G_1. Since f_1 is a source in G^*, G_1 consists of a single cycle C with $\mathrm{d}^+(f_1)$ many edges, where $\mathrm{d}^+(f_1)$ is the outdegree of f_1. All vertices of C receive the x-coordinate x_1 and their y-coordinates are chosen according to the cyclic order as defined by C, where the total order induced by the y-coordinates of the vertices implies the cyclic order as defined by C. See Fig. 3.13(a) for an illustration. The drawing of G_1 guarantees the induction invariants. Especially, G_1^* is an acyclic dipole with source f_1 and a single sink to the right of C.

Now assume that $1 < i < k - 1$. We obtain the situation depicted in Fig. 3.13(b). In the embedding of G_{i+1}^*, all incoming edges are consecutive in the rotation system of f_{i+1} and so are all outgoing edges. This follows from the fact that G^* is an embedded planar acyclic dipole and, therefore, **SUP**-embedded which implies that its rotation system is bimodal [BDBD02]. Denote by $e_1^-, \dots, e_p^-$ (dashed) and $e_1^+, \dots, e_q^+$ (dotted) the primal edges of all incoming and outgoing edges of f_{i+1}, respectively, where the sequence of duals of $e_1^+, \dots, e_q^+, e_p^-, e_{p-1}^- \dots, e_1^-$ in order is the rotation system of f_{i+1}. Note that f_{i+1} has at least one incoming edge as otherwise it would be a source different from f_1. Analogously, f_{i+1} has at least one outgoing edge. Due to the topological ordering of the faces, all faces that have an outgoing edge to f_{i+1} are already present in the drawing of G_i. Additionally, all edges e_j^- are part of the rightmost cycle C_r of G_i. For contradiction assume that is not the case. Then, there is an edge e_j^- $(1 \leq j \leq p)$ where the endpoints of its dual are both to the left of C_r. However, then either e_j^- cannot be an edge bounding f_{i+1} or Γ_i does not respect the embedding of G_i; a contradiction to the induction hypothesis. Moreover, since Γ_i respects the rotation system of f_{i+1}, the edges



(a) Base case.

(b) Induction step.

(c) After the induction step.

Figure 3.13: Inductive construction of a **RUP** drawing from its dual

$e^-_1, \dots, e^-_p$ form a dipath $p^- = (v^-_1, \dots, v^-_{p+1})$ in G_i, which is part of C_r. Let $a \in I_\circ$ and $b \in I_\circ$ be the y-coordinates of v^-_1 and v^-_{p+1}, respectively. Note that v^-_{p+1} must lie "above" v^-_1 as p^- must be drawn upward. If $a > b$, we rotate the drawing around the cylinder by some value Δ until $a < b$ and, w. l. o. g., we assume that $a < b$. Accordingly, the edges $e^+_1, \dots, e^+_q$ correspond to a dipath $p^+ = (v^+_1, \dots, v^+_{q+1})$ in G_{i+1}. Note that $v^+_1 = v^-_1$ and $v^+_{q+1} = v^-_{p+1}$ since p^+ and p^- together bound face f_{i+1}. We assign to each vertex $v^+_2, \dots, v^+_q$ the x-coordinate x_{i+1}. As y-coordinate, we choose for every vertex v^+_j a value y^+_j such that for all $1 \le j < q$ the inequality $a < y^+_j < y^+_{j+1} < b$ holds. Now, the edges of p^+ can be drawn upward as straight lines with a single bend at the first and the last edge of the dipath; see Figs. 3.13(b) and 3.13(c). For the position of the bend in edges e^+_1 and e^+_q, we choose as y-coordinate some value in the intervals (a, y^+_2) and (y^+_q, b), respectively, such that the straight lines from v^-_1 and v^-_{p+1} to the bends cause no crossing with any edge from p^-. This is always possible due to the construction of the drawing. For the x-coordinate of the bends, we choose x_{i+1}. In the case that p^+ consists of a single edge, this edge has two bends. The resulting drawing Γ_{i+1} of G_{i+1} is a **RUP** drawing respecting the embedding of G and it lies within $[x_1, x_{i+1}] \times I_\circ$. In Γ_{i+1} there is a newly formed cycle C'_r containing p^+ on the right border of the drawing such that all faces $f_1, \dots, f_{i+1}$ lie to the left of C'_r, see Fig. 3.13(c).

In the drawing of G_{k-1}, face f_k is already existent as it is the single face to the right of the rightmost cycle in G_{k-1}. Hence, $G_{k-1} = G_k = G$ of which all are **RUP**-embedded. □

Since every embedded acyclic digraph is **SUP**-embedded [Has01], Lem. 3.4 implies:

Corollary 3.1. *The dual of a strongly connected* **RUP***-embedded digraph is* **SUP***-embedded.*

Consider again the component digraph $\mathbb{G}$ and the compound digraph $\overline{\mathbb{G}}$ in Fig. 3.10(c) of the **RUP** digraph G in Fig. 3.10(a). In the dual G^* of G, compounds and transits of G swap their roles, i. e., compounds become transits and vice versa (cf. Fig. 3.10(d)). This is

due to the fact that transits are acyclic and contain at least one edge and, hence, their duals are compounds. Analogously, the duals of compounds are transits. By Lem. 3.4, this duality between compounds and transits is even more profound. For this, consider compound γ_2 in Fig. 3.10(a). As the whole digraph is **RUP**-embedded, especially γ_2 is **RUP**-embedded and, thus, its dual, depicted in Fig. 3.11(a), is a **SUP**-embedded transit. For the transits, the same holds but with swapped roles, i. e., the dual of a transit, which is **SUP**-embedded, is a **RUP**-embedded compound. As an example, the dual of the transit τ_2 in Fig. 3.10(a) is shown in Fig. 3.11(b). The following lemma subsumes these observations.

Corollary 3.2. *Let G be a closed* **RUP***-embedded digraph and let $\overline{\mathbb{G}} = (\mathbb{V}_C, \overline{\mathbb{E}})$ be its compound digraph. Then, the following statements are true:*

(i) The dual of each compound is a **SUP***-embedded transit.*

(ii) The dual of each transit is a **RUP***-embedded compound.*

Proof. (i): The compound of a **RUP**-embedded digraph is also **RUP**-embedded, by Cor. 3.1, it is a **SUP**-embedded transit.

(ii): A transit of a digraph is an acyclic dipole. Since G is (**RUP**-)embedded, the transit is also **SUP**-embedded [Has01]. Thereby, its dual is a **RUP**-embedded compound (Lem. 3.4). □

Note that the orientation of the cycles of a transit's dual depends on the direction of the transit in the primal. For instance, the dual of τ_1 in Fig. 3.10(b) winds around the cylinder in upward direction as τ_1 points from γ_2 to γ_1, whereas the dual of τ_3 winds around the cylinder "downwards" as τ_3 points from γ_3 to γ_4. In particular, the dual of a **RUP**-embedded digraph is not necessarily **RUP**.

Let $\overline{\mathbb{G}}^*$ the compound digraph of the dual G^* of a closed **RUP**-embedded digraph G. We denote the dual of a compound γ of G by γ^*, where γ^* is a transit of G^*. Likewise, we denote the dual of a transit τ of G by τ^*, where τ^* is a compound of G^*. By Lem. 3.3, the dual of a closed **RUP**-embedded digraph is a dipole. We now prove the converse.

Lemma 3.5. *A closed digraph G is* **RUP***-embedded if its dual G^* is a dipole.*

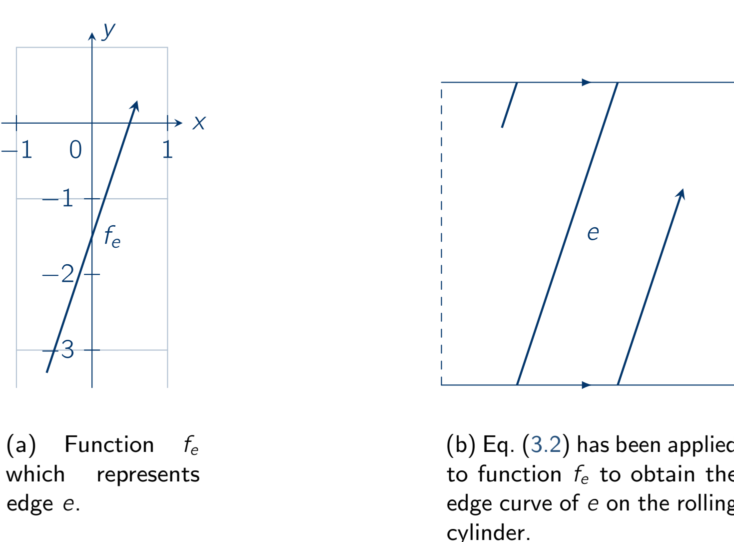
The basic idea of the proof is as follows: Consider again the example in Fig. 3.10(a) and the compound digraph $\overline{\mathbb{G}}^*$ of its dual G^* in Fig. 3.10(d). Since G^* is a dipole, $\overline{\mathbb{G}}^*$ is a dipath $p = (s, \gamma_1^*, \tau_1^*, \gamma_2^*, \tau_2^*, \gamma_3^*, \tau_3^*, \gamma_4^*, t)$, where γ_i^* is the dual of compound γ_i, and τ_i^* is the dual of transit τ_i. Each element on p corresponds to a subgraph in the primal G, i. e., for each τ_i^* there is a transit τ_i in G and for each γ_j^* there is a compound γ_j in G. We construct a **RUP** drawing of G by subsequently attaching the elements of p to each other. We start with transit γ_1^*, which is an acyclic dipole, and obtain a **RUP** drawing of its primal γ_1 which respects the given embedding by Lem. 3.4. Then, we proceed with τ_1^*, a compound in G^*, for which we obtain a **SUP** drawing of its primal τ_1 which also respects the given embedding. However, note that this **SUP** drawing is upward only on the standing cylinder. In particular, rotating the **SUP** drawing by 90 degrees to obtain a drawing on the rolling cylinder does not necessarily produce a **RUP** drawing. Fortunately, we can transform the **SUP** drawing of τ_1, while preserving its embedding, such that it is also upward on the rolling cylinder. The so obtained drawing of τ_1 is then attached to the rightmost cycle of γ_1. Then, the **RUP** drawing of γ_2 is attached to the right side of τ_1, and so forth until we reach t. Note that since all transits γ_j^* point into the same direction in $\overline{\mathbb{G}}^*$, i. e., from s to t, all cycles of the compounds in G have the same

orientation in the obtained drawing which means that they all wind around the cylinder in the same direction. Before we prove Lem. 3.5, we need to show that a **SUP** embedding is also a **RUP** embedding.

Lemma 3.6. *A* **SUP***-embedded digraph is also* **RUP***-embedded.*

Proof. Let Γ be a **SUP** drawing of a digraph $G = (V, E)$ on the standing cylinder where all edge curves are monotonically increasing in y-direction. We transform Γ such that it becomes **RUP** and still implies the same embedding.

If we negate the x-coordinate of each point in Γ and swap the axes of the fundamental polygon, we obtain a drawing Γ' of G on the rolling cylinder where all edge curves are monotonically increasing in x-direction.[1] We assume that all edge curves are differentiable. Otherwise, critical points must be excluded from the following reasoning. The idea is to shear the drawing vertically such that the curves are increasing additionally in y-direction. Then, we have a **RUP** drawing. In the following, we denote by $x \bmod y$ the non-negative remainder $x - y\lfloor \frac{x}{y} \rfloor$ of dividing a real x by a real y.



(a) Function f_e which represents edge e.

(b) Eq. (3.2) has been applied to function f_e to obtain the edge curve of e on the rolling cylinder.

Figure 3.14: Edge curve represented by the partial function $f_e : I \times \mathbb{R} : x \mapsto 3(x - \frac{1}{2})$ with $\mathrm{dom}(f_e) = [-0.6, 0.6]$

Edge curves are usually represented by continuous maps of the interval $[0, 1]$ to points of the surface (see Sect. 1.1.7). In the following, we take a different approach and use real-valued partial functions as they allow for simpler mathematical treatment. As all edge curves are monotonically increasing in x direction, no edge curve has multiple points with the same x-coordinate. Thus, we can represent each curve of an edge e by a differentiable and partial function f_e from I to $\mathbb{R}$, whose domain is a closed interval, which we denote by $\mathrm{dom}(f_e)$. Then, let the drawing of e be the point set

$$\{(x, y) \in \mathrm{dom}(f_e) \times I_\circ : y = f_e(x) \bmod 2 - 1\}, \tag{3.2}$$

[1] Swapping the coordinates alone would alter the embedding as it reverses the cyclic order of incident edges. Negation cancels this effect.

At first glance, this definition may seem odd, but mapping the image of f_e via this function to y-coordinates on the rolling cylinder allows differentiable real-valued functions to represent all curves increasing in x-direction, even if they wind multiple times around the cylinder. See Fig. 3.14 for an example. Let a be the least gradient of all functions representing an edge of G:

$$a = \min_{e \in E} \min_{x \in \mathrm{dom}(f_e)} f_e'(x) .$$

If there are critical points, then choose $a = \min_{e \in E} \inf_{x \in \mathrm{dom}(f_e)} f_e'(x)$. If $f_e' > 0$ for all edge curves, all curves are increasing monotonically in y-direction and we are done. Otherwise, assume for the remainder of the proof that there is at least one function f_e with a non-positive gradient at some point, i. e., $a \leq 0$. We define a shearing transformation S by:

$$S : I \times I_\circ \to I \times I_\circ : (x, y) \mapsto (x, y + (1 - a) \cdot x) .$$

For simplicity, we omit the application of Eq. (3.2) to the y-coordinate. Denote by $S[\Gamma']$ the image of Γ' under S. Observe that S preserves the embedding of G, i. e., $S[\Gamma']$ is plane and implies the same embedding as Γ' and, especially, the same embedding as Γ. It remains to show that $S[\Gamma']$ is a **RUP** drawing. Let f_e represent an edge curve in Γ'. Then the function g_e representing the corresponding edge curve in the transformed drawing $S[\Gamma']$ is:

$$g_e : \mathrm{dom}(f) \to \mathbb{R} : x \mapsto f_e(x) + (1 - a) \cdot x .$$

The derivation of g_e is:

$$\forall_{x \in \mathrm{dom}(f)} \quad g_e'(x) = f_e'(x) + 1 - a > 0$$

since $f_e'(x) \geq a$ by the definition of a. Hence, all edge curves in $S[\Gamma']$ are monotonically increasing in y-direction. Since $S[\Gamma']$ implies the same embedding as Γ' and Γ, we can conclude that the **SUP** embedding of G is also a **RUP** embedding. □

We can now proof Lem. 3.5.

Proof. [Lem. 3.5] Let $G = (V, E)$ be an embedded closed digraph with dual $G^* = (F, E^*)$ and assume that G^* is a dipole. If G consists of a single compound, then it is strongly connected. Thus, G^* is an acyclic dipole, the embedding of G is a **RUP** embedding according to Lem. 3.4, and we are done. In the following we assume that G contains at least two compounds.

Let $s \in F$ and $t \in F$ be the source and the sink of G^*, respectively. We first show that each cycle C in G separates the faces s and t, i. e., s lies to the left and t to the right of C or vice versa. Cycle C defines a dicut $(F^{\mathfrak{l}}, F^{\mathfrak{r}})$ in G^*. Since G^* is a dipole, there is a dipath from s to any face $f \in F$ and from f to t. Hence, $s \in F^{\mathfrak{l}}$ and $t \in F^{\mathfrak{r}}$, and s is to the left- and t to the right of C.

Let C_1 and C_2 be two cycles in G belonging to different compounds. We show that C_1 and C_2 have the same orientation. As C_1 and C_2 belong to different compounds, they are vertex- and edge-disjoint. In the dual, C_1 and C_2 define two dicuts $(F_1^{\mathfrak{l}}, F_1^{\mathfrak{r}})$ and $(F_2^{\mathfrak{l}}, F_2^{\mathfrak{r}})$, respectively, with $s \in F_1^{\mathfrak{l}}, F_2^{\mathfrak{l}}$ and $t \in F_1^{\mathfrak{r}}, F_2^{\mathfrak{r}}$. If C_1 and C_2 have opposite orientations, we obtain the same situation as in the proof of Lem. 3.3 and as displayed in Fig. 3.12(a), where t is situated within region $\overline{F}_s$. In particular, there would be no dipath from s to t in G^* which is a contradiction due to Lem. 3.1.

By the reasoning in the previous paragraph, we can also conclude that the compounds of G properly nest, i. e., there is a total order $\gamma_1, \gamma_2, \ldots, \gamma_k$ of the compounds $\mathbb{V}_C$ of G with the

following properties: The region to the left of any cycle in compound γ_i $(1 < i < k)$ contains all vertices of compounds $\gamma_1, \dots, \gamma_{i-1}$, and the region to the right of any cycle in compound γ_i contains all vertices of compounds $\gamma_{i+1}, \dots, \gamma_k$. Compound γ_1 is the leftmost compound in the sense that no compound is to its left side and all other compounds are to its right side. In the same sense, γ_k is the rightmost compound.

In the following, consider a drawing of G in the plane which respects the given embedding. Fig. 3.15 shows the principle structure of such a drawing. The compounds are displayed as shaded rings and the arrows on the rings' borders indicate the direction of the compounds' cycles. Face s is situated in the middle and lies left of all compounds $\gamma_1, \dots, \gamma_k$. Remember that "left" must not be interpreted geometrically here but in the sense that s lies to the left of each cycle in $\gamma_1, \dots, \gamma_k$. Face t is the outer face to the right of all compounds. Let γ_i and γ_j be two compounds of G with $1 \leq i < j \leq k$ such that $j - i > 1$, i. e., in the ordering of the compounds, there is at least one compound between γ_i and γ_j. We now show that there is no transit between γ_i and γ_j, i. e., neither $(\gamma_i, \gamma_j) \in \overline{\mathbb{E}}$ nor $(\gamma_j, \gamma_i) \in \overline{\mathbb{E}}$. Assume for contradiction that a transit $\hat{\tau} = (\gamma_i, \gamma_j) \in \overline{\mathbb{E}}$ exists. If the transit points in the opposite direction, the following reasoning proceeds analogously. There is a dipath p from a vertex in γ_i to a vertex in γ_j which internally visits only trivial components. Further, there is at least one compound γ_ℓ between γ_i and γ_j $(i < \ell < j)$. In other words, $\hat{\tau}$ must "overleap" γ_ℓ. Compound γ_ℓ contains at least one cycle that encloses a region R such that γ_i is completely contained within R in the drawing. As p internally only visits trivial components, p cannot have a vertex in common with γ_ℓ. Moreover, p starts within region R and must reach a vertex of γ_j which is completely situated outside of R. This inevitably leads to a crossing which contradicts planarity. For instance, in Fig. 3.15, the transit $\hat{\tau}$ points from γ_1 to γ_3 and γ_2 is situated between them, which leads to a crossing.

Remember, that we assume that G is connected. Hence, there must be a transit between adjacent compounds γ_i and γ_{i+1}, i. e., for all i with $1 \leq i < k$, either $(\gamma_i, \gamma_{i+1}) \in \overline{\mathbb{E}}$ or $(\gamma_{i+1}, \gamma_i) \in \overline{\mathbb{E}}$. In the following, let $\gamma_1, \tau_1, \gamma_2, \dots, \tau_{k-1}, \gamma_k$ be the sequence of compounds and transits in G such that τ_i is the transit connecting compounds γ_i and γ_{i+1}. Analogously, let $\gamma_1^*, \tau_1^*, \gamma_2^*, \dots, \tau_{k-1}^*, \gamma_k^*$ be the sequence of compounds and transits in G^* in order of the dipath from the source to the sink in $\overline{\mathbb{G}}^*$.

By Lem. 3.4, we know that each compound in G has a **RUP** embedding. Further, each transit of G is an acyclic, planar dipole whose embedding is **SUP** and also **RUP** (Lem. 3.6). We conclude the proof by showing that the **RUP** embeddings of the individual compounds and transits can be merged into a single consistent **RUP** embedding of the whole digraph G. For this, we construct a **RUP** drawing of G by subsequently processing the elements in the order $\gamma_1, \tau_1, \gamma_2, \tau_2, \dots, \gamma_k$. For an example, see Fig. 3.16. As scaling a drawing in x-direction does not impair its upwardness in y-direction [ABBG11], we do not have to bother with the width of the rolling cylinder.

We start with transit γ_1^* of G^*, which is an acyclic dipole, and obtain a **RUP** embedding of γ_1 by Lem. 3.4 (see Fig. 3.16(b)). Let Γ_{γ_1} be a **RUP** drawing of γ_1 according to its **RUP** embedding. Denote by $C_1^{\mathfrak{r}}$ the rightmost cycle of γ_1. We proceed with τ_1^*, a compound in G^*. The primal τ_1 is a transit and, thus, an acyclic dipole. The embedding of τ_1 is **SUP** by [Has01]. Denote by Γ_{τ_1} the **SUP** drawing of τ_1. First assume that τ_1 points from γ_1 to γ_2. We shear Γ_{τ_1} as shown in the proof of Lem. 3.6 such that it becomes a **RUP** drawing and place it to the right of Γ_{γ_1}. Remember that τ_1 is not a subgraph of G but of its component digraph: τ_1 is an acyclic dipole, where the source s_{τ_1} and sink t_{τ_1} correspond to γ_1 and γ_2,

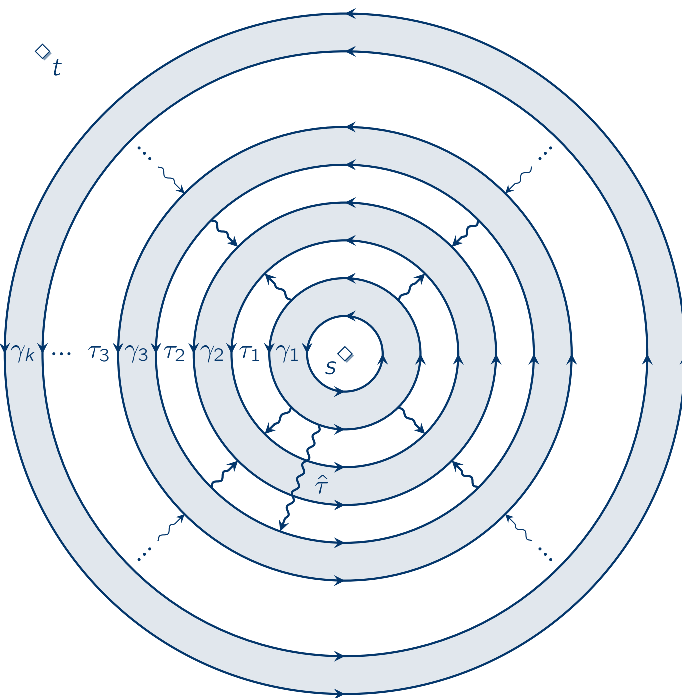
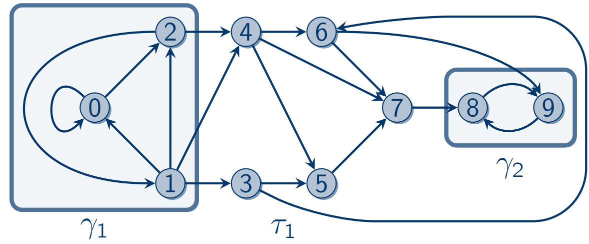


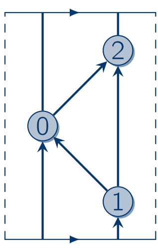
Figure 3.15: A transit which "overleaps" a compound causes a crossing.

respectively. All other vertices of τ_1 are trivial components and, hence, they directly correspond to vertices of G. The edges incident to s_{τ_1} in τ_1 correspond to edges in G which are incident to vertices in γ_1, more precisely vertices of $C_1^{\mathfrak{r}}$. Therefore, we remove s_{τ_1} and all points of its incident edge curves from Γ_{τ_1} within an axis-aligned, rectangular region R around s_{τ_1} that is chosen as follows (see Fig. 3.16(d)): R contains no points of Γ_{τ_1} besides those of s_{τ_1} and its incident edges, and its dimensions are such that all intersection points of R's boundary with Γ_{τ_1} are at the top side of the rectangle. Note such a rectangle exists as Γ_{τ_1} is upward in y-direction. This results in edge curves starting in cutting points rather than in s_{τ_1}, where all cutting point have the same y-coordinate. We rotate Γ_{τ_1} around the rolling cylinder such that the y-coordinate of the cutting points is greater than the y-coordinates of any of the vertices in γ_1. Let $e = (u, v)$ be the edge in G corresponding to the edge in τ_1 whose edge curve has the cutting point with the smallest x-coordinate. Vertex u is part of $C_1^{\mathfrak{r}}$ and v is a vertex in τ_1. Next we rotate the drawing of γ_1 such that u is the topmost vertex, but has a smaller y-coordinate than the cutting points. Since both the embedding of $C_1^{\mathfrak{r}}$ implied by Γ_{γ_1} and the embedding of τ_1 implied by Γ_{τ_1} obey the initial planar embedding of G, the order of the cutting points from right to left corresponds to the order of the vertices in $C_1^{\mathfrak{r}}$ from bottom to top. Hence, we can connect the vertices of $C_1^{\mathfrak{r}}$ with edge curves increasing monotonically in y-direction to the respective cutting points without introducing crossings.

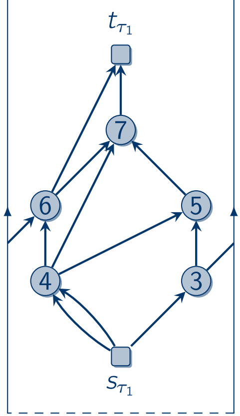
The resulting drawing Γ', see Fig. 3.16(e), forms the basis for the next step, where we obtain, again as in Lem. 3.4, a **RUP** drawing Γ_{γ_2} of compound γ_2 and place it to the right of Γ'. In a similar way, we remove t_{τ_1} from the drawing and reconnect the resulting cutting points to the respective vertices in the leftmost cycle of γ_2. If, contrary to our aforementioned assumption, a transit is directed from right to left, we proceed similarly except that we switch



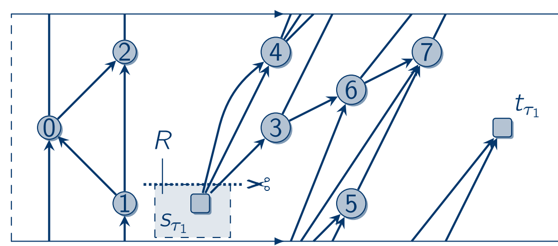
(a) A closed **RUP** digraph. The subgraphs within the shaded boxes are the two compounds γ_1 and γ_2. Transit τ_1 points from γ_1 to γ_2.



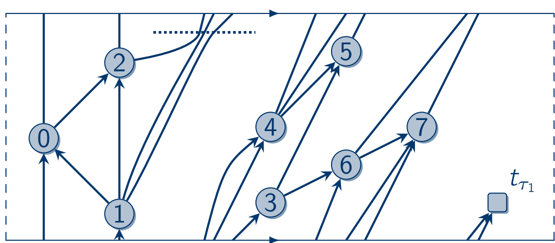
(b) **RUP** embedding of γ_1.



(c) **SUP** embedding of τ_1.



(d) Γ_{τ_1} has been sheared to become a **RUP** drawing and placed to the right of Γ_{γ_1}.



(e) Intermediate result Γ' after the drawings of γ_1 and τ_1 have been merged.

Figure 3.16: Construction of a **RUP** drawing

the roles of s_{τ_1} and t_{τ_1} and rotate the cutting points around t_{τ_1} to the bottom rather than the top. Analogously, we proceed with $\tau_2, \gamma_3, \tau_3, \gamma_4, \ldots$ until we have processed all components, resulting in a **RUP** drawing of G. □

Lemmas 3.3 and 3.5 both require that the digraph at hand contains neither sources nor sinks. In the following lemma, we show that each **RUP** digraph can be augmented by edges such that all sources and sinks vanish while still preserving **RUP**-embeddability.

Lemma 3.7. *A* **RUP** *digraph is a spanning subgraph of a closed* **RUP** *digraph.*

Before we prove this, we need a definition. For $\epsilon > 0$ and an arbitrary metric d on a surface $\mathbb{S}$, the *ϵ-environment* of a point set $P \subseteq \mathbb{S}$ is the union of all open balls with radius ϵ around points in P:

$$\bigcup_{p \in P} \{q \in \mathbb{S} : d(p,q) < \epsilon\} . \tag{3.3}$$

Proof. [Lem. 3.7] Consider an **RUP** drawing of a **RUP** digraph G. We iteratively add edges until all vertices have both incoming and outgoing edges. Let t be a sink of G. Shoot a ray from the position of t in upward direction and determine where it first meets some point p of the drawing. If p belongs to a vertex v, we can introduce a geodesic edge, i. e., a straight line on the fundamental polygon, from t to v. Note that $v = t$ if no other vertex or edge has a point with the x-coordinate of p such that the ray winds exactly once around the cylinder. If p belongs to an edge (u, v), proceed as follows: the drawing of $e = (u, v)$, i. e., the set of points belonging to the edge curve of e, has an ϵ-environment which contains no other point of the drawing except within the ϵ-environment of u and v. Thus, we can route a new edge (t, v) in upward direction and without introducing crossings, which goes first from t towards p on the ray, then runs alongside (u, v) such that it finally meets v. Analogously, we add incoming edges to the sources of G. □

The proof of Thm. 3.1 is now complete. The only-if direction follows from Lemmas 3.3 and 3.7 and the if direction is a consequence of Lem. 3.5 and the fact that every subgraph of a **RUP** digraph is a **RUP** digraph. We have now arrived at a characterization of **RUP**-digraphs by means of their duals. In the next section, we use this characterization to derive a linear-time algorithm that decides whether a closed digraph is **RUP**. Remember that in general deciding whether a digraph is **RUP** is $\mathcal{NP}$-complete [Bra14], which suggests that the complexity of the decision problem lies in the augmentation of a digraph to a closed digraph without violating **RUP**-embeddability. Note that in Lem. 3.7, we assume a **RUP** drawing of the digraph and, hence, we can aptly augment it such that it becomes closed.

3.5 Efficient Rolling Upward Planarity Testing of Closed Digraphs

The ultimate goal of this section is to prove the following theorem:

Theorem 3.2. *There is an algorithm that computes a* **RUP** *embedding for a closed digraph* $G = (V, E)$ *or returns* $\perp$ *if* G *is not* **RUP**. *The running time is in* $\mathcal{O}(|V|)$.

Instead of solving this problem all at once, our strategy is to further divide it into more manageable chunks. Fig. 3.17 sketches how this is done: We start with a closed digraph and use the compound digraph to separate it into its compounds and transits (Sect. 3.5.1). Whereas for the transits we can use a slightly modified, and yet ordinary, planarity testing algorithm, we need to put more effort into compounds. By using the block-cut tree, we divide each compound into its blocks, i. e., its biconnected components (Sect. 3.5.2). Each block is then analyzed separately using its SPQR tree, which describes how a block is composed of series and parallel compositions, and triconnected components (Sect. 3.5.3).

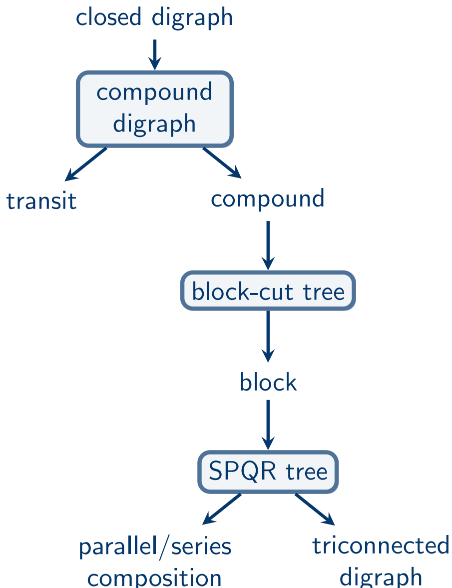


Figure 3.17: Sketch of how the algorithm divides the problem into smaller problems.

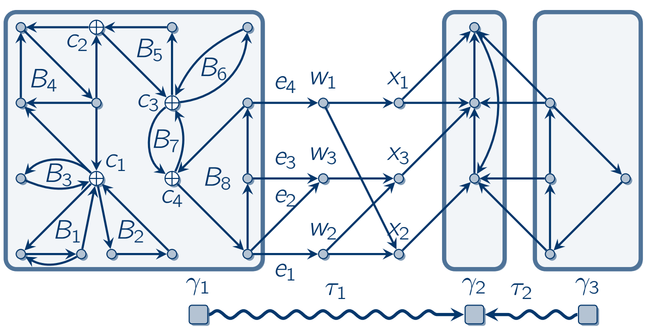
All of the following sections are organized similarly: The first part gives a characterization of **RUP** digraphs and develops tools using compound digraphs, block-cut trees, and SPQR trees, respectively. These characterizations and tools heavily rely on the dual digraph characterization derived in Sect. 3.4. As block-cut trees and SPQR trees in their original version incorporate no information about dual graphs or edge directions, we need to adapt them adequately. We do this in a general way that is not limited to **RUP** digraphs such that the developed ideas are applicable in a wider range. Based on these ideas, we derive an algorithm in the second part of each section that decides whether the input is **RUP** or not.

3.5.1 Closed Digraphs

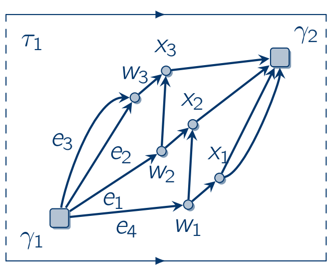
We start with the top level of the algorithm (cf. Fig. 3.17): We are given a closed digraph and separate it into its compounds and transits, and for each we test whether they have a

RUP embedding with some special properties. These special properties, which we derive in Sect. 3.5.1.1, ensure that we can assemble a **RUP** embedding of the whole digraph by "plugging" all compounds and transits together as in the proof of Lem. 3.5. In Sect. 3.5.1.2, we devise the respective algorithm that tests for these properties and assembles a **RUP** embedding for the whole digraph or returns $\perp$ if the digraph is not **RUP**. The algorithm given in Sect. 3.5.1.2 relies on the testing algorithm for compounds which is the topic of Sects. 3.5.2 and 3.5.3. We assume that the given closed digraph is connected as a closed digraph is **RUP** if and only if all of its connected components are **RUP**.

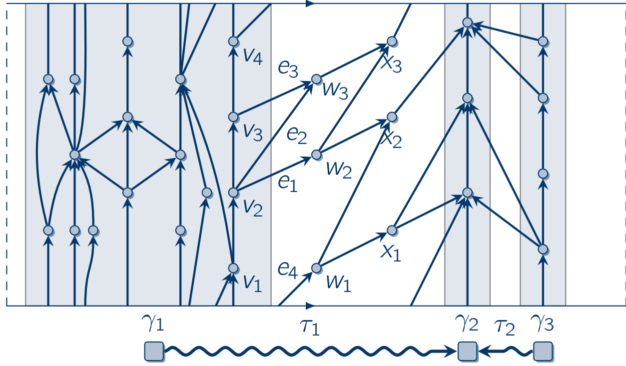
3.5.1.1 Transits and Compounds of Closed RUP Digraphs



(a) A closed digraph consisting of three compounds and two transits.



(b) Transit τ_1 and its **RUP** embedding.



(c) **RUP** embedding of the closed digraph.

Figure 3.18: A closed **RUP** digraph.

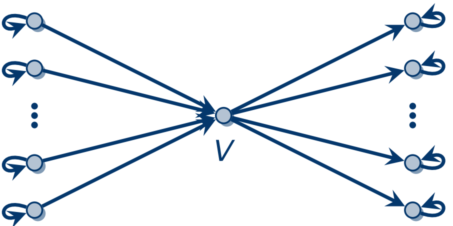
We use the closed digraph G displayed in Fig. 3.18(a) as an illustrative example. Fig. 3.18(c) shows G's **RUP** embedding along with its compound digraph. By Thm. 3.1, we know that the dual G^* of G is a dipole. In particular, the compound digraph $\overline{\mathbb{G}}^*$ of the dual is a dipath $p = s, \gamma_1^*, \tau_1^*, \gamma_2^*, \tau_2^*, \gamma_3^*, t$, where s is the single source and t the single sink, and γ_1^*, γ_2^*, γ_3^* are the transits and τ_1^*, τ_2^* are the compounds of G^*. Remember that compounds and

transits swap their roles when going from primal to dual. In the proof of Lem. 3.5, we use dipath p to obtain a total order on the transits and compounds of both G and G^*. By this total order, we subsequently construct the **RUP** embedding of G from G^*. In particular, the path structure in G^* is carried over to G. Hence, a necessary condition for a closed digraph to be **RUP** is for its compound digraph to be a path, i. e., the underlying undirected graph of the G's compound digraph is a path.

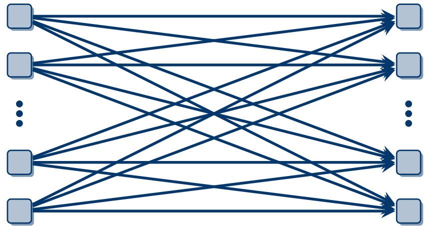
Furthermore, the transits must be independent in the following sense: remember that a transit, pointing from compound γ to another compound γ', corresponds to the subgraph consisting of all dipaths from γ to γ' in the component digraph, where each dipath traverses only trivial components.

Definition 3.2. *Two transits are* independent *if they share no trivial components.*

As an example of a digraph where the transits are not independent, consider Fig. 3.19(a): The depicted digraph consists of r compounds to the left, r compounds to the right, and a trivial component v in the middle. There is a dipath from each left compound to each right compound, which all pass vertex v, and, hence, each transit contains v. Consequently, the transits are not independent. In contrast, the transits of a closed **RUP** digraph are always independent. Consider transit τ_1 in Fig. 3.18(c) which points from γ_1 to γ_2 and, hence, all vertices of τ_1 lie in the region between the rightmost cycle of γ_1 and the leftmost cycle of γ_2. Assume that there is another transit τ' which also starts at γ_1 and shares at least one trivial component with τ_1 but ends at another compound γ'. Then, γ' would also be situated between γ_1 and γ_2. However, since γ' contains at least once cycle that winds around the cylinder this would lead to a crossing. All these observations are summarized in the following corollary.



(a) The digraph consists of r compounds on the left, r compounds on the right side and one trivial component r in the middle.



(b) The compound digraph has r^2 transits (straight lines) which all share the vertex v.

Figure 3.19: A closed digraph in which the transits are not independent and which has a compound digraph of quadratic size.

Corollary 3.3. *Let G be a closed* **RUP** *digraph with compound digraph $\overline{\mathbb{G}}$. The underlying undirected graph of $\overline{\mathbb{G}}$ is a path and all transits are independent.*

Proof. Assume that G is **RUP**-embedded. By Thm. 3.1, the dual digraph G^* is a dipole and its compound digraph is a dipath $p = s, \gamma_1^*, \tau_1^*, \dots, \tau_{k-1}^*, \gamma_k^*, t$. In the proof of Lem. 3.5, we have shown that the compound digraph $\overline{\mathbb{G}}$ of G is the path $\gamma_1, \tau_1, \dots, \tau_{k-1}, \gamma_k$. In particular, between each pair of subsequent compounds γ_i, γ_{i+1} $(1 \leq i < k)$ there is exactly one transit τ_i which is

either directed from γ_i to γ_{i+1} or vice versa. Conversely, each transit in the compound digraph of G, connects a pair of subsequent compounds γ_i, γ_{i+1}. Furthermore, by the construction in the proof of Lem. 3.5, all transits τ_i have no trivial components in common, i. e., the transits are independent. □

The independence of the transits will be of particular importance for the (linear) running time of the algorithm that we develop later.

By traversing the underlying undirected graph of $\overline{\mathbb{G}}$ in either direction, we obtain a total order on the compounds and transits as in the proofs of Lem. 3.5 and Cor. 3.3. In the following, we assume this total order, e. g., $\gamma_1, \tau_1, \gamma_2, \tau_2, \gamma_3$ in Fig. 3.18(c).

A compound may contain vertices which are adjacent to vertices in a neighboring transit. For instance, the vertices v_2, v_3, and v_4 of γ_1 are adjacent to the vertices w_1, w_2, and w_3 in transit τ_1 (see Fig. 3.18(c)). Hence, in the **RUP** embedding, vertices v_2, v_3, and v_4 must lie on the rightmost cycle of γ_1. More generally, all vertices of γ_i adjacent to vertices in τ_i must be rightmost in γ_i. Recall that a vertex of a **RUP**-embedded compound is rightmost if it is incident to the sink of the compound's dual. Accordingly, all vertices of γ_{i+1} which are adjacent to τ_i must be leftmost in γ_{i+1}'s **RUP** embedding. We obtain the following corollary.

Corollary 3.4. *Let G be a* **RUP***-embedded, closed digraph and τ_i be a transit between compounds γ_i and γ_{i+1}. All vertices of γ_i adjacent to vertices of τ_i are rightmost in γ_i and all vertices in γ_{i+1} adjacent to vertices in τ_i are leftmost in γ_{i+1}.*

Proof. This follows again from the proof of Lem. 3.5: In the **RUP** embedding of a closed digraph, there is a transit τ_i between each pair of subsequent compounds γ_i, γ_{i+1}. In the proof of Lem. 3.5, τ_i is "attached" to the rightmost cycle of γ_i. Hence, due to planarity, all vertices of γ_i adjacent to vertices in τ_i are rightmost in γ_i. Analogously, the vertices in γ_{i+1} with neighbors in τ_i are leftmost. □

Let τ_i be a transit between compounds γ_i, γ_{i+1} of a **RUP**-embedded closed digraph G. Recall that we identify transit τ_i with its induced subgraph $G_{\tau_i} = (V_{\tau_i}, E_{\tau_i})$ of the component digraph, which is an acyclic dipole where γ_i is the source and γ_{i+1} the sink or vice versa. For instance, Fig. 3.18(b) shows τ_1 where its source and sink are representing γ_1 and γ_2, respectively. An acyclic dipole is **SUP** if and only if it is planar and, hence, since τ_i is an acyclic dipole and planar, it is also **SUP** [Has01]. In Fig. 3.18(b), the transit is drawn "upward" from left to right. By Lem. 3.6, we know that the **SUP** embedding of a digraph is also a **RUP** embedding. Altogether, we obtain the following corollary:

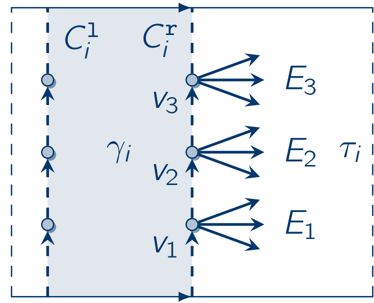
Corollary 3.5. *Any embedding of a transit is* **RUP** *.*

Hence, only planarity has to be tested to find out whether a transit is **RUP**. Additionally, each transit has to be "attachable" to its incident compounds. Remember that Cor. 3.4 ensures that all vertices of a compound γ_i adjacent to vertices in a transit lie on the left- and rightmost cycle of γ_i, respectively. These cycles imply a *partial rotation system*[2] at the source and sink of transit τ_i which the embedding of the transit has to respect. For instance, the rightmost cycle $C_1^{\mathfrak{r}}$ of γ_1 consists of vertices v_1, v_2, v_3, and v_4 in this order. These vertices are connected to w_1, w_2, and w_3 in τ_1 by edges e_1, e_2, e_3, and e_4. The embedding of τ_1 must respect the cyclic order of the edges induced by $C_1^{\mathfrak{r}}$, which is $e_4 \prec \{e_1, e_2\} \prec e_3$, where $e \prec e'$ means

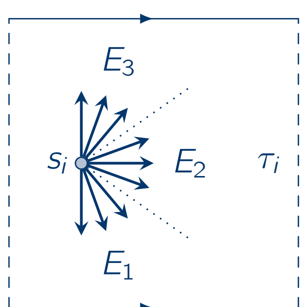
[2] This definition of partial rotation system must not be confused with the definition of partial rotation system as used in the context of the splittable deque in Sect. 2.5.3.

that e precedes e'. The order of e_1 and e_2 can be arbitrary as long as both succeed e_4 and precede e_3. Note that the embedding of τ_1 in Fig. 3.18(b) respects this partial rotation system. Likewise, the leftmost cycle of γ_2 partially induces a rotation system at the sink of τ_1.

More generally, we obtain the situation sketched in Fig. 3.20(a). Let $v_1, \ldots, v_r$ $(r \geq 1)$ be the vertices of compound γ_i that have neighbors in τ_i. By Cor. 3.4, $v_1, \ldots, v_r$ lie on the rightmost cycle C_i^r and we assume w. l. o. g. that $v_1, \ldots, v_r$ is the (cyclic) order on C_i^r. By E_j, we denote the set of outgoing edges of v_j pointing to vertices in τ_i. Let s_i be the source in transit τ_i and assume that we are given a rotation system of s_i. From each E_j, we choose an arbitrary edge e_j with $1 \leq j \leq r$ and thereby obtain the sequence $e_1, \ldots, e_r$. The rotation system of s_i *respects the partial rotation system induced by* γ_i if $e_1 \prec e_2 \prec \ldots \prec e_r$ in s_i's rotation system regardless of which edges we chose from $E_1, \ldots, E_r$. In other words, in s_i's rotation system the edges are ordered according to C_i^r. Fig. 3.20(b) shows the partial rotation system induced by C_i^r in Fig. 3.20(a). All edges in E_i can be ordered arbitrarily, whereas the edges of different sets E_i and E_j with $1 \leq i < j \leq r$ must obey the rotation system induced by C_i^r. In particular, edges between different sets cannot interlace as indicated by the dotted lines in Fig. 3.20(b). Likewise, we can define a partial rotation system of the sink of τ_i as induced by γ_{i+1}. We get the following necessary condition for **RUP** embeddings.



(a) Compound γ_i (shaded) attached to transit τ_i at its rightmost cycle C_i^r. The vertices v_1, v_2 and v_3 are connected to vertices in τ_i via edge sets E_1, E_2 and E_3, respectively.

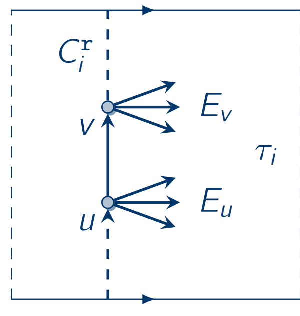


(b) The partial rotation system of transit τ_i induced by cycle C_i^r with $E_1 \prec E_2 \prec E_3$.

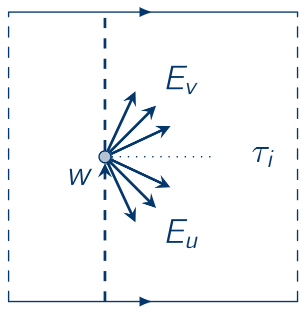
Figure 3.20: Compound γ_i which is attached to transit τ_i via vertices v_1, v_2 and v_3.

Corollary 3.6. *Let G by a closed digraph with a* **RUP** *embedding and τ_i be the transit between compounds γ_i and γ_{i+1}. The embedding of τ_i respects the partial rotation system induced by γ_i and γ_{i+1} at its source and sink.*

Proof. We assume that τ_i points from γ_i to γ_{i+1} where the proof for the opposite direction is similar. By Cor. 3.3, the vertices of γ_i adjacent to vertices in τ_i lie on the rightmost cycle C_i^r of γ_i. Let G' be the subgraph of G that consists solely of cycle C_i^r and transit τ_i. By subsequently contracting the edges of C_i^r, we ultimately obtain the source of τ_i and its rotation system. The single step of such an edge contraction is carried out as follows: Let u and v be two vertices on C_i^r such that u is the immediate predecessor of v. In particular, there is an edge (u, v) that



(a) Edge (u, v) is part of the rightmost cycle of compound γ_i.



(b) After (u, v) is contracted, u and v are merged to vertex w.

Figure 3.21: Edge contraction of an edge (u, v) at the rightmost cycle of a compound.

lies on $C_i^{\mathtt{r}}$. First, we assume that $u \neq v$, i. e., (u, v) is no loop and $C_i^{\mathtt{r}}$ consists of at least two vertices. By the construction in the proof of Lem. 3.5, we obtain a situation as depicted in Fig. 3.21(a). Denote by E_u and E_v the sets of outgoing edges of u and v to vertices in τ_i, respectively. The situation after contracting edge (u, v) is illustrated in Fig. 3.21(b), where vertices u and v are merged to vertex w. In the rotation system of w, the rotation systems of u and v are concatenated. In particular, the edges E_u and E_v do not interlace as indicated by the dotted line. In this sense, the rotation system of w respects the rotation system induced by γ_i. We subsequently contract all edges on $C_i^{\mathtt{r}}$ until only a single vertex with a loop remains. By removing this loop, we obtain a vertex that has the same rotation system as the source s_i in τ_i. As all contraction steps respect the partial rotation system so does the rotation system of s_i. The same edge contractions can be applied to the edges of the leftmost cycle $C_{i+1}^{\mathtt{l}}$ in γ_{i+1} to obtain sink t_i and its rotation system. □

A partial rotation system is unique in the following sense: Assume that a compound γ contains vertices $V^{\mathtt{r}}$ that have neighbors in a transit τ. Further, τ lies right of γ in a **RUP** embedding, i. e., the vertices $V^{\mathtt{r}}$ must be rightmost in γ. Then, in any **RUP** embedding of γ where the vertices in $V^{\mathtt{r}}$ are rightmost, the cyclic order of the vertices in $V^{\mathtt{r}}$ as obtained from the rightmost cycle is always the same. The reason is that all simple cycles in γ that contain the vertices in $V^{\mathtt{r}}$ must wind around the cylinder in the same direction.

The conditions of Corollaries 3.3, 3.4 and 3.6 together yield a characterization.

Lemma 3.8. *Let G be a closed digraph and $\overline{\mathbb{G}}$ be its compound digraph. G is* **RUP** *if and only if the underlying undirected graph of $\overline{\mathbb{G}}$ is a path $\gamma_1, \tau_1, \ldots, \tau_{k-1}, \gamma_k$, all transits are independent, and each compound and transit has a* **RUP** *embedding with the following properties:*

(i) For each $1 \leq i < k$, all vertices of γ_i adjacent to vertices of τ_i are rightmost in γ_i and all vertices in γ_{i+1} adjacent to vertices in τ_i are leftmost in γ_{i+1}.

(ii) The embedding of each τ_i respects the partial rotation system induced by γ_i and γ_{i+1} at its source and sink.

Proof. $\Rightarrow$: Follows from Corollaries 3.3, 3.4 and 3.6.

$\Leftarrow$: From the **RUP** embeddings of the compounds and transits, we inductively construct a **RUP** embedding of G. As the compound digraph $\overline{\mathbb{G}}$ is a path, we have a total order $\gamma_1, \tau_1, \dots, \tau_{k-1}, \gamma_k$ of the compounds and transits. We subsequently construct subgraphs G_i of G with $1 \leq i \leq k$, where G_i consists of compounds $\gamma_1, \dots, \gamma_i$ interconnected by transits $\tau_1, \dots, \tau_{i-1}$. In the following, we will deal with graphs that either have sources and sinks in their primals or duals. To avoid confusion, sources and sinks in the dual are denoted with the superscript * and in the primal there is no superscript.

First observe that each compound γ_i is endowed with a **RUP** embedding and, hence, its dual γ_i^* is an acyclic dipole with source s_i^* and sink t_i^*. In the inductive proof, we maintain the following invariant: The dual G_i^* of digraph G_i is a dipole and, hence, G_i is **RUP**-embedded. Further, the source of G_i^* is the source s_1^* of γ_1^* and the sink of G_i^* is the sink t_i of γ_i^*. The base case follows directly as the embedding of $G_1 = \gamma_1$ is **RUP** and, hence, its dual γ_1^* is an (acyclic) dipole with source s_1^* and sink t_1^*.

For the induction step, let graph G_i fulfill the induction invariant. We construct G_{i+1} and its embedding by "appending" transit τ_i and compound γ_{i+1} to G_i. We assume that τ_i points from γ_i to γ_{i+1}. The reasoning is similar if τ_i is oriented the other direction. Since the transits are independent, they share no trivial components and can be processed independently. In particular, when attaching transit τ_i to γ_i, and γ_{i+1} to τ_i, no other transit τ_j with $j \neq i$ needs to be considered.

The situation before the merge in sketched in Fig. 3.22, where G_i and γ_{i+1} are displayed by shaded regions. The dual G_i^* of G_i is a dipole with source s_1^* and sink t_i^*, where source s_1^* is enclosed by the leftmost cycle $C_1^{\mathtt{l}}$ and t_i^* by the rightmost cycle $C_i^{\mathtt{r}}$. Transit τ_i is an acyclic dipole with source s_i and sink t_i. Compound γ_{i+1} is **RUP**-embedded and its dual γ_{i+1}^* is an acyclic dipole with source s_{i+1}^* and sink t_{i+1}^*. Source s_{i+1}^* is the face enclosed by γ_{i+1}'s leftmost cycle $C_{i+1}^{\mathtt{l}}$ and t_{i+1}^* is enclosed by the rightmost cycle $C_{i+1}^{\mathtt{r}}$.

By the induction invariant, the sink of G_i^* is the sink of γ_i^* and, by property (i), all vertices in γ_i with neighbors in τ_i are rightmost in γ_i. Consequently, these vertices are also rightmost in G_i, e. g., $u_{\mathtt{r}}$, $v_{\mathtt{r}}$, and $w_{\mathtt{r}}$ in Fig. 3.22. We append τ_i to G_i as follows: Let $v_{\mathtt{r}}$ be a vertex on $C_i^{\mathtt{r}}$ with outgoing edges $E_{v_{\mathtt{r}}}$ that have their other endpoints in τ_i. As the rotation system of s_i respects the rotation system induced by γ_i (by (ii)), all edges in $E_{v_{\mathtt{r}}}$ are consecutive in the rotation system of s_i. We adopt the rotation system of s_i for $v_{\mathtt{r}}$ and obtain the situation as shown in Fig. 3.23. Any other vertex $w_{\mathtt{r}}$ on $C_i^{\mathtt{r}}$ also adopts its portion of s_i's rotation system. As the rotation system of s_i respects the cyclic order defined by $C_i^{\mathtt{r}}$, the resulting rotation system is still planar. By (i), all vertices in γ_{i+1} with neighbors in τ_i are leftmost, e. g., u_1, v_1, and w_1. Furthermore, the sink t_i of τ_i respects the partial rotation system induced by γ_{i+1}. By the same reasoning as before, we can thus attach γ_{i+1} to τ_i and obtain a planar rotation system (see again Fig. 3.23).

In fact, the rotation system is not only planar, but also the faces stay essentially the same. Consider for instance, face f between edges $E_{u_{\mathtt{r}}}$ and $E_{v_{\mathtt{r}}}$ in the rotation system of s_i in Fig. 3.22. As shown in Fig. 3.23, face f reappears, though this time also a part of $C_i^{\mathtt{r}}$ is at f's boundary. This observation will be useful in the remainder of the proof.

Denote by G_{i+1} the graph we obtain by the merge process as described above. Since G_{i+1} is endowed with a planar rotation system, we obtain the dual digraph $G_{i+1}^* = (F_{i+1}, E_{i+1}^*)$. We show that G_{i+1}^* is a dipole with source s_1^* and sink t_{i+1}^*. Cycle $C_i^{\mathtt{r}}$ defines a dicut in G_{i+1}^* which partitions the set of faces F_{i+1} into two subsets, i. e., the ones before the dicut and the one beyond. Denote by $F_{G_i} \subsetneq F_{i+1}$ the partition that contains s_1^*. As the subscript suggests,

F_{G_i} contains all faces of G_i^* where only t_i^* is missing. Likewise C_{i+1}^{l} defines a dicut and we denote by $F_{\gamma_{i+1}}$ the partition that contains t_{i+1}^*. As before, $F_{\gamma_{i+1}}$ corresponds to the faces of γ_{i+1}^* where s_{i+1}^* is missing. The remaining faces $F_{\tau_i} = F_{i+1} \setminus (F_{G_i} \cup F_{\gamma_{i+1}})$ are the faces of τ_i^*. At the bottom of Fig. 3.23, the braces indicate where the faces of the three sets are located.

We use Lem. 3.1 to prove that G_{i+1}^* is indeed a dipole. We first show that there is a dipath from s_1^* to each face and from each face to t_{i+1}^*. By the induction hypothesis, we know that G_i^* is a dipole and, thus, there is a dipath from s_1^* to each face in F_{G_i}. Let f be a face in F_{τ_i} which has an incoming edge from the dicut as defined by cycle C_i^{r}, e. g., face f in Fig. 3.23. In G_i^*, there is a dipath from s_1^* to t_i^* (see Fig. 3.22) and, therefore, in G_{i+1}^*, there is a dipath from s_1^* to f. As the dual τ_i^* of transit τ_i is strongly connected, there is a dipath from f to every face $g \in F_{\tau_i}$. Consequently, there is a dipath from s_i^* to g.

Let g' be any face in $F_{\gamma_{i+1}}$. As γ_{i+1}^* is an acyclic dipole with source s_{i+1}^*, there is a dipath $p' = s_{i+1}^* \rightsquigarrow g'$. Denote by e the first edge on this dipath which is part of the dicut defined by C_{i+1}^{l}, i. e., the primal of e is part of C_{i+1}^{l}. As shown in Fig. 3.23, let f' be the face in F_{τ_i} to which e is incident. Since $f' \in F_{\tau_i}$, there is a dipath $s_1^* \rightsquigarrow f'$ by the reasoning from before. Concatenating these dipaths results in the desired dipath $s_1^* \rightsquigarrow f' \rightsquigarrow g'$. Altogether, G_{i+1}^* contains a dipath from s_1^* to any face in $F_{i+1} = F_{G_i} \cup F_{\tau_i} \cup F_{\gamma_{i+1}}$. By the same arguments, there is also a dipath from any face in F_{i+1} to t_{i+1}^*.

What is left to show is that each dipath $p = s_1^* \rightsquigarrow t_{i+1}^*$ visits each of G_{i+1}^*'s compounds. Let p be a simple dipath from s_1^* to t_{i+1}^*. By the induction hypothesis, each dipath $p' = s_i^* \rightsquigarrow t_i^*$ in G_i^* contains at least one face of each of G_i^*'s compounds and so p also visits each compound of G_i^*. During the merge, we have added the compound τ_i^* to G_{i+1}^*. By the definition of F_{G_i} and $F_{\gamma_{i+1}}$ and since p is simple, dipath p first traverses a dual edge e of C_i^{r} and then a dual edge e' of C_{i+1}^{l}. Note that e and e' are distinct. Between traversing e and e', dipath p visits at least one face f which is neither in F_{G_i} nor in $F_{\gamma_{i+1}}$. Hence, f is in F_{τ_i} and, therefore, belongs to compound τ_i^*. Dipath p, therefore, visits all compounds of G_{i+1}^*.

We can conclude that G_{i+1}^* is a dipole with source s_1^* and sink t_{i+1}^* and, hence, the induction invariant is maintained. In particular, the dual of $G_k = G$ is a dipole and, by Thm. 3.1, we have constructed a **RUP** embedding for G. □

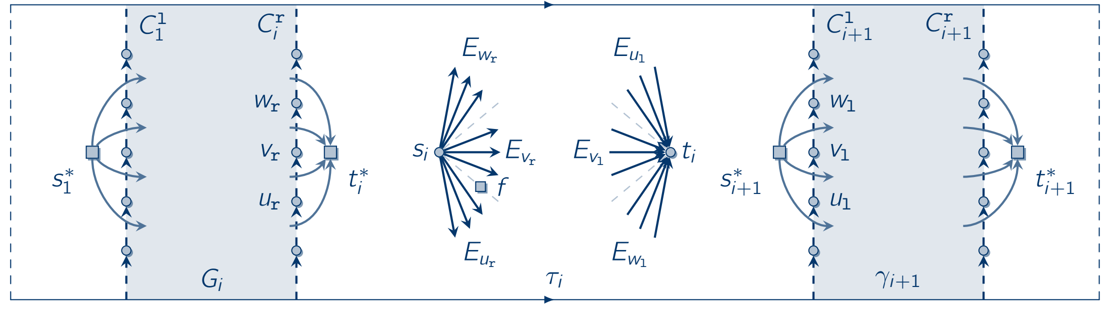


Figure 3.22: Before merging G_i with transit τ_i and compound γ_{i+1}.

Note that the proof of Lem. 3.8 is constructive: If we can find **RUP** embeddings for the compounds and transits fulfilling the properties as listed in Lem. 3.8, we can construct a **RUP** embedding of the whole graph. We transform this construction into an algorithm in Sect. 3.5.1.2. Moreover, this construction can be carried out in linear time.

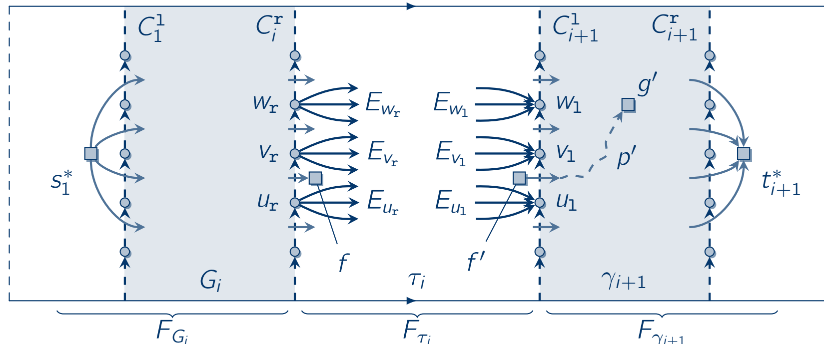


Figure 3.23: The graph G_{i+1} obtained from merging G_i with τ_i and γ_{i+1}. G_{i+1} fulfills the induction invariants.

Corollary 3.7. *Let $G = (V, E)$ be a closed digraph with compound digraph $\overline{\mathbb{G}}$ whose underlying undirected graph is a path $\gamma_1, \tau_1, \dots, \tau_{k-1}, \gamma_k$ and whose transits are independent. Given* **RUP** *embeddings of the compounds and transits that fulfill properties (i) and (ii) of Lem. 3.8, a* **RUP** *embedding of G can be constructed in time $\mathcal{O}(|V| + |E|)$.*

Proof. Let γ_i be a compound to which transit τ_i has to be attached as described in the proof of Lem. 3.8. Let v be a rightmost vertex in γ_i which has neighbors in τ_i connected via edges E_v. In transit τ_i, source s_i is endowed with a rotation system, where edges E_v are consecutive. By removing all edges E_v from s_i and attaching them in the same order to v, we obtain the rotation system of v. This can be done for each rightmost vertex in γ_i which has neighbors in τ_i. All other vertices and edges of γ_i and τ_i stay untouched. Attaching γ_{i+1} to τ_i proceeds analogously. Also observe that the transits are independent and, thus, each trivial component, i. e., all vertices of a transit other than its source and sink, is processed only once. Altogether, each vertex and edge of G is processed at most $\mathcal{O}(1)$ times, which leads to an overall running time of $\mathcal{O}(|V| + |E|)$. □

3.5.1.2 The Algorithm for Closed Digraphs

From Lem. 3.8, we can derive an outline for a **RUP** testing algorithm for closed digraphs: First, we have to check whether the underlying graph of the compound digraph is a path. Then, we have to find **RUP** embeddings of the compounds and transits that fulfill the properties of Lem. 3.8. Alg. 3.2 shows the routine `TestClosedDigraph` which follows this outline. `TestClosedDigraph` takes a closed digraph $G = (V, E)$ as input and returns a **RUP** embedding of G or $\perp$ if G is not **RUP**. `TestClosedDigraph` calls the subroutines `PlanarConstraint`, `TestCompound`, and `ComputeCompoundDigraph`.

The subroutine `PlanarConstraint`$(G, \mathcal{E}')$ takes as input a graph $G = (V, E)$ and a partial rotation system $\mathcal{E}'$ as defined in Sect. 3.5.1.1. `PlanarConstraint` either returns an embedding of the graph which respects $\mathcal{E}'$ or $\perp$ if no such embedding exists. It can be implemented with a running time in $\mathcal{O}(|V|)$ according to [GKM07]. The subroutine `TestCompound`$(\gamma, V^{\mathtt{l}}, V^{\mathtt{r}})$ takes as input a compound $\gamma = (V_\gamma, E_\gamma)$ and two sets $V^{\mathtt{l}}, V^{\mathtt{r}} \subseteq V_\gamma$ and returns a **RUP** embedding of γ where all vertices in $V^{\mathtt{l}}$ are left- and all vertices in $V^{\mathtt{r}}$ are rightmost. If no such embedding exists, `TestCompound` returns $\perp$. For the moment, we take `TestCompound` as given

and postpone its description to Sects. 3.5.2 and 3.5.3. We also assume that `TestCompound` has a running time of $\mathcal{O}(|V_\gamma|)$ which indeed is the case as seen later.

The third subroutine is `ComputeCompoundDigraph` which computes the compound digraph and the subgraph corresponding to each transit. Some care has to be taken here since in general the compound digraph may have a size that is quadratic in the size of the original digraph and this can lead to a quadratic running time. Consider again the digraph displayed in Fig. 3.19(a) on page 125 which consists of r compounds to the left, r compounds to the right, and one trivial component v, i. e., $|V| = 2r + 1$ and $|E| = 2r$. There is a dipath and, hence, also a transit from each left compound to each right compound (cf. Fig. 3.19(b)). The number of transits is therefore r^2 which is quadratic in the size of the digraph. Note that the digraph in Fig. 3.19(a) is even planar and, thus, neither planarity nor $\mathcal{O}(|V|)$ many edges guarantee a compound digraph of linear size.

Fortunately, with **RUP** graphs we can circumvent this problem. The fact that in Fig. 3.19(a) all transits share the trivial component v is to blame for the quadratic size of the compound digraph. Remember that by Cor. 3.3 the transits of a **RUP** graph are independent, i. e., they share no trivial components. In this case, the number of transits is bounded by the number of edges in the original digraph.

Lemma 3.9. *Let $G = (V, E)$ be a digraph with compound digraph $\overline{\mathbb{G}} = (\mathbb{V}_C \cup \mathbb{T}, \overline{\mathbb{E}})$. If all transits are independent, then the number of transits is bounded by $|E|$.*

Proof. Each edge $e \in E$ either belongs to a compound or to a set of transits. We denote by $E' \subseteq E$ the set of edges belonging to transits. Let e be an edge in E' and let $\tau \in \overline{\mathbb{E}}$ be a transit which contains e. Transit τ points from $\sigma_u \in \mathbb{V}_C \cup \mathbb{T}$ to $\sigma_v \in \mathbb{V}_C \cup \mathbb{T}$, where each of σ_u and σ_v is either a compound or a terminal. In the component digraph of G, e lies on a dipath from σ_u to σ_v and this dipath contains only trivial components except for σ_u and σ_v. We show that no other transit can contain e. For contradiction, assume that a transit $\hat{\tau} \neq \tau$ contains e, where $\hat{\tau}$ points from $\hat{\sigma_u} \in \mathbb{V}_C \cup \mathbb{T}$ to $\hat{\sigma_v} \in \mathbb{V}_C \cup \mathbb{T}$. Since $\tau \neq \hat{\tau}$, e cannot be the direct edge from σ_u to σ_v. Consequently, at least one of e's endpoints is a trivial component σ_w. However, transits τ and $\hat{\tau}$ then share σ_w, which is a contradiction to the independence of the transits. Therefore, for each edge there is at most one transit that contains it and, hence, the number of transits is bounded by $|E'|$ and, thus, also by $|E|$. ☐

`ComputeCompoundDigraph` uses Lem. 3.9 to achieve a running time of $\mathcal{O}(|V| + |E|)$ as follows. While computing the compound digraph, it tests whether the transits are independent. In this case, the compound digraph can be computed in time $\mathcal{O}(|V| + |E|)$. If the transits are not independent, `ComputeCompoundDigraph` aborts the computation in time, and returns $\perp$.

Before we analyze `ComputeCompoundDigraph` in detail, we need an additional definition: For each trivial component σ_v, let $\mathrm{Pred}(\sigma_v) \subseteq (\mathbb{V}_C \cup \mathbb{T})$ be the set of compounds/terminals $\sigma_u \in \mathbb{V}_C \cup \mathbb{T}$, for which there is a dipath $\sigma_u \rightsquigarrow \sigma_v$ which visits only trivial components except for its starting point. Likewise, $\mathrm{Succ}(\sigma_v) \subseteq (\mathbb{V}_C \cup \mathbb{T})$ contains all compounds and terminals for which there is a dipath from σ_v internally visiting only trivial components. We call $\mathrm{Pred}(\sigma_v)$ and $\mathrm{Succ}(\sigma_v)$ the set of *predecessors and successors of σ_v (in the component digraph)*, respectively. By the proof of Lem. 3.9, we know that each edge e, which does not belong to a compound, belongs to exactly one transit τ if the transits are independent. By using the

definition of predecessors and successors, we characterize the independence of the transits:

Algorithm 3.1. `ComputeCompoundDigraph`

Input: digraph $G = (V, E)$
Output: compound graph $\overline{\mathbb{G}} = (\mathbb{V}_C \cup \mathbb{T}, \overline{\mathbb{E}})$ with independent transits and digraphs (V_τ, E_τ) as induced by each transit; or $\perp$ if transits are not independent

1 $\mathbb{G} = (\mathbb{V}, \mathbb{E}) \leftarrow$ `ComputeComponentDigraph`(G)
2 Initialize $\overline{\mathbb{G}} = (\mathbb{V}_C \cup \mathbb{T}, \overline{\mathbb{E}})$ with $\mathbb{V}_C, \mathbb{T}, \overline{\mathbb{E}} = \emptyset$
3 **foreach** $\sigma \in \mathbb{V}$ **do**
4   **switch** type of σ **do**
5   **case** compound: $\mathbb{V}_C \leftarrow \mathbb{V}_C \cup \{\sigma\}$
6   **case** terminal: $\mathbb{T} \leftarrow \mathbb{T} \cup \{\sigma\}$
7   **case** trivial component: $\mathrm{Pred}(\sigma) \leftarrow \perp$; $\mathrm{Succ}(\sigma) \leftarrow \perp$
8 $\sigma_1, \dots, \sigma_r \leftarrow$ `TopSort`$(\mathbb{G})$
9 **foreach** $\sigma = \sigma_1, \dots, \sigma_r$ **do**
10   **if** σ is compound or terminal **then** $p \leftarrow \sigma$
11   **else** $p \leftarrow \mathrm{Pred}(\sigma)$
12   **foreach** σ' with $(\sigma, \sigma') \in \mathbb{E}$ and σ' is a trivial component **do**
13   **if** $\mathrm{Pred}(\sigma') = \perp$ **then** $\mathrm{Pred}(\sigma') \leftarrow p$
14   **else if** $\mathrm{Pred}(\sigma') \neq p$ **then return** $\perp$ by Cor. 3.8
15 **foreach** $\sigma =$ `Reverse`$(\sigma_1, \dots, \sigma_r)$ **do**
16   **if** σ is compound or terminal **then** $s \leftarrow \sigma$
17   **else** $s \leftarrow \mathrm{Succ}(\sigma)$
18   **foreach** σ' with $(\sigma', \sigma) \in \mathbb{E}$ and σ' is a trivial component **do**
19   **if** $\mathrm{Succ}(\sigma') = \perp$ **then** $\mathrm{Succ}(\sigma') \leftarrow s$
20   **else if** $\mathrm{Succ}(\sigma') \neq s$ **then return** $\perp$ by Cor. 3.8
21 **foreach** $\sigma \in \mathbb{V}$ and σ is trivial **do**
22   $\tau \leftarrow (\mathrm{Pred}(\sigma), \mathrm{Succ}(\sigma))$
23   **if** $\tau \notin \overline{\mathbb{E}}$ **then**
24     $\overline{\mathbb{E}} \leftarrow \overline{\mathbb{E}} \cup \{\tau\}$
25     $V_\tau \leftarrow \{\mathrm{Pred}(\sigma), \mathrm{Succ}(\sigma)\}$
26     $E_\tau \leftarrow \emptyset$
27   $V_\tau \leftarrow V_\tau \cup \{\sigma\}$
28 **foreach** $(\sigma_u, \sigma_v) \in \mathbb{E}$ **do**
29   **if** σ_u is a trivial component **then** $\tau \leftarrow (\mathrm{Pred}(\sigma_u), \mathrm{Succ}(\sigma_u))$
30   **else if** σ_v is a trivial component **then** $\tau \leftarrow (\mathrm{Pred}(\sigma_v), \mathrm{Succ}(\sigma_v))$
31   **else** $\tau \leftarrow (\sigma_u, \sigma_v)$ where each of σ_u and σ_v is a compound or terminal
32   $E_\tau \leftarrow E_\tau \cup \{(\sigma_u, \sigma_v)\}$
33 **return** $\overline{\mathbb{G}}$, $(V_\tau, E_\tau)_{\tau \in \overline{\mathbb{E}}}$

Corollary 3.8. *The transits of a digraph are independent if and only if both* $\mathrm{Pred}(\sigma_v)$ *and* $\mathrm{Succ}(\sigma_v)$ *are singletons for each trivial component* σ_v*, i. e.,* $\mathrm{Pred}(\sigma_v) = \{\sigma_u\}$ *and* $\mathrm{Succ}(\sigma_v) = \{\sigma_w\}$*, where there is a transit* τ *from* σ_u *to* σ_v *that contains* σ_v*.*

Proof. ⇒: Assume that transit τ points from σ_u to σ_w in the compound digraph. In the proof of Lem. 3.9, we argue that if one of e's endpoints is a trivial component σ_v, then σ_v also belongs to transit τ and only to this transit. Hence, $\mathrm{Pred}(\sigma_v) = \{\sigma_u\}$ and $\mathrm{Succ}(\sigma_v) = \{\sigma_w\}$. In particular, σ_v has exactly one predecessor and one successor.

⇐: If a trivial component σ_v has exactly one predecessor and successor, then only this transit contains σ_v. Therefore, no two distinct transits can both contain σ_v. As this argument holds for all trivial components, all transits must be independent. □

As soon as a trivial component has more than one predecessor or successor, the transits are not independent and `ComputeCompoundDigraph` must return $\perp$. Indeed, this check is at the heart of `ComputeCompoundDigraph`, which we analyze now.

Lemma 3.10. *For each digraph $G = (V, E)$,* `ComputeCompoundDigraph` *in Alg. 3.1 tests whether the transits are independent. If this is the case, the compound digraph of G and all subgraphs induced by the transits are returned. Otherwise, $\perp$ is returned. The running time is in $\mathcal{O}(|V| + |E|)$.*

Proof. The first step in line 1 is to compute the component digraph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ which can be done in time $\mathcal{O}(|V| + |E|)$ by the algorithm of Tarjan [Tar72a]. The size of the component digraph is bounded by the size of G, i. e., $|\mathbb{V}| \leq |V|$ and $|\mathbb{E}| \leq |E|$.

In lines 2 to 7, the compound digraph $\overline{\mathbb{G}} = (\mathbb{V}_C \cup \mathbb{T}, \overline{\mathbb{E}})$ is initialized and for each component $\sigma \in \mathbb{V}$ it is checked whether it is a compound, a terminal, or a trivial component. Recall that a component is a compound if it contains at least one edge, which can be tested in time $\mathcal{O}(1)$ (line 5). For each trivial component σ, two variables $\mathrm{Pred}(\sigma)$ and $\mathrm{Succ}(\sigma)$ are maintained, which are both initially set to $\perp$. Later in the algorithm, $\mathrm{Pred}(\sigma)$ and $\mathrm{Succ}(\sigma)$ are set to the single predecessor and single successor of σ. The loop starting in line 2 has $\mathcal{O}(\mathbb{V}) \subseteq \mathcal{O}(|V|)$ iterations where each iteration takes constant time.

The component digraph $\mathbb{G}$ is acyclic and, thus, has a topological ordering $\sigma_1, \dots, \sigma_r$ of the components. To obtain the topological ordering, the routine `TopSort` is called in line 8 which runs in time $\mathcal{O}(|\mathbb{V}| + |\mathbb{E}|) \subseteq \mathcal{O}(|V| + |E|)$ [Knu97]. The loop starting in line 9 processes the components in the order of the topological ordering. This order ensures that at σ_j's turn, all components σ_i for which there is a dipath $\sigma_i \rightsquigarrow \sigma_j$ in $\mathbb{G}$ have been processed before, i. e., $i < j$.

The idea of the loop is that the information about the predecessor of each trivial component is propagated forward on each dipath: In line 10, the variable p is set to the predecessor which is then propagated forward. If component σ is a compound or terminal, then σ is the starting point of a transit and p is set to σ. Otherwise, σ is a trivial component and the predecessor $\mathrm{Pred}(\sigma)$ of σ has to be propagated. For each trivial component σ' with an edge pointing from σ to σ', its predecessor $\mathrm{Pred}(\sigma')$ is set to p. If $\mathrm{Pred}(\sigma') \neq \perp$ and $\mathrm{Pred}(\sigma') \neq p$, then σ' has two distinct predecessors and the transits are not independent by Cor. 3.8 (see lines 13 and 14). In this case, $\perp$ is returned. Note that after the loop starting in line 9 has finished, $\mathrm{Pred}(\sigma) \neq \perp$ for each trivial component σ. Also, each component and edge is processed at most once in the loop and all other operations have a constant running time and, hence, the loop runs in $\mathcal{O}(|\mathbb{V}| + |\mathbb{E}|) \subseteq \mathcal{O}(|V| + |E|)$. The successor of each trivial component is determined likewise in the loop starting in line 15. Only this time, the components are processed in the reversed topological ordering and the successors are propagated backwards on the dipaths.

In the loop starting in line 21, the edges of the compound digraph, i. e., the transits, are determined and the digraphs belonging to the transits are initialized. Each trivial component σ belongs to the transit τ which points from $\mathrm{Pred}(\sigma)$ to $\mathrm{Succ}(\sigma)$ (line 22). If τ is not in $\overline{\mathbb{E}}$, it

is inserted. Furthermore, the set of vertices V_τ is initialized to $\{\mathrm{Pred}(\sigma), \mathrm{Succ}(\sigma)\}$ which are the source and sink of the digraph corresponding to transit τ and the trivial component σ is inserted to V_τ (line 27). The loop in line 21 is carried out $\mathcal{O}(|\mathbb{V}|) \subseteq \mathcal{O}(|V|)$ times and each iteration takes $\mathcal{O}(1)$ time.

In the loop starting in line 28, the edges E_τ belonging to each transit τ are determined. For each edge $e \in \mathbb{E}$ of the component digraph, the transit τ to which e belongs is determined similar to the proof of Lem. 3.9: If one of e's endpoint is a trivial component σ, then the corresponding transit τ is $(\mathrm{Pred}(\sigma), \mathrm{Succ}(\sigma))$ (lines 29 and 30). Otherwise, e points from a terminal or compound to another terminal or compound and, hence, e itself is the transit τ to which e belongs (line 31). In any case, edge e is inserted into E_τ. The loop has $\mathcal{O}(|\mathbb{E}|) \subseteq \mathcal{O}(|E|)$ iterations of which each has a constant running time.

Finally in line 33, the compound digraph of G and all digraphs (V_τ, E_τ) are returned. Since each trivial component has exactly one predecessor and one successor, all transits are independent by Cor. 3.8. By the analysis from before, each of the loops in `ComputeCompoundDigraph` runs in $\mathcal{O}(|V| + |E|)$ and so does the whole routine. □

Note that `ComputeCompoundDigraph` works for general digraphs with independent transits. For closed digraph, it returns an empty set of terminals. We now analyze `TestClosedDigraph`.

Lemma 3.11. `TestClosedDigraph` *(Alg. 3.2) returns a* **RUP** *embedding of a closed digraph $G = (V, E)$ or $\perp$ if G is not* **RUP**. *Assuming that* `TestCompound` *has linear running time,* `TestClosedDigraph` *runs in time $\mathcal{O}(|V|)$.*

Proof. `TestClosedDigraph` starts with testing whether the input digraph is planar in line 1, which can be done in time $\mathcal{O}(|V|)$ [KW01]. If the digraph is not planar, it is certainly not **RUP** and $\perp$ is returned. The subroutine `ComputeCompoundDigraph` is called in line 2 which takes $\mathcal{O}(|V| + |E|)$ time by Lem. 3.10. If `ComputeCompoundDigraph` returns $\perp$, then transits are not independent and the whole digraph cannot be **RUP** by Cor. 3.3. Otherwise, `ComputeCompoundDigraph` returns the compound digraph $\overline{\mathbb{G}} = (\mathbb{V}_C \cup \mathbb{T}, \overline{\mathbb{E}})$ and for each transit τ its induced digraph (V_τ, E_τ). Note that since G is closed, it contains no terminals and, hence, $\mathbb{T} = \emptyset$. Furthermore, $|\mathbb{V}_C| \in \mathcal{O}(|V|)$ and $\overline{\mathbb{E}} \in \mathcal{O}(|E|)$ by Lem. 3.9. In line 6, `TestClosedDigraph` tests whether the underlying undirected graph of $\overline{\mathbb{G}}$ is a path in time $\mathcal{O}(|\mathbb{V}_C| + |\overline{\mathbb{E}}|) \subseteq \mathcal{O}(|V| + |E|)$ and returns $\perp$ if this is not the case (Cor. 3.3). Otherwise, traversing the path in any direction yields the total order $\gamma_1, \tau_1, \ldots, \tau_{k-1}, \gamma_k$ (line 7).

In the loop starting in line 8, property (i) of Lem. 3.8 is tested for each compound γ_i. In lines 9 and 10, the sets $V^{\mathtt{l}}$ and $V^{\mathtt{r}}$ are obtained. Set $V^{\mathtt{l}}$ contains all vertices in γ_i which have neighbors in τ_{i-1} if $i > 1$ and $V^{\mathtt{l}}$ is empty if $i = 0$. Likewise, $V^{\mathtt{r}}$ contains all vertices in γ_i with neighbors in τ_i if $i < k$ and $V^{\mathtt{r}}$ is empty if $i = k$. More specifically, for each edge (u, v) in both transits τ_i and τ_{i-1}, it is tested if either u or v is in γ_i to obtain $V^{\mathtt{l}}$ and $V^{\mathtt{r}}$. Hence, the edges of all transits are processed at most twice during all iterations. Thus, lines 9 and 10 contribute $\mathcal{O}(|E|)$ to the overall running time of the loop. By Cor. 3.4, compound $\gamma_i = (V_{\gamma_i}, E_{\gamma_i})$ must have a **RUP** embedding such that all vertices in $V^{\mathtt{l}}$ are left- and all vertices in $V^{\mathtt{r}}$ are rightmost which is tested in line 11 by calling `TestCompound`. If `TestCompound` returns $\perp$, G cannot be **RUP**. Otherwise, the **RUP** embedding of γ_i is stored in $\mathcal{E}_{\gamma_i}$. By assumption, the running time of `TestCompound` is $\mathcal{O}(|V_{\gamma_i}| + |E_{\gamma_i}|)$. Since the compounds are all disjoint, the overall running time of the whole loop is $\mathcal{O}(|V| + |E|)$.

The loop starting in line 13 tests for each transit τ_i if it has a **RUP** embedding fulfilling property (ii) of Lem. 3.8: Let τ_i be the transit between compound γ_i and γ_{i+1} with induced

digraph (V_{τ_i}, E_{τ_i}). From the **RUP** embedding $\mathcal{E}_{\gamma_i}$, we compute the dual digraph γ_i^* of γ_i in time $\mathcal{O}(|V_{\gamma_i}| + |E_{\gamma_i}|)$, which is an acyclic dipole. From the sink of γ_i^*, we directly obtain the rightmost cycle $C_i^{\mathtt{r}}$ of γ_i in line 14. Likewise, we obtain the leftmost cycle $C_{i+1}^{\mathtt{l}}$ from the dual of γ_{i+1}. These two cycles define a partial rotation system $\mathcal{E}'_{\tau_i}$ for τ_i as described in Sect. 3.5.1.1, where $\mathcal{E}'_{\tau_i}$ is obtained by traversing $C_i^{\mathtt{r}}$ and $C_{i+1}^{\mathtt{l}}$. Remember, this partial rotation system is unique in the sense that the cyclic order of the vertices on $C_i^{\mathtt{r}}$ and $C_{i+1}^{\mathtt{l}}$ that have neighbors in τ_i is always the same regardless of which **RUP** embeddings are returned for γ_i and γ_{i+1} in line 11. `PlanarConstraint` tests whether an embedding of τ_i exists which respects the partial rotation system $\mathcal{E}'_{\tau_i}$. The embedding is stored in $\mathcal{E}_{\tau_i}$. Recall that the embedding returned by `PlanarConstraint` is also **RUP** by Cor. 3.5. If no embedding respecting $\mathcal{E}'_{\tau_i}$ exists, `PlanarConstraint` and also `TestClosedDigraph` return $\perp$ (Cor. 3.6). The overall running time of the loop starting in line 13 is $\mathcal{O}(|V| + |E|)$ since each call of `PlanarConstraint` takes $\mathcal{O}(|V_{\tau_i}| + |E_{\tau_i}|)$ time steps and the transits are all independent.

As of now, we have obtained **RUP** embeddings of the compounds and transits that fulfill the properties of Lem. 3.8. By Cor. 3.7, the **RUP** embedding of G can be constructed in time $\mathcal{O}(|V| + |E|)$ (line 19). Overall, each step of the algorithm takes $\mathcal{O}(|V| + |E|)$. After line 1, we can assume that G is planar and, hence, $\mathcal{O}(|V| + |E|) = \mathcal{O}(|V|)$, which leads to an overall running time of $\mathcal{O}(|V|)$. □

Algorithm 3.2. `TestClosedDigraph`

Input: closed digraph $G = (V, E)$
Output: **RUP** embedding of G or $\perp$ if G is not **RUP**

1. **if** `Planar`$(G) =$ `false` **then return** $\perp$
2. **if** `ComputeCompoundDigraph`$(G) \neq \perp$ **then**
3. $\quad \overline{\mathbb{G}} = (\mathbb{V}_C, \overline{\mathbb{E}}) \leftarrow$ compound digraph of G
4. $\quad (V_\tau, E_\tau)_{\tau \in \overline{\mathbb{E}}} \leftarrow$ digraphs induced by the transits
5. **else return** $\perp$ by Cor. 3.3
6. **if** underlying undirected graph of $\overline{\mathbb{G}}$ is no path **then return** $\perp$ by Cor. 3.3
7. Traversal of path underlying $\overline{\mathbb{G}}$ yields total order $\gamma_1, \tau_1, \gamma_2, \dots, \tau_{k-1}, \gamma_k$
8. **foreach** Compound $\gamma_i \in \mathbb{V}_C$ **do**
9. $\quad V^{\mathtt{l}} \leftarrow$ vertices in γ_i adjacent to transit τ_{i-1} if $i > 1$, else $\emptyset$
10. $\quad V^{\mathtt{r}} \leftarrow$ vertices in γ_i adjacent to transit τ_i if $i < k$, else $\emptyset$
11. $\quad \mathcal{E}_{\gamma_i} \leftarrow$ `TestCompound`$(\gamma_i, V^{\mathtt{l}}, V^{\mathtt{r}})$
12. $\quad$ **if** $\mathcal{E}_{\gamma_i} = \perp$ **then return** $\perp$ by Cor. 3.4
13. **foreach** Transit $\tau_i \in \overline{\mathbb{E}}$ **do**
14. $\quad C_i^{\mathtt{r}} \leftarrow$ rightmost cycle of γ_i according to $\mathcal{E}_{\gamma_i}$
15. $\quad C_{i+1}^{\mathtt{l}} \leftarrow$ leftmost cycle of γ_{i+1} according to $\mathcal{E}_{\gamma_{i+1}}$
16. $\quad \mathcal{E}'_{\tau_i} \leftarrow$ partial rotation system at the source and sink according to $C_i^{\mathtt{r}}$ and $C_{i+1}^{\mathtt{l}}$
17. $\quad \mathcal{E}_{\tau_i} \leftarrow$ `PlanarConstraint`$((V_{\tau_i}, E_{\tau_i}), \mathcal{E}'_{\tau_i})$
18. $\quad$ **if** $\mathcal{E}_{\tau_i} = \perp$ **then return** $\perp$ by Cor. 3.6
19. $\mathcal{E} \leftarrow$ embedding of G obtained by assembling the embeddings of all compounds and transits according to the proof of Lem. 3.5
20. **return** $\mathcal{E}$

3.5.2 Compounds

A compound is made up of blocks, i. e., biconnected components, that are connected by cut vertices. A useful tool to investigate the blocks and their relationship is the block-cut tree (see Sect. 1.1.6). Remember, the vertices of the block-cut tree $\mathcal{T}_B = (\mathbf{B}, \mathbf{C}, \mathbf{E}_B)$ of a graph G are the blocks $B_i = (V_i, E_i) \in \mathbf{B}$ and the cut vertices $\mathbf{C}$, and there is an edge between a block B_i and a cut vertex c if B_i contains c, i. e., $c \in V_i$. We assume that a vertex v with a directed loop is also a block and v is a cut vertex, i. e., removing v disconnected the loop from the rest. We first establish some general properties of block-cut trees of planar graphs and their duals. Although, the results are proved for undirected graphs they apply to digraphs, i. e., their underlying undirected graphs, as well. In all of the following sections, we assume that all graphs and digraphs a planar.

3.5.2.1 Cut Vertices and Cut Faces

A neat property of the block-cut tree of an embedded graph is that a cut vertex carries its feature, namely that it separates the graph, over to faces to which it is incident. We call a cut vertex of the dual graph *cut face*. Let c be a cut vertex of an embedded graph and $B_1, \dots, B_k$ be all blocks that contain c. Further, let e and e' be two distinct edges incident to c such that e and e' belong to two distinct blocks B_i and B_j and e is the immediate predecessor of e' in c's rotation system. Between edges e and e' lies a face f_c, which is defined by its clockwise traversal according to the embedding. We say that f_c *lies between blocks* B_i *and* B_j *at cut vertex* c. Fig. 3.24(a) illustrates a block-cut tree, where the blocks are the shaded regions and the cut vertices are displayed by the symbol $\oplus$. The blocks B_1, B_2, and B_3 are attached to each other by cut vertex c. Face f_c, between e and e', is defined by the (non-simple) circle in the graph which surrounds all blocks, i. e., f_c is the outer face of the drawing. Note that several cut vertices may be incident to f_c, e. g., c and c' in Fig. 3.24(a). Conversely, not all cut vertices must be incident to f_c, e. g., cut vertex c''. It turns out that face f_c is a cut face.

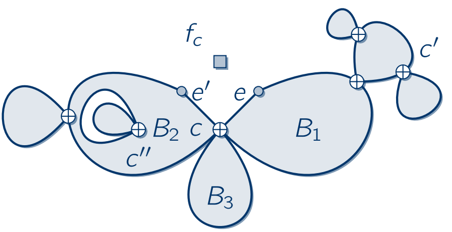
Lemma 3.12. *Let c be a cut vertex of an embedded graph G. A face that lies between blocks B_i and B_j at c is a cut face.*

Proof. Fig. 3.24(b) illustrates the situation we obtain in the proof. Let $B_i = (V_i, E_i)$ and $B_j = (V_j, E_j)$ be two distinct blocks which contain c and let f_c be a face that lies between B_i and B_j at c. First, we construct two simple circles C_i and C_j that enclose B_i and B_j, respectively, in the following way. Face f_c corresponds to a non-simple circle C_{f_c} in the graph which visits c at least twice and may also contain other cut vertices. Consider C_{f_c} as a sequence of edges. Then the subsequence of edges in C_{f_c} that are in E_i, i. e., edges that belong to B_i, form another circle C_i. In particular, this subsequence does not contain any "gaps" for the following reasons: Whenever C_{f_c} contains two subsequent edges e and e' such that $e \in E_i$ and $e' \notin E_i$, a cut vertex c' is visited between e and e'. Note that c' is not necessarily c. Eventually, C_{f_c} must return to c' and continues via another edge $e'' \in E_i$. Since circle C_i consists only of edges in E_i, C_i traverses e'' directly after e without taking the "detour". Note that before C_{f_c} traverses edge e'', it may revisit c' several times. Also observe that C_i is vertex-disjoint since any vertex visited more than once would be a cut vertex of B_i. In contrast, circle C_i may not be edge-disjoint if B_i consists of a single edge only. For the moment, we assume that C_i is edge-disjoint and deal with the non-disjoint case later. Circle C_j is defined analogously for block B_j and is also assumed to be edge-disjoint. The set of edges in C_i and C_j are disjoint

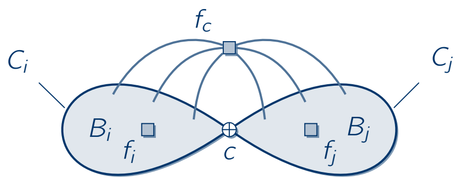
since otherwise B_i and B_j would have at least two vertices in common and, thus, c would be no cut vertex.

Recall the definition of cut and cut-set from Sect. 1.1.8. Both circles C_i and C_j define cut-sets $E^*_{C_i} \subseteq E^*$ and $E^*_{C_j} \subseteq E^*$ of the dual graph $G^* = (F, E^*)$, respectively, where the edges in $E^*_{C_i}$ ($E^*_{C_j}$) are the duals of the edges in C_i (C_j). All edges in $E^*_{C_i}$ and $E^*_{C_j}$ are incident to face f_c by construction. The cut-set $E^*_{C_i}$ defines the cut F_{C_i} and $F \setminus F_{C_i}$, where w. l. o. g. $f_c \notin F_{C_i}$. Intuitively, F_{C_i} contains all faces of B_i enclosed by C_i. Likewise, F_{C_j} and $F \setminus F_{C_j}$ is the cut according to cut-set $E^*_{C_j}$ with $f_c \notin F_{C_j}$. For a face $f_i \in F_{C_i}$, consider a simple path $p = f_i \rightsquigarrow f_c$, which must exist since G^* is connected. As $f_i \in F_{C_i}$ and $f_c \in F \setminus F_{C_i}$, p contains exactly one edge from $E^*_{C_i}$, which is in fact the last edge before reaching f_c. Since p is simple and f_c is incident to all edges $E^*_{C_j}$, p cannot contain an edge from $E^*_{C_j}$. Analogously, all simple paths from any face $f_j \in F_{C_j}$ to f_c contain exactly one edge from $E^*_{C_j}$ and none from $E^*_{C_i}$. Hence, there can be no face which is in both sets F_{C_i} and F_{C_j}, i. e., $F_{C_i} \cap F_{C_j} = \emptyset$. In particular, any simple path from a face $f_i \in F_{C_i}$ to a face $f_j \in F_{C_j}$ contains exactly one edge from each of the sets $E^*_{C_i}$ and $E^*_{C_j}$ and, hence, the path also visits f_c. Thus, removing face f_c from G^* disconnects F_{C_i} from F_{C_j} and, hence, f_c is a cut face of G^*.

The previous reasoning only works if B_i and B_j contain at least two edges and, hence, F_{C_i} and F_{C_j} are not empty. If, for instance, B_i contains only one edge e, then f_c lies to the left and to the right of this face and therefore $F_{C_i} = \emptyset$. The dual of e is a loop from f_c to itself. Removing f_c from G^* does not disconnect the faces in F_{C_i} from the faces in F_{C_j} since there are no faces in F_{C_i} anyway; not to mention that F_{C_j} might also be empty. However, f_c is still a cut vertex in the following sense: In the dual, there is a loop from f_c to itself which is the single edge in B_i. In particular, any path in G^* which contains this loop must also visit f_c. In this sense, f_c is still a cut face. □



(a) Several cut vertices (but not all, e. g., c'') are incident to cut face f_c.



(b) Face f_c is a cut face since every path from a face f_i within C_i to a face f_j within C_j contains f_c.

Figure 3.24: The cut vertices are incident to cut faces.

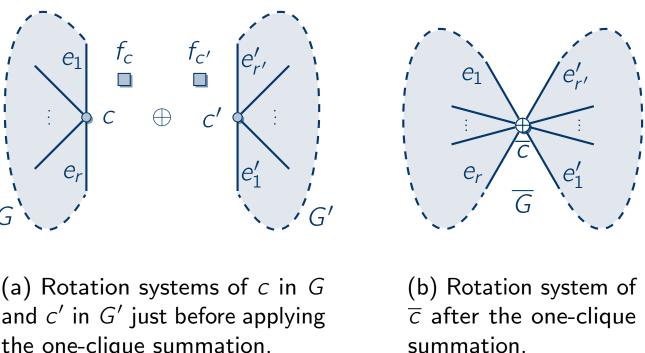
From Lem. 3.12, we obtain the following corollary:

Corollary 3.9. *The dual of an embedded and biconnected graph is also biconnected.*

Proof. Let G be embedded and biconnected with dual G^*. First note that G^* is always connected. If G^* would not be biconnected, we could find a cut face f_c. By Lem. 3.12 there is a cut vertex in the dual of G^*, which is (isomorphic to) G; a contradiction as G is biconnected. □

By Cor. 3.9 we can conclude that the duals of the blocks of G are also blocks of the dual graph G^*. Moreover, these blocks in the dual graph are connected by cut faces. In the following, we will see that the relationship between the block-cut tree of G and the block-cut tree of G^* is even more profound.

Assume that we are given the block-cut tree $\mathcal{T}_B = (\mathbf{B}, \mathbf{C}, \mathbf{E}_B)$ of a graph. In order to "reassemble" G from $\mathcal{T}_B$, we use an operation called *one-clique summation*. The name "one-clique sum" derives from the general operation of k-clique-sums: Let G_1 and G_2 be two graphs of which each contains a complete subgraph (clique) with at least k vertices. Given two cliques $H_1 \subseteq G_1$ and $H_2 \subseteq G_2$ with k vertices, the k-clique-sum of G_1 and G_2 is obtained by "glueing" them together at H_1 and H_2, that is, each vertex in H_1 is identified with a vertex in H_2. In case of the block-cut trees, we use the special case of $k = 1$ as follows. Let $B_i = (V_i, E_i)$ and $B_j = (V_j, E_j)$ be two blocks that are incident to the same cut vertex c in $\mathcal{T}_B$, i. e., both B_i and B_j contain c with $V_i \cap V_j = \{c\}$. The *one-clique sum* $B_i \oplus B_j$ of B_i and B_j is the union of B_i and B_j with $B_{i,j} = (V_i \cup V_j, E_i \cup E_j)$. In other words, B_i and B_j are "glued" together at c to obtain another subgraph of G. By applying the one-clique summation to graph $B_i \oplus B_j$ and another block $B_{j'} = (V_{j'}, E_{j'})$ with $V_{j'} \cap (V_i \cup V_j) = \{c'\}$, we obtain a third graph subgraph $B_i \oplus B_j \oplus B_{j'}$ of G, and so forth. In general, if G is connected, we can order the blocks $B_1, \dots, B_k$ totally such that each block B_i with $1 < i \leq k$ shares a cut vertex with at least one block from $B_1, \dots, B_{i-1}$. Such a total order, called *block sequence*, can be obtained by a depth-first search traversal of the block-cut tree. Let $\overline{B}_i = \overline{B}_{i-1} \oplus B_i$ with $\overline{B}_1 = B_1$ and $\overline{B}_k = G$ be a sequence of subgraphs of G obtained from subsequent one-clique summations of G's blocks. We call this sequence *block series* as it is the (one-clique-)sum of the blocks.



(a) Rotation systems of c in G and c' in G' just before applying the one-clique summation.

(b) Rotation system of $\overline{c}$ after the one-clique summation.

Figure 3.25: One-clique sum of two graphs G and G'.

Since we are dealing with embedded graphs, we also need to incorporate the rotation system during a one-clique summation. In general, let $G = (V, E)$ and $G' = (V', E')$ be two embedded graphs and c and c' bet two vertices in G and G', respectively. Note that this time we do not identify c and c' with each other. Vertex c has the rotation system $e_1, \dots, e_r$ in G and c' the rotation system $e'_1, \dots, e'_{r'}$ in G', where $r, r' \geq 1$. There is a face between each pair of subsequent edges in a rotation system. If the degree of, for instance, c is one, then there is a single face between $e_1 = e_r$ and itself. In a one-clique summation, we have to decide at which faces we want to unite G and G'. In the following, we choose w. l. o. g. the face f_c between e_r and e_1 in G and the face $f_{c'}$ between $e'_{r'}$ and e'_1 in G'. The situation before the

one-clique summation is depicted in Fig. 3.25(a). We call the result of merging G and G' at vertices c and c' at faces f_c and $f_{c'}$ the (c, f_c)-$(c', f_{c'})$*-one-clique sum (of* G *and* G'*)* denoted by $\overline{G} = G \oplus G'$. $\overline{G}$ is shown in Fig. 3.25(b) where c and c' are replaced by the (cut) vertex $\overline{c}$. The rotation system of $\overline{c}$ is $e_1, \dots, e_r, e'_1, \dots, e'_{r'}$ and, hence, the concatenation of the rotation system of c, split between e_r and e_1, and of c', split between $e'_{r'}$ and e'_1. Note that the face between edges e_r and e'_1 is equal to the face between $e'_{r'}$ and e_1. Also observe that the rotation system of $\overline{G}$ is planar as the rotation systems of G and G' are planar.

Interestingly, a one-clique summation of two (primal) graphs is also reflected in their duals: it turns out that the (c, f_c)-$(c', f_{c'})$-one-clique sum of G and G' corresponds to a (f_c, c)-$(f_{c'}, c')$-one-clique sum of G^* and G'^*. Remember that the faces of the dual graph, where "faces" is used in the very sense of the word, are the vertices of the primal graph. To distinguish the (primal) one-clique sum from the *dual one-clique sum*, we denote by $G^* \boxplus G'^*$ the (f_c, c)-$(f_{c'}, c)$*-one-clique sum* of G^* and G'^*. We obtain the following lemma:

Lemma 3.13 (De Morgan's Law for One-Clique Sums). *Let $G = (V, E)$ and $G' = (V, E)$ be two embedded graphs with duals $G^* = (F, E^*)$ and $G'^* = (F', E'^*)$, respectively. Further, let $c \in V$ and $c' \in V$ be two vertices and $f_c \in F$ and $f_{c'} \in F'$ such that c is incident to f_c and c' is incident to $f_{c'}$. The following equality holds:*

$$(G \oplus G')^* = G^* \boxplus G'^*,$$

where $(G \oplus G')^*$ denotes the dual of the (c, f_c)-$(c', f_{c'})$-one-clique sum of G and G', and $G^* \boxplus G'^*$ the (f_c, c)-$(f_{c'}, c')$-one-clique sum of G^* and G'^*.

We call Lem. 3.13 also De Morgan's Law for One-Clique Sums. De Morgan's law, from Boolean algebra, states that for two Boolean variables x and y, the following equation holds:

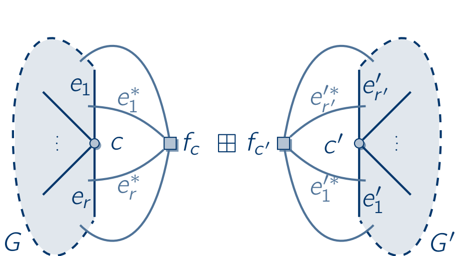
$$(x \wedge y)^c = x^c \vee y^c \,, \tag{3.4}$$

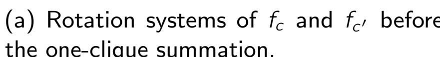
where x^c is the negation of x. When replacing ^{c} by *, $\wedge$ by $\oplus$, and $\vee$ by $\boxplus$, Eq. (3.4) resembles the equation from Lem. 3.13. The strategy of the proof is to show that the dual obtained from a one-clique summation of two primal graphs is equal to the one-clique sum of their duals. This is illustrated by the following commutative diagram:

$$\begin{array}{ccc} G, G' & \xrightarrow{*} & G^*, G'^* \\ \downarrow \oplus & & \downarrow \boxplus \\ G \oplus G' & \xrightarrow{*} & G^* \boxplus G'^* \end{array}$$

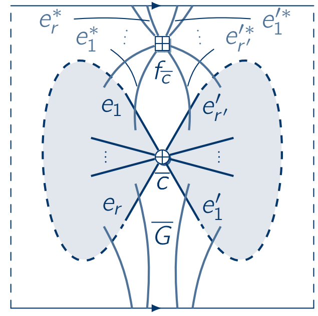
For this, we investigate what happens to the dual graph during a one-clique summation in the primal.

Proof. We start with the situation depicted in Fig. 3.26(a). The face between e_r and e_1 is face f_c. In the rotation system of face f_c the dual e_r^* of edge e_r is the predecessor of the dual e_1^* of e_1. Between e_1^* and e_r^*, the duals of the edges belonging to the circle indicated by the dashed line to the left in Fig. 3.26(a) are incident to f_c. As only e_1^* and e_r^* are relevant in the following, we neglect the other edges incident to f_c. Let $e_r^*, \dots, e_1^*$ be the rotation system of f_c. Note that vertex c lies between e_1^* and e_r^* in the rotation system of f_c. Likewise, in the dual of G' we obtain face $f_{c'}$ with rotation system $e_r'^*, \dots, e_1'^*$ and c' lies between $e_1'^*$ and $e_{r'}'^*$. Consider Fig. 3.26(b) that depicts the situation after the one-clique summation of c and c' at





(a) Rotation systems of f_c and $f_{c'}$ before the one-clique summation.



(b) Rotation system of $f_{\overline{c}}$ after the one-clique summation.

Figure 3.26: One-clique sum of two graphs G and G', and their duals.

faces f_c and $f_{c'}$, respectively. For the sake of clarity, the graph is displayed on the fundamental polygon of the rolling cylinder such that some dual edges can conveniently wind around the cylinder. We obtain face $f_{\overline{c}}$ with rotation system $e_r^*, \dots, e_1^*, e_{r'}'^*, \dots, e_1'^*$. The rotation system of $f_{\overline{c}}$ is the concatenation of the rotation system of f_c, split between e_1^* and e_r^*, and the rotation system of $f_{c'}$, split between $e_1'^*$ and $e_{r'}'^*$. Also, vertex $\overline{c}$ lies between edges e_1^* and $e_{r'}'^*$ and between $e_1'^*$ and e_r^*. The duals G^* and G'^* stay essentially unchanged during the one-clique summation. Only faces f_c and $f_{c'}$ are replaced by face $f_{\overline{c}}$. Altogether, we have obtained the (f_c, c)-$(f_{c'}, c')$-one-clique sum of G^* and G'^* in the dual graph $\overline{G}^*$ and, hence, the equality holds. □

Remember that we can reassemble a connected graph G and its embedding by a sequence of one-clique summations of its blocks by using the block series $\overline{B}_i$ of G. For each $1 < i \leq k$, c in $\overline{B}_{i-1}$ and c' in B_i are merged at faces f_c and $f_{c'}$ to obtain $\overline{B}_i$ and c is a cut vertex of $\overline{B}_i$. By De Morgan's law for one-clique sums (Lem. 3.13), each one-clique summation in the primal is also a one-clique summation in the dual. In the dual, faces f_c and $f_{c'}$ are replaced by face $f_{\overline{c}}$ to which cut vertex c is incident. By construction, face $f_{\overline{c}}$ lies between a block B_j in $\overline{B}_{i-1}$ and block B_i with $B_i \neq B_j$ and, hence, face $f_{\overline{c}}$ is a cut face in $\overline{B}_i$ by Lem. 3.12. Furthermore, the dual of each block is biconnected (Cor. 3.9) and, hence, a block in $\overline{B}_i^*$. All these observations together yield the following corollary:

Corollary 3.10. *For each embedded and connected graph G with block series $\overline{B}_i$, we obtain for the block series $\overline{B}_i^*$ of the dual graph G^*:*

$$\overline{B}_i^* = \overline{B}_{i-1}^* \boxplus B_i^* ,$$

with $\overline{B}_1^* = B_1^*$ and $\overline{B}_k^* = G^*$.

Let H be a subgraph of an embedded graph G and H^* be the dual of H. In general, H^* is no subgraph of G^*: Consider a triangle, i. e., the embedded complete graph with three vertices, and the subgraph obtained by removing any of the three edges. The dual of the triangle consists of two faces connected by three edges, whereas the dual of the subgraph consists of a single

face with two loops. In particular, the dual of the subgraph is no subgraph of the dual. In the case of blocks, however, we obtain the following corollary:

Corollary 3.11. *The dual of a block B_i of an embedded graph G is a subgraph of G^*.*

Proof. The dual B_i^* of B_i is a block of G^* by Cor. 3.10 and also a subgraph of G^*. □

Remember, technically spoken, B_i^* is *isomorphic* to a subgraph of G^*.

This concludes our general observations about block-cut trees of embedded graphs and their duals. These observations will be of great use in the following sections when we study the block-cut trees of **RUP** compounds and their duals.

3.5.2.2 Block-Cut Trees of Acyclic Dipoles

In this section, we investigate the block-cut trees of acyclic dipoles. In turns out that the linear structure of acyclic dipoles are reflected in their block-cut trees. First, note that each block of an acyclic digraph is also acyclic and if each block is acyclic, then so is the whole digraph. The latter follows from the fact the block-cut tree contains no circles.

Proposition 3.3. *A digraph is acyclic if and only if each block is acyclic.*

If a digraph is not only acyclic, but also a dipole, then its block-cut tree is a path. In fact, we obtain a characterization of acyclic dipoles by means of their block-cut trees.

Lemma 3.14. *An acyclic digraph is a dipole with source s and sink t if and only if the block-cut tree is a path $B_1, c_1, B_2, \ldots, B_{k-1}, c_{k-1}, B_k$, where each c_i with $1 \le i < k$ is the cut vertex in block B_i and B_{i+1}, and each block B_i is an acyclic dipole with source c_{i-1} and sink c_i, where, with a slight abuse of notation, $c_0 := s$ and $c_k := t$.*

In order to prove Lem. 3.14, we need further lemmas:

Lemma 3.15. *In an acyclic dipole, neither its source nor its sink is a cut vertex.*

Proof. Suppose for contradiction that s is a cut vertex. Then, there are at least two distinct blocks B_i and B_j containing s. Removing s partitions the block-cut tree $\mathcal{T}_B$ of G into at least two subgraphs $\mathcal{T}_B^i$ and $\mathcal{T}_B^j$ where $\mathcal{T}_B^i$ contains B_i and $\mathcal{T}_B^j$ contains B_j, and none contains s. Denote by $\overline{G}_i$ and $\overline{G}_j$ the union of all blocks in $\mathcal{T}_B^i$ and $\mathcal{T}_B^j$, respectively. Note that $\overline{G}_i$ and $\overline{G}_j$ are disjoint except for cut vertex s. By Prop. 3.3, both $\overline{G}_i$ and $\overline{G}_j$ are acyclic and, hence, each contains at least one source, which is s, and one sink. Hence, G itself contains at least two sinks contradicting the assumption that G is a dipole. By a similar reasoning, t cannot be a cut vertex. □

Lemma 3.16. *If G is an acyclic dipole and p a dipath from G's source to its sink, then the following statements hold true:*

(i) If p contains an edge of block B_i, then B_i contains at most two cut vertices and all cut vertices of B_i are visited by p.

(ii) If p visits a cut vertex c, then c is in at most two blocks and p traverses at least one edge of each block containing c.

Proof. (i): Let p be a simple path from the source s to the sink t of G. Denote by $B_1, \dots, B_k$ ($k \geq 1$) the blocks in the order in which they are encountered by p, that is, p traverses an edge of a block. Note that $B_i \neq B_j$ for all $1 \leq i < j \leq k$ as otherwise p would not be simple. Source s is in B_1 and sink t is in B_k. First, suppose that $k \geq 2$. For each $1 < i < k$, dipath p enters block B_i via a cut vertex c_{i-1} and leaves block B_i by another cut vertex $c_i \neq c_{i-1}$. Towards a contradiction, suppose that B_i contains a third cut vertex c' with $c_{i-1} \neq c'$ and $c_i \neq c'$. Cut vertex c' connects block B_i to another block B', where p does not contain an edge of B'. Further, as s is the single source and t the single sink, there is a dipath $p' = s \rightsquigarrow v \rightsquigarrow t$, where v is a vertex in B' distinct from c'. Since c_{i-1}, c_i and c' are distinct cut vertices, and s is in B_1 and t in B_k, we obtain for dipath $p' = s \rightsquigarrow c_{i-1} \rightsquigarrow c' \rightsquigarrow v \rightsquigarrow c' \rightsquigarrow c_i \rightsquigarrow t$ and, thus, G would contain a cycle which is a contradiction. Therefore, each B_i with $1 < i < k$ contains exactly two cut vertices c_{i-1} and c_i which are both visited by p. By a similar reasoning, B_1 contains exactly one cut vertex (c_1) by which p leaves B_1, and B_k contains exactly one cut vertex (c_{k-1}) by which p enters B_k. Altogether, dipath p visits all cut vertices of the blocks $B_1, \dots, B_k$. If $k = 1$, i. e., s and t are both in B_1, then, by the same argument as before, B_1 does not contain any cut vertex and the statement is vacuously true.

(ii): The reasoning is similar to (i). As before, let $B_1, \dots, B_k$ ($k \geq 1$) be the blocks in the order in which they are visited by p. For contradiction, suppose that p visits a cut vertex c_i while leaving block B_{i-1} and entering block B_i and there is a block B' that contains c_i but p does not contain an edge of B'. In other words, there are at least three blocks that contain c_i. Let v be a vertex in B' that is distinct from c_i. As before, there is a dipath $p' = s \rightsquigarrow v \rightsquigarrow t$ which visits c_i twice, i. e., $p' = s \rightsquigarrow c_i \rightsquigarrow v \rightsquigarrow c_i \rightsquigarrow t$; a contradiction. □

Corollary 3.12. *Let G be an acyclic dipole and p be a dipath from G's source to its sink. Dipath p visits each cut vertex exactly once and contains an edge of each block.*

Proof. If $s = t$, then G consists of a single vertex and the statement is vacuously true. Otherwise, $s \neq t$ and p contains at least one edge from a block B_i and, hence, p contains all cut vertices in B_i by Lem. 3.16. The cut vertices of B_i are themselves connected to other blocks and so forth. Since the underlying undirected digraph of G is connected, p visits each cut vertex at least once and contains an edge of each block. As p is simple, it visits each cut vertex at most once and, thus, exactly once. □

We can now prove Lem. 3.14:

Proof. [Lem. 3.14] ⇒: If G is biconnected, the block-cut tree consists only of G itself and the statement is vacuously true. Otherwise, let G be an acyclic dipole with source s and sink t, at least one cut vertex, and block-cut tree $\mathcal{T}_B = (\mathbf{B}, \mathbf{C}, \mathbf{E}_B)$. Observe that the underlying undirected graph of G is connected and so is $\mathcal{T}_B$.

By Lem. 3.16 we know that each block contains at most two cut vertices and each cut vertex is in at most two blocks. Consequently, the degree of each cut vertex and block in $\mathcal{T}_B$ of G is at most two and, since $\mathcal{T}_B$ is connected, it is a path $B_1, c_1, B_2, \dots, B_{k-1}, c_{k-1}, B_k$. As any dipath $s \rightsquigarrow t$ visits each cut vertex exactly once (Cor. 3.12) and as s and t are no cut vertices (Lem. 3.15), s and t cannot be in the same block. For the same reasons, s must be in B_1 and t in B_k or vice versa. In the following, we assume w. l. o. g. that $s \in B_1$ and $t \in B_k$.

What is left to show is that each block is a dipole. Let B_i be a block with $1 < i < k$. B_i is acyclic and, hence, contains at least one source and one sink. Also, B_i contains two cut vertices of G, namely, c_{i-1} and c_i. c_{i-1} is the single source and c_i the single sink of B_i for

the following reasons: Any vertex v with $v \neq c_{i-1}$ and $v \neq c_i$ of B_i can neither be a source nor a sink since otherwise G would be no dipole. Hence, either c_{i-1} is the source and c_i the sink or vice versa. Suppose that c_{i-1} is the sink. Then, c_{i-1} is the source in block B_{i-1} since otherwise c_{i-1} is a sink of G which is not possible due to Lem. 3.15. Since G is a dipole, there is a dipath $p = s \rightsquigarrow c_{i-1}$ with s in B_1. Necessarily, p traverses an edge of B_{i-1} to reach c_{i-1}. However, all edges in B_{i-1} incident to c_{i-1} point away from c_{i-1}; a contradiction. Thus, c_{i-1} is the source and c_i the sink of B_i. By a similar reasoning, $c_0 := s$ is the source and c_1 the sink of B_1 and, likewise, for B_k, c_{k-1} is the source and $c_k := t$ the sink.

$\Leftarrow$: First note that G is connected since its block-cut tree is connected. Each cut vertex c_i with $1 \leq i < k$ is neither a source nor a sink in G since c_i is a sink in B_i and source in B_{i+1}. Since each block B_i is acyclic, G itself is acyclic by Prop. 3.3. Further, each block B_i is also a dipole with source c_{i-1} and sink c_k, where $c_0 = s$ is the single source and $c_k = t$ the single sink of G. Hence, G is an acyclic dipole with source s and sink t. □

3.5.2.3 Characterizing RUP Compounds by their (Directed) Block-Cut Trees

In this section, we characterize the block-cut trees of **RUP** compounds. This characterization is then used in Sect. 3.5.2.4 for an efficient testing algorithm. As in Sect. 3.5.1, we first derive a series of necessary conditions for an embedding to be **RUP** and these conditions together are also sufficient and, hence, yield a characterization. As a tool, we introduce the directed block-cut tree in which the direction of the edges encode the embedding of each block to a certain degree. It turns out that the compound at hand has a **RUP** embedding if and only if its directed block-cut tree has a Eulerian dipath. In the following, we assume that if the compound in endowed with a **RUP** embedding, then each block inherits this embedding.

Using Block Chains to Embed RUP Compounds A useful property of compounds is that each of its blocks is a compound. Conversely, if each block is a compound, then the whole digraph is. This follows from the fact that the blocks are connected to each other at common cut vertices. We get the following proposition:

Proposition 3.4. *A connected digraph is a compound if and only if each block is a compound.*

Remember that the dual γ^* of a **RUP**-embedded compound γ is an acyclic dipole by Lem. 3.4. Hence, we can apply Lem. 3.14, i. e., the block-cut tree of γ^* is a path and each block is an acyclic dipole. Further, the blocks of γ^* are connected to each other at their sources and sinks. The blocks of γ^* are the duals of the blocks of γ by Cor. 3.10, which also applies for digraphs. All these observations together yield the following corollary:

Corollary 3.13. *Let γ be an embedded compound with dual γ^*. The embedding of γ is* **RUP** *if and only if the blocks of γ can be totally ordered from $B_1, \ldots, B_k$ such that the block-cut tree of γ^* is a path $B_1^*, f_{c_1}, B_2^*, \ldots, B_{k-1}^*, f_{c_{k-1}}, B_k^*$ where all f_{c_i} with $1 \leq i < k$ are the cut faces and each block B_i^* is an acyclic dipole with source $f_{c_{i-1}}$ and sink f_{c_i} where f_{c_0} is the source of B_1^* and f_{c_k} the sink of B_k^*.*

Since the block-cut tree of a **RUP**-embedded compound is a path, a traversal of this path in any direction defines a total order of the blocks of the dual and, hence, also of the blocks of the primal. This total order $B_1, \ldots, B_k$ has the property that each pair B_i, B_{i+1} of subsequent blocks shares a cut vertex since B_i^*, B_{i+1}^* share a cut face in the dual. This type of total order plays an important role in the following and deserves its own name:

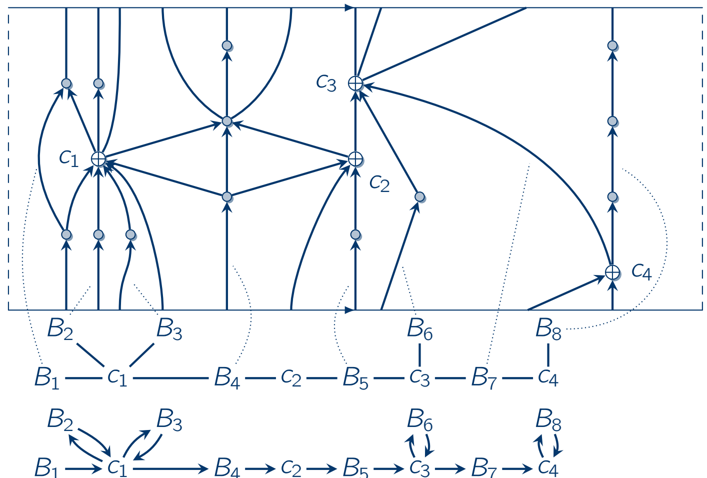


Figure 3.27: **RUP** embedding of compound γ_1 with its block-cut tree in the middle and its directed block-cut tree (Def. 3.4) at the bottom.

Definition 3.3. *A total order $B_1, \dots, B_k$ of the set of blocks of a digraph is called* block chain *if and only if each pair B_i, B_{i+1} of subsequent blocks share a cut vertex.*

In Sect. 3.5.2.1, we defined the block sequence of a connected digraph which is a total order of the blocks $B_1, \dots, B_k$ such that B_i shares a cut vertex with at least one block of $B_1, \dots, B_{i-1}$. A block chain is a special kind of block sequence. However, not every block sequence is a block chain and not every digraph has a block chain as we will see later. For the moment, we obtain:

Corollary 3.14. *A* **RUP***-embedded compound has a block chain and so has its dual.*

In the following we assume that all blocks of a **RUP**-embedded compound are ordered according to a block chain $B_1, \dots, B_k$. As running example, we consider the compound γ_1 in Fig. 3.18(a) on page 124 whose **RUP** embedding is displayed in Fig. 3.27. It consists of blocks $B_1, \dots, B_8$ connected by cut vertices $c_1, \dots, c_4$. Again, the cut vertices are displayed by the symbol $\oplus$. A block chain of γ_1 is $B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8$. Note that a single cut vertex can be shared by more than two blocks, e. g., c_1 by B_1, B_2, B_3, and B_4. Fig. 3.28 shows the dual of γ_1: At the top, the drawing of γ_1^* is laid upon the drawing of γ_1 and all cut faces are displayed by the symbol $\boxplus$. Immediately below the fundamental polygon, a more concise drawing of γ_1^* is shown where, for illustration purposes, all multiple edges are replaced by a single one and a more comprehensible embedding is chosen. At the bottom, the block-cut tree of γ_1^* is shown, the cut faces are numbered from f_{c_1} to f_{c_7}. For convenience, the cut vertices of all three drawings are aligned, i. e., f_{c_i} has the same x-coordinate in all three drawings for all $1 \leq i \leq 7$.

Consider blocks B_4 and B_5 (Fig. 3.27) and their duals B_4^* and B_5^* (Fig. 3.28). The sink of B_4^* is cut face f_{c_4}, which is the source of B_5^*. In the primal digraph B_4, face f_{c_4} corresponds to the rightmost cycle of B_4's embedding and, likewise, f_{c_4} to the leftmost cycle of B_5. Furthermore, B_4 is connected to block B_5 by cut vertex c_2. In order to attach B_4 and B_5 to each other, their common cut vertex c_2 must be rightmost in B_4 and leftmost in the B_5. In general, we obtain that a cut vertex shared by two blocks must be leftmost in one block and rightmost in the other block in any **RUP** embedding.

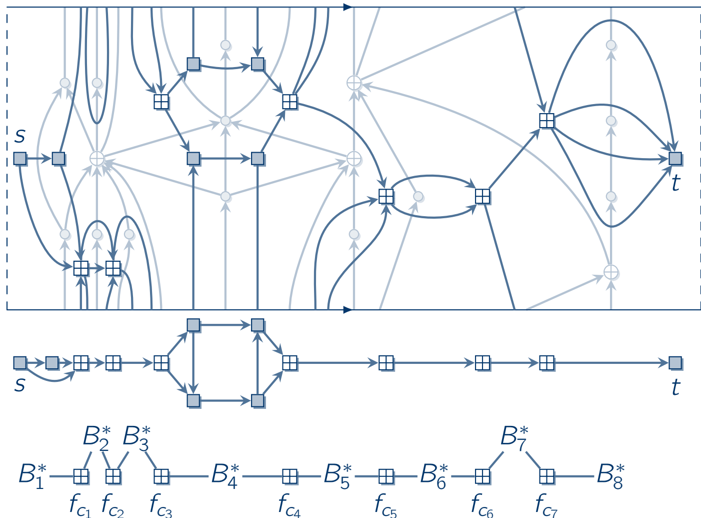


Figure 3.28: Dual γ_1^* of compound γ_1 on top. In the middle, a more concise drawing of γ_1^* is shown, where all multiple edges are removed and a different embedding is used. At the bottom, the block-cut tree of γ_1^* is shown.

Lemma 3.17. *If B_i and B_j with $1 \leq i < j \leq k$ are two blocks of a* **RUP***-embedded compound γ which share cut vertex c, then c is rightmost in B_i and leftmost in B_j.*

Proof. Denote by $\overline{\gamma} \subseteq \gamma$ the union of blocks B_i and B_j. $\overline{\gamma}$ is strongly connected (Prop. 3.4) and inherits the **RUP** embedding of γ. The situation is illustrated in Fig. 3.29, where B_i and B_j are displayed as shaded regions. We can apply Cor. 3.13: Consider the block-cut tree $\mathcal{T}_B^* = (\mathbf{B}^*, \mathbf{C}^*, \mathbf{E}_B^*)$ of γ^* which consists of blocks $\mathbf{B}^* = \{B_i^*, B_j^*\}$, a single cut face $\mathbf{C}^* = \{f_c\}$, and edges $\mathbf{E}_B^* = \{\{B_i^*, f_c\}, \{B_j^*, f_c\}\}$. $\mathcal{T}_B^*$ is displayed at the bottom of Fig. 3.29. Cut vertex c is incident to cut face f_c, which in turn is the sink of B_i^* and the source of B_j^* and, thus, c is rightmost in B_i and leftmost in B_j. □

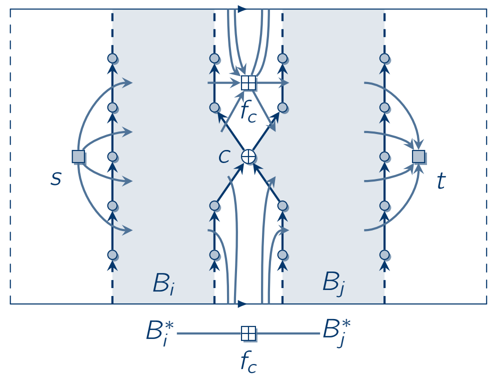


Figure 3.29: Cut vertex c is shared between block B_i and B_j and cut face f_c lies between them. The block-cut tree of the dual digraph is shown at the bottom.

Note that in the proof of Lem. 3.17, not necessarily $i = j - 1$ holds. For instance, block B_1 and block B_4 in Fig. 3.27 share cut vertex c_1, which is rightmost in B_1 and leftmost in B_4. By the contrapositive of Lem. 3.17, we get that if a cut vertex is neither left- nor rightmost in the **RUP** embedding of a block, the embedding of the whole compound cannot be **RUP**. We call a **RUP** embedding of a block B_i *feasible* if each of the block's cut vertices is left- or rightmost, or both. We obtain the following corollary from Lem. 3.17:

Corollary 3.15. *In a* **RUP***-embedded compound, the* **RUP** *embedding of each block is feasible.*

By combining Cor. 3.13 with Lem. 3.17, we obtain the following lemma which is a major leap towards our decision algorithm:

Lemma 3.18. *A compound γ is* **RUP** *if and only if γ has a block chain $B_1, \dots, B_k$ with:*

(i) Each block B_i has a feasible **RUP** *embedding.*

(ii) Let B_i and B_{i+1} be two blocks which share cut vertex c_i. In the **RUP** *embedding of B_i, cut vertex c_i is rightmost and, in the* **RUP** *embedding of B_{i+1}, c_i is leftmost.*

Proof. First note that in (ii), the cut vertex shared between B_i and B_{i+1} is denoted by c_i for convenience. As c_i may be shared between several blocks, $c_i = c_{i+1}$ is possible if cut vertex c_i is shared by blocks B_i, B_{i+1}, and B_{i+2}.

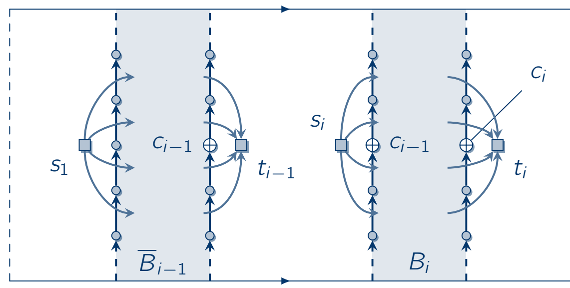
$\Rightarrow$: Since γ is endowed with a **RUP** embedding, so is each of its blocks. By Cor. 3.13, we know that γ^*'s block-cut tree is a path from which we obtain the block chain $B_1, \dots, B_k$. By Lem. 3.17, the cut vertex shared between B_i and B_{i+1} is rightmost in B_i and leftmost in B_{i+1} and, hence, (i) and (ii) follow.

$\Leftarrow$: We construct a **RUP** embedding by attaching the blocks $B_1, \dots, B_k$ to one another at their cut vertices and at the sources and sinks of the blocks' duals. Each block B_i has a feasible **RUP** embedding, where B_i^* is its dual with source s_i and sink t_i. Let $\overline{B}_i$ be the block series of γ in order of the block chain. That is, $\overline{B}_i = \overline{B}_{i-1} \oplus B_i$ for all $1 < i \leq k$ with $\overline{B}_1 = B_1$ and where $\overline{B}_i = \overline{B}_{i-1} \oplus B_i$ is the (c_{i-1}, t_{i-1})-(c_{i-1}, s_i)-one-clique sum of $\overline{B}_{i-1}$ and B_i, and c_{i-1} is the cut vertex shared between $\overline{B}_{i-1}$ and B_i. For the dual $\overline{B}_i^*$ of $\overline{B}_{i-1}$, we get by Cor. 3.10:

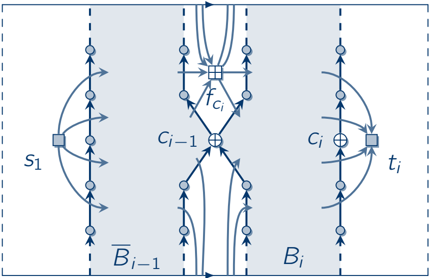
$$\overline{B}_i^* = (\overline{B}_{i-1} \oplus B_i)^* = \overline{B}_{i-1}^* \boxplus B_i^* ,$$

where $\overline{B}_{i-1}^* \boxplus B_i^*$ is the (t_{i-1}, c_{i-1})-(s_i, c_{i-1})-one-clique sum of $\overline{B}_{i-1}^*$ and B_i^*. The situation before the one-clique summation is shown in Fig. 3.30(a) and the situation afterwards is shown in Fig. 3.30(b). By this construction, we obtain for the block-cut tree of $\overline{B}_k^*$ a path $B_1^*, f_{c_1}, B_2^*, f_{c_2}^*, \dots, f_{c_{k-1}}^*, B_k^*$, where each cut face $f_{c_i}^*$ is the sink of B_i^* and the source of B_{i+1}^* for all $1 \leq i < k$. The source of B_1^* is s_1 and the sink of B_k^* is t_k. Hence, all properties of Cor. 3.13 are fulfilled and we can conclude that G is **RUP**-embedded. □

Lem. 3.18 breaks down the complexity of testing whether a compound is **RUP** into two tasks: First, we have to find a block chain $B_1, \dots, B_k$. Then, we need to find a feasible **RUP** embedding for each block such that the cut vertex shared between B_i and B_{i+1} is rightmost in B_i and leftmost in B_{i+1} for each $1 \leq i < k$. Also, the proof of Lem. 3.18 is constructive: under the assumption that we already have obtained a block chain and a **RUP** embedding of each block that fulfills the properties of Lem. 3.18, we can construct the **RUP** embedding of the whole compound in time linear in the size of the compound by locally attaching the cut vertices to each other.



(a) The situation before the (c_{i-1}, t_{i-1})-(c_{i-1}, s_i)-one-clique summation of $\overline{B}_{i-1}$ and block B_i.



(b) The situation after the one-clique summation of $\overline{B}_{i-1}$ and block B_i.

Figure 3.30: Situation obtained in the proof of Lem. 3.18.

Caterpillars and Block-Cut Trees of RUP Compounds A necessary condition for a compound to be **RUP** is that has a block chain. This raises the following general question: When does a graph have a block chain? In particular, what is the structure of the block-cut tree of these graphs? It turns out that the structure is a very special one, namely, that of a *caterpillar*: A caterpillar, first studied by Harary and Schwenk [HS73], is a tree where the removal of all leaves results in a path. A single vertex is also a caterpillar. We obtain the following lemma:

Lemma 3.19. *A graph has a block chain if and only if its block-cut tree is a caterpillar.*

Proof. Let $\mathcal{T}_B = (\mathbf{B}, \mathbf{C}, \mathbf{E}_B)$ be the block-cut tree of a graph G. For both implications, we assume that the graph and, hence, also $\mathcal{T}_B$ are connected since otherwise neither the graph has a block chain nor is $\mathcal{T}_B$ a caterpillar. Remember, a graph is biconnected if and only if its block-cut tree consists only of a single vertex, which is a caterpillar and the graph has a block chain in this case. In following, we assume that the graph contains at least one cut vertex.

We use the following characterization of caterpillars: a tree is a caterpillar if and only if there is a simple path that contains each vertex of degree two or more. The proof of this characterization is as follows: After removing the leaves from the caterpillar, a path remains. This path is simple in the original graph and contains all vertices of degree two or more. Conversely, if a graph has a simple path that contains each vertex of degree two or more, all other vertices have degree one and are leaves. After removing the leaves, the path remains.

$\Rightarrow$: The block chain is a total order on the blocks that corresponds to a path $p = (B_1, c_1, B_2, \dots, c_{k-1}, B_k)$ $(k > 1)$ in $\mathcal{T}_B$, where B_i and B_{i+1} share cut vertex c_i and, thus, $\{B_i, c_i\}$ and $\{c_i, B_{i+1}\}$ are edges of $\mathcal{T}_B$. Path p visits each block exactly once but may contain a cut vertex several times. Note, as in the proof of Lem. 3.18, $c_i = c_{i+1}$ is possible. Some of the blocks visited by p are leaves. Note that a cut vertex is never a leaf in a block-cut tree. We obtain path p' from p by skipping each leaf in p: Whenever p visits a leaf B_i, i. e., $p = \dots c_{i-1}, B_i, c_i, \dots$ with $c_{i-1} = c_i$, p' only visits $c_i = c_{i-1}$ and skips B_i.

We now show that p' is simple. Since p contains each block exactly once, p' visits each block at most once. Now suppose for contradiction that p' visits a cut vertex c_i at least twice. Let B_j be the block in p' just after c_i is visited for the first time and let $B_{j'}$ be the block just before returning to c_i. The respective subsequence in p' is a circle $C = c_i, B_j, \dots, B_{j'}, c_i$, which is not necessarily simple. We now show that $B_j \neq B_{j'}$. Neither B_j nor $B_{j'}$ is a leaf and, thus,

both are adjacent to cut vertices different from c_i. In particular, since both B_j and $B_{j'}$ are visited exactly once, there are cut vertices c_ℓ and $c_{\ell'}$ such that $C = c_i, B_j, c_\ell, \dots, c_{\ell'}, B_{j'}, c_i$. Therefore, circle C leaves and enters c_i via different blocks and does not contain c_i in between. Thus, the block-cut tree $\mathcal{T}_B$ contains circles which is a contradiction. Altogether, path p' is a simple path that contains all vertices of degree two or more since we obtained p' from p by skipping only the leaves, and we conclude that $\mathcal{T}_B$ is a caterpillar.

$\Leftarrow$: Let p be a simple path in $\mathcal{T}_B$ that contains all vertices, i. e., blocks and cut vertices, of degree two or more. Since p is simple, each block in p is visited exactly once. We can subsequently extend p such that it also visits all leaves of $\mathcal{T}_B$. Note that a leaf in $\mathcal{T}_B$ is always a block which is adjacent to a cut vertex. Whenever p visits a cut vertex for the first time that is adjacent to one or more leaves, we can take a "detour" to each of these leaves. The so obtained path p', which may not be simple, visits each block exactly once and, thus, induces a total order $B_1, \dots, B_k$ on the set of blocks. Between each subsequent pair B_i, B_{i+1} of blocks, p' visits cut vertex c_i and, thus, B_i and B_{i+1} share c_i. Consequently, $B_1, \dots, B_k$ is a block chain of G. □

Corollary 3.16. *If the block-cut tree of a graph is a caterpillar, then each block contains at most two cut vertices.*

Proof. First note that all leaves of the block-cut tree are blocks, which contain only one cut vertex. Furthermore, all leaves are connected to cut vertices and, hence, removing the leaves does not change the degree of all other blocks. After the removal of all leaves, we obtain a path in which all blocks have degree at most two and, consequently, these blocks contain at most two cut vertices. □

Corollary 3.17. *The block-cut tree of a* **RUP** *compound is a caterpillar and each block contains at most two cut vertices.*

The block-cut tree of γ_1 is shown directly beneath the fundamental polygon in Fig. 3.27 and it is indeed a caterpillar.

The path that remains after removing all leaves form a caterpillar is called *spine*. If the block-cut tree is a caterpillar, we call the blocks on the spine *spine blocks* and all other blocks *leaf blocks*. Further, traversing the spine of a caterpillar in any direction induces a total order $c_1, \dots, c_\ell$ on the set of cut vertices, where ℓ is the number of cut vertices. We say that this total order is *induced by the caterpillar*. In the following discussion, it makes no difference of whether we traverse the spine in one or the other direction to obtain this total order. Note that each spine block has degree two and is incident to two cut vertices c_j and c_{j+1} with $1 \leq j < \ell$ in the block-cut tree.

Directed Block-Cut Trees of RUP Compounds By Cor. 3.17, the block-cut tree of a **RUP** compound is a caterpillar and, in addition, Lem. 3.18 demands that each block has a feasible **RUP** embedding with certain properties, i. e., the cut vertex between blocks B_i and B_{i+1} must be rightmost in B_i and leftmost in B_{i+1}. Using these properties, we equip the block-cut tree with some additional attributes that store relevant information about the embedding of each block. For this, we define the directed block-cut tree.

Definition 3.4 (Directed Block-Cut Tree). *Let $\mathcal{T}_B = (\mathbf{B}, \mathbf{C}, \mathbf{E}_B)$ be the block-cut tree of a compound. If each block B_i is feasibly* **RUP***-embedded and $\mathcal{T}_B$ is a caterpillar, we define the* directed block-cut tree *$\vec{\mathcal{T}}_B = (\mathbf{B}, \mathbf{C}, \vec{\mathbf{E}}_B)$ as follows. Let $c_1, \dots, c_\ell$ be the total order of*

the cut vertices as induced by the caterpillar. For each block B_i, we introduce directed edges depending on the following cases:

- **B_i is a spine block containing cut vertices c_j and c_{j+1} with $1 \leq j < \ell$**
  There is an edge $(c_j, B_i) \in \vec{\mathbf{E}}_B$ if and only if c_j is leftmost in B_i, and there is an edge $(B_i, c_{j+1}) \in \vec{\mathbf{E}}_B$ if and only if c_{j+1} is rightmost in B_i.
- **B_i is a leaf block containing cut vertex c_j**
  There is an edge $(c_j, B_i) \in \vec{\mathbf{E}}_B$ if and only if c_j is leftmost in B_i, and there is an edge $(B_i, c_j) \in \vec{\mathbf{E}}_B$ if and only if c_j is rightmost in B_i.

Note that Def. 3.4 assumes some preconditions before it can be applied: First, each block must be feasibly **RUP**-embedded. In contrast, an (undirected) block-cut tree, does not presume an embedding (of any kind). Second, the (undirected) block-cut tree must be a caterpillar which induces a total order $c_1, \dots, c_\ell$ of the cut vertices. Only if these conditions are fulfilled, we can derive the directed block-cut tree. Another subtlety of the directed block-cut tree is that its definition heavily relies on the total order of the cut vertices. For a spine block B_i with cut vertices c_j and c_{j+1}, there is an edge (c_j, B_i) if and only if c_j is leftmost in B_i. Conversely, there is never an edge pointing from B_i to c_j, even if c_j is rightmost in B_i. Analogously, there can only be an edge from B_i to c_{j+1} but never in the opposite direction. Suppose that B_i has a feasible **RUP** embedding, however, both c_j and c_{j+1} are leftmost but not rightmost in B_i. Then, there is an edge from c_j to B_i but no edge between B_i and c_{j+1} as c_{j+1} is not rightmost in B_i. That is, even if there is an edge between a block and a cut vertex in the (undirected) block-cut tree, there may be no edge between those two in the directed version, i. e., the directed block-cut tree might not be connected.

The directed block-cut tree of γ_1 is shown at the bottom of Fig. 3.27. In the **RUP** embedding of B_4, c_1 is leftmost, whereas is c_2 rightmost. Hence in $\vec{\mathcal{T}}_B$, block B_4 has an incoming edge from c_1 and an outgoing edge to c_2. Also observe that there is neither an edge from B_5 to c_2 nor an edge from c_3 to B_5 although both cut vertices are left- and rightmost in B_5. Note that $\vec{\mathcal{T}}_B$ is technically no tree since it contains antiparallel edge pairs whenever a cut vertex is both left- and rightmost in the same block, e. g., B_2 or B_3, or it may be unconnected. However, its overall structure is still that of a tree and, hence, we stick to the term. From Def. 3.4, we obtain:

Corollary 3.18. *For each block B_i in a directed block-cut tree, the indegree and outdegree of B_i is at most one, i. e., $\mathrm{d}^-(B_i) \leq 1$ and $\mathrm{d}^+(B_i) \leq 1$.*

A neat property of the directed block-cut tree is as follows. Consider again Fig. 3.27. The blocks $B_1, \dots, B_8$ of γ_1 in that order already comprise a block chain that fulfills the properties of Lem. 3.18. In the directed block-cut tree of γ_1, we can trace a dipath p according to the block chain, which is:

$$p = B_1, c_1, B_2, c_1, B_3, c_1, B_4, c_2, B_5, c_3, B_6, c_3, B_7, c_4, B_8\,.$$

Dipath p is Eulerian, i. e., it contains each edge exactly once. Also note that p is Hamiltonian on the set of blocks as it visits each block exactly once. We obtain the following lemma:

Lemma 3.20. *The directed block-cut tree of a* **RUP***-embedded compound has a Eulerian dipath.*

Proof. Let γ be a **RUP**-embedded compound with dual γ^*. If γ is biconnected, then it contains not cut vertex and its directed block-cut tree consists of only a single vertex with no edges and the statement is vacuously true. Otherwise, let $B_1, \dots, B_k$ $(k > 1)$ be the block chain as induced by γ^* as in Cor. 3.13. The block-cut tree of γ is a caterpillar and each of its blocks is feasibly **RUP**-embedded. Thus, we can apply Def. 3.4 to obtain the directed block-cut tree $\vec{\mathcal{T}}_B = (\mathbf{B}, \mathbf{C}, \vec{\mathbf{E}}_B)$. Remember that each pair B_i, B_{i+1} of subsequent blocks share cut vertex c_i $(1 \leq i < k)$, where c_i is rightmost in B_i and leftmost in B_{i+1} by Lem. 3.18. Hence, in $\vec{\mathcal{T}}_B$, there are edges $(B_i, c_i), (c_i, B_{i+1}) \in \vec{\mathbf{E}}_B$. From the block chain, we obtain a dipath $p = B_1, c_1, B_2, \dots, c_{k-1}, B_k$ in the directed block-cut tree that visits each block exactly once.

We show that p, or a slight modification of p, is Eulerian. We first investigate all blocks B_i with $1 < i < k$. Dipath p enters B_i via an incoming edge e and leaves B_i via an outgoing edge e'. By Cor. 3.18, these two edges are the only edges incident to B_i and, hence, $\mathrm{d}^-(B_i) = \mathrm{d}^+(B_i)$. Since B_i is visited exactly once, e and e' are traversed exactly once by p.

For B_1 and B_k, we possibly need to extend dipath p at its ends. Note that $\vec{\mathcal{T}}_B$ contains edge (B_1, c_1); see also Fig. 3.27 where B_1 has an outgoing to c_1. If also $(c_1, B_1) \in \vec{\mathbf{E}}_B$, then we let p start at c_1 instead of B_1, i. e., $p = c_1, B_1, c_1, B_2, \dots, c_{k-1}, B_k$. Otherwise, we leave p as it is and let it start at B_1. Likewise, we know that $(c_{k-1}, B_k) \in \vec{\mathbf{E}}_B$ and if also $(B_k, c_{k-1}) \in \vec{\mathbf{E}}_B$, e. g., for B_8 and c_4, we let p end at c_{k-1} instead of B_k. Then, p also traverses each incident edge of B_1 and B_k exactly once. Since $\vec{\mathcal{T}}_B$ contains only edges between blocks and cut vertices, we can conclude that p is a Eulerian dipath. □

Now suppose the directed block-cut tree has a Eulerian dipath. By Cor. 3.18, each block has at most one incoming and at most one outgoing edge. Hence, the Eulerian dipath visits each block exactly once. This induces a total order on the set of blocks which is, in fact, the block chain that fulfills the properties of Lem. 3.18. We get the converse of Lem. 3.20.

Lemma 3.21. *Let γ be a compound whose block-cut tree is a caterpillar and whose blocks are feasibly* **RUP***-embedded. If the directed block-cut tree of γ contains a Eulerian dipath, γ is* **RUP***.*

Proof. If γ is biconnected, it consists of only a single block that is (feasibly) **RUP**-embedded and so is γ. Otherwise, γ contains at least one cut vertex. We use Lem. 3.18 for the proof. By assumption, each block has a feasible **RUP** embedding and (i) follows immediately. Let $\vec{\mathcal{T}}_B = (\mathbf{B}, \mathbf{C}, \vec{\mathbf{E}}_B)$ be the directed block-cut tree of γ and denote by p the Eulerian dipath in $\vec{\mathcal{T}}_B$. Each block has in- and outdegree at most one in $\vec{\mathcal{T}}_B$ and, hence, each block is visited exactly once by p. Let $B_1, \dots, B_k$ be the (total) order in which the blocks are visited. For each pair of subsequent blocks B_i, B_{i+1} with $1 \leq i < k$, the Eulerian dipath p visits a cut vertex c_i and, thus, $(B_i, c_i) \in \vec{\mathbf{E}}_B$ and $(c_i, B_{i+1}) \in \vec{\mathbf{E}}_B$. This implies that B_i and B_{i+1} share a cut vertex and, hence, $B_1, \dots, B_k$ is a block chain. Also, c_i is rightmost in B_i and leftmost in B_{i+1}, which implies (ii). □

In fact, from a Eulerian dipath in the directed block-cut tree we not only can conclude that the compound at hand is **RUP**, we can also construct its **RUP** embedding: The Eulerian dipath induces a block chain and we can assemble the **RUP** embeddings of the blocks by a series of one-clique summations as in the proof of Lem. 3.18,.

Lemmas 3.20 and 3.21 together yield a characterization of **RUP**-embeddable compounds, which will be most useful for the decision algorithm in Sect. 3.5.2.4. This characterization aptly summarizes this section.

Theorem 3.3. *A compound is* **RUP***-embeddable if and only if its block-cut tree is a caterpillar and each of its blocks is feasibly* **RUP***-embeddable such that the directed block-cut tree has a Eulerian dipath.*

3.5.2.4 The Algorithm for Compounds

In the **RUP** testing algorithm for closed digraphs in Sect. 3.5.1, we have assumed that we are given a routine `TestCompound`, which takes as input a compound $\gamma = (V, E)$ and two sets $V^{\mathtt{l}}, V^{\mathtt{r}} \subseteq V$. The output of this routine is a **RUP** embedding of γ such that all vertices in $V^{\mathtt{l}}$ are left- and all vertices in $V^{\mathtt{r}}$ are rightmost. If no such embedding exists, `TestCompound` returns $\perp$. Before we delve into the algorithm which solves this problem, we need to adjust Thm. 3.3 such that it incorporates the sets $V^{\mathtt{l}}$ and $V^{\mathtt{r}}$.

Corollary 3.19. *Let γ be a compound whose block-cut tree is a caterpillar and whose blocks are feasibly* **RUP***-embedded. Compound γ has a* **RUP** *embedding such that all vertices in $V^{\mathtt{l}}$ are left- and all vertices in $V^{\mathtt{r}}$ are rightmost if and only if the directed block-cut tree has a Eulerian dipath with the following properties: Denote by $B^{\mathtt{l}}$ and $B^{\mathtt{r}}$ the first and last block as visited by the Eulerian dipath, respectively. All vertices in $V^{\mathtt{l}}$ are leftmost in $B^{\mathtt{l}}$ and all vertices in $V^{\mathtt{r}}$ are rightmost in $B^{\mathtt{r}}$.*

Proof. If γ consists of only one block B, then B is the first and last block visited by the Eulerian dipath and B is **RUP**-embedded where all vertices $V^{\mathtt{l}}$ are left- and all vertices $V^{\mathtt{r}}$ are rightmost. In the following, we assume that γ contains at least one cut vertex.

$\Rightarrow$: By Cor. 3.13, the block-cut tree of γ^* is a path $B_1^*, f_{c_1}, \dots, f_{c_{k-1}}, B_k^*$. The source s_1 of B_1^* is the source s of γ^* and the sink t_k of B_k^* is the sink t of γ^*. Each vertex $v \in V^{\mathtt{l}}$ is leftmost in γ and, thus, incident to s. This implies that v is also incident to s_1 and, therefore, v is leftmost in B_1. Analogously, the vertices in $V^{\mathtt{r}}$ are rightmost in B_k. Further by Cor. 3.13, we obtain a block chain $B_1, \dots, B_k$ and by the proof of Lem. 3.20, we can construct a Eulerian dipath from this block chain which visits B_1 first and B_k last. Thus, $B_1 = B^{\mathtt{l}}$ and $B_k = B^{\mathtt{r}}$.

$\Leftarrow$: Let $B^{\mathtt{l}} = B_1, \dots, B_k = B^{\mathtt{r}}$ be the block chain as induced by the Eulerian dipath, i. e., the order in which the blocks are visited by the Eulerian dipath. By this block chain, we can construct a **RUP** embedding of γ according to the proof of Lem. 3.18 such that the block-cut tree of the dual is a path $(B^{\mathtt{l}})^*, \dots, (B^{\mathtt{r}})^*$ as in Cor. 3.13. Again, the source of $(B^{\mathtt{l}})^*$ is the source s of γ^* and the sink of $(B^{\mathtt{r}})^*$ is the sink t of γ^*. Hence, all vertices in $V^{\mathtt{l}}$ are leftmost and all vertices in $V^{\mathtt{r}}$ are rightmost in the embedding of γ. $\square$

Note that if, for instance, $V^{\mathtt{l}}$ is empty, Cor. 3.19 also applies. In this case, no particular block $B^{\mathtt{l}}$ must be the first in the block chain since no vertex must be leftmost. Likewise, if $V^{\mathtt{r}}$ is empty, no particular block must be the last. By Cor. 3.19, we immediately obtain the following corollary.

Corollary 3.20. *Let $\gamma = (V, E)$ be a* **RUP***-embedded compound such that all vertices in $V^{\mathtt{l}} \subseteq V$ ($V^{\mathtt{r}} \subseteq V$) are leftmost (rightmost). Furthermore, let $\mathcal{T}_B$ and $\vec{\mathcal{T}}_B$ be the block-cut tree of γ and its directed version, respectively. There exist blocks $B^{\mathtt{l}}$ and $B^{\mathtt{r}}$ with the following properties:*

(i) All vertices in $V^{\mathtt{l}}$ ($V^{\mathtt{r}}$) are leftmost (rightmost) in $B^{\mathtt{l}}$ ($B^{\mathtt{r}}$).

(ii) $\vec{\mathcal{T}}_B$ has a Eulerian dipath that visits $B^{\mathtt{l}}$ as first and $B^{\mathtt{r}}$ as last block.

(iii) Assume that γ contains at least one cut vertex. Both B^{l} and B^{r} are leaf blocks which contain cut vertices c_1 and c_ℓ, respectively, where c_1 and c_ℓ are the two ends of the spine of $\mathcal{T}_B$. Note that $c_1 = c_\ell$ is possible if γ contains only one cut vertex.

(iv) B^{l} contains all vertices in V^{l} and B^{r} contains all vertices in V^{r}.

Proof. (i) and (ii) follow directly from Cor. 3.19. Since the Eulerian dipath visits B^{l} before any other block, it must be a leaf block containing only cut vertex c_1. Likewise, B^{r} is a leaf block containing only c_ℓ. Thus, (iii) follows. Since all vertices in V^{l} are leftmost in B^{l}, it must also contain all vertices in V^{l} and the same holds for B^{r} and V^{r}, which implies (iv). □

Alg. 3.3 shows the routine `TestCompound`, which is our desired algorithm. Before we prove the correctness of `TestCompound`, we give a high-level description of the algorithm's strategy: `TestCompound` first computes the block-cut tree $\mathcal{T}_B$ of the compound and tests if it is a caterpillar. If it is no caterpillar, γ is not **RUP** and $\perp$ is returned. Otherwise, it calls the subroutine `ComputeDBCT&fRUPEmbeddings`, which stands for "compute directed block-cut tree and feasible **RUP** embeddings". `ComputeDBCT&fRUPEmbeddings` has three tasks: It tests for each block whether it has a feasible **RUP** embedding and based on these embeddings, it derives the corresponding directed block-cut tree $\vec{\mathcal{T}}_B$, and guarantees that $\vec{\mathcal{T}}_B$ has a Eulerian dipath. If one of the blocks has no feasible **RUP** embedding such that $\vec{\mathcal{T}}_B$ has a Eulerian dipath, `ComputeDBCT&fRUPEmbeddings` returns $\perp$. Back in `TestCompound`, traversing the Eulerian dipath of $\vec{\mathcal{T}}_B$ yields a block chain and, by iterating this block chain, the desired **RUP** embedding is obtained by a series of one-clique summations.

Both `TestCompound` and `ComputeDBCT&fRUPEmbeddings` use `TestBiconnected`. For a block $B_i = (V_i, E_i)$ and sets $V^{\mathrm{l}}, V^{\mathrm{r}} \subseteq V_i$, `TestBiconnected` returns a **RUP** embedding of B_i in time $\mathcal{O}(|V_i|)$ such that all vertices in V^{l} and V^{r} are left- and rightmost, respectively. If no such embedding exists, it returns $\perp$. For the moment, we use `TestBiconnected` as a black box as it is the topic of Sect. 3.5.3. We now prove that Alg. 3.3 really delivers what it promises:

Lemma 3.22. *`TestCompound` in Alg. 3.3 returns a **RUP** embedding of a compound $\gamma = (V, E)$ such that all vertices in $V^{\mathrm{l}} \subseteq V$ are left- and all vertices in $V^{\mathrm{r}} \subseteq V$ are rightmost or $\perp$ if no such embedding exists. Under the assumption that `TestBiconnected` has a linear running time, `TestCompound` runs in time $\mathcal{O}(|V|)$.*

Proof. The proof has two parts: In the first part, the correctness and running time of `Test-Compound` as shown in Alg. 3.3 is analyzed. The second part is devoted to the analysis of the subroutine `ComputeDBCT&fRUPEmbeddings` in Alg. 3.5.

TestCompound First, `TestCompound` calls `ComputeBlockCutTree` in line 1 to obtain the block-cut tree $\mathcal{T}_B = (\mathbf{B}, \mathbf{C}, \mathbf{E}_B)$. $\mathcal{T}_B$ can be computed in time $\mathcal{O}(|V| + |E|)$ by the algorithm in [HT73]. If the block-cut tree consists only of a single block, i. e., γ is biconnected, `Test-Biconnected` is called directly to test if a suitable **RUP** embedding of γ exists (line 2). The running time for this step is $\mathcal{O}(|V|)$ by assumption. Otherwise, the algorithm tests whether the block-cut tree $\mathcal{T}_B$ is a caterpillar (line 3). The test removes all leaves from the block-cut tree and tests if the remaining graph is a path, which takes $\mathcal{O}(|\mathbf{B}| + |\mathbf{C}| + |\mathbf{E}_B|) \subseteq \mathcal{O}(|V| + |E|)$ running time. If $\mathcal{T}_B$ is no caterpillar, then the compound cannot be **RUP** by Cor. 3.17 and $\perp$ is returned. In line 4, the spine of the caterpillar is traversed in any direction in $\mathcal{O}(|\mathbf{B}|) \subseteq \mathcal{O}(|V|)$ many steps to obtain the total order $c_1, \dots, c_\ell$ of the cut vertices. Note that the spine can be traversed in two direction yielding either the total order $c_1, \dots, c_\ell$ or its reversed version.

Algorithm 3.3. TestCompound

Input: planar compound $\gamma = (V, E)$, $V^{\mathtt{l}}, V^{\mathtt{r}} \subseteq V$

Output: **RUP** embedding of γ such that all vertices in $V^{\mathtt{l}}$ are leftmost and all vertices in $V^{\mathtt{r}}$ are rightmost; $\perp$ is returned if no such embedding exists

```
1  T_B = (B, C, E_B) ← ComputeBlockCutTree (γ)
2  if B = {B_1} then return TestBiconnected (B_1, V^l, V^r)
3  if T_B is no caterpillar then return ⊥
4  c_1, …, c_ℓ is the total order induced by the caterpillar T_B
5  if FindBlock(T_B, c_1, V^l) = ⊥ ∨ FindBlock(T_B, c_ℓ, V^r) = ⊥ then
6  |   c_1, …, c_ℓ ← Reverse (c_1, …, c_ℓ)
7  |   if FindBlock(T_B, c_1, V^l) = ⊥ ∨ FindBlock(T_B, c_ℓ, V^r) = ⊥ then return ⊥
8  (𝓔_i)_{i=1…k}, →T_B, ε_start, ε_end ← ComputeDBCT&fRUPEmbeddings(T_B, V^l, V^r, (c_1, …, c_ℓ))
9  if return value of ComputeDBCT&fRUPEmbeddings is ⊥ then return ⊥
10 if ε_start is a block B_i = (V_i, E_i) then
11 |   if V^l ⊆ V_i then B^l ← B_i
12 |   else return ⊥
13 else B^l ← FindBlock (T_B, c_1, V^l)
14 if ε_start is a block B_i = (V_i, E_i) then
15 |   if V^r ⊆ V_i then B^r ← B_i
16 |   else return ⊥
17 else B^r ← FindBlock (T_B, c_ℓ, V^r)
18 B_1, …, B_k ← block chain induced by Eulerian dipath starting at ε_start and ending at ε_end
   with B_1 = B^l and B_k = B^r
19 B̄_1 ← B_1
20 foreach block B_i = B_2, …, B_k do (see also proof of Lem. 3.18)
21 |   c_i ← cut vertex shared between B_{i-1} and B_i
22 |   t_{i-1} ← sink of B̄*_{i-1} according to its embedding 𝓔̄_{i-1}
23 |   s_i ← source of B*_i according to its embedding 𝓔_i
24 |   Compute the (c_i, t_{i-1})-(c_i, s_i)-one-clique sum of B̄_{i-1} and B_i:
25 |   B̄_i ← B̄_{i-1} ⊕ B_i
26 |   𝓔̄_i ← embedding of B̄_i
27 return embedding 𝓔̄_k of γ
```

According to properties (ii) and (iii) of Cor. 3.20, there must exist blocks $B^{\mathtt{l}}$ and $B^{\mathtt{r}}$ such that $B^{\mathtt{l}}$ contains c_1 and all vertices in $V^{\mathtt{l}}$, and $B^{\mathtt{r}}$ contains c_ℓ and all vertices in $V^{\mathtt{r}}$. In lines 5 to 7, the subroutine `FindBlock` is called to find $B^{\mathtt{l}}$ and $B^{\mathtt{r}}$. If the search is not successful, the total order of the cut vertices is reversed (see line 6) and the same test is executed again. If the second test also fails, the compound cannot be **RUP** by Cor. 3.20 and $\perp$ is returned.

`FindBlock` in Alg. 3.4 takes as input the block-cut tree $\mathcal{T}_B$, a cut vertex c, and a set of vertices V'. Note that $c \in V'$ is possible. It returns a leaf block containing c and V' or $\perp$ if no such leaf block exists. For each leaf block $B_i = (V_i, E_i)$ which contains c, `FindBlock` initializes a counter K to $|V'|$ and decrements K by 1 for each vertex in V_i that is also in V'. If K reaches 0, all vertices in V' are in V_i, and B_i is returned. Otherwise, if K never reaches 0, $\perp$ is returned. For each non-cut vertex, its membership to V' is tested at most once. For c its membership to V' is tested at most $|\mathbf{B}_c|$ many times where $\mathbf{B}_c$ is the set of leaf blocks that contain c. For $\mathbf{B}_c$, we get $|\mathbf{B}_c| \in \mathcal{O}(|V|)$ and, thus, the overall running time of `FindBlock` is in $\mathcal{O}(|V|)$ and so is the running time of lines 5 to 7 in `TestCompound`. Note that for the case $V' = \emptyset$, `FindBlock` simply returns a leaf block that contains c.

Algorithm 3.4. `FindBlock`

Input: block-cut tree $\mathcal{T}_B = (\mathbf{B}, \mathbf{C}, \mathbf{E}_B)$, cut vertex $c \in \mathbf{C}$, and set of vertices V'
Output: leaf block containing c and V'; $\perp$ if no such block exists

1 **foreach** leaf block $B_i = (V_i, E_i)$ containing c **do**
2     $K \leftarrow V'$
3   **foreach** $v \in V_i$ **do**
4   **if** $v \in V'$ **then** $K \leftarrow K - 1$
5   **if** $K = 0$ **then return** B_i
6 **return** $\perp$

In line 8, the subroutine `ComputeDBCT&fRUPEmbeddings` is called which is analyzed in detail later. For the moment is suffices to know that `ComputeDBCT&fRUPEmbeddings` takes as input the block-cut tree $\mathcal{T}_B$, the sets $V^{\mathtt{l}}$ and $V^{\mathtt{r}}$, and the total order of cut vertices $c_1, \ldots, c_\ell$. It returns a feasible **RUP** embedding $\mathcal{E}_i$ for each block B_i, the respective directed block-cut tree $\vec{\mathcal{T}}_B = (\mathbf{B}, \mathbf{C}, \vec{\mathbf{E}}_B)$, and the start $\varepsilon_{\text{start}}$ and end ε_{end} of the Eulerian dipath in $\vec{\mathcal{T}}_B$. `ComputeDBCT&fRUPEmbeddings` also guarantees that in the **RUP** embedding of each leaf block $B_i = (V_i, E_i)$, all vertices $V_i \cap V^{\mathtt{l}}$ and $V_i \cap V^{\mathtt{r}}$ are left- and rightmost, respectively. In particular, all vertices $V^{\mathtt{l}}$ ($V^{\mathtt{r}}$) are leftmost in the block that is visited first (last) by the Eulerian dipath. If `ComputeDBCT&fRUPEmbeddings` either finds no suitable **RUP** embeddings for the blocks or the directed block-cut tree has no Eulerian dipath, it returns $\perp$. The start $\varepsilon_{\text{start}}$ of the Eulerian dipath is either c_1 or a block B_i which contains c_1. In the latter case, c_1 is right- but not leftmost in B_i, i. e., $(B_i, c_1) \in \vec{\mathbf{E}}_B$ and $(c_1, B) \notin \vec{\mathbf{E}}_B$. Likewise, the end ε_{end} of the Eulerian dipath is either c_ℓ or a block B_i in which c_ℓ is left- but not rightmost.

In lines 10 to 17, two blocks $B^{\mathtt{l}}$ and $B^{\mathtt{r}}$ are determined that fulfill the properties as listed in Cor. 3.20. If $\varepsilon_{\text{start}}$ is a block B_i, then it is the start of the Eulerian dipath and, hence, must contain all vertices in $V^{\mathtt{l}}$. In this case, $B^{\mathtt{l}}$ is set to $\varepsilon_{\text{start}}$. Otherwise if $V^{\mathtt{l}} \not\subseteq V_i$, $\perp$ is returned. If $\varepsilon_{\text{start}}$ is c_1, then the Eulerian dipath starts at c_1 and $B^{\mathtt{l}}$ is set to any block which contains c_1 and all vertices in $V^{\mathtt{l}}$ by calling `FindBlock`. Note that $B^{\mathtt{l}}$ must exist by the tests in lines 5 to 7. Likewise, we obtain $B^{\mathtt{r}}$. The running time of lines 10 to 17 is $\mathcal{O}(|V|)$.

In line 18, the Eulerian dipath from $\varepsilon_{\text{start}}$ to ε_{end} is traversed such that $B^{\mathtt{l}}$ is the first and $B^{\mathtt{r}}$ the last visited block. Similar to the proof of Lem. 3.20, the Eulerian dipath p is canonically obtained as follows (see also Fig. 3.27): If $\varepsilon_{\text{start}} = B^{\mathtt{l}}$, p first goes to c_1. Otherwise, if $\varepsilon_{\text{start}} = c_1$, p first visits leaf block $B^{\mathtt{l}}$. Then, p traverses all remaining leaf blocks containing c_1 and proceeds to the spine block containing c_1 and c_2. At c_2, p visits all leaf blocks containing c_2 and then proceeds to the next spine block, and so forth. When reaching c_ℓ, $B^{\mathtt{r}}$ must be the last leaf block that is visited by p. If $\varepsilon_{\text{end}} = B^{\mathtt{r}}$, the Eulerian dipath ends at $B^{\mathtt{r}}$. Otherwise, $\varepsilon_{\text{end}} = c_\ell$ and the Eulerian dipath visits $B^{\mathtt{r}}$ before ending at c_ℓ. As in the proof of Lem. 3.21, the block chain $B_1, \dots, B_k$ is obtained from the Eulerian dipath, where $B_1 = B^{\mathtt{l}}$ and $B_k = B^{\mathtt{r}}$. Altogether, line 18 takes $\mathcal{O}(|\mathbf{B}| + |\mathbf{C}| + |\mathbf{E}_B|) \subseteq \mathcal{O}(|V| + |E|)$ running time.

Finally in lines 20 to 26, the desired **RUP** embedding of γ is constructed as in the proof of Lem. 3.18. Starting with the embedding of B_1, the embeddings of the blocks $B_2, \dots, B_k$ are subsequently merged by a series of one-clique summations. For each one-clique summation, the faces at which the cut vertices are merged have to be obtained, i. e., for each block $B_i = (V_i, E_i)$ its dual B_i^* has to be computed which takes $\mathcal{O}(|V_i| + |E_i|)$ running time. The one-clique summation itself merges the respective cut vertices and adapts the rotation systems. Altogether, we obtain a running time of $\mathcal{O}(|V| + |E|)$ for lines 20 to 26.

The so obtained embedding, denoted by $\overline{\mathcal{E}}_k$, fulfills all properties of Cor. 3.19. First and foremost, $\overline{\mathcal{E}}_k$ is indeed a **RUP** embedding by Lem. 3.21. Second, all vertices in $V^{\mathtt{l}}$ are leftmost and all vertices in $V^{\mathtt{r}}$ are rightmost which follows from Cor. 3.19.

Under the assumption that `TestBiconnected` and `ComputeDBCT&fRUPEmbeddings` have a linear running time, all steps of `TestCompound` have a running time of either $\mathcal{O}(|V|)$ or $\mathcal{O}(|V| + |E|)$. Consequently, the overall running time is $\mathcal{O}(|V| + |E|)$ and, hence, $\mathcal{O}(|V|)$ since γ is planar.

ComputeDBCT&fRUPEmbeddings `ComputeDBCT&fRUPEmbeddings` is shown in Alg. 3.5. The directed block-cut tree $\vec{\mathcal{T}}_B$ is derived by testing the feasible **RUP**-embeddability of each block. The respective embedding of block B_i is stored in $\mathcal{E}_i$. $\vec{\mathcal{T}}_B = (\mathbf{B}, \mathbf{C}, \vec{\mathbf{E}}_B)$ is initialized with $\vec{\mathbf{E}}_B = \emptyset$. Simultaneously, `ComputeDBCT&fRUPEmbeddings` stores the start $\varepsilon_{\text{start}}$ and the end ε_{end} of the Eulerian dipath with initial values $\varepsilon_{\text{start}} = c_1$ and $\varepsilon_{\text{end}} = c_\ell$. For reasons that are described below, we also need a set $\mathbf{B}_{\text{defer}}$ which is initialized to an empty set in line 3. The loop starting in line 4 processes all blocks $B_i = (V_i, E_i)$ and for each two cases are distinguished: either B_i is a leaf block or it is a spine block.

In the first case, B_i contains exactly one cut vertex c_j. Let $V_i^{\mathtt{l}} = V_i \cap V^{\mathtt{l}}$ and $V_i^{\mathtt{r}} = V_i \cap V^{\mathtt{r}}$ according to line 6. In line 7, `TestBiconnected` is called to test whether B_i has a **RUP** embedding such all vertices in $V_i^{\mathtt{l}} \cup \{c_j\}$ are left- and all vertices in $V_i^{\mathtt{r}} \cup \{c_j\}$ are rightmost. In particular, c_j is both left- and rightmost. If B_i has such a **RUP** embedding, the antiparallel pair of edges (B_i, c_j) and (c_j, B_i) is introduced to $\vec{\mathcal{T}}_B$. Otherwise, B_i must either be the beginning or the end of the Eulerian dipath for γ to be **RUP**. In case $c_j = c_1$ (line 10), `TestBiconnected` is used to test whether B_i has a **RUP** embedding $\mathcal{E}_i^{\text{start}}$ in which c_1 is rightmost. If this is the case, B_i is a candidate for the beginning of the Eulerian dipath. Likewise, if $c_j = c_\ell$, `TestBiconnected` is used to find a **RUP** embedding $\mathcal{E}_i^{\text{end}}$ of B_i where c_ℓ is leftmost and B_i is a candidate for the end of the Eulerian dipath. If B_i does not contain c_1 (c_ℓ) or does not have the respective **RUP** embedding, $\mathcal{E}_i^{\text{start}}$ ($\mathcal{E}_i^{\text{end}}$) is set to $\perp$. Note that if $c_j \neq c_1$ and $c_j \neq c_\ell$ both embeddings are set to $\perp$. In line 14, four cases are distinguished: neither of $\mathcal{E}_i^{\text{start}}$ and $\mathcal{E}_i^{\text{end}}$ is $\perp$, exactly one is $\perp$, or both are $\perp$. The first case (line 15) occurs only if the compound

Algorithm 3.5. `ComputeDBCT&fRUPEmbeddings`

Input: block-cut tree $\mathcal{T}_B = (\mathbf{B}, \mathbf{C}, \mathbf{E}_B)$ of planar compound $\gamma = (V, E)$ which is a caterpillar, total order $c_1, \dots, c_\ell$ of cut vertices, and $V^{\mathtt{l}}, V^{\mathtt{r}} \subseteq V$

Output: **RUP** embedding of each leaf block $B_i = (V_i, E_i)$ such that all vertices in $V^{\mathtt{l}} \cap V_i$ ($V^{\mathtt{r}} \cap V_i$) are leftmost (rightmost), feasible **RUP** embedding of each spine block such that its cut vertices are left-/rightmost, directed block-cut tree $\vec{\mathcal{T}}_B$ which has a Eulerian dipath, and start $\varepsilon_{\text{start}}$ and end ε_{end} of Eulerian dipath; or $\perp$ if no such embeddings of the blocks exist

```
1  Initialize T⃗_B = (B, C, E⃗_B) with E⃗_B = ∅
2  ε_start ← c_1; ε_end ← c_ℓ
3  B_defer ← ∅
4  foreach block B_i = (V_i, E_i) do
5      if B_i is a leaf block with cut vertex c_j then
6          V_i^l ← V_i ∩ V^l; V_i^r ← V_i ∩ V^r
7          ℰ_i ← TestBiconnected(B_i, V_i^l ∪ {c_j}, V_i^r ∪ {c_j})
8          if ℰ_i ≠ ⊥ then E⃗_B ← E⃗_B ∪ {(c_j, B_i), (B_i, c_j)}
9          else B_i has to be either the beginning or the end of the Eulerian dipath
10             if c_j = c_1 then ℰ_i^start ← TestBiconnected(B_i, V_i^l, V_i^r ∪ {c_j})
11             else ℰ_i^start ← ⊥
12             if c_j = c_ℓ then ℰ_i^end ← TestBiconnected(B_i, V_i^l ∪ {c_j}, (V_i^r))
13             else ℰ_i^end ← ⊥
14             switch values of ℰ_i^start and ℰ_i^end do
15                 case both are ≠ ⊥: B_defer ← B_defer ∪ {B_i}
16                 case ℰ_i^start ≠ ⊥ ∧ ℰ_i^end = ⊥
17                     if ε_start = c_1 then ε_start ← B_i; E⃗_B ← E⃗_B ∪ {(B_i, c_j)}
18                     else return ⊥
19                 case ℰ_i^start = ⊥ ∧ ℰ_i^end ≠ ⊥
20                     if ε_end = c_ℓ then ε_end ← B_i; E⃗_B ← E⃗_B ∪ {(c_j, B_i)}
21                     else return ⊥:
22                 case both are ⊥ return ⊥
23     else B_i is a spine block with cut vertices c_j, c_{j+1}
24         ℰ_i ← TestBiconnected(B_i, {c_j}, {c_{j+1}})
25         if ℰ_i ≠ ⊥ then E⃗_B ← E⃗_B ∪ {(c_j, B_i), (B_j, c_{j+1})}
26         else return ⊥
27 while B_defer ≠ ∅ do
28     B_i ← remove one element from B_defer
29     if ε_start = c_1 then ε_start ← B_i; E⃗_B ← E⃗_B ∪ {(B_i, c_j)}; ℰ_i ← ℰ_i^start
30     else if ε_end = c_ℓ then ε_end ← B_i; E⃗_B ← E⃗_B ∪ {(c_j, B_i)}; ℰ_i ← ℰ_i^end
31     else return ⊥
32 return RUP embeddings ℰ_i for all blocks B_i, T⃗_B, ε_start, ε_end
```

contains exactly one cut vertex $c_1 = c_\ell = c_j$. Then, the decision whether B_i is the beginning or the end of the Eulerian dipath has to be deferred as there may be another block B_p with $p \neq i$ that must either be the beginning or the end of the Eulerian dipath. For this, we maintain the set $\mathbf{B}_{\text{defer}}$ into which B_i is inserted. In the second case (line 16), $c_j = c_1$ and B_i must be the beginning of the Eulerian dipath as it only has a **RUP** embedding where c_j is rightmost. If there is no other block that already is the beginning of the Eulerian dipath ($\varepsilon_{\text{start}} = c_1$), $\varepsilon_{\text{start}}$ is set to B_i and edge (B_i, c_j) is introduced to $\vec{\mathcal{T}}_B$. Otherwise, $\perp$ is returned as the compound cannot be **RUP** (Cor. 3.20). Line 19 corresponds to the symmetric case with $c_j = c_\ell$ and B_i must be the end of the Eulerian dipath. If B_i has no suitable **RUP** embedding (line 22), $\perp$ must be returned (Thm. 3.3).

In the second case, B_i is a spine block containing cut vertices c_j and c_{j+1}. In line 24, `TestBiconnected` is called. If it succeeds, it returns a **RUP** embedding of B_i where c_j is left- and c_{j+1} is rightmost and edges (c_j, B_i) and (B_i, c_{j+1}) are added to $\vec{\mathcal{T}}_B$. If `TestBiconnected` returns $\perp$, then (c_j, B_i) and (B_i, c_{j+1}) cannot be introduced to the directed block-cut tree which in turn cannot have a Eulerian dipath and $\perp$ is returned.

Note that the sets V^{l} and V^{r} are irrelevant for a spine block: If a spine block contains a vertex from, say, V^{l}, this vertex must be a cut vertex since by lines 5 to 7 in Alg. 3.3 it is already guaranteed that all vertices in V^{l} are in a leaf block B^{l}. Denote by c this cut vertex in $V_i \cap V^{\text{l}}$. The case $c = c_j$ is already covered by the test in line 24. Otherwise, $c = c_{j+1}$. In this case, c_{j+1} must be in leaf block B^{l} as well. However, B^{l} contains only cut vertex c_1 and, hence, $c_1 = c_{j+1}$, which is a contradiction since $j + 1 > 1$. The same reasoning applies for V^{r}.

The loop starting in line 27, subsequently processes the blocks in $\mathbf{B}_{\text{defer}}$. Remember that $\mathbf{B}_{\text{defer}}$ contains the blocks that can be both the beginning and the end of the Eulerian dipath and this can only happen if the compound has only one cut vertex $c_1 = c_\ell$. For each block $B_i \in \mathbf{B}_{\text{defer}}$, `ComputeDBCT&fRUPEmbeddings` tests whether B_i can be the beginning or the end of the Eulerian dipath, that is, $\varepsilon_{\text{start}}$ or ε_{end} still has its initial value of c_1 or c_ℓ, respectively. B_i is set to the "free position" and, if both are already occupied, $\perp$ is returned.

After successfully processing all blocks, the directed block-cut tree has a Eulerian dipath which starts at $\varepsilon_{\text{start}}$ and ends at ε_{end}. Each leaf block B_i is connected to its cut vertex via an antiparallel pair of edges with the following two exceptions. If $\varepsilon_{\text{start}}$ is a block B_i, then $(B_i, c_1) \in \vec{\mathbf{E}}_B$ and the Eulerian dipath starts at B_i. Likewise, if ε_{end} is a block $B_{i'}$, $(c_\ell, B_{i'}) \in \vec{\mathbf{E}}_B$ and B_i is the end of the Eulerian dipath. For the embedding $\mathcal{E}_i$, the respective embedding $\mathcal{E}_i^{\text{start}}$ or $\mathcal{E}_i^{\text{end}}$ is chosen. Furthermore, for each spine block containing cut vertices c_j and c_{j+1}, $(c_j, B_i) \in \vec{\mathbf{E}}_B$ and $(B_i, c_{j+1}) \in \vec{\mathbf{E}}_B$.

The running time of `ComputeDBCT&fRUPEmbeddings` is $\mathcal{O}(|V| + |E|)$: For each block, `TestBiconnected` is called at most three times. All other operations can be implemented in linear time, e. g., adding edges to the directed block-cut tree. Under the assumption that `TestBiconnected` runs in time $\mathcal{O}(|V_i| + |E_i|)$ for each block $B_i = (V_i, E_i)$, the running time of `ComputeDBCT&fRUPEmbeddings` is in $\mathcal{O}(|V| + |E|)$. ☐

3.5.3 Biconnected Compounds

The last piece missing in our **RUP** testing algorithm deals with biconnected compounds and, thus, we assume in the following that all graphs and digraphs are biconnected unless stated otherwise. It turns out that this part of the algorithm is the most involved one. A neat tool to study biconnected graphs are SPQR trees as introduced by Di Battista and Tamassia [DBT96]. In their original form, SPQR trees deal with undirected graphs and, in the case of embedded

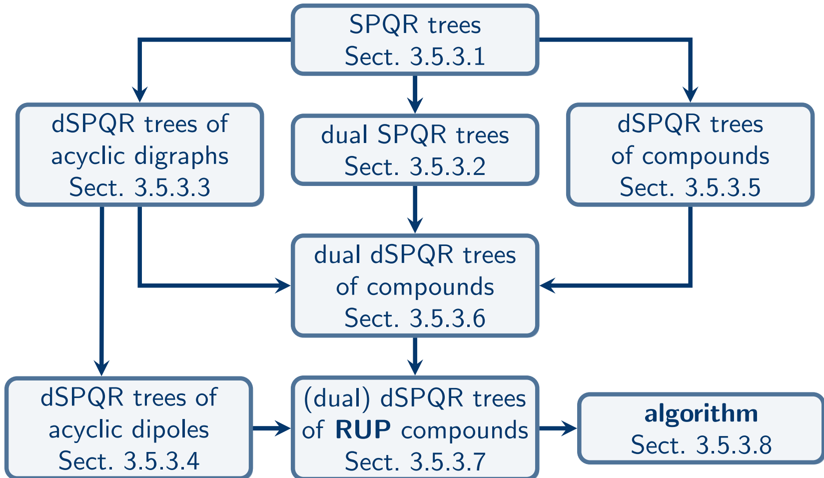


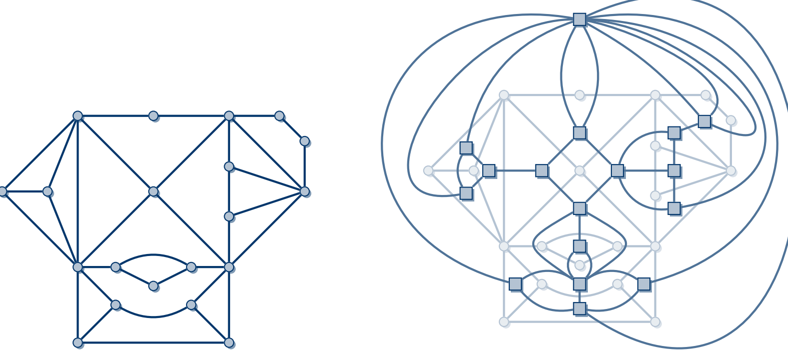
Figure 3.31: Overview of Sect. 3.5.3.

graphs, with the primal graph only. Thus, before we can devise our algorithm, we need to equip SPQR trees with capabilities to deal with digraphs and duals. In order to do this, we introduce several notions of SPQR trees and combine them in various ways. To avoid getting lost in this jungle of various notions of SPQR trees, we can consult Fig. 3.31, which shows a chart with an overview of this section. We start with a primer on SPQR trees (Sect. 3.5.3.1). Then, we introduce dual SPQR trees to study the SPQR trees of dual graphs (Sect. 3.5.3.2). By introducing edge directions to SPQR trees, we obtain *directed SPQR trees* or *dSPQR trees* for short. We do this in two steps: First, we consider acyclic digraphs (Sect. 3.5.3.3) and characterize dSPQR trees of acyclic dipoles (Sect. 3.5.3.4). Afterwards, we define dSPQR trees of compounds (Sect. 3.5.3.5). By combining dual SPQR trees with dSPQR trees of acyclic digraphs and of compounds, we obtain dual dSPQR trees of compounds (Sect. 3.5.3.6) and a characterization of biconnected **RUP** compounds by means of their (dual) dSPQR trees (Sect. 3.5.3.7). In particular, we see how to decide whether or not a compound is **RUP** and, if it is **RUP**, how to obtain a **RUP** embedding by using the compound's dSPQR tree and its dual. This leads to our algorithm in Sect. 3.5.3.8.

3.5.3.1 A Primer on SPQR Trees

We start with an introduction to SPQR trees as given by Di Battista and Tamassia [DBT96]. In a nutshell, SPQR trees are to split pairs and triconnected components what block-cut trees are to cut vertices and blocks. Remember that a split pair is a pair of vertices whose removal disconnects the graph and a graph is triconnected if it contains no split pair. The triconnected components of a biconnected graph are interconnected via split pairs. Just like a block-cut tree is a high-level description of the blocks and cut vertices, an SPQR tree describes the interrelationship of the triconnected components and the split pairs.

Let G be a biconnected. As an example, we consider the graph displayed in Fig. 3.32(a) and its SPQR tree in Fig. 3.33. The SPQR tree $\mathcal{T}$ of G is an unrooted tree and its vertices are called *nodes* for which we use the symbol μ. A node is depicted by a rectangle in Fig. 3.33. Associated with each node μ is a graph that is homeomorphic to a subgraph of G called the *skeleton* $\mathrm{skel}(\mu)$, which is displayed within the node's rectangle. "Homeomorphic to a subgraph



(a) Biconnected and embedded graph.

(b) Dual of the graph in Fig. 3.32(a).

Figure 3.32: An embedded biconnected graph with its dual.

of G" means that the edges of a skeleton can be replaced by paths connecting their endpoints to obtain a graph that is isomorphic to a subgraph of G. There are four different types of nodes and each type has a different type of skeleton:

- An *S node* represents a series composition of split pairs and its skeleton is a circle of length at least three, e. g., node μ_4.
- A *P node* represents a parallel composition of a single split pair and its skeleton consists of this pair connected by three or more multiple edges, e. g., node μ_3.
- A *Q node* stands for a single edge $e = \{u, v\}$ and its skeleton consists of u and v connected by two parallel edges, e. g., node μ_5. For reasons described later, we usually neglect Q nodes.
- An *R node* corresponds to a triconnected component and its skeleton is a triconnected graph with no multiple edges, i. e., it is simple. Nodes μ_1 and μ_2 are R nodes.

In its original definition, every edge of a skeleton, except for one of a Q node, is a *virtual edge* (displayed by a bold line), e. g., edge $\{u, v\}$ in node μ_1. The pair u, v is a split pair of G and the virtual edge $\{u, v\}$ in μ_1 represents the subgraph which is disconnected from node μ_1 if u and v are removed. In this sense, a virtual edge stands for a whole subgraph. In general, the subgraph corresponding to a skeleton's virtual edge in a node μ is referred to as the *expansion graph* $\text{expg}_\mu(e)$ of e in μ. The expansion graph $\text{expg}_{\mu_1}(\{u, v\})$ is displayed in the box located in the top left corner of Fig. 3.33 and it corresponds to the nodes μ_2, μ_3, and μ_4. In fact, we can reconstruct the subgraph $\text{expg}_{\mu_1}(e)$ by adequately merging the nodes μ_2, μ_3, and μ_4 as we will see later. For every virtual edge e in the skeleton of a node μ, there is another node μ' that *refines* the structure of $\text{skel}(\mu)$. This link is represented by an edge between μ and μ' in $\mathcal{T}$ (dashed line) and it connects two virtual edges whose endpoints are the same. For instance, edge $\{u, v\}$ in node μ_1 is refined by node μ_2. This relationship is symmetric as node μ_2 also contains the virtual edge between u and v which is refined by node μ_1. As stated before, all edges of a skeleton, except for a single edge in a Q node, are virtual. In particular,

in Fig. 3.33, all non-bold edges are actually virtual edges refined by Q nodes, except for the non-bold edge in the skeleton of node μ_5, which is the only Q node displayed for illustration purposes. For the sake of convenience, we represent edges of the graph directly in the skeletons instead of attaching them to Q nodes via virtual edges and omit Q nodes in our representation of SPQR trees unless stated otherwise.

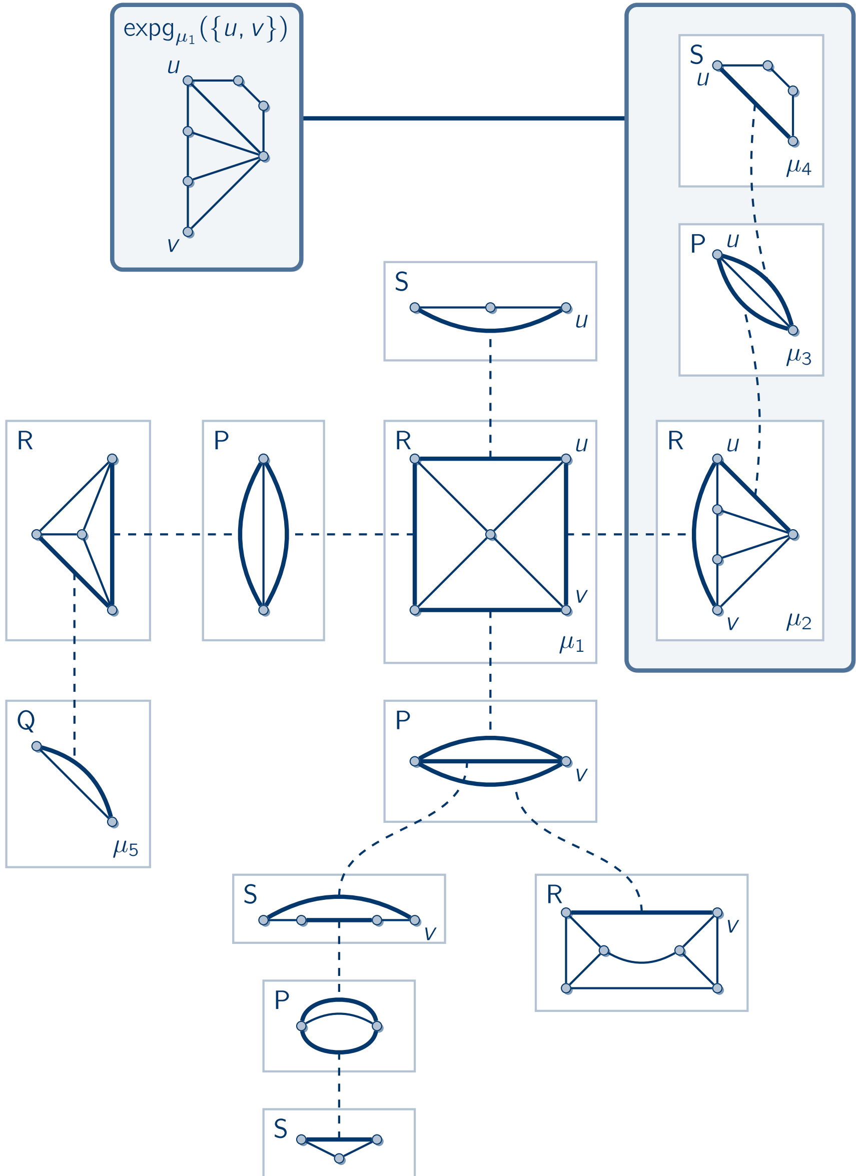


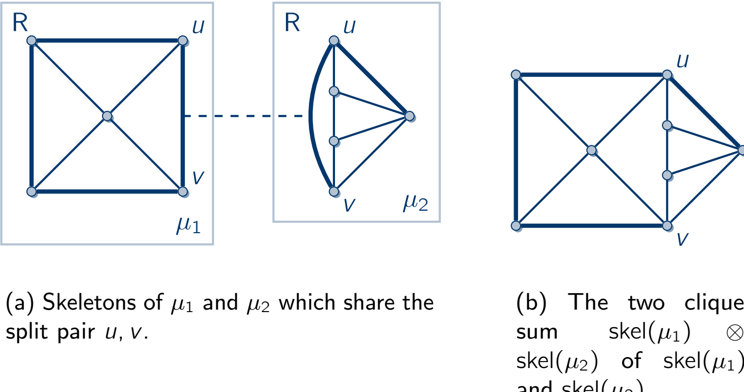
Figure 3.33: The SPQR tree of the embedded graph in Fig. 3.32(a) where only one Q node is shown for illustration purposes.

SPQR trees are minimal in the sense that no two P nodes are adjacent as they can be merged into a single P node. Likewise, no two S nodes are adjacent. In contrast, two R nodes can be linked together if they share a split pair, e. g., nodes μ_1 and μ_2. The minimality also

implies that the SPQR tree of a graph is unique. The following lemma from [DBT96] limits the size of an SPQR tree with respect to the size of the original graph.

Proposition 3.5 ([DBT96]). *The SPQR tree $\mathcal{T}$ of a biconnected graph $G = (V, E)$ has $|E|$ Q nodes and the total number of S, P, and R nodes is in $\mathcal{O}(|V|)$. The total number of edges in all skeletons is in $\mathcal{O}(|E|)$.*

Two-Clique Summation Let μ be a node and $e = \{u, v\}$ be a virtual edge of μ's skeleton that is refined by another node μ'. The vertices u and v are part of both skeletons skel(μ) and skel(μ'). Consider, for instance, virtual edge $\{u, v\}$ in node μ_1 and its refinement in node μ_2 as depicted in Fig. 3.34(a). We can merge the skeletons skel(μ_1) and skel(μ_2) at their common vertices u and v by an operation called *two-clique summation*. The result of this operation is shown in Fig. 3.34(b). In the *two-clique sum* of the skeletons of μ_1 and μ_2, denoted by skel(μ_1) $\otimes$ skel(μ_2), the vertices u and v are identified and the virtual edge $\{u, v\}$ is removed from both skeletons. Note that u, v is a split pair in skel(μ_1) $\otimes$ skel(μ_2).



(a) Skeletons of μ_1 and μ_2 which share the split pair u, v.

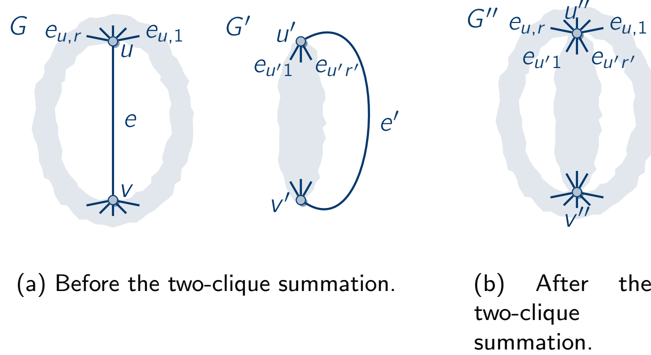
(b) The two clique sum skel(μ_1) $\otimes$ skel(μ_2) of skel(μ_1) and skel(μ_2).

Figure 3.34: Edge $\{u, v\}$ in μ_1 is refined by μ_2 and their skeletons can be merged by a two-clique summation.

The two-clique summation is similar to the one-clique summation we used for block-cut trees in Sect. 3.5.2.1. Only this time, two two-cliques (two pairs of adjacent vertices) are merged instead of two single vertices. Remember, by a block-cut tree we are able to reconstruct the whole graph by a series of one-clique summations. The same is true for SPQR trees. Given an SPQR tree of a graph G, its nodes can be totally ordered according to $\mu_1, \dots, \mu_k$ such that each μ_i $(1 < i \leq k)$ refines exactly one virtual edge from one of the skeletons of nodes $\mu_1, \dots, \mu_{i-1}$, i. e., in the SPQR tree, node μ_i is connected to one of $\mu_1, \dots, \mu_{i-1}$. We call this order *node sequence* and it can be obtained by a depth-first traversal of the SPQR tree. Based on this total order, we define a sequence of graphs $\overline{G}_i$ with $\overline{G}_i = \overline{G}_{i-1} \otimes$ skel(μ_i) for $1 < i \leq k$ and $\overline{G}_1 =$ skel(μ_1), where $\overline{G}_{i-1} \otimes$ skel(μ_i) is the two-clique sum of $\overline{G}_{i-1}$ and μ_i's skeleton. Note that μ_i is a refinement of a virtual edge $e = \{u, v\}$ in a skeleton of one of $\mu_1, \dots, \mu_{i-1}$ and, hence, u and v are in $\overline{G}_{i-1}$ and μ_i. Merging all nodes of the SPQR tree results in G, i. e., $\overline{G}_k = G$. As with block series, we call the sequence $\overline{G}_1, \dots, \overline{G}_k$ *node series* of G and $\mathcal{T}$.

Two-clique summations can also be used to define the expansion graph of a virtual edge. For this, consider again the virtual edge $\{u, v\}$ in node μ_1 in Fig. 3.32(a). The expansion graph $\mathrm{expg}_{\mu_1}(\{u, v\})$ is the two-clique sum of the skeletons of nodes μ_2, μ_3, and μ_4, where virtual edge $\{u, v\}$ in $\mathrm{skel}(\mu_2)$ is removed.

SPQR Trees of Planar Graphs SPQR trees have proven especially useful in the context of planarity testing. If the underlying graph is planar, its SPQR tree can be used to maintain and store all of its embeddings. Especially, a graph is planar if and only if each of the skeletons of its SPQR tree is planar [DBT96]. Assume that we are given an SPQR tree of a planar graph. The skeleton of an R node is triconnected and simple. Hence, its embedding is unique up to inversion due to Whitney's theorem [Whi33] (cf. Sect. 1.1.7). In the context of SPQR trees, the inversion of an R node's embedding is called a *flip*. The embedding of a P node's skeleton is uniquely defined by the rotation system of either of its two vertices: Let u and v be the two vertices of a P node connected by the edges $e_1, \dots, e_r$ ordered according to the rotation system of u. Due to planarity, the rotation system of v is $e_r, \dots, e_1$, i. e., the inverted rotation system of u. Exchanging the positions of two edges in the skeleton of a P node is called a *swap*. Finally, the embedding of the skeleton of an S node, which is a circle, is always unique as each vertex is of degree two.



(a) Before the two-clique summation.

(b) After the two-clique summation.

Figure 3.35: A two-clique summation.

Given an embedding of each skeleton, we can obtain an embedding of the whole graph by two-clique summations according to a node series where we have to incorporate the skeletons' embeddings. Suppose we are given two embedded graphs G and G' and let $e = \{u, v\}$ and $e' = \{u', v'\}$ be two edges in G and G', respectively. The situation is shown in Fig. 3.35(a). The rotation system of u is $e, e_{u,1}, \dots, e_{u,r}$ and the rotation system of u' is $e', e_{u'1}, \dots, e_{u'r'}$. The rotation systems of v and v' are ignored for the moment. In the two clique-sum of G and G', vertices u and u' are identified and so are v and v', where the first pair is replaced by u'' and the latter by v''. The resulting graph is denoted by $G'' = G \otimes G'$ and is called (e, e')-two-clique sum (see Fig. 3.35(b)). The rotation systems of u and u' are split at e and e', respectively, and are then concatenated to obtain the rotation system for u'' which is $e_{u,1}, \dots, e_{u,r}, e_{u'1}, \dots, e_{u'r'}$. The rotation system of v'' is obtained analogously from the rotation systems of v and v'.

Given two edges $e = \{u, v\}$ and $e' = \{u', v'\}$ for a two-clique summation, we can either identify u with u' and v with v', or u with v' and v with u'. However, in the following it is

clear from the context which vertices are identified. Recall that with one-clique summations, we needed to define at which faces the cut vertices are merged (see Sect. 3.5.2.1). In the case of two-clique summations, the face is uniquely defined by the two involved edges e and e' and their positions within the rotation system.

Given a node series of an SPQR tree and an embeddings for each skeleton, an embedding of the whole graph can be obtained by subsequently applying two-clique-summations. For instance, the embeddings of the skeletons in Fig. 3.33 result in the embedding shown in Fig. 3.32(a). By applying swaps to P nodes and flips to R nodes beforehand, we can obtain any possible embedding of a planar graph [DBT96]. Thereby, the SPQR tree implicitly stores all possible embeddings. Note that we neglect the outer face here, i. e., the embedding is determined up to the outer face. Conversely, an embedding of a biconnected graph G implies embeddings of all skeletons in its SPQR tree, that is, exactly those embeddings of the skeletons which result in the original embedding of the whole graph.

3.5.3.2 Dual SPQR Trees

In our studies of **RUP** graphs, dual graphs play an important role. In the following, we combine SPQR trees with dual graphs to define dual SPQR trees. Let G be an embedded graph and let $\mathcal{T}$ be its SPQR tree, where the skeletons of $\mathcal{T}$ are embedded according to G's embedding. From the embedding of each skeleton $\text{skel}(\mu)$ we obtain its dual $\text{skel}(\mu)^*$ called the *dual skeleton*. Dual SPQR trees are defined as follows.

Definition 3.5 (Dual SPQR Tree). *Given the SPQR tree $\mathcal{T}$ of a biconnected and embedded graph, the dual SPQR tree $\mathcal{T}^*$ is defined as follows. For each node μ in $\mathcal{T}$, $\mathcal{T}^*$ contains node μ^*, which is called the* dual node *of μ, with the following properties:*

- *The skeleton $\text{skel}(\mu^*)$ of μ^* is the dual skeleton $\text{skel}(\mu)^*$ of μ.*
- *An edge e^* of $\text{skel}(\mu^*)$ is virtual if its primal edge e^* is virtual in μ.*
- *If the virtual edge $e = \{u, v\}$ in node μ is refined by μ' in $\mathcal{T}$, then the dual of e in μ^* is refined by μ'^* in $\mathcal{T}^*$.*
- *If the type of μ is X for $X \in \{\text{S}, \text{P}, \text{Q}, \text{R}\}$ in $\mathcal{T}$, then the type of μ^* is X^* in $\mathcal{T}^*$.*

The dual SPQR tree of the SPQR tree in Fig. 3.33 is shown in Fig. 3.36. Note that the dual SPQR tree inherits the overall structure of the (primal) SPQR tree and the skeletons are replaced by their duals. In particular, the dual SPQR tree adopts the links between the virtual edges of the nodes.

In the remainder of this section, we show that the dual SPQR tree $\mathcal{T}^*$ of G is the SPQR tree of the dual G^*. The dual graph of our running example is shown in Fig. 3.32(b) on page 160 and its SPQR tree is indeed the one displayed in Fig. 3.36, where only the node types need some adaptation. For instance, node μ_3 in Fig. 3.33 is a P node and its dual μ_3^* in Fig. 3.36 is an S node. In general, S and P nodes swap their roles when going from the primal to the dual. In contrast, Q nodes stay unchanged and the same holds for R nodes. Recall that the skeleton of an R node is a triconnected and simple graph. The dual of a triconnected and simple graph is also triconnected due to [MT01, Thm. 2.6.7]. The dual is also simple since any multiple edges in the dual would imply vertices of degree two in the primal, which would then not be triconnected.

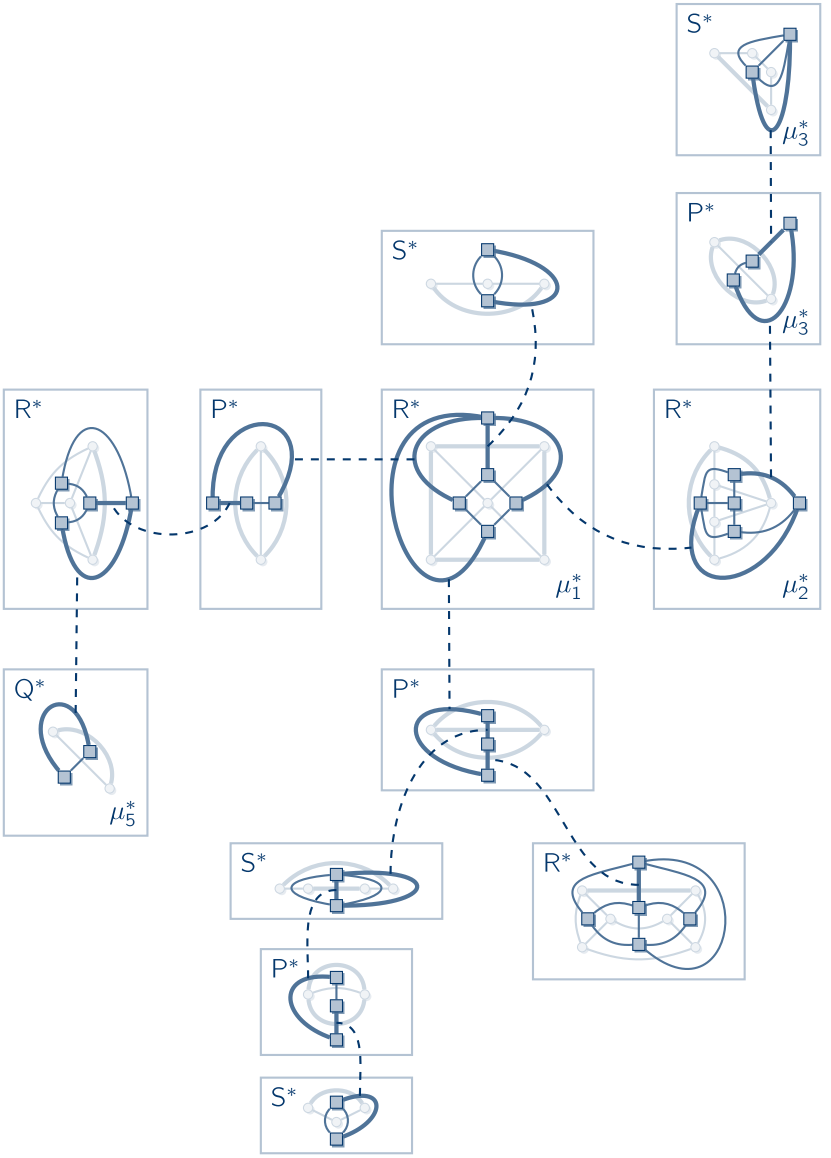


Figure 3.36: Dual SPQR tree of the SPQR tree in Fig. 3.33.

Proposition 3.6. *The following relationship holds between the node types of SPQR trees and dual SPQR trees:*

$$\mathsf{P} \leftrightarrow \mathsf{S}^*\,, \qquad \mathsf{S} \leftrightarrow \mathsf{P}^*\,, \qquad \mathsf{Q} \leftrightarrow \mathsf{Q}^*\,, \qquad \mathsf{R} \leftrightarrow \mathsf{R}^*\,.$$

where $X \leftrightarrow Y$ means that type X in the primal SPQR tree corresponds to type Y in the dual SPQR tree.

In an SPQR tree, P and S nodes can never be adjacent to nodes of their own type. The same is true for dual SPQR trees as P and S nodes swap their roles and R and Q nodes keep their type. From these observations, we can conclude that a dual SPQR tree is again an SPQR tree.

In Sect. 3.5.2.1, we have proved De Morgan's law for one-clique sums, i. e., the dual of a one-clique sum of two primal graphs is the one-clique sum of their duals (Lem. 3.13). Next, we prove De Morgan's law for two-clique sums. As in Sect. 3.5.2.1, we distinguish the two-clique sum in the dual from the two-clique sum in the primal by using the binary operator $\boxtimes$, i. e., $G^* \boxtimes G'^*$ is the (e^*, e'^*)-two-clique sum of two duals G^* and G'^* at dual edges e^* and e'^*.

Lemma 3.23 (De Morgan's Law for Two-Clique Sums). *Let $G = (V, E)$ and $G' = (V', E')$ be two embedded and biconnected graphs with duals $G^* = (F, E^*)$ and $G'^* = (F', E'^*)$, respectively. Further, let $e \in E$ and $e \in E'$ be two edges with respective duals $e^* \in E^*$ and $e'^* \in E'^*$. We obtain the following equality:*

$$(G \otimes G')^* = G^* \boxtimes G'^*\,,$$

where $(G \otimes G')^*$ denotes the dual of the (e, e')-two-clique sum of G and G' and $G^* \boxtimes G'^*$ the (e^*, e'^*)-two-clique sum of G^* and G'^*.

As with De Morgan's law for one-clique sums (Lem. 3.13), the strategy of the proof is to show that the dual obtained from a two-clique summation of two primal graphs is equal to the two-clique sum of their duals. This is illustrated by the following commutative diagram:

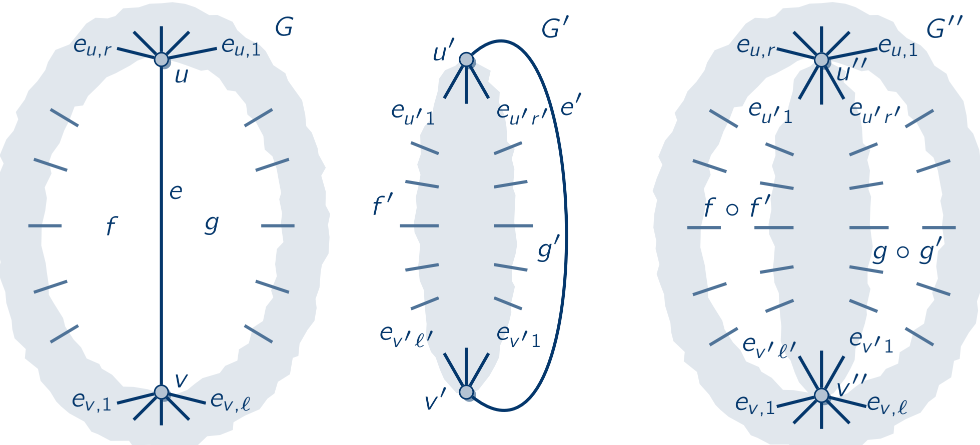
$$\begin{array}{ccc} G, G' & \xrightarrow{*} & G^*, G'^* \\ \downarrow \otimes & & \downarrow \boxtimes \\ G \otimes G' & \xrightarrow{*} & G^* \boxtimes G'^* \end{array}$$

In the proof, we investigate what happens to the dual graph during a (primal) two-clique summation.

Proof. The two graphs G and G' are sketched in Fig. 3.37(a). The endpoints of e and e' are u, v and u', v', respectively. Let $G'' = G \otimes G'$ be the (e, e')-two-clique sum of G and G', were u and u' are identified and so are v and v'. Graph G'' is sketched in Fig. 3.37(b), where u and u' are replaced by u'', and v and v' by v''.

Let $e, e_{u,1}, \dots, e_{u,r}$ be the rotation system of u and let g be the face between edges e and $e_{u,1}$ and f be the face on the opposite side of e, i. e., the face between $e_{u,r}$ and e. Since f lies opposite to g, they are connected in G^* by edge e^* which is the dual of e. Note that vertex v is also incident to f and g as v is connected to u by edge e. Moreover, $f \neq g$ since G is biconnected. The rotation system of v is $e, e_{v,1}, \dots, e_{v,\ell}$.

Analogously in G', the rotation system of u' is $e', e_{u'1}, \dots, e_{u'r'}$, f' is the face between e' and $e_{u'1}$, and g' lies between $e_{u'r'}$ and e'. The rotation system of v' is $e', e_{v'1}, \dots, e_{v'\ell'}$. As before, f' and g' are connected in G'^* by edge e'^* which is the dual of e'.



(a) Again, G and G' are merged at e and e'. This time the duals of both graphs are additionally sketched.

(b) The two-clique summation of G and G' is also a two-clique-summation of G^* and G'^*.

Figure 3.37: A (primal) two-clique summation corresponds to a two-clique sum in the dual.

By the definition of the two-clique sum (cf. Sect. 3.5.3.1), the rotation system of u'' is obtained by splitting the rotation systems of u and u' at e and e', respectively, and by concatenating them. The rotation system of u'' is thus $e_{u,1}, \dots, e_{u,r}, e_{u'1}, \dots, e_{u'r'}$. Between edges $e_{u,r}$ and $e_{u'1}$ lies the face $f \circ f'$ and between edges $e_{u'r'}$ and $e_{u,1}$ lies the face $g \circ g'$ (see Fig. 3.37(b)). The names of these faces are chosen deliberately as it turns out that f and f' in G^* and G'^* are merged to face $f \circ f'$ in G''^* and the same holds for g, g' and $g \circ g'$.

The rotation system of face f is obtained by the counterclockwise traversal of its boundary in G. The duals of the edges on this boundary are the incident edges of f in G^* in the cyclic order as defined by the boundary. f's rotation system is thus $e^*, e^*_{u,r}, \dots, e^*_{v,1}$, where e^* is the dual of e, $e^*_{u,r}$ the dual of $e_{u,r}$, and so forth. All edges between $e^*_{u,r}$ and $e^*_{v,1}$ are sketched in Fig. 3.37(a) as shaded truncated lines around f, reaching into G. Likewise, the rotation system of g is $e^*, e^*_{v,\ell}, \dots, e^*_{u,1}$, that of f' is $e'^*, e^*_{v'\ell'}, \dots, e^*_{u'1}$, and that of g' is $e'^*, e^*_{u'r'}, \dots, e^*_{v'1}$.

Consider now G''^* and face $f \circ f'$ in Fig. 3.37(b): Its rotation system is obtained as before and is $e^*_{u,r}, \dots, e^*_{v,1}, e^*_{v'\ell'}, \dots, e^*_{u'1}$. Thus, it is the concatenation of the rotation systems of f and f' split at e and e', respectively. The same holds for the rotation system of $g \circ g'$ which is obtained from the rotation systems of g and g'. Also, faces f and f' are replaced by face $f \circ f'$ and faces g and g' are replaced by $g \circ g'$. All other faces and rotation systems of the dual graph stay untouched. In other words, G''^* is the (e^*, e'^*)-two-clique sum of G^* and G'^* and, hence, the equality holds. □

We can reconstruct G and its embedding from its SPQR tree $\mathcal{T}$ by a sequence of two-clique summations. The same technique can be applied to the dual SPQR tree $\mathcal{T}^*$ and the resulting graph is indeed the dual graph G^*. From Prop. 3.6 and De Morgan's law for two-clique sums (Lem. 3.23), we thus obtain the following theorem.[3]

[3] This result was discovered independently and simultaneously by [ABR13].

Theorem 3.4. *The dual SPQR tree of an embedded graph G is the SPQR tree of the dual G^*.*

Proof. By Prop. 3.6, we know $\mathcal{T}^*$ is an SPQR tree, where the roles of P and S nodes swap when going from the primal to the dual. We now show that we can obtain the dual G^* from the dual SPQR tree $\mathcal{T}^*$ by a sequence of two-clique summations of $\mathcal{T}^*$'s skeletons. Let $\overline{G}_1, \dots, \overline{G}_k$ be a node series of G and its SPQR tree $\mathcal{T}$. Recall that the node series is defined as:

$$\overline{G}_i = \overline{G}_{i-1} \otimes \mathsf{skel}(\mu_i)\,,$$

with $\overline{G}_1 = \mathsf{skel}(\mu_1)$ and $\overline{G}_k = G$. For the duals $(\overline{G}_i)^*$ of the node series, denoted by $\overline{G}_i^*$, we obtain:

$$\overline{G}_i^* = (\overline{G}_{i-1} \otimes \mathsf{skel}(\mu_i))^*\,,$$

and, by De Morgan's law for two-clique sums (Lem. 3.23), we can rewrite this equation to:

$$\overline{G}_i^* = \overline{G}_{i-1}^* \boxtimes \mathsf{skel}(\mu_i)^*\,.$$

The graph $\mathsf{skel}(\mu_i)^*$ is the dual skeleton of node μ_i and, by the definition of dual SPQR trees (Def. 3.5), it is the skeleton of μ_i^* in $\mathcal{T}^*$, i. e., $\mathsf{skel}(\mu_i)^* = \mathsf{skel}(\mu_i^*)$. Inserting this equality yields:

$$\overline{G}_i^* = \overline{G}_{i-1}^* \boxtimes \mathsf{skel}(\mu_i^*)\,,$$

with $\overline{G}_1^* = \mathsf{skel}(\mu_1^*)$ and $\overline{G}_k^* = G^*$. Thus, $\mathcal{T}^*$ is the SPQR tree of G^*. □

Denote by σ the function that transform a biconnected and embedded graph into its SPQR tree. The following commutative diagram illustrates the relationship between embedded graphs, SPQR trees and their duals:

$$\begin{array}{ccc} G & \xrightarrow{*} & G^* \\ \downarrow^{\sigma} & & \downarrow^{\sigma} \\ \mathcal{T} & \xrightarrow{*} & \mathcal{T}^* \end{array}$$

3.5.3.3 dSPQR Trees of Acylic Digraphs

As we investigate (**RUP**) digraphs, we need to incorporate edge directions into SPQR trees. We start off with the definition of directed SPQR trees, or simply dSPQR trees, of acyclic digraphs. As an example, we consider the acyclic digraph displayed in Fig. 3.38 which is a directed and acyclic version of the graph in Fig. 3.32(a). Sources and sinks are displayed by diamond shapes and are white. In the following, we call an acyclic dipole with source u and sink v a uv-digraph.

Definition 3.6 (dSPQR Trees of Acyclic Digraphs). *Let G be a biconnected and acyclic digraph and $\mathcal{T}$ be the SPQR tree of its underlying undirected graph. The* dSPQR tree *of G, denoted by $\vec{\mathcal{T}}$, is obtained from $\mathcal{T}$ by directing and labeling the edges of the skeletons. The non-virtual edges in the Q nodes inherit the direction as defined in G. For each virtual edge, we define the* expansion digraph *in accordance to the expansion graph, where all edges are directed as in G. For each virtual edge $\{u, v\}$ of a node μ, we distinguish between the following cases: If the expansion digraph* $\mathsf{expg}_\mu(\{u, v\})$ *...*

- *... is a uv-digraph (vu-digraph), then the virtual edge $\{u, v\}$ is directed from u to v (v to u) in μ.*

- ▶ *... contains a source distinct from* u *and* v*, then the virtual edge* $\{u, v\}$ *is a* source edge *in* μ*, indicated by the label* ◀▶.
- ▶ *... contains a sink distinct from* u *and* v*, then the virtual edge* $\{u, v\}$ *is a* sink edge *in* μ*, indicated by the label* ▶◀.

By applying these substitutions in the skeleton skel(μ) *of node* μ*, we obtain the* directed skeleton $\overrightarrow{\text{skel}}(\mu)$.

Since in the following, we are mostly dealing with digraphs and directed skeletons, we simply speak of skeletons. If a virtual edge is a source edge, a sink edge, or both, we call it *terminal edge*. Note that a virtual edge may be both a source and a sink edge. The dSPQR tree of the digraph in Fig. 3.38 is shown in Fig. 3.39. The expansion digraph $\text{expg}_{\mu_1}(\{u, v\})$, displayed at the top left side of Fig. 3.39, contains a source distinct from u and v, and, hence, $\{u, v\}$ is a source edge in μ_1. The order of the labels ◀▶ and ▶◀ on an edge that is both a source and a sink edge is arbitrary. The expansion digraph of a virtual edge $e = \{u, v\}$ which is refined a Q node is always a uv- or vu-digraph and, hence, e is directed, e. g., the virtual edge refined by μ_5. As before, we omit Q nodes and directly display edges and their directions, therefore, an edge in a skeleton is either virtual and refined by an S, P, or R node, or it is non-virtual.

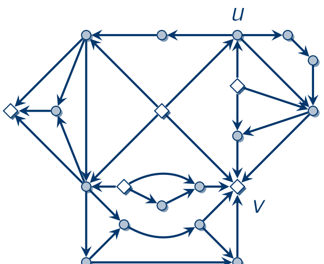


Figure 3.38: Directed acyclic version of the planar graph in Fig. 3.32(a). Sources and sinks are displayed by diamond shapes.

If a virtual edge is directed, it cannot be a terminal edge by Def. 3.6. In turn, a terminal edge cannot be directed. Also, there is no undirected virtual edge which is also no terminal edge: Suppose that $\{u, v\}$ is undirected and no terminal edge. Then its expansion digraph is neither a uv- nor a vu-digraph and contains no other sources or sinks. Therefore, the expansion digraph must contain a cycle contradicting the assumption that the original digraph is acyclic.

Proposition 3.7. *A virtual edge in a dSPQR tree of an acyclic digraph is either directed or a terminal edge.*

Consider the virtual edge $e = \{u, v\}$ in node μ_2 in Fig. 3.39 which is refined by μ_1. Node μ_1 contains a source and a sink edge different from e. Hence, the expansion digraph $\text{expg}_{\mu_2}(\{u, v\})$ contains a source and a sink different from u and v. Therefore, e must be a source and sink edge in μ_2. In general, we get the following property of transitivity of terminal edges.

Proposition 3.8 (Transitivity of Terminal Edges). *Let* μ *be a node containing a virtual edge* e *that is refined by node* μ'*. If any virtual edge* $e' \neq e$ *in* μ' *is a source (sink) edge, then so is virtual edge* e *in* μ*.*

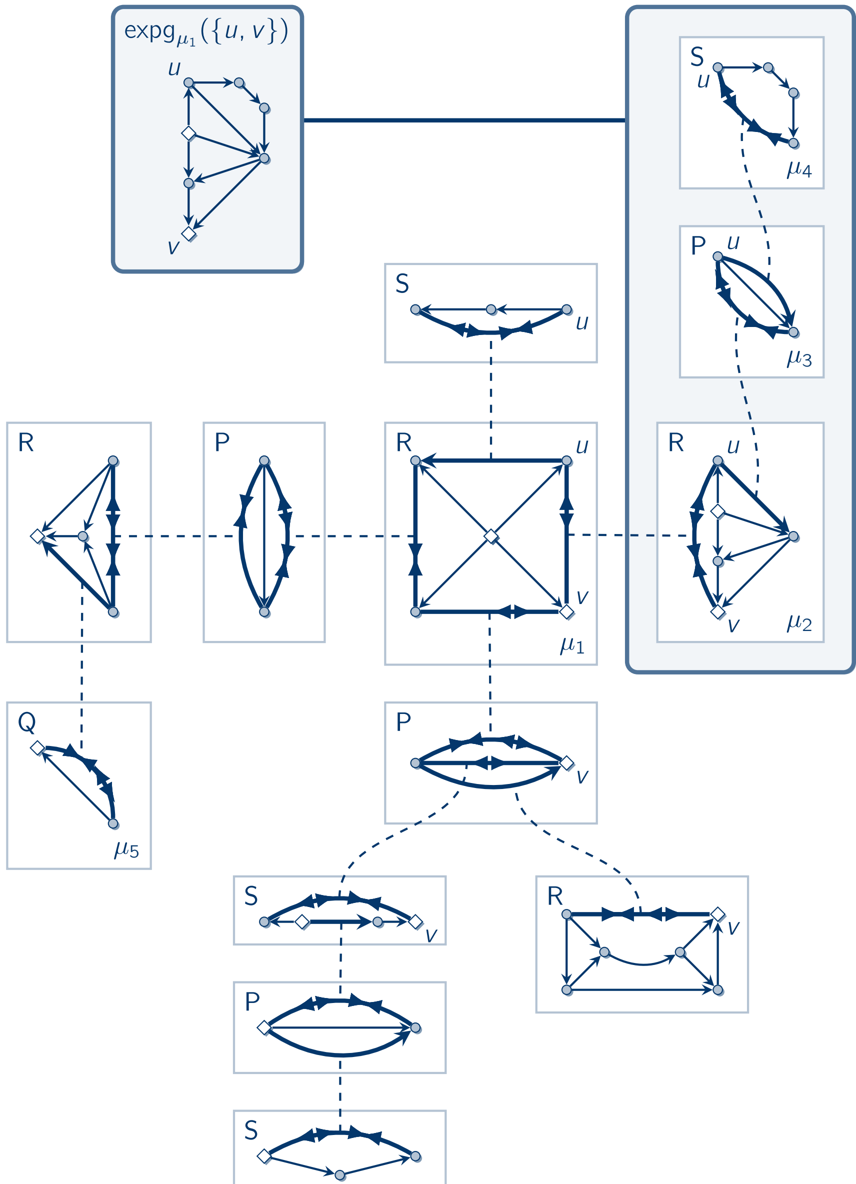
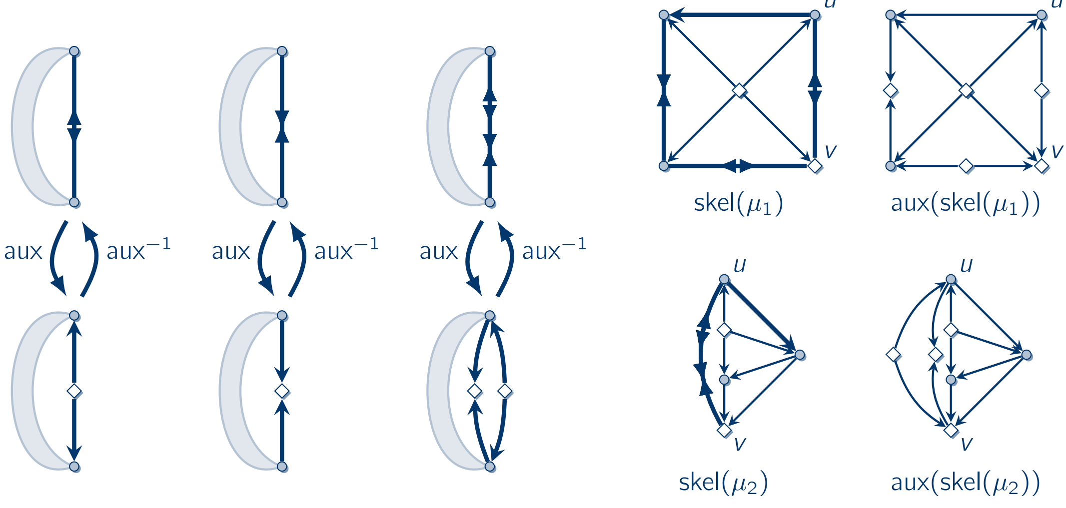


Figure 3.39: dSPQR tree of the planar and acyclic digraph in Fig. 3.38.

A source edge e stands for "remote" source in the expansion digraph of e. Given a skeleton of a node, we can replace the source and sink edges by *auxiliary sources* and *sinks* as displayed in Fig. 3.40(a). Let μ be a node with directed skeleton $\overrightarrow{\text{skel}}(\mu)$. By replacing all terminal edges in this manner and keeping the directed virtual edges as they are, we obtain the *auxiliary skeleton* $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ from $\overrightarrow{\text{skel}}(\mu)$. The inverse operation, i. e., replacing the auxiliaries by the respective terminal edges, is denoted by aux^{-1}. Fig. 3.40(b) shows the skeletons of nodes μ_1 and μ_2 along with their auxiliaries. Vertex v is a sink in the whole digraph and also in the auxiliary skeleton of node μ_1. In contrast, in the auxiliary skeleton of μ_2, v is no sink. The reason is that each skeleton and its auxiliary always display only a local scope of the whole digraph; an important fact we have to keep in mind. Auxiliary skeletons play an important role in the following sections.



(a) The auxiliaries of terminal edges.

(b) The skeletons of μ_1 and μ_2 along with their auxiliary skeletons.

Figure 3.40: Auxiliary skeletons.

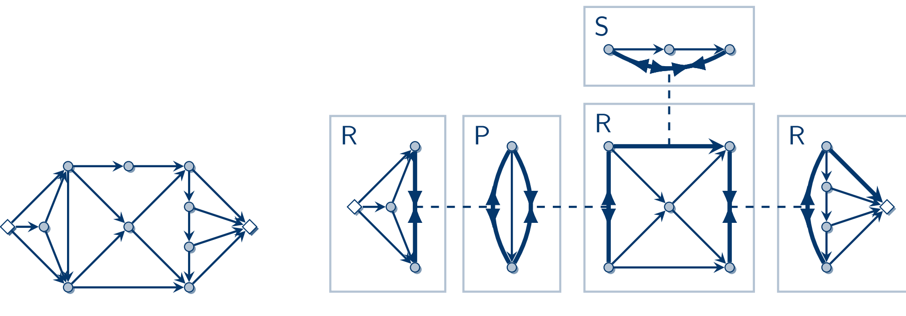
3.5.3.4 dSPQR Trees of Acyclic Dipoles

As acyclic dipoles play an important role for the duals of **RUP** compounds, we investigate their dSPQR trees. It turns out that the property of being an acyclic dipole is visible in each node of the dSPQR tree of an acyclic dipole. We start with a definition.

Definition 3.7. *A node μ and its skeleton in a dSPQR tree of an acyclic digraph is called* acyclic dipole *if and only if the auxiliary skeleton* $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ *is an acyclic dipole.*

Consider the acyclic dipole in Fig. 3.41(a) and its dSPQR tree in Fig. 3.41(b). Note that each node is an acyclic dipole. In fact, this holds true in general and we obtain the following characterization of acyclic dipoles and their dSPQR trees.

Lemma 3.24. *An acyclic digraph is a dipole if and only if each node of its dSPQR tree is an acyclic dipole.*



(a) An acyclic dipole.

(b) The dSPQR tree of the acyclic dipole in Fig. 3.41(a) in which each node is an acyclic dipole.

Figure 3.41: An acyclic dipole and its dSPQR tree.

Proof. $\Rightarrow$: Let G be an acyclic dipole with dSPQR tree $\vec{\mathcal{T}}$. First, we observe that replacing the sink and source edges by their auxiliaries (Fig. 3.40(a)) in a node's skeleton cannot introduce any cycles. Thus, each auxiliary skeleton is acyclic as G is acyclic. In the following, we assume for contradiction that there is a node μ in $\vec{\mathcal{T}}$ whose auxiliary skeleton contains at least two sources u, u'. The proof proceeds analogously if $\operatorname{aux}(\overrightarrow{\operatorname{skel}}(\mu))$ contains two or more sinks.

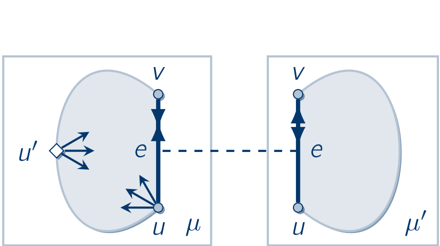
There are two cases: First, both u and u' are vertices in $\overrightarrow{\operatorname{skel}}(\mu)$, that is, neither u nor u' is a source as obtained from the auxiliary of a source edge. Second, exactly one of u and u' is an auxiliary source, i. e., obtained from a source edge. Note that not both can be auxiliary sources as then G would contain two distinct sources.

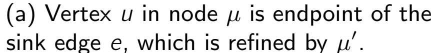
First, assume that neither of u and u' is an auxiliary source. Our first observation is that at least one of u and u' must be part of a split pair: If this is not the case, then no virtual edge is incident to u and u' and, hence, both u and u' are sources in G which contradicts the assumption that G is a dipole. Therefore, there is at least one virtual edge incident to u or u' and, thus, at least one of them is part of a split pair. Since u is a source in the auxiliary skeleton, a virtual edge incident to u is either a sink edge or directed and pointing away from u. The same holds for virtual edges incident to u'. Moreover, at least one virtual edge incident to u and u' must be a sink edge: Assume for contradiction that all these virtual edges are directed away from u and u'. This implies that all expansion digraphs of these virtual edges are uv- and $u'v'$-digraphs, where v and v' are the other endpoints of the virtual edges. However, then u and u' are two sources in G as well, which is again a contradiction.

For the second case, assume that u' is an auxiliary source and u is not. This implies that there is a source s distinct from u' in the expansion digraph $\operatorname{expg}_\mu(e')$ where e' is the source edge that is replaced by the auxiliary source u' in $\operatorname{aux}(\overrightarrow{\operatorname{skel}}(\mu))$. Vertex u must be part of a split pair as otherwise it would be a source in G distinct from s. As u is a source in the auxiliary, all virtual edges incident to u are either sink edges or pointing away from u. Similar to before, at least one of these virtual edge must be a sink edge as otherwise all directed virtual edges incident to u would point away from u, and u would be a source distinct from s in G.

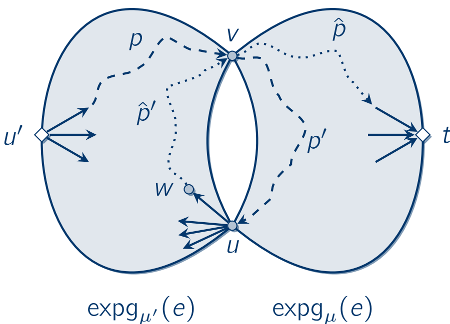
Regardless of which case applies, we assume w. l. o. g. that u is endpoint of a sink edge e whose other endpoint is v. The relevant part of the dSPQR tree is sketched in Fig. 3.42(a). The expansion digraph $\operatorname{expg}_\mu(e)$ of e in node μ contains the sink t of G with $t \neq u$ and $t \neq v$.

As t is the single sink of G, no other virtual edge of node μ can be a sink edge. In particular, all (virtual) edges $e' \neq e$ incident to u (u') are directed away from u (u').





(a) Vertex u in node μ is endpoint of the sink edge e, which is refined by μ'.



(b) Expansion digraphs of e in μ and μ': There is a dipath from u' to u via v (dashed) and a dipath from u to t also via v (dotted). Combining both dipaths results in a cycle.

Figure 3.42: Situation obtained in the proof of Lem. 3.24.

Let μ' be the node which refines e in $\vec{\mathcal{T}}$. Recall that μ' also contains virtual edge e. Let $\mathrm{expg}_{\mu'}(e)$ be the expansion digraph of e in μ'. Fig. 3.42(b) sketches both expansion digraphs $\mathrm{expg}_{\mu}(e)$ and $\mathrm{expg}_{\mu'}(e)$. All virtual and non-virtual edges incident to u point away from u in $\overrightarrow{\mathrm{skel}}(\mu)$ and, hence, u is a source in $\mathrm{expg}_{\mu'}(e)$. For u' there are two cases: If u' is no auxiliary source, then by the same reasoning as for u, u' is a source in $\mathrm{expg}_{\mu'}(e)$. If u' is an auxiliary source, then $\mathrm{expg}_{\mu'}(e)$ contains a source s as $\mathrm{expg}_{\mu}(e')$ contains the same source s where e' is the source edge in $\overrightarrow{\mathrm{skel}}(\mu)$ that is replaced by the auxiliary source u' in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu))$. For the sake of simplicity, we identify s with u' in this case in the following as the reasoning is similar to the case where u' is no auxiliary source.

The expansion digraph $\mathrm{expg}_{\mu'}(e)$ is acyclic and, thus, it must contain a sink. As $\mathrm{expg}_{\mu}(e)$ contains the single sink t of G which is distinct from u and v, either u or v must be a sink in $\mathrm{expg}_{\mu'}(e)$. Further, since u is a source, v must be the single sink of $\mathrm{expg}_{\mu'}(e)$. This also implies that v and u' are distinct as u' is a source as well. Hence, $\mathrm{expg}_{\mu'}(e)$ contains the source u' distinct from u and v, and vertex u' is the single source of G. This also implies that u is no source of G.

Let p be a dipath in G from its source u' to u (dashed in Fig. 3.42(b)). The dipath p must exist since G contains only one source u'. Vertex u is a source in $\mathrm{expg}_{\mu'}(e)$, therefore, dipath p must leave $\mathrm{expg}_{\mu'}(e)$ to reach u. Dipath p can only leave $\mathrm{expg}_{\mu'}(e)$ by passing vertex v before reaching u because u, v is a split pair. Hence, from p we obtain a dipath p' from v to u in $\mathrm{expg}_{\mu}(e)$. Let w be a vertex in $\mathrm{expg}_{\mu'}(e)$ such that there is a directed edge from u to w. Such a vertex and edge must exist since $\mathrm{expg}_{\mu'}(e)$ is biconnected and at least contains vertices u, u', and v. Note that $w = v$ is possible while $w = u'$ is not as u' is a source. Let $\hat{p}$ be a dipath in G from u to t via w, i.e., $\hat{p} = u \rightarrow w \rightsquigarrow t$ (dotted in Fig. 3.42(b)). Again, this dipath must exist since G contains only sink t. By the same reasoning as before, $\hat{p}$ must contain vertex v. Therefore, we obtain a dipath $\hat{p}'$ from u to v. By combining both dipaths $p' = v \rightsquigarrow u$ and $\hat{p}' = u \rightsquigarrow v$, we obtain a cycle in G which is a contradiction.

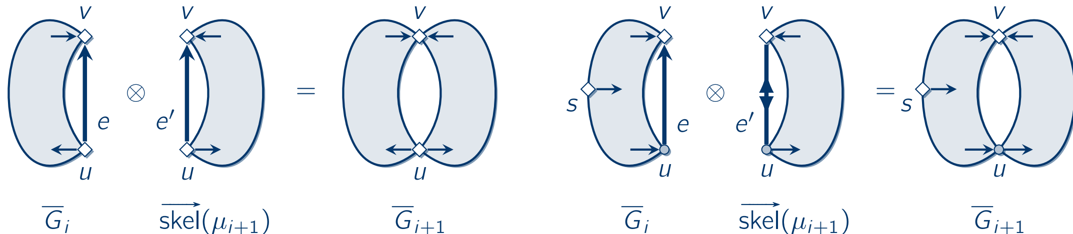
$\Leftarrow$: Let G be an acyclic digraph with dSPQR tree $\overrightarrow{\mathcal{T}}$. Further, let $\mu_1, \ldots, \mu_k$ be a node sequence of $\overrightarrow{\mathcal{T}}$ by which we obtain the node series $\overline{G}_i$ with $\overline{G}_i = \overline{G}_{i-1} \otimes \overrightarrow{\mathrm{skel}}(\mu_i)$ for $1 < i \leq k$ and $\overline{G}_1 = \overrightarrow{\mathrm{skel}}(\mu_1)$. Recall that $\overline{G}_{i-1} \otimes \overrightarrow{\mathrm{skel}}(\mu_i)$ is the two-clique sum of $\overline{G}_{i-1}$ and μ_i's skeleton. By adopting the definition of terminal and directed virtual edges, each digraph $\overline{G}_i$ with $i < k$ contains at least one virtual edge which is either directed or a terminal edge. We canonically apply the definition of the auxiliary skeleton to each graph $\overline{G}_i$ to obtain the auxiliary digraph $\mathrm{aux}(\overline{G}_i)$, where all virtual edges are replaced as shown in Fig. 3.40(a).

By mathematical induction, we show that each auxiliary digraph $\mathrm{aux}(\overline{G}_i)$ is an acyclic dipole and, hence, $\mathrm{aux}(\overline{G}_k) = \mathrm{aux}(G) = G$ is an acyclic dipole. The digraph $\mathrm{aux}(\overline{G}_1) = \mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_1))$ is an acyclic dipole by assumption. For the induction step, suppose that $\mathrm{aux}(\overline{G}_i)$ is an acyclic dipole. There is a virtual edge e in $\overline{G}_i$ that is refined by node μ_{i+1} and e also appears in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$. For clarity, we distinguish e in $\overline{G}_i$ from e in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ and denote the first by e and the latter by e'.

In the following, we use a complete case differentiation to show that $\overline{G}_{i+1} = \overline{G}_i \oplus \overrightarrow{\mathrm{skel}}(\mu_{i+1})$ is an acyclic dipole: Edge e can either be directed from u to v, vice versa, or e is a sink or a source edge, or both. This makes 5 cases and, since the same 5 cases also apply to e', we get 25 cases altogether. Fortunately, some of these cases cannot occur. For instance, e and e' cannot be source edges at the same time: If e is a source edge, then $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ either contains a source edge e'' distinct from e' or a source distinct from u and v. In particular, $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1}))$ contains a source s with $s \neq u, v$. However, e' is also a source edge in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ and, therefore, $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1}))$ contains two sources which is a contradiction to $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1}))$ being an acyclic dipole. For the same reasons, e and e' cannot be sink edges at the same time. Note that the reasoning from before also applies if both e and e' are source edges and any of them is additionally a sink edge or vice versa. This reduces the amount of cases by 7 to a total number of 18. Moreover, it is not possible that e is directed from u to v and e' from v to u, or vice versa, as this would imply a cycle, and 16 cases remain.

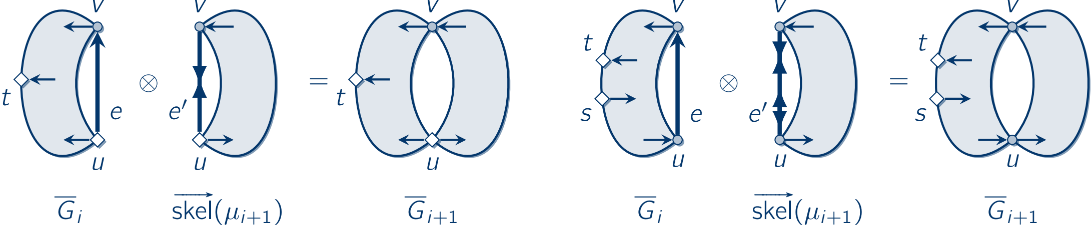
The cases we investigate in the following are sketched in Fig. 3.43. All other cases not explicitly depicted in Fig. 3.43 are symmetric in the sense that the roles of $\overline{G}_i$ and $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ swap or the orientation of the directed edge is reversed. In each diagram, the digraphs $\overline{G}_i$, $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ and $\overline{G}_{i+1}$ are sketched. Again, terminals, i. e., sources and sinks, are displayed by a diamond shape. Some edges have only one endpoint and are used to indicate the existence of at least one incoming/outgoing edge to/from the corresponding vertex. For instance, in Fig. 3.43(a) vertex v in $\overline{G}_i$ has at least one incoming edge apart from e and u has at least one additional outgoing edges. Observe that during the two-clique summation of $\overline{G}_i$ and $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ only vertices u and v are changed, whereas all other vertices stay unchanged. In order to prove that $\mathrm{aux}(\overline{G}_{i+1})$ is an acyclic dipole, we show that if u is a source, a sink, or none of both in $\mathrm{aux}(\overline{G}_i)$ and $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1}))$ then u is a source, a sink, or none of both in $\mathrm{aux}(\overline{G}_{i+1})$, respectively. The same also holds for v.

- ***e* and *e′* are directed — two cases** (Fig. 3.43(a))
  W. l. o. g., we assume that $e = (u, v)$ and $e' = (u, v)$. Due to transitivity (Prop. 3.8), neither $\overline{G}_i$ nor $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ contains any terminal edge. This implies that in $\mathrm{aux}(\overline{G}_i)$ and $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1}))$ only u and v can be terminals. Since both e and e' are directed from u to v, and $\mathrm{aux}(\overline{G}_i)$ and $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1}))$ are acyclic dipoles, u is a source and v a sink in both digraphs. The same is then true for $\mathrm{aux}(\overline{G}_{i+1})$ which is an acyclic dipole with source u and sink v.



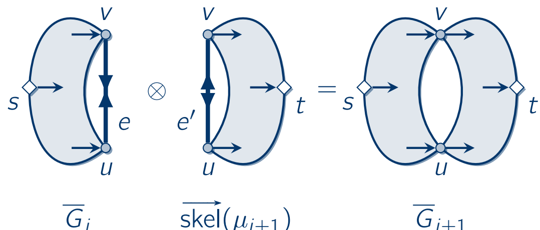
(a) Both e and e' are directed.

(b) Virtual edge e is directed and e' is a source but no sink edge.



(c) Virtual edge e is directed and e' is a sink but no source edge.

(d) Virtual edge e is directed and e' is a source and a sink edge.



(e) Virtual edge e is a sink but no source edge and e' a source but no sink edge.

Figure 3.43: Case differentiation as obtained in the second part of the proof of Lem. 3.24.

- ▶ ***e* is directed and *e′* is a source/no sink edge — four cases** (Fig. 3.43(b))
  Since e' is a source edge, there is one source s with $s \neq u$ and $s \neq v$ in aux($\overline{G}_i$), either represented by another source edge or by a vertex. Also, e' is no sink edge and, hence, either u or v must be a sink in aux($\overline{G}_i$). Again, we assume w. l. o. g. that e is directed from u to v which implies that v is the sink. As aux($\overline{G}_i$) is a dipole with source s, there must be at least one incoming edge to u in aux($\overline{G}_i$). In $\overrightarrow{\text{skel}}(\mu_{i+1})$ source s is represented by the source edge e'. Further, since e is directed from u to v in $\overline{G}_i$, u has at least one outgoing and v at least one incoming edge in aux($\overrightarrow{\text{skel}}(\mu_{i+1})$). Consequently, v is the single sink in aux($\overrightarrow{\text{skel}}(\mu_{i+1})$). Putting all things together, we obtain aux($\overline{G}_{i+1}$) where s is the single source and v the single sink.

- ▶ ***e* is directed and *e′* is a sink/no source edge — four cases** (Fig. 3.43(c))
  This case is symmetric to the previous case where e is directed and e' a source edge only that this time u is the single source and there is a single sink t in $\overline{G}_i$.

- ▶ ***e* is directed and *e′* is a source and a sink edge — four cases** (Fig. 3.43(d))
  Again, w. l. o. g. e points from u to v. As e' is a both a source and a sink edge, there must be a source s and a sink t in aux($\overline{G}_i$) distinct from u and v. This time neither u nor v can be terminals of the auxiliaries of $\overline{G}_i$ and $\overrightarrow{\text{skel}}(\mu_{i+1})$. Since $e = (u, v)$ and $\overline{G}_i$ is a dipole, u has at least one incoming edge and v at least one outgoing edge in aux($\overline{G}_i$). The direction of e also implies that u has at least one outgoing and v at least one incoming edge in aux($\overrightarrow{\text{skel}}(\mu_{i+1})$) distinct from the auxiliary of e'. Hence, neither u nor v is a terminal in aux($\overline{G}_{i+1}$), which must then be an acyclic dipole with source s and sink t.

- ▶ ***e* is a sink/no source edge and *e′* a source/no sink edge — two cases** (Fig. 3.43(e))
  There is a source s in aux($\overline{G}_i$) and a sink t in aux($\overrightarrow{\text{skel}}(\mu_{i+1})$) both distinct from u and v. In aux($\overline{G}_i$), neither u nor v is a source and, hence, both must have at least one incoming edge. Further, u and v are no sinks in aux($\overrightarrow{\text{skel}}(\mu_{i+1})$) and both must have at least one outgoing edge. Consequently, s is the single source and t the single sink in aux($\overline{G}_{i+1}$).

In any case, aux($\overline{G}_{i+1}$) is an acyclic dipole and, hence, G itself is an acyclic dipole. □

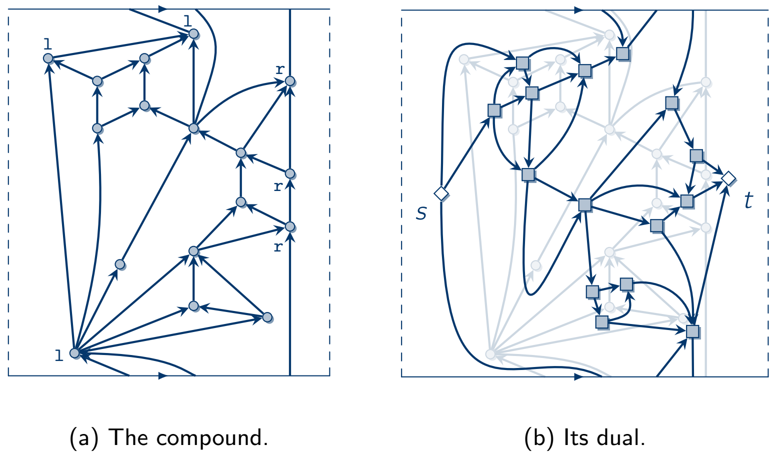
3.5.3.5 dSPQR Trees of Compounds

In this section, we combine dual SPQR trees (Sect. 3.5.3.2) with dSPQR trees of acyclic digraphs (Sect. 3.5.3.3) to define dSPQR trees of compounds. We assume that the compound at hand is embedded for reasons that will become clear later. From the previous sections, we readily obtain a characterization of **RUP** compounds by means of dual SPQR and dSPQR trees.

Corollary 3.21. *An embedding of a biconnected compound γ is* **RUP** *if and only if all nodes of the dSPQR tree of γ^* are acyclic dipoles.*

Proof. Let γ be an embedded and biconnected compound. The dual SPQR tree $\mathcal{T}^*$ of the underlying undirected graph of γ is the SPQR tree of the underlying undirected graph of γ^* by Thm. 3.4. Since γ^* is acyclic, we obtain from $\mathcal{T}^*$ the dSPQR tree of γ^* by Def. 3.6. The embedding of γ is **RUP** if and only if its dual γ^* is an acyclic dipole by Lem. 3.4 which, in turn, holds true if and only if each node of the dSPQR tree of γ^* is an acyclic dipole (Lem. 3.24). □

As an example consider the biconnected **RUP** compound in Fig. 3.44(a) whose dual is depicted in Fig. 3.44(b). The labels `l`, `r`, and `lr` can be ignored for the moment. The dSPQR tree of the compound's dual is shown in Fig. 3.45 and all of its nodes are acyclic dipoles.



(a) The compound. (b) Its dual.

Figure 3.44: A biconnected, **RUP**-embedded compound and its dual.

Remember that a source (sink) edge in the dSPQR tree of an acyclic digraph stands for a "remote" source (sink) in the respective expansion digraph. We use a similar idea for dSPQR trees of compounds, where we introduce *cyclic edges*, which are the duals of terminal edges and stand for remote left- and rightmost cycles.

Definition 3.8 (dSPQR Trees of Compounds). *Let γ be an embedded and biconnected compound and let $\mathcal{T}$ be the SPQR tree of its underlying undirected graph. The dSPQR tree of γ is defined as follows: As with dSPQR trees of acyclic digraphs (Def. 3.6), the non-virtual edges in the Q nodes are directed as their counterparts in γ. Let $\{u, v\}$ be a virtual edge of a node μ. If the expansion digraph* $\text{expg}_\mu(\{u, v\})$ *. . .*

- *. . . is a uv-digraph (vu-digraph), then $\{u, v\}$ is directed from u to v (v to u) in μ.*
- *. . . contains a leftmost cycle of γ, then the virtual edge $\{u, v\}$ is a cyclic-L edge in μ, indicated by the symbol $\odot_L$.*
- *. . . contains a rightmost cycle of γ, then the virtual edge $\{u, v\}$ is a cyclic-R edge in μ, indicated by the symbol $\odot_R$.*

Observe, a virtual can be both cyclic-L and cyclic-R. We call a virtual edge *cyclic* if it is a cyclic-L or a cyclic-R edge, or both. Note that the property of containing a left- or rightmost cycle depends on the current embedding. This is also the reason why Def. 3.8 only works for embedded compounds. Fig. 3.46 shows the dSPQR tree of the embedded compound in Fig. 3.44(a). Consider the expansion digraph $\text{expg}_{\mu_2}(e)$ of the cyclic-L edge e in μ_2 which is refined by μ_1. $\text{expg}_{\mu_2}(e)$ is essentially the skeleton of μ_1 without the cyclic-R edge and it contains a leftmost cycle of the compound in Fig. 3.44(a). Hence, the virtual edge in μ_2 refined by μ_1 is cyclic-L. Respectively, the expansion digraph of the cyclic-R edge in μ_2, which

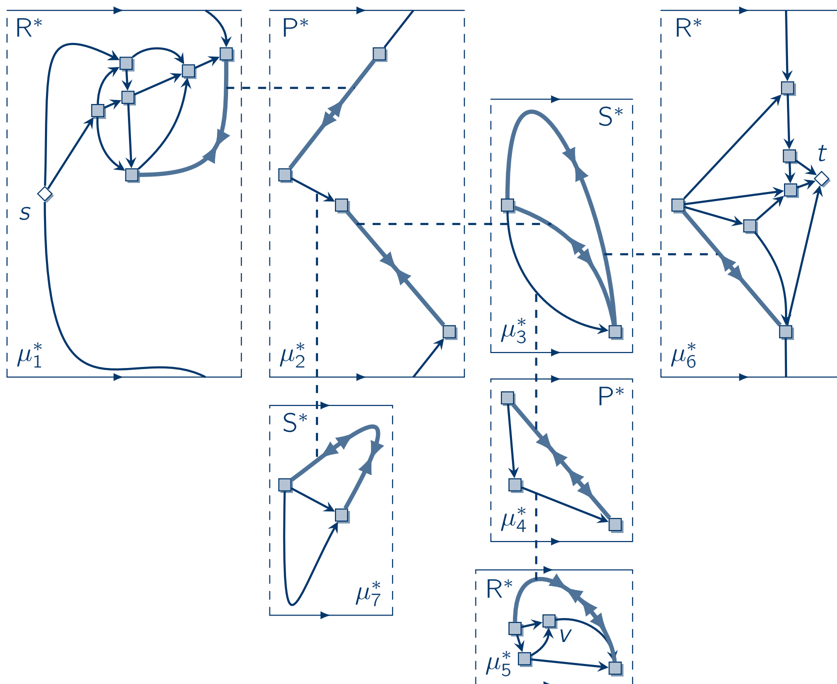


Figure 3.45: dSPQR tree of the dual in Fig. 3.44(b).

is refined by μ_3, contains a rightmost cycle of the compound. Note that for an edge to be cyclic-L (cyclic-R), the respective expansion digraph must fully contain a leftmost (rightmost) cycle of the compound. In particular, it must contain all vertices and edges that are incident to a leftmost (rightmost) cycle.

Recall that terminal edges of dSPQR trees fulfill a transitivity property (Prop. 3.8). The same holds for compounds: In Fig. 3.46, the virtual edge in μ_2, refined by μ_1, is cyclic-L and its expansion digraph contains a leftmost cycle. Thus, also the expansion digraph of the virtual edge in μ_3, refined by μ_2, contains a leftmost cycle and it is also cyclic-L.

Proposition 3.9 (Transitivity of Cyclic Edges). *In a dSPQR tree of an embedded and biconnected compound, let μ be a node containing a virtual edge e that is refined by node μ'. If any virtual edge $e' \neq e$ in μ is a cyclic-L (cyclic-R) edge, then so is virtual edge e in μ'.*

As with the acyclic version of dSPQR trees, each virtual edge of a compound's dSPQR tree is either directed or cyclic.

Lemma 3.25. *In the dSPQR tree of an embedded compound, each virtual edge is either directed or cyclic.*

Proof. Let γ be an embedded compound with dSPQR tree $\vec{\mathcal{T}}$ and let $e = \{u, v\}$ be a virtual edge in a node μ. As γ is strongly connected, the only vertices that can be terminals in the expansion digraph $\text{expg}_\mu(e)$ are u and v, where not both u and v can be a source (sink). If u is a source and v a sink, then $\text{expg}_\mu(e)$ is a uv-digraph and e is directed from u to v. Accordingly, if v is a source and u a sink, $\text{expg}_\mu(e)$ is a vu-digraph and e is directed from v to u. Otherwise, at least one of u or v is no terminal and $\text{expg}_\mu(e)$ contains a cycle C. Let μ^* be

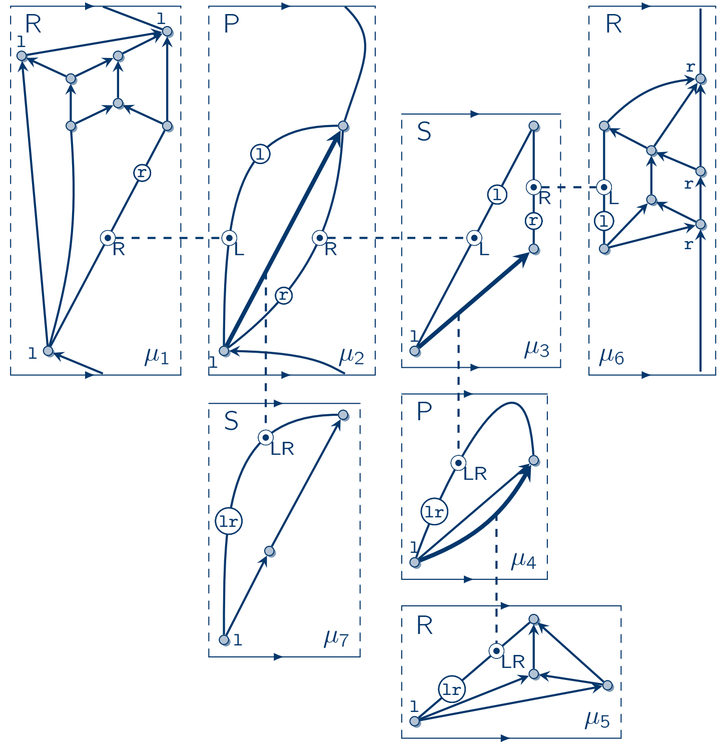


Figure 3.46: dSPQR tree of the biconnected compound in Fig. 3.44(a).

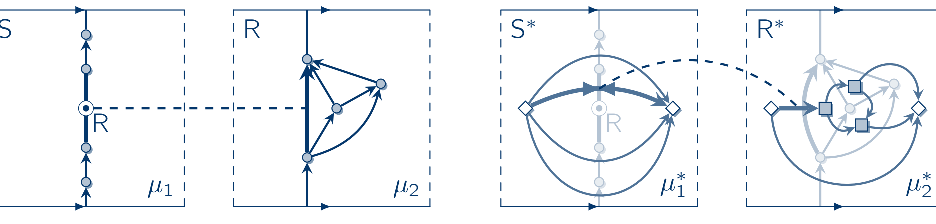
the dual of μ in the dSPQR tree $\vec{\mathcal{T}}^*$ of the dual γ^*. In the expansion digraph $\operatorname{expg}_{\mu^*}(e^*)$ of the dual, cycle C defines a dicut, which partitions the set of faces of $\operatorname{expg}_{\mu^*}(e^*)$ into subsets $F^{\mathtt{l}}$ and $F^{\mathtt{r}}$. At least one of $F^{\mathtt{l}}$ or $F^{\mathtt{r}}$ must contain a terminal as γ^* is acyclic. Which of $F^{\mathtt{l}}$ or $F^{\mathtt{r}}$ contains a terminal depends on the orientation of C. W. l. o. g. $F^{\mathtt{l}}$ contains a terminal and so does $\operatorname{expg}_{\mu^*}(e^*)$. Hence, $\operatorname{expg}_{\mu}(e)$ contains a left- or rightmost cycle of the compound and e is cyclic in μ. □

From the proof of Lem. 3.25, we also obtain the following, which will be of use later in our algorithm:

Corollary 3.22. *If the expansion digraph* $\operatorname{expg}_{\mu}(e)$ *of a virtual edge* e *in node* μ *contains a cycle, then* $\operatorname{expg}_{\mu}(e)$ *also contains a left- or rightmost cycle, i. e.,* e *is cyclic.*

3.5.3.6 Dual dSPQR Trees of Compounds

As the proof of Lem. 3.25 suggests, there is a duality between dSPQR trees of compounds and the dSPQR trees of their duals. For an example, consider the dSPQR tree of the embedded compound in Fig. 3.47(a). There, μ_1 contains a cyclic-R edge which is refined by μ_2 which, in turn, contains a directed virtual edge. Fig. 3.47(b) shows the dSPQR tree of the dual, where in each skeleton the primal is shown semi-transparent behind the dual. All non-virtual edges are directed as in the duals of digraphs. The same also applies to the directed virtual edges in μ_2 and μ_2^*: The dual of the directed virtual edge in μ_2^* points from the face to the left of its primal to the face on its right side. Moreover, the dual of the cyclic-R edge in μ_1 is a sink edge in μ_1^*. We show that this duality holds true in general:



(a) The dSPQR tree of a compound.

(b) The dSPQR tree of the dual of Fig. 3.47(a).

Figure 3.47: The dSPQR tree of an embedded compound and of its dual.

Lemma 3.26. *Let γ be an embedded compound with dSPQR tree $\vec{\mathcal{T}}$ which contains node μ. In the dSPQR tree of γ^*, let μ^* be the corresponding dual node of μ. Let e be an edge between vertices u and v in $\overrightarrow{\text{skel}}(\mu)$ and, in $\overrightarrow{\text{skel}}(\mu^*)$, let e^* be the respective dual edge. For e and e^*, exactly one of the following two statements holds true:*

(i) **Edge e is directed.**
Let f and g be the faces to the left and right of e, respectively. Edge e^* is directed from f to g in $\overrightarrow{\text{skel}}(\mu^*)$.

(ii) **Edge e is cyclic-L (cyclic-R).**
Edge e^* is a source (sink) edge.

Before we prove this lemma, we recall some concepts from the theory of graph minors: In a connected graph G, contracting an edge means identifying its vertices. If G is embedded, then contracting an edge e corresponds to removing the corresponding dual e^* from G^*. Conversely, deleting an edge from G corresponds to an edge contraction in G^*. Consider a directed virtual edge e in a dSPQR tree and its expansion digraph which is a uv-digraph that contains a dipath p from u to v. Removing all edges in the expansion digraph that are not on p and, afterwards, contracting all edges of p except for one yields e itself, i. e., e itself is a minor of its expansion digraph. Using these observations, we prove Lem. 3.26.

Proof. [Lem. 3.26] If e is non-virtual, then e^* is directed from the face to the left of e to the face to the right of e by the definition of the duals of digraphs and (i) follows for non-virtual edges. From now on, we assume that e is virtual. By Lem. 3.25, we know that e is either directed or cyclic. Assume that f and g are the endpoints of e^* in μ^*. If e is, say, cyclic-L, then the expansion digraph $\text{expg}_\mu(e)$ contains a leftmost cycle of the compound and, in the dual, $\text{expg}_{\mu^*}(e^*)$ there is a source distinct from f and g and, therefore, e^* is a source edge in μ^*. Conversely, if e^* is a source edge, there is a source in $\text{expg}_{\mu^*}(e^*)$ distinct from f and g. Consequently, $\text{expg}_\mu(e)$ contains a leftmost cycle of the compound and, hence, e is cyclic-L in μ. The reasoning for cyclic-R edges and sink edges is analogous. Thus, statement (ii) follows. The reasoning for statement (i) is as follows.

$\Rightarrow$: As e is directed, we know by Lem. 3.25 that it is not cyclic and, thus, e^* is no terminal edge. By Prop. 3.7, e^* must also be directed. W. l. o. g., we assume that $e = (u, v)$ and, thus, $\text{expg}_\mu(e)$ is an acyclic dipole with source u and sink v. Further, $e^* = (u^*, v^*)$, where u^* is the

source and v^* is the sink of $\text{expg}_{\mu^*}(e^*)$. We have to show that e^* is directed from f to g, i. e., $u^* = f$ and $v^* = g$. First note that for every edge in $\text{expg}_\mu(e)$ there is exactly one dual edge in $\text{expg}_{\mu^*}(e^*)$. In $\text{expg}_\mu(e)$, there is a dipath $p = u \rightsquigarrow v$. Let $\hat{e}$ be any edge on p with respective dual $\hat{e}^*$. Edge $\hat{e}^*$ points from a face $\hat{f}$ to a face $\hat{g}$ in $\text{expg}_{\mu^*}(e^*)$. In particular, $\hat{f}$ is left and $\hat{g}$ is right of $\hat{e}$. As $\text{expg}_{\mu^*}(e^*)$ is a u^*v^*-digraph, there is a dipath $p^* = u^* \rightsquigarrow \hat{f} \rightarrow \hat{g} \rightsquigarrow v^*$. By subsequently contracting and removing edges, we can transform $\text{expg}_\mu(e)$ into a digraph which consists solely of edge $\hat{e}$. Edge $\hat{e}$ then corresponds to virtual edge e in the skeleton of μ. For the dual $\text{expg}_{\mu^*}(e^*)$, we also obtain the single edge $\hat{e}^*$, which corresponds to the virtual edge e^* in the skeleton of μ^*. Hence, we can conclude that there is a dipath from f to g in $\text{expg}_{\mu^*}(e^*)$ and, consequently, $f = u^*$ is the source and $g = v^*$ is the sink of $\text{expg}_{\mu^*}(e^*)$. Then, virtual edge e^* must point from f to g.

$\Leftarrow$: The reasoning is analogous to before only that the roles of primal and dual swap. □

We use Lem. 3.26 as a motivation to define dual dSPQR trees.

Definition 3.9. *Let γ be an embedded compound with dSPQR tree $\vec{\mathcal{T}}$. The* dual dSPQR tree*, denoted by $\vec{\mathcal{T}}^*$, is obtained from $\vec{\mathcal{T}}$ by replacing the skeleton of each node μ by its directed dual to obtain the dual node μ^*, where cyclic-L edges are replaced by source edges and cyclic-R edges by sink edges.*

By our latest findings (Lem. 3.26) and since the dual SPQR tree is the SPQR tree of the dual (Thm. 3.4), we get the following theorem.

Theorem 3.5. *The dual dSPQR tree of an embedded compound is the dSPQR tree of the dual.*

The dual dSPQR-tree of the dSPQR-tree in Fig. 3.46 is shown in Fig. 3.45. We can give a commutative diagram that illustrates the relationship between embedded compounds, dSPQR trees and their duals. Let σ be the function that maps a compound or acyclic digraph to its dSPQR tree and let γ be an embedded compound.

$$\begin{array}{ccc} \gamma & \overset{*}{\rightarrow} & \gamma^* \\ \downarrow^{\sigma} & & \downarrow^{\sigma} \\ \vec{\mathcal{T}} & \overset{*}{\rightarrow} & \vec{\mathcal{T}}^* \end{array}$$

3.5.3.7 dSPQR Trees of RUP Compounds

With the help of dual dSPQR trees, we characterize **RUP** compounds by the embeddings of the skeletons of its dSPQR tree. Remember that an acyclic digraph is an acyclic dipole if and only if each of the skeletons of its dSPQR tree is an acyclic dipole by Lem. 3.24, and a skeleton is an acyclic dipole if its auxiliary skeleton is an acyclic dipole. We obtain the following characterization:

Corollary 3.23. *An embedding of a compound is* **RUP** *if and only if each node of its dual dSPQR tree is an acyclic dipole.*

Proof. Let $\vec{\mathcal{T}}^*$ be the dual dSPQR tree of an embedded compound γ. The embedding of γ is **RUP** if and only if γ^* is acyclic dipole, if and only if each node in the dSPQR tree of γ^* is an acyclic dipole. Since the dSPQR-tree of γ^* is the dual dSPQR tree of γ by Thm. 3.5, the statement follows. □

This motivates the following definition:

Definition 3.10. *Let μ be a node of a dSPQR tree of an embedded compound. The embedding of μ's skeleton $\overrightarrow{\text{skel}}(\mu)$ is called* **RUP***(-embedded) if its dual $\overrightarrow{\text{skel}}(\mu^*)$ is an acyclic dipole. In this case, node μ is called* **RUP***(-embedded).*

Hence, an embedding of a compound is **RUP** if and only if each node of its dSPQR tree is **RUP**; a result that we will refine in the following. Consider again the dSPQR tree shown in Fig. 3.46, where all nodes are is indeed **RUP**-embedded as their duals in Fig. 3.45 are acyclic dipoles.

By Cor. 3.23, we can bound the number of cyclic-L and cyclic-R edges in a node of the dSPQR tree of a **RUP**-embedded compound:

Corollary 3.24. *If a compound is* **RUP***-embedded, then the skeleton of each node in its dSPQR tree has at most one cyclic-L and at most one cyclic-R edge.*

Proof. If a skeleton contains two or more cyclic-L edges, then its dual contains at least two source edges and the dual cannot be an acyclic dipole. Therefore, the whole compound is not **RUP**-embedded. The same reasoning applies to cyclic-R edges. □

Note that a cyclic-LR edge counts as a cyclic-L and a cyclic-R edge.

Auxiliary Skeletons of RUP Nodes For dSPQR trees of acyclic digraphs, we have introduced auxiliary skeletons in which source and sink edges are replaced by actual sources and sinks. We apply the same idea to dSPQR trees of compounds, that is, we replace cyclic edges by actual cycles. Let μ be a node of the dSPQR tree of an embedded compound. The auxiliary skeleton $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ is obtained from $\overrightarrow{\text{skel}}(\mu)$ by replacing each cyclic edge $e = \{u, v\}$ by one or two pairs of antiparallel edges as shown in Fig. 3.48. A cyclic-L edge is replaced by a single pair of antiparallel edges which form an *(auxiliary) L-cycle* whose edges carry the label L. Likewise, a cyclic-R edge is replaced by an *(auxiliary) R-cycle* with label R. A cyclic-LR edge is replaced by both an L- and an R-cycle.

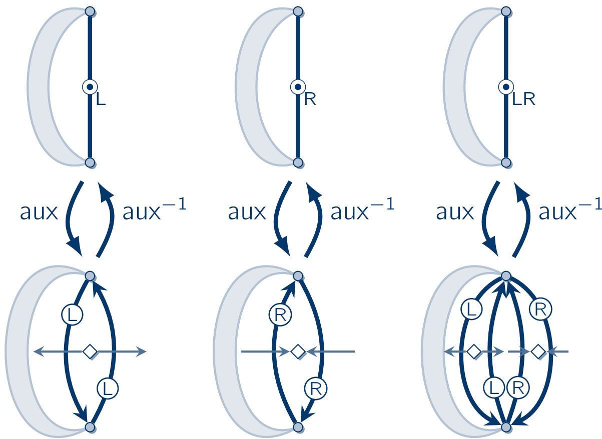


Figure 3.48: Replacements for cyclic edges in the auxiliary skeleton.

The idea is that, instead of directly finding the **RUP** embedding of a node's skeleton, we shift the problem to its auxiliary. For the auxiliary, we have to find a **RUP** embedding such that each L-cycle is leftmost and each R-cycle is rightmost, that is, an L-cycle encloses a source and

an R-cycle a sink in the dual. In turn, from the **RUP** embedding of the auxiliary, we obtain a **RUP** embedding of the original skeleton by replacing the L- and R-cycles by their original cyclic edges. The position of the original cyclic edge $e = \{u, v\}$ in the rotation systems of its endpoints is determined by the position of its auxiliary cycles in the rotation system of the auxiliary skeleton. For this step to be well defined, we need to make sure that in the embedding of the auxiliary skeleton the auxiliary edges of e are all consecutive in the rotation systems of u and v. Then, the position of e in the rotation system is uniquely defined by the position of the "bundle" of auxiliary edges in the auxiliary skeleton. This leads to the following definition:

Definition 3.11. *Let $\overrightarrow{\text{skel}}(\mu)$ be the skeleton of a node μ in the dSPQR tree of an embedded compound. An embedding of the auxiliary $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ is called* feasible *if it fulfills all of the following properties:*

(i) Every L-cycle is leftmost.

(ii) Every R-cycle is rightmost.

(iii) For any cyclic edge $e = \{u, v\}$, all of its auxiliary edges are consecutive in the rotation systems of u and v.

We define the function aux^{-1} (cf. Fig. 3.48) which maps a feasibly embedded auxiliary skeleton $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ to the respective embedded skeleton $\overrightarrow{\text{skel}}(\mu)$.

Remember, the idea is to obtain a **RUP** embedding of a node's skeleton from a feasible **RUP** embedding of its auxiliary. For an example, consider Fig. 3.49. In the bottom left corner, the auxiliary skeleton of μ_1 is feasibly **RUP**-embedded. In particular, its dual is an acyclic dipole. By applying aux^{-1}, we obtain an embedding of $\overrightarrow{\text{skel}}(\mu_1)$ in the top left corner. The dual of μ is shown in top right corner and its auxiliary $\text{aux}(\overrightarrow{\text{skel}}(\mu_1^*))$ is placed beneath. It is an acyclic dipole and, thereby, $\overrightarrow{\text{skel}}(\mu_1)$ is **RUP**-embedded. We prove this observation in general:

Lemma 3.27. *Let μ be a node of the dSPQR tree of an embedded compound. If a feasible embedding of an auxiliary skeleton $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ is* **RUP**, *then the embedding obtained by applying aux^{-1} to $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ is* **RUP**.

Proof. We show that if the dual $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$ of the auxiliary is an acyclic dipole, then so is $\overrightarrow{\text{skel}}(\mu^*)$. For the latter, we have to show that $\text{aux}(\overrightarrow{\text{skel}}(\mu^*))$ is an acyclic dipole. Let $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ be feasibly **RUP**-embedded. Its dual $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$ is an acyclic dipole with source s and sink t. First, suppose that there is a cyclic-L edge $e = \{u, v\}$ in $\overrightarrow{\text{skel}}(\mu)$ which is not cyclic-R. In $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$, there is a dipath from s to every other face, where s has two outgoing edges whose primals are the auxiliary edges of e (cf. Fig. 3.48). We assume that the outgoing edges of s point to faces f and g. Applying aux^{-1} to $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ means that first the two auxiliary edges of e are removed from $\text{aux}(\overrightarrow{\text{skel}}(\mu))$. In the dual, this operations corresponds to contracting the two auxiliary edges incident to s and, in particular, s is merged with f and g to a new face h. Afterwards, a cyclic-L edge is introduced in the primal between u and v to obtain $\overrightarrow{\text{skel}}(\mu)$ and, in the dual, we obtain $\overrightarrow{\text{skel}}(\mu^*)$, where h is split into two faces u^* and v^* with a source edge in between them. Since there is a dipath from s to any other face in $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$, there is a dipath from u^* or v^* to any other face in $\overrightarrow{\text{skel}}(\mu^*)$. In $\text{aux}(\overrightarrow{\text{skel}}(\mu^*))$, the source edge $\{u^*, v^*\}$ is replaced by a source s' with edges (s', u^*) and (s', v^*) and, again, there is a dipath from s' to any other face in $\text{aux}(\overrightarrow{\text{skel}}(\mu^*))$. Therefore, s' is the single source in $\text{aux}(\overrightarrow{\text{skel}}(\mu^*))$.

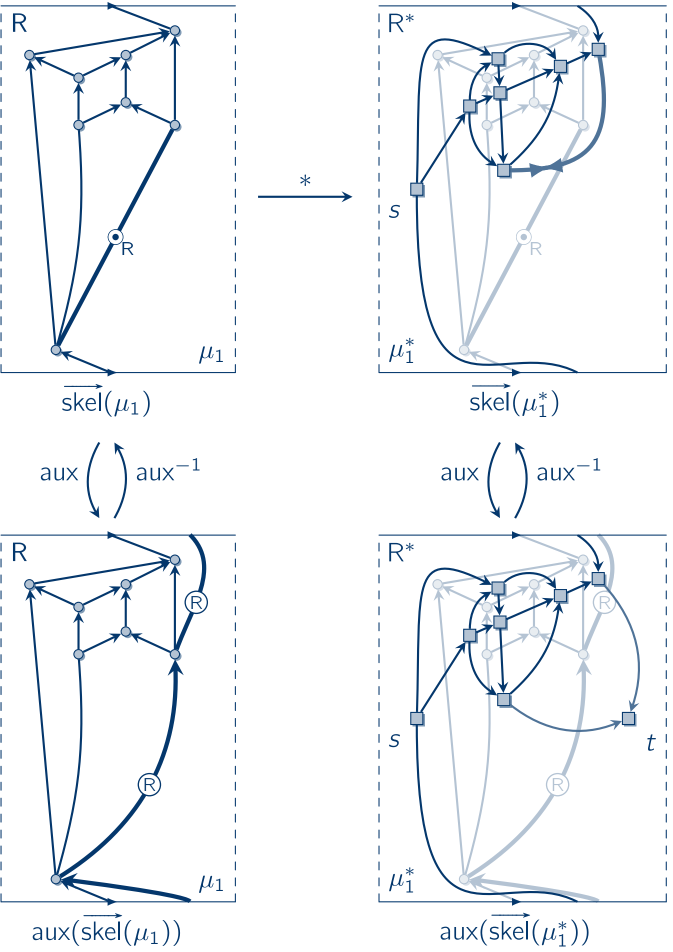


Figure 3.49: The primal and dual skeletons of μ_1 with their auxiliary skeletons.

If e is a cyclic-R, then $\text{aux}(\overrightarrow{\text{skel}}(\mu^*))$ contains a single sink by the symmetric reasoning. In case e is cyclic-LR, s is merged with four instead of two faces and one of these four faces is the single source and one the single sink of $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$ (see Fig. 3.48). The remaining reasoning is similar only that the cyclic-LR edge is replaced by an edge that is both a source and a sink edge. If μ contains no cyclic-L or cyclic-LR edge, then its dual contains a (non-auxiliary) single source in both $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$ and in $\text{aux}(\overrightarrow{\text{skel}}(\mu^*))$. Similarly, if μ neither contains a cyclic-R or cyclic-LR edge, there is a single sink in both. Altogether, we can conclude that $\text{aux}(\overrightarrow{\text{skel}}(\mu^*))$ is an acyclic dipole and, thus, the embedding of $\overrightarrow{\text{skel}}(\mu)$ is **RUP**. □

For our running example in Fig. 3.44(a) and its dSPQR tree in Fig. 3.46, all respective auxiliary skeletons are shown in Fig. 3.50, where the labels `l`, `r`, and `lr` can be ignored for the moment. All auxiliary skeletons are feasibly **RUP**-embedded and, hence, their embeddings obtained for the skeletons in Fig. 3.44(a) are **RUP**. From Lem. 3.27, we also obtain that in this case the whole compound can be **RUP**-embedded:

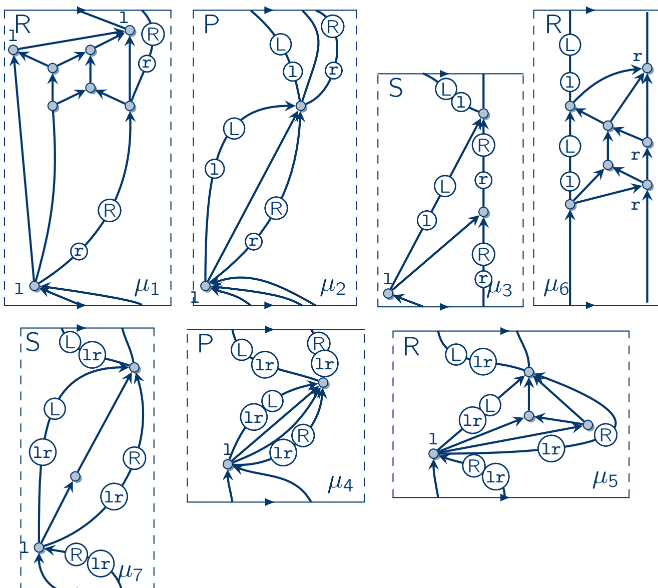


Figure 3.50: Auxiliary skeletons with feasible **RUP** embeddings for all nodes of the example in Fig. 3.46.

Corollary 3.25. *Let γ be an embedded compound with dSPQR tree $\vec{\mathcal{T}}$. If the auxiliary skeleton of each node of $\vec{\mathcal{T}}$ has a feasible* **RUP** *embedding, then applying* aux^{-1} *to every auxiliary yields a* **RUP** *embedding of each skeleton and, hence, a* **RUP** *embedding of γ.*

The converse version of Cor. 3.25 the following:

Lemma 3.28. *Let γ be an embedded compound with dSPQR tree $\vec{\mathcal{T}}$. If the whole compound is* **RUP***-embedded and, thus, every node is* **RUP***-embedded, then the auxiliary of each node's skeleton has a feasible* **RUP** *embedding.*

The reasoning for Lem. 3.28 is the converse reasoning of Lem. 3.27 where we additionally have to argue that the auxiliary indeed has a **RUP** embedding that is feasible. In particular,

the auxiliary edges have to be consecutive in the rotation system. To show this, we need the following lemma:

Lemma 3.29. *Let μ be a node of the dSPQR tree of a* **RUP***-embedded compound. If* $\overrightarrow{\text{skel}}(\mu)$ *contains a cyclic-LR edge e, then* $\overrightarrow{\text{skel}}(\mu) \setminus \{e\}$ *is an acyclic dipole.*

Proof. Let μ' be the node which refines edge e in μ. $\overrightarrow{\text{skel}}(\mu)$ contains the cyclic-LR edge e and, thus, all all other edges have to be directed since in a **RUP**-embedded compound a node can contain at most one cyclic-L and at most one cyclic-R edge (Cor. 3.24). For contradiction, suppose that $\overrightarrow{\text{skel}}(\mu) \setminus \{e\}$ contains a cycle or a cyclic edge. Then, virtual edge e in μ' is cyclic-L, cyclic-R, or cyclic-LR. Furthermore, by the assumption that e is cyclic-LR in μ, we can distinguish the following cases which follow from the transitivity of cyclic edges (Prop. 3.9):

- $\overrightarrow{\text{skel}}(\mu') \setminus \{e\}$ contains a cyclic-LR edge.
- $\overrightarrow{\text{skel}}(\mu') \setminus \{e\}$ contains a cyclic-L (cyclic-R) edge and a rightmost (leftmost) cycle.
- $\overrightarrow{\text{skel}}(\mu') \setminus \{e\}$ contains no cyclic edge and both a left- and a rightmost cycle.

In either case, however, $\overrightarrow{\text{skel}}(\mu'^*)$ cannot be an acyclic dipole as it either contains more than one source or more than one sink which contradicts the assumption that the compound is **RUP**-embedded. Hence, $\overrightarrow{\text{skel}}(\mu) \setminus \{e\}$ is acyclic and, since the compound is strongly connected, the endpoints of e must be the single source and single sink. □

Proof. [Lem. 3.28] The following reasoning the reversed version of the proof of Lem. 3.27, where we additionally show that the **RUP** embedding of the auxiliary skeleton is feasible. Let $\overrightarrow{\text{skel}}(\mu)$ be the **RUP**-embedded skeleton of a node μ in the dSPQR tree of a **RUP**-embedded compound. Consider the dual node μ^* and its skeleton $\overrightarrow{\text{skel}}(\mu^*)$. We first tackle the case where $\overrightarrow{\text{skel}}(\mu)$ contains no cyclic-LR edge. Assume that $\overrightarrow{\text{skel}}(\mu^*)$ contains a source edge $e^* = \{f, g\}$. We have to show that if $\text{aux}(\overrightarrow{\text{skel}}(\mu^*))$ contains a single source, then so does $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$. By assumption, $\text{aux}(\overrightarrow{\text{skel}}(\mu^*))$ is an acyclic dipole with source s and sink t, and there is a dipath $s \rightsquigarrow h$ from s to every other face in $\text{aux}(\overrightarrow{\text{skel}}(\mu^*))$. Hence, in $\overrightarrow{\text{skel}}(\mu^*)$ for every face h, there is a dipath from f to h or from g to h. The primal of e^* is a cyclic-L edge $e = \{u, v\}$ in $\overrightarrow{\text{skel}}(\mu)$. As shown in Fig. 3.48, e is replaced in $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ by an L-cycle, where the antiparallel pair of edges inherits the position of e in rotation systems of u and v in $\overrightarrow{\text{skel}}(\mu)$ such that the L-cycle is leftmost. In the dual $\overrightarrow{\text{skel}}(\mu^*)$, this operation corresponds to introducing a source s with two outgoing edges to the faces f and g, and we thereby obtain the dual of the auxiliary, i. e., $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$. As there is a dipath from f or g in $\overrightarrow{\text{skel}}(\mu^*)$ to every other face, there is dipath from s to every other face in $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$. s is therefore the single source in $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$. Also note that the since the L-cycle is leftmost, the auxiliary edges of e must be consecutive in the rotation systems of u and v as the L-cycle encloses s. If e is a cyclic-R edge, the symmetric reasoning shows that $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$ contains a single sink and the auxiliary edges of e are consecutive. Also, as in the proof of Lem. 3.27, if $\overrightarrow{\text{skel}}(\mu)$ neither contains a cyclic-L nor a cyclic-LR edge, then its dual contains a (non-auxiliary) single source in both $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$ and in $\text{aux}(\overrightarrow{\text{skel}}(\mu^*))$. Similarly, if μ neither contains a cyclic-R or cyclic-LR edge, there is a single sink in both. Altogether, if e is no LR-edge, we can conclude that $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$ has a feasible **RUP** embedding.

If $e = \{u, v\}$ is a cyclic-LR edge, then $\overrightarrow{\text{skel}}(\mu) \setminus \{e\}$ is an acyclic dipole by Lem. 3.29, where, w. l. o. g., u is the source and v the sink. The situation we obtain in shown to the left in

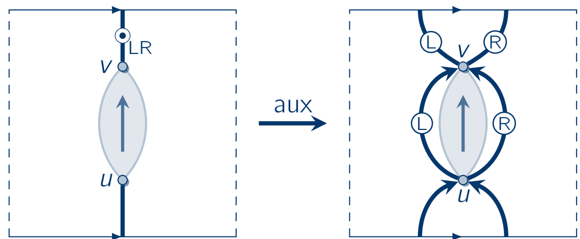


Figure 3.51: If e is cyclic-LR, the remainder of the skeleton is an acyclic dipole.

Fig. 3.51, where the acyclic dipole is symbolized by the arrow in the shaded area. Remember, $\overrightarrow{\mathsf{skel}}(\mu)$ is **RUP**-embedded and, especially, embedded. Therefore, $\overrightarrow{\mathsf{skel}}(\mu)$ is an embedded acyclic dipole with an edge that connects its source with its sink and, thus, its embedding is **UP**, that is, upward planar in the plane [Kel87, DBT88]. In particular, we can introduce the L- and R-cycle corresponding to the cyclic-LR edge e as depicted on the right side of Fig. 3.51, where the L-cycle encloses a source and the R-cycle a sink in the dual. As there is no edge from v to u in $\overrightarrow{\mathsf{skel}}(\mu) \setminus \{e\}$, all four auxiliary edges in $\mathsf{aux}(\overrightarrow{\mathsf{skel}}(\mu))$ are consecutive. Thus, the obtained embedding is feasible and **RUP**. □

In the following, we assume that if the skeletons of the nodes are **RUP**-embedded, then the auxiliary of the skeletons are embedded according to the proof of Lem. 3.28. That is, L-cycles are leftmost, R-cycles are rightmost, and for cyclic-LR edges we have the embedding as shown in Fig. 3.51. We obtain the following theorem that wraps up all previous sections about dSPQR trees and their duals, and paves the way for our testing algorithm.

Theorem 3.6. *For any embedded and biconnected compound γ with dSPQR tree $\vec{\mathcal{T}}$ and dual dSPQR tree $\vec{\mathcal{T}}^*$, the following statements are equivalent:*

(i) The embedding of γ is **RUP**.

(ii) Each node of $\vec{\mathcal{T}}$ is **RUP**.

(iii) The auxiliary skeleton of each node of $\vec{\mathcal{T}}$ has a feasible **RUP** *embedding.*

(iv) The dual of γ is an acyclic dipole.

(v) Each node of γ's dual dSPQR tree is an acyclic dipole, i. e., its auxiliary skeleton is an acyclic dipole.

Proof. Follows from Lemmas 3.4 and 3.28 and Corollaries 3.21, 3.23 and 3.25. □

3.5.3.8 The Algorithm for Biconnected Compounds

During the last sections, we have forged the tools we now use to devise our testing algorithm. As input, the algorithm receives a planar and biconnected compound $\gamma = (V, E)$, and two sets $V^{\mathtt{l}}, V^{\mathtt{r}} \subseteq V$. The output is either a **RUP** embedding, where all vertices in $V^{\mathtt{l}}$ and $V^{\mathtt{r}}$ are left- and rightmost, respectively, or $\perp$ if no such embedding exists. We use the planar compound in Fig. 3.52 as a running example, where vertices in $V^{\mathtt{l}}$ are labeled with $\mathtt{l}$ and vertices in $V^{\mathtt{r}}$ with $\mathtt{r}$.

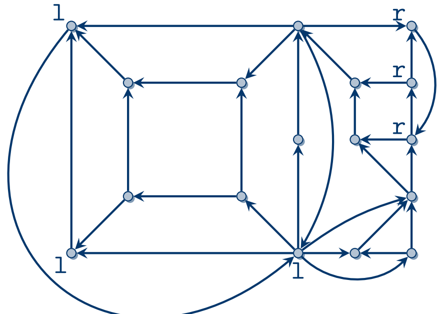


Figure 3.52: Example of a planar compound.

Fig. 3.53 shows a preliminary version of the dSPQR-tree, the *pre-dSPQR tree*, of the compound in Fig. 3.52. The pre-dSPQR tree is equal to the dSPQR tree only that its cyclic edges do not carry the labels L, R, or LR. Remember, the information that a cyclic edge is cyclic-L or -R presumes that the compound is embedded (cf. Def. 3.8). However, as the task of our algorithm is to find a **RUP** embedding, we do not know yet whether a virtual edge is cyclic-L or cyclic-R. Instead, the virtual edges in Fig. 3.53 are either directed if the expansion digraph is an acyclic dipole, or cyclic if the expansion digraph contains a cycle. Observe, these attributes of virtual edges do not presume an embedding.

dSPQR Trees with Vertices that must be Left-/Rightmost Our first step in this section is to adapt dSPQR trees to accommodate for vertices that must be left- and rightmost. Let $\vec{\mathcal{T}}$ be the dSPQR tree of an embedded compound and let μ be one of $\vec{\mathcal{T}}$'s nodes with skeleton $\overrightarrow{\mathsf{skel}}(\mu) = (V_\mu, E_\mu)$. A vertex in $v \in V^{\mathtt{l}}$ is either present explicitly in the skeleton if $v \in V_\mu$ or implicitly whenever v belongs to the expansion digraph of one of the skeleton's virtual edges. For instance, node μ_1 in Fig. 3.53, contains three vertices from $V^{\mathtt{l}}$, where one belongs to a split pair and, thus, reappears in nodes μ_2, μ_3, μ_4, μ_5, and μ_7. In contrast, node μ_6 does not explicitly contain any vertex from $V^{\mathtt{l}}$. However, the expansion digraph of its virtual edge contains all vertices from $V^{\mathtt{l}}$, indicated by the label l. In general, we define for each node μ two sets $\mathbf{E}^{\mathtt{l}}(\mu), \mathbf{E}^{\mathtt{r}}(\mu) \subseteq E_\mu$ of virtual edges. A virtual edge $e = \{u, v\}$ is in $\mathbf{E}^{\mathtt{l}}(\mu)$ if and only if the expansion digraph $\mathsf{expg}_\mu(e)$ contains a vertex from $V^{\mathtt{l}}$ distinct from u and v. Likewise, $e = \{u, v\} \in \mathbf{E}^{\mathtt{r}}(\mu)$ if and only if $\mathsf{expg}_\mu(e) \setminus \{u, v\}$ contains a vertex from $V^{\mathtt{r}}$. In the following, all virtual edges in $\mathbf{E}^{\mathtt{l}}(\mu)$ and $\mathbf{E}^{\mathtt{r}}(\mu)$ are labeled with l and r, respectively. If a virtual edge is in both sets, its label is lr.

The virtual edge in μ_7 is refined by μ_2 which contains two cyclic virtual edges labeled with l and r. Hence, the expansion digraph of the virtual edge in μ_7 contains vertices from $V^{\mathtt{l}}$ and $V^{\mathtt{r}}$ and, thus, is labeled with lr. Hence, as with terminal and cyclic edges, we also have a property of transitivity here:

Proposition 3.10. *In a dSPQR tree of an embedded compound, let μ be a node containing a virtual edge e, with endpoints u and v, that is refined by node μ' and let $V^{\mathtt{l}}$ and $V^{\mathtt{r}}$ be sets of vertices that must lie left- and rightmost, respectively. For node μ, let $\mathbf{E}^{\mathtt{l}}(\mu)$ and $\mathbf{E}^{\mathtt{r}}(\mu)$ be the set of virtual edges that must lie left- and rightmost respectively. Likewise, we have $\mathbf{E}^{\mathtt{l}}(\mu')$ and $\mathbf{E}^{\mathtt{r}}(\mu')$ for node μ'. If μ contains any vertex in $V^{\mathtt{l}}$ ($V^{\mathtt{r}}$) distinct from u and v or any virtual edge $e' \neq e$ with $e' \in \mathbf{E}^{\mathtt{l}}(\mu)$ ($e' \in \mathbf{E}^{\mathtt{r}}(\mu)$), then $e \in \mathbf{E}^{\mathtt{l}}(\mu')$ ($e \in \mathbf{E}^{\mathtt{r}}(\mu')$).*

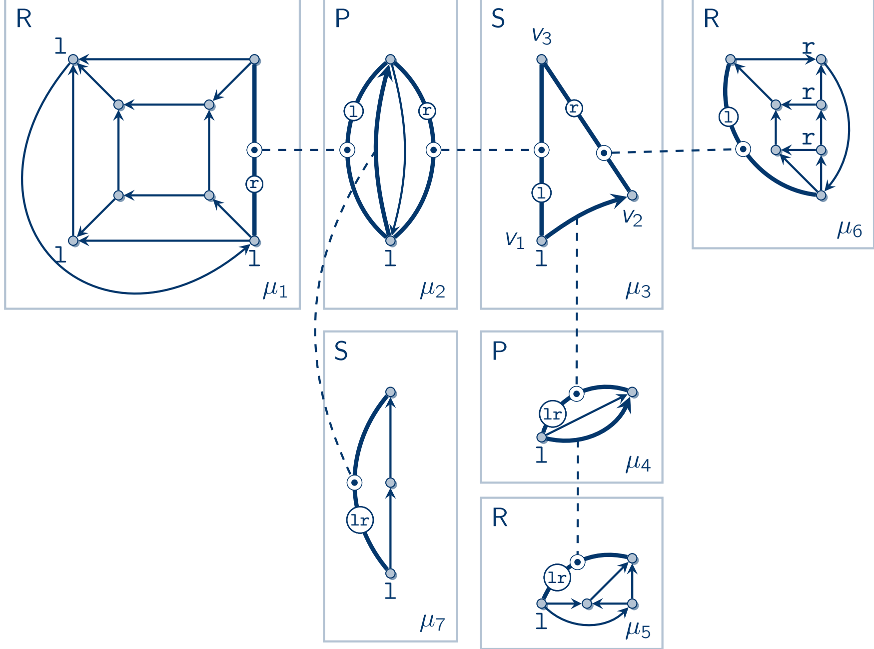
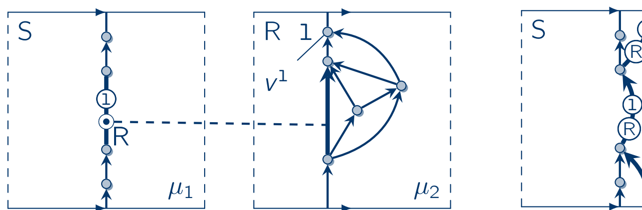
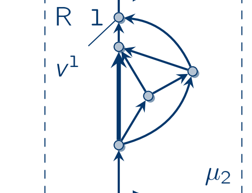


Figure 3.53: Pre-dSPQR tree of the example in Fig. 3.52.

Remember, in the auxiliary of a skeleton $\overrightarrow{\mathsf{skel}}(\mu)$, a cyclic-L or cyclic-R edge is replaced by an antiparallel pair of edges, and a cyclic-LR edge by two pairs of antiparallel edges. In the following, we will see how the feasible **RUP** embeddings of the auxiliary skeletons must look like such that all vertices in $V^{\mathtt{l}}$ and $V^{\mathtt{r}}$ are left- and rightmost in the **RUP** embedding of the whole compound. For an example, consider Fig. 3.54(a) which shows an R node μ_2 which contains vertex $v^{\mathtt{l}} \in V^{\mathtt{l}}$. The directed virtual edge is refined by S node μ_1 which, in turn, contains a cyclic-R edge which is in $\mathbf{E}^{\mathtt{l}}(\mu_1)$. Note that the embedding of the whole compound, as defined by the embeddings of the two skeletons, is **RUP** where vertex $v^{\mathtt{l}}$ is leftmost. The first observation we gain from this example is that a cyclic-R can contain the label $\mathtt{l}$. For the second observation, first remember that the idea of our algorithm is to find a **RUP** embedding of the compound by finding feasible **RUP** embeddings of the auxiliary skeletons. Fig. 3.54(b) shows the auxiliary skeletons where each is feasible **RUP**-embedded. Note that the auxiliary R-cycle in μ_1 is rightmost. The following conditions must hold in order for $v^{\mathtt{l}}$ to be leftmost in the whole **RUP** embedding. Vertex $v^{\mathtt{l}}$ must be leftmost in $\mathsf{aux}(\overrightarrow{\mathsf{skel}}(\mu_2))$ and, at least one of the two auxiliary edges of the R-cycle in μ_1 must be leftmost in $\mathsf{aux}(\overrightarrow{\mathsf{skel}}(\mu_1))$. Note that if none of the auxiliary edges of the R-cycle is leftmost, then there would be a leftmost cycle in μ_1 that encloses $v^{\mathtt{l}}$ in the whole **RUP** embedding such that $v^{\mathtt{l}}$ cannot be leftmost. Intuitively, a single leftmost auxiliary edge of an R-cycle guarantees that there there is a "gap" which ensures that the vertices in $V^{\mathtt{l}}$ are leftmost. The following lemma generalizes this observation. In the following, we denote by $\mathsf{aux}(e)$ the set of edges by which e is replaced in the auxiliary skeleton. That is, if e is a directed virtual edge, then $\mathsf{aux}(e) = \{e\}$, and if e is cyclic, it contains either one or two pairs of antiparallel edges depending on whether it is cyclic-L, cyclic-R, or cyclic-LR.



(a) The R node μ_2 contains vertex $v^1 \in V^1$ and, therefore, the cyclic-R edge in S node μ_1 is in $\mathbf{E}^1(\mu_1)$



(b) The auxiliaries of the skeletons in Fig. 3.54(a).

Figure 3.54: An R node with vertex v^1 that must be leftmost and an S node which contains a cyclic-R edge which is in $\mathbf{E}^1(\mu_1)$.

Lemma 3.30. *The embedding of a biconnected compound $\gamma = (V, E)$ is* **RUP***, where the vertices in $V^1 \subseteq V$ and $V^r \subseteq V$ are left- and rightmost, respectively, if and only if the auxiliary skeleton* $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu))$ *of each node μ of γ's dSPQR tree is feasibly* **RUP***-embedded with the following properties:*

(i) All vertices in V^1 (V^r) that are also in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu))$ *are leftmost (rightmost).*

(ii) For each virtual edge $e \in \mathbf{E}^1(\mu)$, at least one edge of $\mathrm{aux}(e)$ *is leftmost.*

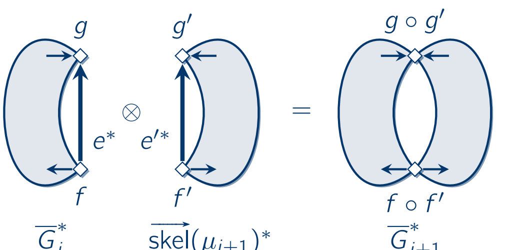
(iii) For each virtual edge $e \in \mathbf{E}^r(\mu)$, at least one edge of $\mathrm{aux}(e)$ *is rightmost.*

Proof. By Thm. 3.6, we know that γ is **RUP**-embedded if and only if the skeletons of the nodes are **RUP**-embedded and their auxiliary skeletons are feasibly **RUP**-embedded. Next, we show that properties (i) to (iii) are also fulfilled.

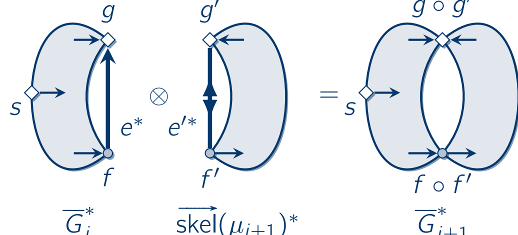
$\Leftarrow$: Let $\mu_1, \dots, \mu_k$ be a node sequence of γ's dSPQR tree $\vec{\mathcal{T}}$ which induces the node series $\overline{G}_1, \dots, \overline{G}_k$ with $\overline{G}_i = \overline{G}_{i-1} \otimes \overrightarrow{\mathrm{skel}}(\mu_i)$, $\overline{G}_1 = \overrightarrow{\mathrm{skel}}(\mu_1)$ and $\overline{G}_k = \gamma$. Note that any $\overline{G}_i$ with $i < k$ has exactly one virtual edge e that is also in the skeleton of μ_i. We adopt the definition of auxiliaries for $\overline{G}_i$, i. e., $\mathrm{aux}(\overline{G}_i)$ is the auxiliary of $\overline{G}_i$. Further, we assume that each $\mathrm{aux}(\overline{G}_i)$ is endowed with a feasible **RUP** embedding as each auxiliary skeleton of $\mu_1, \dots, \mu_k$ has a feasible **RUP** embedding. By mathematical induction, we show that all $\mathrm{aux}(\overline{G}_i)$ fulfill properties (i) to (iii) and, ultimately, the embedding of $\mathrm{aux}(\overline{G}_i) = \overline{G}_i = \gamma$, which has no virtual edges, fulfills property (i).

By assumption, $\mathrm{aux}(\overline{G}_1)$ fulfills all properties and the base case follows. For the induction step, $\mathrm{aux}(\overline{G}_i)$ and $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1}))$ fulfill all properties. As in the proof of Lem. 3.24 and for clarity, we denote by e the virtual edge in $\overline{G}_i$ that is refined by μ_{i+1} and by e' the corresponding virtual edge in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$. The digraph $\overline{G}_{i+1} = \overline{G}_i \otimes \overrightarrow{\mathrm{skel}}(\mu_{i+1})$ is the (e, e')-two-clique sum of $\overline{G}_i$ and $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$. By De Morgan's Law for two-clique sums (Lem. 3.23), we get for the dual $\overline{G}^*_{i+1} = \overline{G}^*_i \boxtimes \overrightarrow{\mathrm{skel}}(\mu_{i+1})^*$ which is the (e^*, e'^*)-two-clique sum of $\overline{G}^*_i$ and $\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*$ (see Figs. 3.37(a) and 3.37(b) on page 167), where e^* and e'^* are the duals of e and e', respectively. Denote by f, g, and f', g' the faces that are the endpoints of e^* and e'^*, respectively. In $\overline{G}^*_{i+1}$, f and f' are replaced by a new face $f \circ f'$, and g and g' by $g \circ g'$. In the proof of Lem. 3.24 (page 171), we have shown that an acyclic digraph is a dipole if and

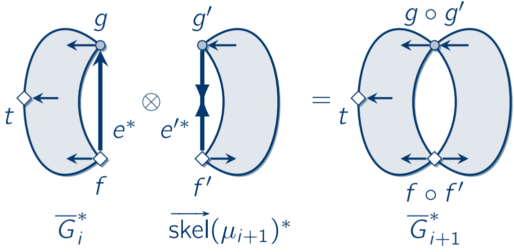
only if each of the nodes of its dSPQR tree is an acyclic dipole. Here, we import some of the insights of the proof of Lem. 3.24, where in the following the dual nodes are the acyclic dipoles. For illustration purposes, we use Fig. 3.55 which is the corresponding version of Fig. 3.43 on page 175 used in the proof of Lem. 3.24 with the difference that all figures display the duals $\overline{G}_i^*$, $\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*$, and $\overline{G}_{i+1}^*$, and e (e') is exchanged with e^* (e'^*), and the vertices are replaced by the faces f, g, f', g', $f \circ f'$, and $g \circ g'$. From the proof of Lem. 3.24, we also obtain that in $\mathrm{aux}(\overline{G}_i^*)$ and $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$ both f and f' are either a source, a sink, or none of both. Further, if f and f' are a source, a sink, or none of both, then $f \circ f'$ is a source, a sink, or none of both, respectively, in $\mathrm{aux}(\overline{G}_{i+1}^*)$. The same holds for g, g' and $g \circ g'$.



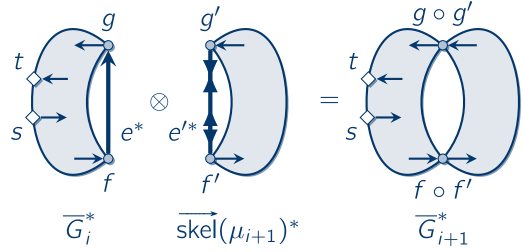
(a) Both e^* and e'^* are directed.



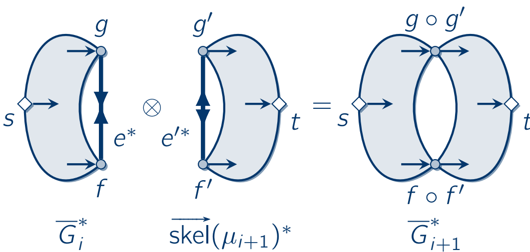
(b) Virtual edge e^* is directed and e'^* is a source but no sink edge.



(c) Virtual edge e^* is directed and e'^* is a sink but no source edge.



(d) Virtual edge e^* is directed and e'^* is a source and sink edge.



(e) Virtual edge e^* is a sink but no source edge and e'^* a source but no sink edge.

Figure 3.55: Case differentiation as obtained in the second part of the proof of Lem. 3.30.

We first show that $\overline{G}_{i+1}$ fulfills property (i). Let v^1 be a vertex in V^1. If v^1 is in both $\overline{G}_i$ and $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$, then v^1 is an endpoint of e and e'. If f is the single source in $\mathrm{aux}(\overline{G}_i^*)$, then so is f' in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$ and $f \circ f'$ in $\mathrm{aux}(\overline{G}_{i+1}^*)$ (cf. Figs. 3.55(a) and 3.55(c)). As v is leftmost in $\overline{G}_i$ and $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$, v is incident to f and f'. Thus, v is also incident to $f \circ f'$ and, thus, leftmost in $\overline{G}_{i+1}$. The same holds true if g is the single source in $\overline{G}_i^*$. Now suppose that neither of f, g, f', and g' is a source. In this case, either e^* or e'^* is a source edge. W. l. o. g., assume that e'^* is the source edge (cf. Figs. 3.55(b), 3.55(d) and 3.55(e)) and, by Lem. 3.26, e' is cyclic-L in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$. This also implies that e^* is no source edge and e is not cyclic-L. Consequently, there is a face h in $\overline{G}_i^*$ distinct from f and g that is a source in $\mathrm{aux}(\overline{G}_i^*)$ and v^1 is incident to h. During the two-clique summation h stays unchanged and, hence, h is also in $\overline{G}_{i+1}^*$ and v^1 is incident to h. Therefore, v^1 is leftmost in $\mathrm{aux}(\overline{G}_{i+1})$.

Now, suppose that v^1 is only in one of $\overline{G}_i$ and $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ and, thus, v^1 is no endpoint of e and e'. W. l. o. g., v^1 is in $\overline{G}_i$ and, thereby, $e' \in \mathbf{E}^1(\mu_{i+1})$. By property (ii), at least one auxiliary edge of $\mathrm{aux}(e')$ is leftmost in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1}))$. Since the embedding of $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1}))$ is feasible, all edges in $\mathrm{aux}(e')$ are consecutive and consequently one of f' and g' is a source in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$. We assume that f' is the source; the reasoning if g' is the source is similar. As f' is a source, so are f in $\mathrm{aux}(\overline{G}_i^*)$ and $f \circ f'$ in $\mathrm{aux}(\overline{G}_{i+1}^*)$. Since v^1 is leftmost in $\mathrm{aux}(\overline{G}_i)$, v^1 is incident to f and also incident to $f \circ f'$. Therefore, v^1 is leftmost in $\mathrm{aux}(\overline{G}_{i+1})$. The proof is analogous for vertices in V^{r} and, altogether, property (i) is fulfilled.

Next, we show that property (ii) is maintained. Note that during the two-clique summation, e and e' are replaced and they are not part of $\overline{G}_{i+1}$. If, for instance, $e' \in \mathbf{E}^1(\mu_{i+1})$, then property (ii) is vacuously true for $\overline{G}_{i+1}$. Now, suppose that there is a virtual edge $e^1 \neq e$ in $\overline{G}_i$ which must be leftmost, i. e., there is a node μ_j with $1 \leq j \leq i$ with $e^1 \in \mathbf{E}^1(\mu_j)$. Then, at least one auxiliary edge in $\mathrm{aux}(e^1)$ is leftmost in $\mathrm{aux}(\overline{G}_i^*)$. By transitivity (Prop. 3.10), $e' \in \mathbf{E}^1(\mu_{i+1})$ and at least one of $\mathrm{aux}(e')$ is leftmost in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$. Hence, either f' or g' is a source in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$. Again, we assume that f' is the source and, hence, so are f in $\mathrm{aux}(\overline{G}_i^*)$ and $f \circ f'$ in $\mathrm{aux}(\overline{G}_{i+1}^*)$. Since at least one of $\mathrm{aux}(e^1)$ is leftmost, we can conclude that e^1 is incident to f and, therefore, also incident to $f \circ f'$. This implies that at least one of $\mathrm{aux}(e^1)$ is leftmost in $\mathrm{aux}(\overline{G}_{i+1})$. Property (iii) is maintained by the same reasons. This concludes the inductive proof and, in particular, property (i) is fulfilled for the whole compound.

$\Rightarrow$: The following proof is the reversed version of the inductive proof from before. We start with $\overline{G}_k = \gamma$ and subsequently split the nodes' skeletons off the compound, i. e., for each $1 \leq i < k$ and $\overline{G}_{i+1} = \overline{G}_i \otimes \overrightarrow{\mathrm{skel}}(\mu_{i+1})$, we show that if $\mathrm{aux}(\overline{G}_{i+1})$ fulfills properties (i) to (iii), then so do $\mathrm{aux}(\overline{G}_i)$ and $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1}))$. Especially, all auxiliary skeletons fulfill all properties.

By assumption, $\mathrm{aux}(\overline{G}_k)$ fulfills property (i) and, since it contains no virtual edges, it vacuously also satisfies (ii) and (iii). For the induction step, assume that $\mathrm{aux}(\overline{G}_{i+1})$ $(1 \leq i < k)$ fulfills all properties and $\mathrm{aux}(\overline{G}_{i+1}) = \overline{G}_i \otimes \overrightarrow{\mathrm{skel}}(\mu_{i+1})$. We adopt the definitions from the proof of the "$\Leftarrow$"-direction, i. e., there is a virtual edge e in $\overline{G}_i$ refined by $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ which contains the corresponding edge e'. Further, we have the dual edges e^* and e'^* with endpoints f, g, and f', g', respectively, that are replaced in $\overline{G}_{i+1}$ by $f \circ f'$ and $g \circ g'$. Again, by the case differentiation in the proof of Lem. 3.24 (cf. Fig. 3.43 on page 175), if $f \circ f'$ is a source, a sink, or none of both in $\mathrm{aux}(\overline{G}_{i+1}^*)$, then f in $\mathrm{aux}(\overline{G}_i^*)$ and f' in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$ are both a source, a sink, or none of both, respectively. The same holds for $g \circ g'$, g, and g'.

By assumption, a vertex $v^1 \in V^1$ is leftmost in $\mathrm{aux}(\overline{G}_{i+1})$. First, assume that v^1 is an endpoint of e and e'. If $f \circ f'$ is a source in $\mathrm{aux}(\overline{G}_{i+1})$, then so are f and f'. Since v^1 is incident to $f \circ f'$, it is also incident to f and f' and, therefore, v^1 is leftmost in $\mathrm{aux}(\overline{G}_i)$ and

$\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$. The reasoning is analogous if $g \circ g'$ is the source. Now assume that v^1 is in $\overline{G}_i$ but not in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ and, thus, $e' \in \mathbf{E}^1(\mu_{i+1})$. We distinguish three cases: In $\mathrm{aux}(\overline{G}^*_{i+1})$, either $f \circ f'$, $g \circ g'$, or none of both is the source. If $f \circ f'$ is the source, then so are f and f'. Since v^1 is incident to $f \circ f'$, it is also incident to f in $\overline{G}_i$ and, hence, also to the source in $\mathrm{aux}(\overline{G}^*_i)$, i. e., $\mathrm{aux}(\overline{G}_i)$ fulfills property (i). Furthermore, edge e' is incident to f' and, hence, at least one of $\mathrm{aux}(e')$ is incident to the source of $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$, i. e., $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1}))$ fulfills property (ii). If $g \circ g'$ is the source, the reasoning is similar.

Now, suppose that none of $f \circ f'$ and $g \circ g'$ is the source . This implies that there is a face h distinct from $f \circ f'$ and $g \circ g'$ in $\overline{G}^*_{i+1}$ such that h is the source in $\mathrm{aux}(\overline{G}^*_{i+1})$ and v^1 is incident to h. Either h is in $\mathrm{aux}(\overline{G}^*_i)$ or in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$. First, assume that h is in $\mathrm{aux}(\overline{G}^*_i)$ (Figs. 3.55(b), 3.55(d) and 3.55(e)). Since $h \neq f \circ f'$ and $h \neq g \circ g'$, face h stays unchanged during the two-clique summation and it is also a source in $\mathrm{aux}(\overline{G}^*_i)$ to which v^1 is incident which implies property (i) for $\mathrm{aux}(\overline{G}_i)$. Further, the expansion digraph of e'^* contains h as source and, by the definition of dSPQR trees, e'^* is a source edge. Duality implies that e' is cyclic-L. In particular, all of $\mathrm{aux}(e')$ are leftmost in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$ and property (ii) follows.

Now assume that h is in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$. We obtain the situations as depicted in Figs. 3.55(b), 3.55(d) and 3.55(e) where the roles of $\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*$ and $\overline{G}^*_i$ are swapped. Fig. 3.56 shows the modified versions with swapped roles of $\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*$ and $\overline{G}^*_i$ where Figs. 3.56(a), 3.56(b) and 3.56(c) correspond to Figs. 3.55(b), 3.55(d) and 3.55(e), respectively. Further, h is the source in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*$. We now show that these cases violate the induction hypothesis. By assumption, vertex v^1 is in $\overline{G}_i$ but not in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$. Further, v^1 is incident to face h in $\mathrm{aux}(\overline{G}_{i+1})$ with $h \neq f \circ f'$ and $h \neq g \circ g'$, and h is the single source in $\mathrm{aux}(\overline{G}^*_{i+1})$. Face h is in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$ but not in $\mathrm{aux}(\overline{G}^*_i)$. However, as v^1 is not in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ it cannot be incident to h in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ and, thus, v^1 is also not incident to h in $\overline{G}_{i+1}$. Hence, v^1 is not leftmost in $\overline{G}_{i+1}$, which violates the induction hypothesis.

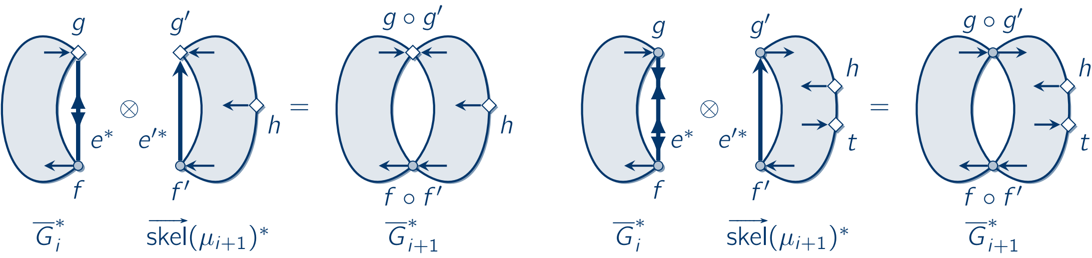
By the reasoning from before, property (ii) follows also for vertices $v^1 \in V^1$ in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ that are not in $\overline{G}_i$ and property (iii) for vertices $v^{\mathrm{r}} \in V^{\mathrm{r}}$.

Finally, let e^1 be an edge in $\mathbf{E}^1(\mu_{i+1})$. Here, two cases are distinguished: Either, $e^1 \neq e'$ or $e^1 = e'$, i. e., either e^1 is in $\overline{G}_{i+1}$ or it is not. The case $e^1 = e'$ implies that $e' \in \mathbf{E}^1(\mu_{i+1})$ which we have already discussed in the previous paragraphs for the case where there is a vertex v^1 in $\overline{G}_i$ but not in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$. Hence, the reasoning is similar. For the cases that violate the induction hypothesis (Figs. 3.56(a) to 3.56(c)), note that $e' \in \mathbf{E}^1(\mu_{i+1})$ implies that there is either a vertex $v^1 \in V^1$ in $\overline{G}_i$ but not in $\overrightarrow{\mathrm{skel}}(\mu_{i+1})$ or an edge $e^1 \in \mathbf{E}^1(\overline{G}_i)$ with $e^1 \neq e$. In both cases, a similar reasoning as before shows that the induction hypothesis is violated.

Suppose now that $e^1 \neq e'$. In this case, e^1 is incident to a face h which is a source in $\mathrm{aux}(\overline{G}^*_{i+1})$. Since $h \neq f \circ f'$ and $h \neq g \circ g'$, h is also the source in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1})^*)$ and e^1 is incident to h. In particular, at least one auxiliary edge of $\mathrm{aux}(e^1)$ is incident to h and, therefore, leftmost which implies (ii).

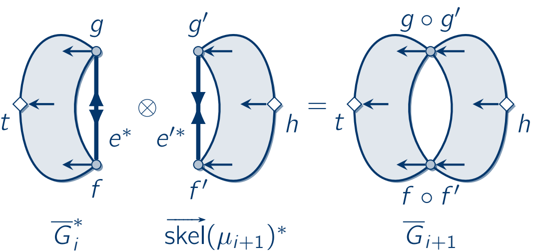
The proof is analogous for edges that must be leftmost in $\overline{G}_i$ and for edges that must be rightmost. Altogether, both $\mathrm{aux}(\overline{G}_i)$ and $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu_{i+1}))$ fulfill all properties. □

We call a feasible **RUP** embedding of a node's auxiliary skeleton *LR-feasible* if it fulfills the properties of Lem. 3.30. From an LR-feasible **RUP** embedding of the auxiliary, we obtain the *LR-feasible* embedding of the skeleton itself by applying aux^{-1}. Hence, we have to find LR-feasible **RUP** embeddings of the skeletons to obtain desired **RUP** embedding of the compound.



(a) Virtual edge e'^* is directed and e^* is a source but no sink edge.

(b) Virtual edge e'^* is directed and e^* is a source and sink edge.



(c) Virtual edge e'^* is a source but no sink edge and e^* a sink but no source edge.

Figure 3.56: Cases in Figs. 3.55(b), 3.55(d) and 3.55(e), where $\overrightarrow{\mathsf{skel}}(\mu_{i+1}^*)$ assumes the role of $\overline{G}_i$ and $\overline{G}_i^*$ the role of $\overrightarrow{\mathsf{skel}}(\mu_{i+1})$.

The Algorithm Alg. 3.6 shows `TestBiconnected` which is the top-level routine of our algorithm. In line 1, the SPQR tree of the underlying undirected graph of γ is computed in time $\mathcal{O}(|V| + |E|)$ by the algorithm in [GM01]. The remainder of the algorithm is divided into two steps:

- **Step 1** (lines 2 to 5): First, a preliminary version of the dSPQR tree is computed. The *pre-dSPQR tree* $\widetilde{\mathcal{T}}$ is equal to the dSPQR tree only that it requires no embedding and the cyclic edges do not carry the labels L, R, or LR. Further, for each node μ, the sets $\mathbf{E}^{\mathrm{l}}(\mu)$ and $\mathbf{E}^{\mathrm{r}}(\mu)$ are determined.
- **Step 2** (lines 6 to 8): In the second step, the routine `ComputeCompoundEmbedding` derives possible dSPQR trees from the pre-dSPQR tree and computes the respective embeddings of the skeletons. If one of these dSPQR trees and skeleton embeddings lead to a **RUP** embedding of γ, then this embedding is returned. Otherwise, $\perp$ is the output.

Step 1: Computation of the Pre-dSPQR Tree and of $\mathbf{E}^{\mathrm{l}}$ and $\mathbf{E}^{\mathrm{r}}$ Remember that in a dSPQR tree (cf. Def. 3.8), a virtual edge is cyclic-L if its expansion digraph contains a leftmost cycle, which implicitly assumes that the digraph is embedded. However, as the task of `TestBiconnected` is to find an embedding, we cannot rely on such information. Fortunately,

Algorithm 3.6. `TestBiconnected`

Input: planar compound $\gamma = (V, E)$ and sets $V^{\mathtt{l}}, V^{\mathtt{r}} \subseteq V$
Output: **RUP** embedding of γ where all vertices in $V^{\mathtt{l}}$ are left- and all vertices in $V^{\mathtt{r}}$ are rightmost or $\perp$ if no such embedding exists

1 $\mathcal{T} \leftarrow$ `ComputeSPQRTree`(γ)
2 **begin** Step 1: Computation of the Pre-dSPQR Tree and of $\mathbf{E}^{\mathtt{l}}$ and $\mathbf{E}^{\mathtt{r}}$
3 $\quad \widetilde{\mathcal{T}} \leftarrow$ `ComputePre-dSPQRTree`$(\mathcal{T})$
4 $\quad$ **if** $\widetilde{\mathcal{T}} = \perp$ **then return** $\perp$
5 $\quad \mathbf{E}^{\mathtt{l}}, \mathbf{E}^{\mathtt{r}} \leftarrow$ `ComputeLREdges`$(\mathcal{T}, V^{\mathtt{l}}, V^{\mathtt{r}})$
6 **begin** Step 2: Computation of the Embedding
7 $\quad$ **global** $V^{\mathtt{l}}, V^{\mathtt{r}}, \mathbf{E}^{\mathtt{l}}, \mathbf{E}^{\mathtt{r}}$
8 $\quad$ **return** `ComputeCompoundEmbedding`$(\widetilde{\mathcal{T}})$

we can use the observation from Cor. 3.22: If the expansion digraph $\mathsf{expg}_\mu(e)$ of a virtual edge e in node μ contains a cycle, then, given an embedding, $\mathsf{expg}_\mu(e)$ also contains a left- or rightmost cycle. In case no embedding is given, we can at least conclude that e is cyclic. By using this observation, we define that the pre-dSPQR tree of a compound is equal to the compound's dSPQR tree (Def. 3.8) only that all cyclic edges do not carry the labels L and R, i. e., a virtual edge is cyclic if its expansion digraph contains a cycle. For our running example in Fig. 3.52, the pre-dSPQR tree is shown in Fig. 3.53 on page 189. We denote by $\widetilde{\mathcal{T}}$ the pre-dSPQR tree and for a node μ its skeleton is denoted by $\widetilde{\mathsf{skel}}(\mu)$.

Alg. 3.7 shows the routine `ComputePre-dSPQRTree` which takes as input an SPQR tree of a compound and returns the pre-dSPQR tree. Under certain circumstances, which are discussed later, `ComputePre-dSPQRTree` returns $\perp$ when a **RUP** embedding is not possible. Before we analyze `ComputePre-dSPQRTree` in detail and prove its correctness, we need a few observations. We first prove an extended version of transitivity for cyclic edges:

Lemma 3.31. *Let e be a virtual edge in a node μ of a pre-dSPQR tree such that e is refined by μ'. Virtual edge e is cyclic if and only if $\widetilde{\mathsf{skel}}(\mu') \setminus \{e\}$ contains a cyclic edge or a cycle of directed, i. e., non-cyclic, edges only.*

Proof. $\Rightarrow$: Either $G_{\mu'} := \widetilde{\mathsf{skel}}(\mu') \setminus \{e\}$ contains a cyclic edge or not. In the first case, the implication follows. Let us assume the second case where all virtual edges in $G_{\mu'}$ are directed. Since e is cyclic in μ, its expansion digraph $\mathsf{expg}_\mu(e)$ contains a simple cycle C_{expg}. Let $C = v_1, \dots, v_\ell$ ($v_1 = v_\ell$ and $\ell > 1$) be the sequence of vertices in $G_{\mu'}$ as visited by cycle C_{expg} in order. We now show that C is a cycle in $G_{\mu'}$: Let v_i, v_{i+1} be two consecutive vertices on C with $1 \leq i < \ell$. If C_{expg} traverses only a single edge e_i between v_i and v_{i+1} such that e_i is also in $G_{\mu'}$, then v_i and v_{i+1} are also connected by e_i in $G_{\mu'}$. Otherwise, C_{expg} follows a dipath $p = v_i \rightsquigarrow v_{i+1}$ in $\mathsf{expg}_\mu(e)$. In this case, there is a directed virtual edge e' from v_i to v_{i+1} in $G_{\mu'}$, where the expansion digraph $\mathsf{expg}_{\mu'}(e')$ contains p. Hence, C is a cycle in $G_{\mu'}$.

$\Leftarrow$: If μ' contains a cyclic edge $e' \neq e$, then $\mathsf{expg}_{\mu'}(e')$ contains a cycle and, since $\mathsf{expg}_{\mu'}(e') \subsetneq \mathsf{expg}_\mu(e)$, we can conclude that e is cyclic in μ. Otherwise, let $C = v_1, \dots, v_\ell$ ($v_1 = v_\ell$ and $\ell > 1$) be a cycle in $G_{\mu'} := \widetilde{\mathsf{skel}}(\mu') \setminus \{e\}$ that consists of directed edges only. We construct a cycle C_{expg} in $\mathsf{expg}_\mu(e)$ as follows: If the edge between v_i and v_{i+1} is non-virtual,

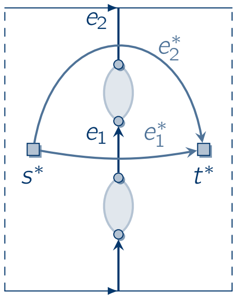
then $\mathrm{expg}_\mu(e)$ also contains this edge which is used in C_{expg}. If the edge e' between v_i and v_{i+1} is virtual, then its expansion digraph $\mathrm{expg}_{\mu'}(e')$ is a uv-digraph with $u = v_i$ and $v = v_{i+1}$. Hence, there is a dipath $p = v_i \rightsquigarrow v_{i+1}$ in $\mathrm{expg}_\mu(e)$. C_{expg} uses p to go from v_i to v_{i+1}. We obtain a cycle C_{expg} in $\mathrm{expg}_\mu(e)$ and can conclude that e is cyclic in μ. □

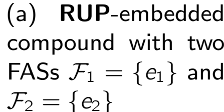
By Lem. 3.31, instead of testing whether the whole expansion digraph of a virtual edge contains a cycle, we only need to consider the skeleton of the node that refines it. `ComputePre-dSPQRTree` tests whether the skeleton contains a cyclic edge or, more complicated, a cycle of directed edges. This problem is solvable in time linear in the number of vertices and edges of the skeleton by using a depth-first search. However, it turns out that `ComputePre-dSPQRTree` needs to solve this problem multiple times for each node μ, i. e., one time for each node connected to μ in the SPQR tree.

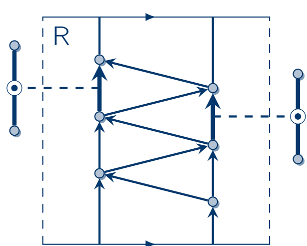
Assume that μ is a node with virtual edges $e_1, \dots, e_\ell$, where $e_1, \dots, e_\ell$ are refined by $\mu_1, \dots, \mu_\ell$, respectively. Edge e_i has to be cyclic in μ_i if $\widetilde{\mathrm{skel}}(\mu) \setminus \{e_i\}$ contains a cyclic edge or a cycle of directed edges. In the worst case, all virtual edge of μ are directed and we need to test for a cycle for all edges e_i, which leads to a quadratic running time. For a linear running time, we exploit properties of the skeletons of SPQR nodes and of **RUP** compounds. In the case of an S node lacking cyclic edges, removing an arbitrary edge leaves a uv-digraph and, hence, edge e_i has to be directed. For a P node with vertices u and v, and no cyclic edges, removing an edge e leaves an acyclic digraph if and only if e is either the single edge from u to v or from v to u. For R nodes, the problem becomes more involved and we need to import ideas from related concepts.

A problem which also deals with finding cycles in a digraph is the *feedback arc set problem*: Given a digraph $G = (V, E)$, a set of edges $\mathcal{F} \subsetneq E$ is called *feedback arc set (FAS)* if its removal leaves G acyclic. An FAS is called *minimum* if its cardinality is minimum among all FASs. Finding an FAS of cardinality $k \geq 0$ is one of Karp's 21 $\mathcal{NP}$-complete problems from his seminal paper [Kar72]. For planar digraphs, there is a polynomial-, though not linear-, time algorithm by Luccehsi and Younger [LY78] which uses the dual to find an FAS. There the idea is to contract edges in the dual until it becomes strongly connected and, thereby, the primal becomes acyclic. Since contracting a dual edge is equivalent to removing a primal edge, the primals of the contracted dual edges constitute an FAS. We use the same idea in the following.

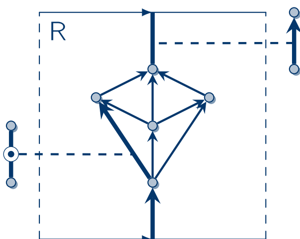
Suppose we are given the skeleton $\overrightarrow{\mathrm{skel}}(\mu)$ of an R node μ where all virtual edges $e_1, \dots, e_\ell$ are directed. If a minimum FAS $\mathcal{F}$ of $\overrightarrow{\mathrm{skel}}(\mu)$ has cardinality at least two, then removing a single edge from $\overrightarrow{\mathrm{skel}}(\mu)$ always leaves a cycle. Hence, the respective virtual edges in the neighboring nodes have to be cyclic. If $|\mathcal{F}| = 1$ with $\mathcal{F} = \{e\}$, removing e results in an acyclic digraph. However, in general, there might be several FASs of cardinality one. Finding all FASs of cardinality 1 takes time $\mathcal{O}(|E| \cdot (|V| + |E|))$ by subsequently removing all edges and using depth-first search to test for a cycle. Fortunately, in the case of R nodes and, even more general, **RUP**-embedded compounds, we can do considerably better. Consider the **RUP**-embedded compound sketched in Fig. 3.57(a): Its dual is an acyclic dipole with source s^* and sink t^*. In the following, we call a primal edge whose dual points from s^* to t^*, an s^*t^*-edge. In Fig. 3.57(a), e_1 and e_2 are s^*t^*-edges and we assume that the shaded regions do not contain s^*t^*-edges. Removing any of e_1 or e_2 results in an acyclic digraph, i. e., we have two minimum FASs $\mathcal{F}_1 = \{e_1\}$ and $\mathcal{F}_2 = \{e_2\}$. In fact, if a compound is **RUP**-embedded and there is an s^*t^*-edge e, then $\mathcal{F} = \{e\}$ is a minimum FAS. We now prove the following general result and apply it to R nodes afterwards.




(a) **RUP**-embedded compound with two FASs $\mathcal{F}_1 = \{e_1\}$ and $\mathcal{F}_2 = \{e_2\}$



(b) Skeleton of an R node with no s^*t^*-edge. An FAS has at least size two (three in this example).



(c) Skeleton of an R node with one s^*t^*-edge, which then constitutes a unique minimum FAS.

Figure 3.57: All minimum FASs of a **RUP**-embedded compounds have cardinality 1 if and only if there is an s^*t^*-edge.

Lemma 3.32. *Let $\gamma = (V, E)$ be a* **RUP***-embedded compound and let γ^* be the dual of γ with source s^* and sink t^*. Denote by* **F** *the set of minimum FASs of γ. Each minimum FAS $\mathcal{F} \in$* **F** *consists of a single s^*t^*-edge e if and only if there is at least one s^*t^*-edge.*

Proof. $\Rightarrow$: Since γ is strongly connected, **F** contains at least one FAS and, therefore, there is at least one edge from s^* to t^*.

$\Leftarrow$: Remember that removing an edge e from γ is equivalent to contracting the dual e^* of e in γ^*. Hence, removing all edges of a set $\mathcal{F} \subset E$ from γ makes it acyclic if and only if contracting all duals edges of $\mathcal{F}$ in γ^* results in a strongly connected digraph.

If there is an edge e^* from s^* to t^*, contracting it is equivalent to identifying the source with the sink of γ^*. Denote by $\overline{\gamma}^*$ the resulting digraph and let f and g be two faces in $\overline{\gamma}^*$. Since γ^* is an acyclic dipole, there are dipaths $f \rightsquigarrow t^*$ and $s^* \rightsquigarrow g$ and, therefore, there is a dipath $f \rightsquigarrow g$ in $\overline{\gamma}^*$. Thus, $\overline{\gamma}^*$ is strongly connected and $\{e\}$ is an FAS of γ which is also minimum since γ is strongly connected. Further, contracting any e^* that not points from s^* to t^* either leaves s^* as a source or t^* as a sink and, hence, results in a digraph that is not strongly connected. Edge e^*, therefore, constitutes no FAS of γ. Altogether, **F** contains only singletons, where each one consists of an s^*t^*-edge, and, conversely, all s^*t^*-edges are in a singleton of **F**. □

If γ does not contain an s^*t^*-edge, any FAS of γ contains at least two edges, which follows from the proof of Lem. 3.32:

Corollary 3.26. *If a* **RUP***-embedded compound does not contain an s^*t^*-edge, each FAS of γ contains at least two edges.*

If the dual does not contain multiple edges, then there is at most one edge from s^* to t^*. In this case, there is exactly one FAS. The skeletons of R nodes are triconnected and, therefore, their duals contain no multiple edges.

Corollary 3.27. *Let $\overrightarrow{\text{skel}}(\mu)$ be a* **RUP***-embedded skeleton of an R node where all virtual edges are directed and let $\overrightarrow{\text{skel}}(\mu^*)$ be its dual with source s^* and sink t^*. The set $\mathcal{F} = \{e\}$ is the unique minimum FAS of γ if and only if e is an s^*t^*-edge. If there is no s^*t^*-edge, any FAS of $\overrightarrow{\text{skel}}(\mu)$ contains at least two edges.*

For examples consider Figs. 3.57(b) and 3.57(c). Both show R nodes which contain two directed virtual edges. The R node in Fig. 3.57(b) contains no s^*t^*-edge and, hence, its FAS contains at least two edges; in fact, three in the example. Therefore, its two virtual edges are cyclic in the nodes which refine them. In Fig. 3.57(c), there is a virtual edge e that is also an s^*t^*-edge, which then constitutes the unique minimum FAS. Hence, the virtual edge e is directed in the node which refines it.

We now analyze `ComputePre-dSPQRTree`.

Lemma 3.33. *For an SPQR tree $\mathcal{T}$ of a biconnected compound $\gamma = (V, E)$,* `ComputePre-dSPQRTree` *(Alg. 3.7) computes the pre-dSPQR tree. If* `ComputePre-dSPQRTree` *returns $\perp$, γ is not* **RUP***. The running time is $\mathcal{O}(|V|)$.*

Proof. The algorithm starts with a pre-dSPQR tree $\widetilde{\mathcal{T}} = \mathcal{T}$, which is initially equal to the input SPQR tree (line 1), and subsequently *adjusts* each virtual edge, i. e., it becomes either directed or cyclic. This is done in two sweeps: first, bottom-up, from the leaves to the inner portion of the tree and then, top-down, backwards to the leaves. The whole algorithm heavily relies on Lem. 3.31, i. e., instead of determining the whole expansion digraph of an edge, only neighboring nodes are investigated to adjust an edge. First, an arbitrary node μ_r is chosen to be the root of the pre-dSPQR tree and connections between nodes point from the children to their parent (line 2). In line 3, a topological ordering $\mu_1, \dots, \mu_k$ of the nodes is obtained. These two steps take $\mathcal{O}(|V|)$ time since the number of nodes is in $\mathcal{O}(|V|)$ by Prop. 3.5.

The first sweep is carried out in the loop from lines 4 to 8. The nodes $\mu_1, \dots, \mu_{k-1}$ are processed in order of the topological ordering. Let μ be the node of the current iteration with skeleton $\widetilde{\text{skel}}(\mu) = (V_\mu, E_\mu)$. It contains a virtual edge e that is refined by μ_p which is the parent of μ in the pre-dSPQR tree. The invariant of the iteration is that all virtual edges $e' \neq e$ in μ have already been adjusted. This is true if μ is a leaf since there are no other virtual edges. Otherwise, we maintain the invariant by adjusting edge e in μ_p. Line 7 tests whether $\widetilde{\text{skel}}(\mu) \setminus \{e\}$ contains a cyclic edge or a cycle. To test for a cycle, a depth-first search is used that runs in time $\mathcal{O}(|V_\mu| + |E_\mu|) \subseteq \mathcal{O}(|V_\mu|)$. If any of the conditions is true, edge e is set cyclic in μ_p. Otherwise, $\widetilde{\text{skel}}(\mu) \setminus \{e\}$ is an acyclic dipole, where either of the endpoints of e is a source and the other endpoint a sink, and e must be directed from the source to the sink in μ_p. By Prop. 3.5, we know that the total sum of all edges of all skeletons is in $\mathcal{O}(|E|) \subseteq \mathcal{O}(|V|)$ and the number of nodes is in $\mathcal{O}(|V|)$. Hence, all operations in lines 4 to 8 have an overall running time in $\mathcal{O}(|V|)$. Note that each iteration adjust the virtual edge e in the parent node, whereas the virtual edge e in μ is not adjusted. This is the task of the second sweep.

In the second sweep, starting in line 9, the nodes are processed in reversed topological ordering. Again, let μ be the node of the current iteration. If μ is a leaf, then μ has no children and nothing has to be done. Otherwise, μ contains virtual edges $e_1, \dots, e_\ell$ with $\ell > 0$ refined by its children $\mu'_1, \dots, \mu'_\ell$ of μ, respectively. Due to the first sweep, all edges $e_1, \dots, e_\ell$ are adjusted in μ but not adjusted in nodes $\mu'_1, \dots, \mu'_\ell$. Three cases are distinguished: First, if two or more edges of $e_1, \dots, e_\ell$ are cyclic (line 12), then, for all i $(1 \leq i \leq \ell)$, digraph $\widetilde{\text{skel}}(\mu) \setminus \{e_i\}$ contains at least one cyclic edge. Hence, e_i must be set cyclic for all children μ'_i (line 13).

Algorithm 3.7. `ComputePre-dSPQRTree`

```
Input: SPQR tree 𝒯
Output: pre-dSPQR tree 𝒯̃ or ⊥ if compound is not RUP
1  𝒯̃ ← 𝒯
2  Choose an arbitrary root node μ_r in 𝒯̃ and direct all edges of 𝒯̃ towards the root
3  μ_1, ..., μ_k ← TopSort(𝒯̃)
4  foreach μ = μ_1, ..., μ_{k−1} do
5      μ_p ← parent of μ in 𝒯̃
6      e ← virtual edge in μ refined by μ_p
7      if skel̃(μ) \ {e} contains a cyclic edge or a cycle then set e cyclic in μ_p
8      else direct e in μ_p
9  foreach μ = Reverse(μ_1, ..., μ_k) do
10     μ'_1, ..., μ'_ℓ ← children of μ in 𝒯̃
11     e_1, ..., e_ℓ ← virtual edges in μ refined by μ'_1, ..., μ'_ℓ, respectively
12     if μ contains more than one cyclic edge then
13         foreach e_i ∈ {e_1, ..., e_ℓ} do set e_i cyclic in μ'_i
14     else if μ contains exactly one cyclic edge e_j then
15         foreach e_i ∈ {e_1, ..., e_ℓ} \ {e_j} do set e_i cyclic in μ'_i
16         if skel̃(μ) \ {e_j} contains a cycle then set e_j cyclic in μ'_j
17         else direct e in μ'
18     else all e_1, ..., e_ℓ are directed
19         switch type of μ do
20             case S-node
21                 foreach e_i ∈ {e_1, ..., e_ℓ} do direct e_i in μ'_i
22             case P-node with vertices u and v
23                 foreach e_i ∈ {e_1, ..., e_ℓ} do
24                     if skel̃(μ) \ {e_i} contains a cycle then set e_i cyclic in μ'_i
25                     else direct e_i in μ'_i
26             case R-node
27                 skel̃(μ*) = (F_μ, E*_μ) ← ComputeDual(skel̃(μ))
28                 if skel̃(μ*) is acyclic dipole with source s* and sink t* then
29                     if e*_j = (s*, t*) ∈ E*_μ then
30                         direct e_j in μ'_j where e_j be the primal of e*_j
31                         foreach e_i ∈ {e_1, ..., e_ℓ} \ {e_j} do set e_i cyclic in μ'_i
32                     else minimum FAS of skel̃(μ) contains more than one edge
33                         foreach e_i ∈ {e_1, ..., e_ℓ} do set e_i cyclic in μ'_i
34                 else return ⊥
35 return 𝒯̃
```

Second, if μ contains exactly one cyclic edge e_j ($1 \le j \le \ell$), all digraphs $\widetilde{\mathrm{skel}}(\mu) \setminus \{e_i\}$ with $i \neq j$ contain a cyclic edge, namely e_j (line 15). For $\overrightarrow{\mathrm{skel}}(\mu) \setminus \{e_j\}$, a depth-first search reveals whether it contains a cycle (line 16) and the virtual edge of μ'_j is adjusted accordingly. For all these operations, the running time is $\mathcal{O}(|V_\mu| + |E_\mu|) \subseteq \mathcal{O}(|V_\mu|)$.

In the third case, all virtual edges in $\widetilde{\mathrm{skel}}(\mu)$ are directed (line 18). Depending on the type of node μ, different operations are carried out: If μ is an S node, then removing any edge results in a uv-digraph and, hence, the corresponding virtual edges are directed (line 21). In case of a P node with vertices u and v, removing an edge e results in an acyclic digraph if and only if e is the single edge from either u to v or from v to u. Hence, for each e_i, we can decide in time $\mathcal{O}(1)$ whether $\widetilde{\mathrm{skel}}(\mu) \setminus \{e_i\}$ contains a cycle and adjust the corresponding virtual edges (line 24). Finally, if μ is an R node (line 26), its skeleton $\widetilde{\mathrm{skel}}(\mu)$ is triconnected and simple. Hence, its embedding is unique (up to inversion). In line 27, `ComputeDual` is called which determines the embedding and the dual $\widetilde{\mathrm{skel}}(\mu)^*$ of $\widetilde{\mathrm{skel}}(\mu)$ in time $\mathcal{O}(|V_\mu|)$, e. g., by an algorithm given in [KW01]. As $\widetilde{\mathrm{skel}}(\mu)$ contains no cyclic edges, $\overrightarrow{\mathrm{skel}}(\mu)$ and the auxiliary $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu))$ also contain no cyclic edges. In particular, $\widetilde{\mathrm{skel}}(\mu)$ is isomorphic to $\overrightarrow{\mathrm{skel}}(\mu)$ and $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu))$. If $\widetilde{\mathrm{skel}}(\mu)^*$ is no acyclic dipole, then $\widetilde{\mathrm{skel}}(\mu)$ is not **RUP** and, hence, $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu))$ has no feasible **RUP** embedding which implies that the whole compound is not **RUP** by Thm. 3.6. Then, $\perp$ is returned in line 34. Otherwise, $\widetilde{\mathrm{skel}}(\mu)^*$ is an acyclic dipole with source s^* and sink t^*. Testing whether $\widetilde{\mathrm{skel}}(\mu)^* = (F_\mu, E^*_\mu)$ is an acyclic dipole takes time $\mathcal{O}(|F_\mu| + |E^*_\mu|) \subseteq \mathcal{O}(|V_\mu|)$. If there is an edge e^*_j that points from s^* to t^*, we know by Cor. 3.27 that $\mathcal{F} = \{e_j\}$ is the unique minimum FAS of the primal (line 29). In this case, $\widetilde{\mathrm{skel}}(\mu) \setminus \{e_j\}$ is acyclic (line 30) and contains a cycle for all $e_i \neq e_j$ (line 31). If there is no edge from s^* to t^*, then all FASs of $\widetilde{\mathrm{skel}}(\mu)$ contain at least two edges and all virtual edges of μ's children must be cyclic (line 33).

After all virtual edges are adjusted, the pre-dSPQR tree $\widetilde{\mathcal{T}}$ is returned in line 35. The overall running time invested for a single node μ is in $\mathcal{O}(|V_\mu|)$ and, hence, `ComputePre-dSPQRTree` runs in time $\mathcal{O}(|V|)$. □

The result of `ComputePre-dSPQRTree` for our running example is shown in Fig. 3.53 on page 189, where the labels `l` and `r`, and `lr` can be ignored for the moment. If μ_2 is chosen to be the root, a possible topological ordering for the first sweep could be

$$\mu_1, \mu_7, \mu_5, \mu_6, \mu_4, \mu_3 \, .$$

Since μ_1 contains a cycle, the virtual edge on the left hand side of μ_2 is set cyclic. Further, the skeleton of μ_7 without its virtual edge is an acyclic dipole and, hence, the corresponding virtual edge in μ_2 is directed. In the second sweep, the nodes are processed in reversed order starting with node μ_2. There, the virtual edge in μ_1 is set cyclic since the virtual edge on the right hand side of μ_2 is cyclic. In our example, we do not have to compute the FAS of any of the R nodes since all three contain a single cyclic edge. For examples, where all edges are directed, and the minimum FAS contains at least two edges or a single edge, see Fig. 3.57(b) or Fig. 3.57(c), respectively.

The second task of Step 1 of `TestBiconnected` in Alg. 3.6 is to determine the mappings $\mathbf{E}^{\mathrm{l}}$ and $\mathbf{E}^{\mathrm{r}}$. As a reminder, $\mathbf{E}^{\mathrm{l}}(\mu)$ for a node μ contains all virtual edges $e = \{u, v\}$ of μ's skeleton whose expansion digraph contains a vertex from $V^{\mathrm{l}} \setminus \{u, v\}$. The routine `ComputeLREdges` in Alg. 3.8 computes $\mathbf{E}^{\mathrm{l}}$ and $\mathbf{E}^{\mathrm{r}}$. `ComputeLREdges` follows the same strategy as `Compute-Pre-dSPQRTree` and, consequently, many of the arguments from the proof of Lem. 3.33 apply equally; only that the analysis of `ComputeLREdges` is significantly simpler.

Algorithm 3.8. `ComputeLREdges`

```
   Input: SPQR tree 𝒯, subsets V^l, V^r of vertices
   Output: E^l and E^r
 1 Choose an arbitrary root node μ_r in 𝒯 and direct all edges of 𝒯 towards the root
 2 μ_1, ..., μ_k ← TopSort(𝒯)
 3 foreach node μ of 𝒯 do E^l(μ) ← ∅; E^r(μ) ← ∅
 4 foreach μ = μ_1, ..., μ_{k−1} do
 5 |   μ_p ← parent of μ in 𝒯
 6 |   e = {u, v} ← virtual edge in μ refined by μ_p
 7 |   if E^l(μ) ≠ ∅ ∨ skel(μ) contains vertex in V^l \ {u, v} then E^l(μ_p) ← E^l(μ_p) ∪ {e}
 8 |   if E^r(μ) ≠ ∅ ∨ skel(μ) contains vertex in V^r \ {u, v} then E^r(μ_p) ← E^r(μ_p) ∪ {e}
 9 foreach μ = Reverse(μ_1, ..., μ_k) do
10 |   μ'_1, ..., μ'_ℓ ← children of μ in 𝒯
11 |   foreach μ' ∈ {μ'_1, ..., μ'_ℓ} do
12 |   |   e = {u, v} ← virtual edge in μ refined by μ'
13 |   |   if E^l(μ) \ {e} ≠ ∅ ∨ skel(μ) contains vertex in V^l \ {u, v} then
14 |   |   |   E^l(μ') ← E^l(μ') ∪ {e}
15 |   |   if E^r(μ) \ {e} ≠ ∅ ∨ skel(μ) contains vertex in V^r \ {u, v} then
16 |   |   |   E^r(μ') ← E^r(μ') ∪ {e}
17 return E^l, E^r
```

Lemma 3.34. *For an SPQR tree $\mathcal{T}$ of a compound $\gamma = (V, E)$ and sets $V^{\mathtt{l}}, V^{\mathtt{r}} \subseteq V$,* `ComputeLREdges` *(Alg. 3.8) computes for each node the sets $\mathbf{E}^{\mathtt{l}}(\mu)$ and $\mathbf{E}^{\mathtt{r}}(\mu)$ in time $\mathcal{O}(|V|)$.*

Proof. As in `ComputePre-dSPQRTree` (cf. proof of Lem. 3.33), two sweeps are used. Again, node μ_r is chosen to be the root of $\mathcal{T}$ to obtain a topological ordering $\mu_1, \dots, \mu_k$ of the nodes (line 2). In the loop in lines 4 to 8, for node μ its unique parent node μ_p is determined which refines edges $e = \{u, v\}$. If the set $\mathbf{E}^{\mathtt{l}}(\mu)$ is not empty or if the skeleton of μ contains a vertex from $V^{\mathtt{l}}$ other than u and v, then e must be in $\mathbf{E}^{\mathtt{l}}(\mu_p)$ (line 7). The same test occurs for the vertices and edges that must be rightmost (line 8). In the second sweep, the nodes are processed in reversed order. For each node μ, the same conditions as in lines 7 to 8 are checked for all of μ's virtual edges $e_1, \dots, e_\ell$. The overall running time invested in each node μ is in $\mathcal{O}(|V_\mu| + |E_\mu|) \subseteq \mathcal{O}(|V_\mu|)$ and the overall running time of `ComputeLREdges` is $\mathcal{O}(|V|)$. □

The result of `ComputeLREdges` is shown in Fig. 3.53 on page 189, where for each node μ_i ($i \in \{1, \dots, 7\}$) the label `l` on a virtual edge e indicates that $e \in \mathbf{E}^{\mathtt{l}}(\mu_i)$. Likewise, the label `r` indicates that $e \in \mathbf{E}^{\mathtt{r}}(\mu_i)$.

Step 2: Computation of the Embedding Based on the pre-dSPQR tree, we compute a **RUP** embedding or reject if this is not possible. This task involves two aspects: First, each of the cyclic edges must be *aligned*, i. e., a cyclic edge becomes either a cyclic-L, a cyclic-R, or a cyclic-LR edge. These alignments must preserve the transitivity of cyclic edges (cf. Prop. 3.9) and, for a given alignment, an LR-feasible **RUP** embedding must exist. Finding such an LR-feasible **RUP** embedding is the second aspect. To avoid bloated parameter lists, $V^{\mathtt{l}}$, $V^{\mathtt{r}}$,

$\mathbf{E}^{\mathrm{l}}$, and $\mathbf{E}^{\mathrm{r}}$ are globally accessible. In the following, we assume that each skeleton contains at most two cyclic edges as otherwise the compound is not **RUP** (Cor. 3.24).

Computation of Compatible Alignments Given the pre-dSPQR tree from the first step, the second step of our algorithm is to compute possible alignments of the cyclic edges and to obtain the corresponding LR-feasible embeddings. If a node μ contains two cyclic edges e_1 and e_2, there are two possible alignments, i. e., either e_1 is cyclic-L and e_2 cyclic-R, denoted by (L, R), or vice versa, i. e., (R, L). If μ contains only one cyclic edge e, there are three possible alignments, namely, e can be cyclic-L, cyclic-R, or cyclic-LR, denoted by L, R, or LR, respectively. In case μ contains no cyclic edge, we have a "null" alignment denoted by $\oslash$. An alignment of a node μ is denoted by σ_μ and it can take any value from the set $\{(\mathsf{L}, \mathsf{R}), (\mathsf{R}, \mathsf{L}), \mathsf{L}, \mathsf{R}, \mathsf{LR}, \oslash\}$, depending on the number of cyclic edges in μ. For each possible alignment σ_μ, we obtain the directed skeleton $\overrightarrow{\mathsf{skel}}(\mu, \sigma_\mu)$ from $\widetilde{\mathsf{skel}}(\mu)$ by aligning the cyclic edges according to σ_μ, where the second parameter in $\overrightarrow{\mathsf{skel}}(\mu, \sigma_\mu)$ indicates the used alignment. Afterwards, we have to find out whether $\overrightarrow{\mathsf{skel}}(\mu, \sigma_\mu)$ can be LR-feasibly **RUP**-embedded. This is the task of `EmbedSkeleton` which is discussed later. If the alignment σ_μ allows for an LR-feasible **RUP** embedding, we call alignment σ_μ *LR-feasible* or simply *feasible* and obtain an LR-feasible **RUP** embedding $\mathcal{E}(\sigma_\mu)$ of $\overrightarrow{\mathsf{skel}}(\mu, \sigma_\mu)$.

For instance, node μ_3 in Fig. 3.53 on page 189 contains two cyclic edges $e_1 = \{v_1, v_3\}$ and $e_2 = \{v_2, v_3\}$. Of the two alignments (L, R) and (R, L) only (L, R), i. e., e_1 is cyclic-L and e_2 cyclic-R, is feasible since virtual edge $\{v_1, v_3\}$ must be left- and $\{v_2, v_3\}$ must be rightmost due to vertices in their expansion digraphs that must be left- and rightmost. Note that the alignment (L, R) induces alignments in neighboring nodes, e. g., in μ_2 the virtual edge on the right hand side must be cyclic-R by transitivity and, thus, the one on the left hand side is cyclic-L. Similarly, in μ_4 the single cyclic edge must be cyclic-LR. Hence, choosing an alignment in one node may restrict the number of possible alignments in adjacent nodes, an important observation that we use to guarantee a linear running time.

Alg. 3.9 on page 205 shows `ComputeCompoundEmbedding` which implements the second step of our algorithm. Before we analyze `ComputeCompoundEmbedding` in detail, we give a high-level description: `ComputeCompoundEmbedding` starts with a node $\overline{\mu}$ that contains the maximum number of cyclic edges and, depending on this number, determines all possible alignments of $\overline{\mu}$. From each alignment $\sigma_{\overline{\mu}}$, it derives the directed skeleton $\overrightarrow{\mathsf{skel}}(\overline{\mu}, \sigma_{\overline{\mu}})$ where all cyclic edges are aligned and calls `EmbedSkeleton` to determine whether the alignment is feasible. `EmbedSkeleton` takes as input $\overrightarrow{\mathsf{skel}}(\overline{\mu}, \sigma_{\overline{\mu}})$ and returns a respective LR-feasible **RUP** embedding or $\perp$ if no such embedding exists. Afterwards, `ComputeCompoundEmbedding` calls `Compute-Alignments` which implements a depth-first search traversal on the pre-dSPQR tree starting with $\overline{\mu}$. Whenever `ComputeAlignments` visits a node μ, the set of feasible alignments of the parent node μ_p, according to the depth-first traversal, is provided. For each feasible alignment σ_{μ_p} of μ_p, `ComputeAlignments` determines a set of alignments for μ that are "compatible" with σ_{μ_p}. "Compatible" means that, for instance, the transitivity of cyclic edges is ensured. As with `ComputeCompoundEmbedding`, for each candidate alignment σ_μ, `ComputeAlignments` derives the directed skeleton $\overrightarrow{\mathsf{skel}}(\mu, \sigma_\mu)$ and calls `EmbedSkeleton` to determine whether the alignment is feasible. In contrast to `ComputeCompoundEmbedding`, however, `ComputeAlignments` may provide additional parameters to `EmbedSkeleton`. Let e be the virtual edge in μ that is refined by μ_p. Assume that e is directed in μ and there is a feasible alignment σ_{μ_p} such that e is cyclic-L in μ_p. Then, there must either be a cyclic-L edge in $\overrightarrow{\mathsf{skel}}(\mu)$ that is distinct from e

or a leftmost cycle in $\overrightarrow{\mathsf{skel}}(\mu) \setminus \{e\}$. Assume the latter case and denote by σ_μ the respective alignment of μ. In a feasible **RUP** embedding of $\overrightarrow{\mathsf{skel}}(\mu, \sigma_\mu)$, e must not be part of the leftmost cycle as e is cyclic-L in μ_p and the expansion digraph $\mathsf{expg}_{\mu_p}(e)$, which does not contain e, must contain a leftmost cycle. In particular, EmbedSkeleton must ensure that in the LR-feasible embedding of $\overrightarrow{\mathsf{skel}}(\mu, \sigma_\mu)$, the directed virtual edge e is not leftmost. For this, EmbedSkeleton additionally receives an *embedding constraint* $\mathcal{C}$ for e. An embedding constraint $\mathcal{C}$ takes a value from $\{\bot, \overline{\mathtt{l}}, \overline{\mathtt{r}}, \mathtt{l}\overline{\mathtt{r}}, \overline{\mathtt{l}}\mathtt{r}, \overline{\mathtt{l}}\overline{\mathtt{r}}\}$. If $\overline{\mathtt{l}}$ appears in $\mathcal{C}$, then virtual edge e must *not* be leftmost as in the example discussed before. Symmetrically, for $\overline{\mathtt{r}}$, e must *not* be rightmost. In contrast, if $\mathtt{l}$ or $\mathtt{r}$ appears in $\mathcal{C}$, then e *has to be* leftmost or rightmost, respectively, which are cases that are discussed later. If $\mathcal{C} = \bot$, there is no constraint. EmbedSkeleton returns an LR-feasible **RUP** embedding of $\overrightarrow{\mathsf{skel}}(\mu, \sigma_\mu)$ which adheres to the embedding constraint $\mathcal{C}$. If no such embedding exists, EmbedSkeleton returns $\bot$. After determining which candidate alignment are feasible, ComputeAlignments recursively calls itself for each children of μ and the depth-first traversal continues.

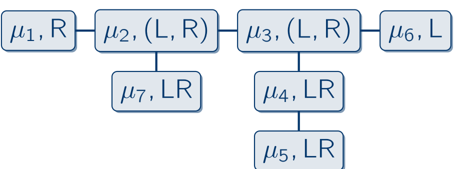


Figure 3.58: Graph of compatible alignments of the running example.

In order to store which alignments of a node are feasible and compatible to other alignments in neighboring nodes, ComputeCompoundEmbedding and ComputeAlignments use a graph data structure to store these relationships. Suppose that an alignment σ_μ induces an alignment $\sigma_{\mu'}$ of an adjacent node such that both alignments are feasible. We say that σ_μ and $\sigma_{\mu'}$ are *compatible* and define the *graph* $G_\Sigma = (V_\Sigma, E_\Sigma)$ *of compatible alignments*. The vertices V_Σ are triples $(\mu, \sigma_\mu, \mathcal{E}(\sigma_\mu))$, each consisting of (a pointer to) μ, a feasible alignment σ_μ and the corresponding LR-feasible embedding $\mathcal{E}(\sigma_\mu)$. There is an undirected edge between two vertices if the respective alignments are compatible. If G_Σ contains a connected subgraph $G'_\Sigma \subseteq G_\Sigma$ such that for each node there is exactly one vertex, we obtain a **RUP** embedding of the whole compound by assembling the respective embeddings. For our running example, G_Σ is shown in Fig. 3.58, where the corresponding LR-feasible embeddings are shown in Fig. 3.46 on page 179. Whereas in Fig. 3.58 there is only one feasible alignment for each node, in general a node may have several feasible alignments. For instance, in Fig. 3.59, μ_2 has two feasible alignments (L, R) and (R, L). The alignment (L, R) induces the alignment R for μ_1 and L for μ_3. The second alignment (R, L) of μ_2, induces the alignment L for μ_1 and R for μ_3 (see lower part of Fig. 3.59). Whereas μ_3's alignment is feasible, μ_1's alignment is not: if the cyclic edge in μ_1 is cyclic-L, the three vertices with the label $\mathtt{l}$ cannot be leftmost. G_Σ is shown in the middle of Fig. 3.59, where each vertex is connected to the respective embedded skeleton by a dashed line. The upper connected component of G_Σ contains a feasible alignment for each node from which a **RUP** embedding of the whole compound can be derived.

With these preliminaries, we analyze ComputeCompoundEmbedding. For the proof, we assume that for an aligned skeleton $\overrightarrow{\mathsf{skel}}(\mu, \sigma_\mu)$ with vertices V_μ and edges E_μ, EmbedSkeleton runs in time $\mathcal{O}(|V_\mu| + |E_\mu|)$.

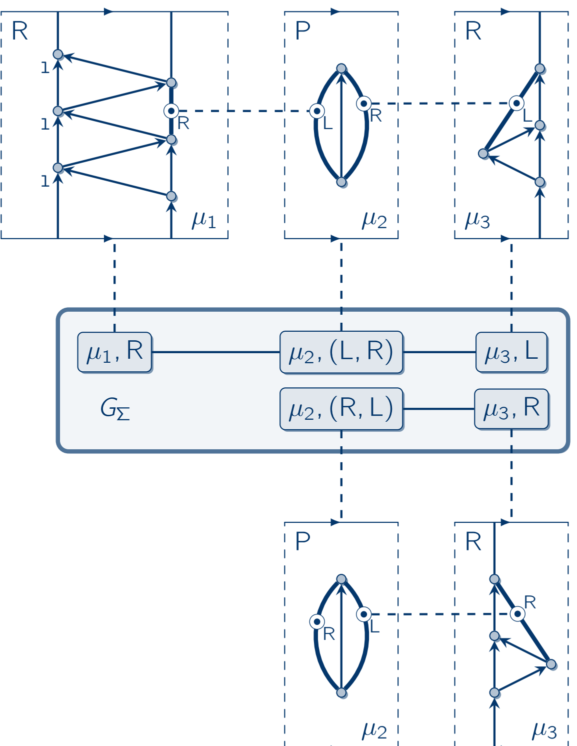


Figure 3.59: Example where each of μ_2 and μ_3 have two feasible alignments. However, only one of each leads to a feasible **RUP** embedding.

Lemma 3.35. *Given a pre-dSPQR tree,* `ComputeCompoundEmbedding` *(Alg. 3.9) returns a* **RUP** *embedding of a compound $\gamma = (V, E)$, where all vertices in $V^{\mathtt{l}}$ are left- and all vertices in $V^{\mathtt{r}}$ are rightmost. If no such embedding exists, $\perp$ is returned. Assuming that* `EmbedSkeleton` *(Alg. 3.12) runs in time $\mathcal{O}(|V_\mu| + |E_\mu|)$ for an aligned skeleton $\overrightarrow{\mathsf{skel}}(\mu, \sigma_\mu) = (V_\mu, E_\mu)$, the running time of* `ComputeCompoundEmbedding` *is in $\mathcal{O}(|V|)$.*

Proof. `ComputeCompoundEmbedding` uses `EmbedSkeleton` from Alg. 3.12 which is discussed later and `ComputeAlignments` in Alg. 3.11. We start with the analysis of `ComputeCompound-Embedding` and then proceed with `ComputeAlignments`.

ComputeCompoundEmbedding If there exists a node with more than two cyclic edges, the compound cannot be **RUP** by Cor. 3.24 (line 1). Let $\overline{\mu}$ be a node with the maximum number of cyclic edges. The first step of `ComputeCompoundEmbedding` is to compute all feasible alignments of $\overline{\mu}$. For this, three cases are distinguished (line 4): There are either two, one, or no cyclic edges in $\overline{\mu}$. If $\overline{\mu}$ contains two cyclic edges, there are two alignment candidates (L, R) and (R, L) which are stored in the set $\mathbf{\Sigma}_{\overline{\mu}}$ (line 5). Likewise, for one cyclic edge, $\mathbf{\Sigma}_{\overline{\mu}} = \{\mathsf{L}, \mathsf{R}, \mathsf{LR}\}$ (line 6) and $\mathbf{\Sigma}_{\overline{\mu}} = \{\oslash\}$ if $\overline{\mu}$ contains no cyclic edge (line 7). For each of these candidates $\sigma_{\overline{\mu}}$ (line 8), the directed skeleton $\overrightarrow{\mathsf{skel}}(\overline{\mu}, \sigma_{\overline{\mu}})$ is obtained from $\widetilde{\mathsf{skel}}(\overline{\mu})$ by aligning its cyclic edges according to $\sigma_{\overline{\mu}}$. Passing $\overrightarrow{\mathsf{skel}}(\overline{\mu}, \sigma_{\overline{\mu}})$ to `EmbedSkeleton` reveals whether $\sigma_{\overline{\mu}}$ is feasible. If this is the case, a new vertex $(\overline{\mu}, \sigma_{\overline{\mu}}, \mathcal{E}(\sigma_{\overline{\mu}}))$ is added to the graph G_Σ. Note that no embedding constraint $\mathcal{C}$ is passed to `EmbedSkeleton` and, hence, the respective

Algorithm 3.9. `ComputeCompoundEmbedding`

Input: pre-dSPQR tree $\widetilde{\mathcal{T}}$ of compound γ

Output: **RUP** embedding of compound where all vertices in V^1 are left- and all vertices in V^r are rightmost or $\perp$ if no such embedding exists

1 **if** $\exists$ node μ with > 2 cyclic edges **then return** $\perp$ by Cor. 3.24
2 $\overline{\mu} \leftarrow$ node in $\widetilde{\mathcal{T}}$ with maximum number of cyclic edges
3 $G_\Sigma = (V_\Sigma, E_\Sigma) \leftarrow$ initialize with empty set of vertices and edges
4 **switch** number of cyclic edges in $\widetilde{\text{skel}}(\overline{\mu})$ **do**
5   **case** two: $\mathbf{\Sigma}_{\overline{\mu}} \leftarrow \{(\text{L}, \text{R}), (\text{R}, \text{L})\}$
6   **case** one: $\mathbf{\Sigma}_{\overline{\mu}} \leftarrow \{\text{L}, \text{R}, \text{LR}\}$
7   **case** none: $\mathbf{\Sigma}_{\overline{\mu}} \leftarrow \{\oslash\}$
8 **foreach** $\sigma_{\overline{\mu}} \in \mathbf{\Sigma}_{\overline{\mu}}$ **do**
9   $\overrightarrow{\text{skel}}(\overline{\mu}, \sigma_{\overline{\mu}}) \leftarrow \widetilde{\text{skel}}(\overline{\mu})$, where cyclic edges are aligned according to $\sigma_{\overline{\mu}}$
10   $\mathcal{E}(\sigma_{\overline{\mu}}) \leftarrow$ `EmbedSkeleton`$(\overrightarrow{\text{skel}}(\overline{\mu}, \sigma_{\overline{\mu}}), \perp, \perp)$
11   **if** $\mathcal{E}(\sigma_{\overline{\mu}}) \neq \perp$ **then** $V_\Sigma \leftarrow V_\Sigma \cup \{(\overline{\mu}, \sigma_{\overline{\mu}}, \mathcal{E}(\sigma_{\overline{\mu}}))\}$
12 Make $\overline{\mu}$ of root of $\widetilde{\mathcal{T}}$ and direct all edges to the children
13 **foreach** child μ' of $\overline{\mu}$ **do** $G_\Sigma \leftarrow$ `ComputeAlignments`$(\mu', \overline{\mu}, G_\Sigma)$
14 **foreach** vertex $(\overline{\mu}, \sigma_{\overline{\mu}}, \mathcal{E}(\sigma_{\overline{\mu}}))$ in G_Σ that belongs to $\overline{\mu}$ **do**
15   $V'_\Sigma \leftarrow$ `FindCompatibleAlignments`$(\overline{\mu}, G_\Sigma, (\overline{\mu}, \sigma_{\overline{\mu}}, \mathcal{E}(\sigma_{\overline{\mu}})))$
16   **if** $V'_\Sigma \neq \perp$ **then**
17     $\mu_1, \dots, \mu_k \leftarrow$ node sequence of $\widetilde{\mathcal{T}}$
18     $\mathcal{E}_1, \dots, \mathcal{E}_k \leftarrow$ embeddings of $\overrightarrow{\text{skel}}(\mu_1), \dots, \overrightarrow{\text{skel}}(\mu_k)$ as in V'_Σ
19     $\mathcal{E} \leftarrow$ assemble embeddings $\mathcal{E}_1, \dots, \mathcal{E}_k$
20   **return** $\mathcal{E}$
21 **return** $\perp$

Algorithm 3.10. `FindCompatibleAlignments`

Input: node μ, graph $G_\Sigma = (V_\Sigma, E_\Sigma)$ of compatible alignments, vertex $(\mu, \sigma_\mu, \mathcal{E}(\sigma_\mu)) \in V_\Sigma$ that belongs to μ

Output: set of vertices $V'_\Sigma \subseteq V_\Sigma$ that induces a connected subgraph $G'_\Sigma \subseteq G_\Sigma$ which contains exactly one alignment for each node in the subtree of μ; $\perp$ if no such subgraph of G_Σ exists

1 $\mu_1, \dots, \mu_\ell \leftarrow$ children of μ in the pre-dSPQR tree
2 $V'_\Sigma \leftarrow \{(\mu, \sigma_\mu, \mathcal{E}(\sigma_\mu))\}$
3 **foreach** $\mu_i = \mu_1, \dots, \mu_\ell$ **do**
4   $V'_{\Sigma,i} \leftarrow \perp$
5   **foreach** $(\mu_i, \sigma_{\mu_i}, \mathcal{E}(\sigma_{\mu_i})) \in V_\Sigma$ with edge to $(\mu, \sigma_\mu, \mathcal{E}(\sigma_\mu))$ in G_Σ **do**
6     $V'_{\Sigma,i} \leftarrow$ `FindCompatibleAlignments`$(\mu_i, G_\Sigma, (\mu_i, \sigma_{\mu_i}, \mathcal{E}(\sigma_{\mu_i})))$
7   **if** $V'_{\Sigma,i} \neq \perp$ **then break**
8   **if** $V'_{\Sigma,i} \neq \perp$ **then** $V'_\Sigma \leftarrow V'_\Sigma \cup V'_{\Sigma,i}$
9   **else return** $\perp$
10 **return** V'_Σ

parameters are $\perp$. EmbedSkeleton is called at most three times and its running time is in $\mathcal{O}(|V_{\overline{\mu}}| + |E_{\overline{\mu}}|)$ by assumption, where $\overrightarrow{\text{skel}}(\overline{\mu}) = (V_{\overline{\mu}}, E_{\overline{\mu}})$. Hence, the running time of lines 4 to 7 is in $\mathcal{O}(|V_{\overline{\mu}}| + |E_{\overline{\mu}}|) \subseteq \mathcal{O}(|V|)$.

In the following, we assume that $\overline{\mu}$ is the root of $\widetilde{\mathcal{T}}$ and that all edges are directed from the parent to its children. In the loop starting in line 13, ComputeAlignments is called for each child μ' of $\overline{\mu}$ in $\widetilde{\mathcal{T}}$. For each child μ' and all nodes in the respective subtree, ComputeAlignments computes the feasible alignments and updates G_Σ accordingly. In the discussion of ComputeAlignments that follows later, we prove that the overall running time of the loop in line 13 is in $\mathcal{O}(|V|)$ and that G_Σ contains at most five vertices for each node.

In Line 15, ComputeCompoundEmbedding calls FindCompatibleAlignments (Alg. 3.10). FindCompatibleAlignments is a modified depth-first traversal on the pre-dSPQR tree and its takes as input the current node μ of the traversal, the graph of compatible alignments $G_\Sigma = (V_\Sigma, E_\Sigma)$ and a vertex $(\mu, \sigma_\mu, \mathcal{E}(\sigma_\mu)) \in V_\Sigma$ that belongs to μ. As output, it returns a subset V'_Σ of V_Σ that induces a connected subgraph of G_Σ which contains exactly one feasible alignment for each node in the subtree of μ. For each child μ_i of μ in the pre-dSPQR tree, FindCompatibleAlignments recursively calls itself for each alignment that is compatible with the alignment of μ, i. e., there is edge between the respective vertices of μ and μ_i in G_Σ. If any of these calls returns a subset $V'_{\Sigma,i}$, then these vertices of G_Σ are added to V'_Σ which initially only contains the feasible alignment and embedding of μ. If for one child all recursive calls of FindCompatibleAlignments return $\perp$, FindCompatibleAlignments also returns $\perp$. Otherwise, FindCompatibleAlignments returns the desired set V'_Σ.

ComputeCompoundEmbedding calls FindCompatibleAlignments for all vertices of G_Σ that belong to $\overline{\mu}$ (line 15). As there are at most five vertices in G_Σ for each node, which we will prove later, the running time of FindCompatibleAlignments is linear in the number of nodes and, hence, in $\mathcal{O}(|V|)$. If FindCompatibleAlignments always returns $\perp$, then the compound has not **RUP** embedding as there is no connected subgraph of G_Σ that contains exactly one feasible alignment for each node. Otherwise, FindCompatibleAlignments returns a set V'_Σ. Based on this set, the **RUP** embedding of γ is determined in lines 17 to 20 as follows: Let $\mu_1, \dots, \mu_k$ be a node sequence of $\widetilde{\mathcal{T}}$ and let $\mathcal{E}_1, \dots, \mathcal{E}_k$ be the respective embeddings of the skeletons according to V'_Σ. Each embedding is LR-feasible and **RUP**, i. e., it fulfills the properties of Lem. 3.30, and assembling them by a series of two-clique summations yields the **RUP** embedding of γ. These last steps are carried out in $\mathcal{O}(|V|)$. Hence, the overall running time of ComputeCompoundEmbedding is $\mathcal{O}(|V|)$ assuming that all calls of ComputeAlignments result in a linear running time, which we prove next.

ComputeAlignments Remember, ComputeAlignments (Alg. 3.11) implements a depth-first traversal on the pre-dSPQR tree. As input, ComputeAlignments receives the current node μ of the depth-first traversal, its parent node μ_p and the (current) graph of compatible alignments $G_\Sigma = (V_\Sigma, E_\Sigma)$. ComputeAlignments extends G_Σ to encompass μ and all nodes in its subtree. In the following, e is the virtual edge in μ refined by its parent μ_p. For each vertex of G_Σ that belongs to μ_p, we obtain a feasible alignment σ_{μ_p} and the respective embedding $\mathcal{E}(\sigma_{\mu_p})$ (line 2). As an invariant of the recursion, we assume that $\mathcal{E}(\sigma_{\mu_p})$ is an LR-feasible **RUP** embedding of $\overrightarrow{\text{skel}}(\mu_p, \sigma_{\mu_p})$. We maintain this invariant for all vertices that we introduce to G_Σ for node μ.

The alignment σ_{μ_p} induces candidate alignments of μ. In order to determine these candidate alignments, different cases are distinguished that depend on the cyclic edges in μ and μ_p (line 3).

Algorithm 3.11. `ComputeAlignments`

Input: node μ, parent node μ_p of μ, graph $G_\Sigma = (V_\Sigma, E_\Sigma)$ of compatible alignments
Output: graph G_Σ extended to μ and all nodes of μ's subtree

1 $e \leftarrow$ virtual edge in μ refined by μ_p
2 **foreach** $(\mu_p, \sigma_{\mu_p}, \mathcal{E}(\sigma_{\mu_p})) \in V_\Sigma$ **do**
3   **switch** number of cyclic edges in $\widetilde{\text{skel}}(\mu)$ **do**
4   **case** two cyclic edges of which one is e and the other is e'
5          $\sigma_\mu \leftarrow$ alignment according to Fig. 3.60
6          $\mathbf{T} \leftarrow \{(\sigma_\mu, \perp)\}$
7   **case** one cyclic edge e refined by μ_p
8          $\sigma_\mu \leftarrow$ alignment according to one of Figs. 3.61(a) to 3.61(f)
9          $\mathbf{T} \leftarrow \{(\sigma_\mu, \perp)\}$
10   **case** one cyclic edge e' not refined by μ_p
11          $\sigma_\mu, \mathcal{C} \leftarrow$ alignment and embedding constraints according to Fig. 3.62(a)
12          $\mathbf{T} \leftarrow \{(\sigma_\mu, \mathcal{C})\}$
13   **if** e is cyclic-LR in μ_p **then** case of Fig. 3.62(b) applies
14             $\mathbf{T} \leftarrow \mathbf{T} \cup \{(\mathsf{L}, \overline{\mathtt{r}}), (\mathsf{R}, \overline{\mathtt{l}})\}$
15   **case** no cyclic edge, i. e., e is directed
16   **switch** e in μ_p **do**
17   **case** directed: $\mathbf{T} \leftarrow \{(\oslash, \perp)\}$
18   **case** L: $\mathbf{T} \leftarrow \{(\oslash, \overline{\mathtt{l}}\mathtt{r})\}$
19   **case** R: $\mathbf{T} \leftarrow \{(\oslash, \mathtt{l}\overline{\mathtt{r}})\}$
20   **case** LR: $\mathbf{T} \leftarrow \{(\oslash, \overline{\mathtt{l}}\overline{\mathtt{r}})\}$
21   **foreach** $(\sigma_\mu, \mathcal{C}) \in \mathbf{T}$ **do**
22       $\overrightarrow{\text{skel}}(\mu, \sigma_\mu) \leftarrow \widetilde{\text{skel}}(\mu)$, where cyclic edges (if any) are aligned according to σ_μ
23       $\mathcal{E}(\sigma_\mu) \leftarrow$ `EmbedSkeleton`$(\overrightarrow{\text{skel}}(\mu, \sigma_\mu), e, \mathcal{C})$
24   **if** $\mathcal{E}(\sigma_\mu) \neq \perp$ **then**
25          $V_\Sigma \leftarrow V_\Sigma \cup \{(\mu, \sigma_\mu, \mathcal{E}(\sigma_\mu))\}$
26          $E_\Sigma \leftarrow E_\Sigma \cup \{(\mu, \sigma_\mu, \mathcal{E}(\sigma_\mu)), (\mu_p, \sigma_{\mu_p}, \mathcal{E}(\sigma_{\mu_p}))\}$
27 **foreach** child μ' of μ **do**
28    $G_\Sigma \leftarrow$ `ComputeAlignments`(μ', μ, G_Σ)
29 **return** G_Σ

For each case, we obtain one or more pairs $(\sigma_\mu, \mathcal{C})$ consisting of an alignment candidate σ_μ of μ and of an embedding constraint $\mathcal{C}$ of e, where e is the virtual edge in μ that is refined by μ_p. If e is not directed, then $\mathcal{C}$ is set to $\perp$. Each of these pairs $(\sigma_\mu, \mathcal{C})$ is stored in the set **T** and, later in the algorithm, `EmbedSkeleton` is called with σ_μ and $\mathcal{C}$ as parameters to find out whether the alignment candidate is feasible. As the used alignment can be inferred from the context in the following, we denote the skeleton of μ by $\overrightarrow{\text{skel}}(\mu)$ instead of $\overrightarrow{\text{skel}}(\mu, \sigma_\mu)$.

In order to determine the candidate alignments of μ, `ComputeAlignments` distinguishes between numerous cases. Instead of giving the case differentiation as pseudocode in Alg. 3.11, we use the diagrams shown in Figs. 3.60 to 3.63. In each of the diagrams, e. g., Fig. 3.60, node μ_p is sketched on top and node μ below. To determine which case applies, the alignment of μ_p's cyclic edges, the existence of leftmost cycles, rightmost cycles, or both (L-, R-, or LR-cycles) in $\overrightarrow{\text{skel}}(\mu_p) \setminus \{e\}$, and the number of cyclic edges in μ are taken into account. If the case applies, we obtain an alignment or an embedding constraint of e in node μ as shown at the bottom. A label $a|b$ of a cyclic edge in μ_p means that it is either cyclic a or cyclic b for $a, b \in \{\text{L}, \text{R}, \text{LR}\}$. Likewise, a label $a|b$ within a cycle, e. g., in Fig. 3.61(a), means that there is an a- or b-cycle for $a, b \in \{\text{L}, \text{R}, \text{L\&R}\}$ in $\overrightarrow{\text{skel}}(\mu_p) \setminus \{e\}$, where L&R means that there is an L- and an R-cycle. Depending on whether a or b applies, a cyclic edge of μ with label $x|y$ is aligned to x if case a applies in μ_p and it is aligned to y for case b. For instance, if edge e'_p is a cyclic-R edge in μ_p in Fig. 3.60, then edge e becomes a cyclic-R edge in μ and e' a cyclic-L edge in μ. If e is a directed virtual edge in μ, it is labeled with an embedding constraint. For example, if e is cyclic-L in μ_p in Fig. 3.63(b), then e receives the embedding constraint $\overline{\texttt{l}}\texttt{r}$. The following cases are distinguished in `ComputeAlignments`:

- **Two cyclic edges in μ** (line 4)
  First, we show that one of the two cyclic edges in μ is the virtual edge refined by μ_p, namely, edge e. Let $\overline{\mu}$ be the node in $\widetilde{\mathcal{T}}$ with the maximum number of cyclic edges, i. e., the node at which the depth-first search starts. In particular, $\mu \neq \overline{\mu}$ and $\overline{\mu}$ contains two cyclic edges since μ contains two cyclic edges. In $\widetilde{\mathcal{T}}$, there is a dipath $p = \overline{\mu} \rightsquigarrow \mu_p \rightarrow \mu$ which corresponds to the current call hierarchy of the depth-first search. The second node μ' on p whose parent is $\overline{\mu}$, contains a cyclic edge e' that is refined by $\overline{\mu}$. Since $\overline{\mu}$ contains two cyclic edge, e' is cyclic in μ'. The same argument can be subsequently applied to all nodes on p and, hence, also to μ. Thus, edge e is cyclic in μ.

  Denote by e' the other cyclic edge of μ. Since e' is a cyclic edge different from e, we can immediately conclude by transitivity (Prop. 3.9) that edge e is also cyclic in μ_p. Also, by the arguments from before, μ_p contains a cyclic edge e'_p distinct from e. The case in Fig. 3.60 applies. Depending on whether e'_p is cyclic-L or cyclic-R, e must also be aligned to cyclic-L or cyclic-R in μ due to transitivity and e' receives the opposite alignment.

- **Edge e is the single cyclic edge in μ** (line 7)
  Since e is cyclic in μ, $\overrightarrow{\text{skel}}(\mu_p) \setminus \{e\}$ either contains a cyclic edge or a cycle of directed edges by Lem. 3.31. One of the cases in Figs. 3.61(a) to 3.61(f) applies. First, we assume that e is directed in μ_p (Figs. 3.61(a) to 3.61(d)).

  - If μ_p contains no cyclic edge, it contains a cycle of directed edges (Fig. 3.61(a)). Since all edges in $\overrightarrow{\text{skel}}(\mu_p)$ are directed, the feasible alignment of μ_p is $\emptyset$. Hence, $\overrightarrow{\text{skel}}(\mu_p)$ is **RUP**-embedded with a leftmost and a rightmost cycle. Note that these cycles are not necessarily disjoint. If e is part of only the leftmost cycle in $\overrightarrow{\text{skel}}(\mu_p)$, i. e., $\text{skel}(\mu_p) \setminus \{e\}$ contains the rightmost cycle, the alignment of e in μ is R.

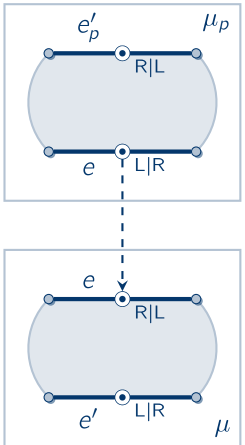


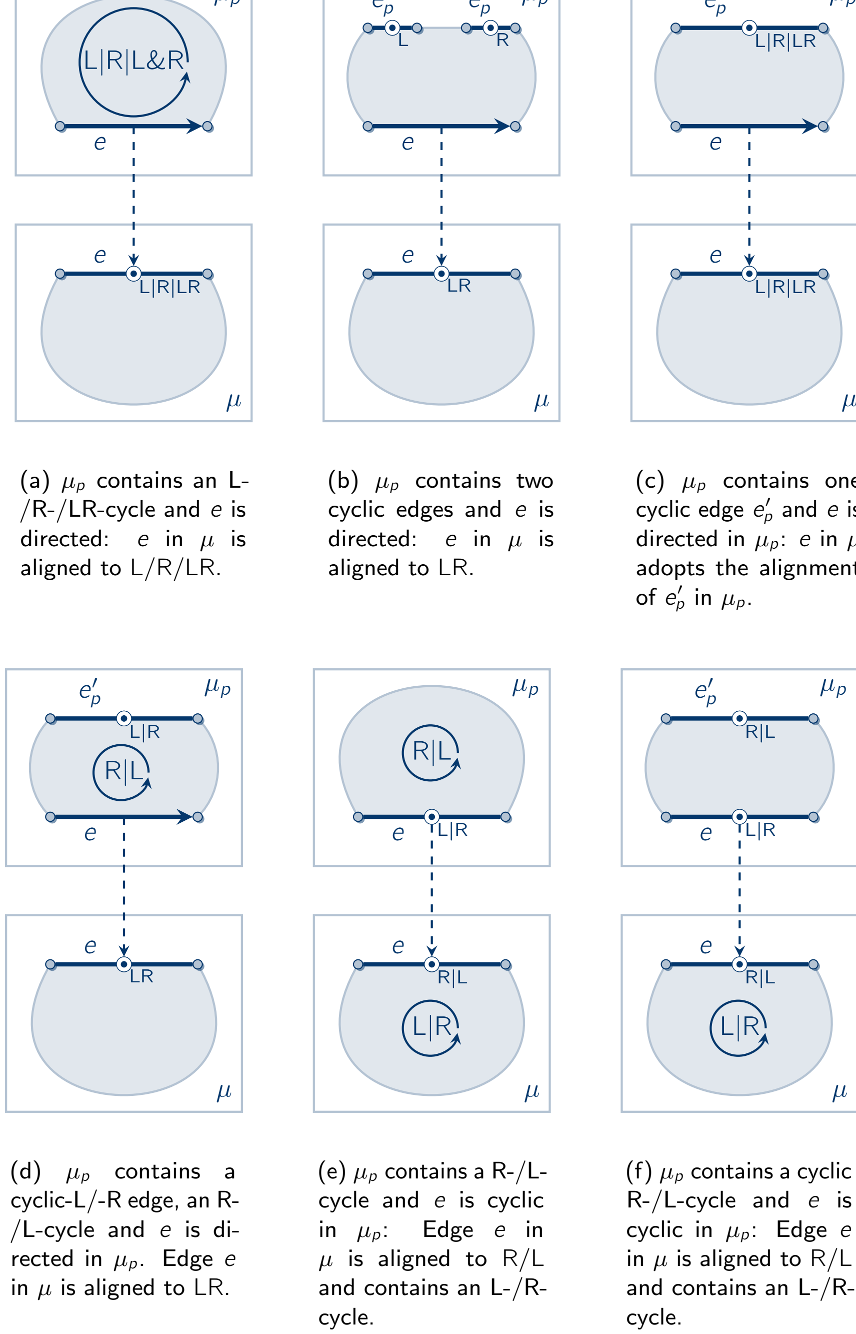
Figure 3.60: Two cyclic edges in μ: Edge e' in μ adopts the alignments of e in μ_p.

Symmetrically, if e is part of only the rightmost cycle, its alignment is L. Finally, if e is neither part of $\overrightarrow{\text{skel}}(\mu_p)$'s left- nor of its rightmost cycle, e becomes a cyclic-LR edge. Note that e cannot be part of both the left- and the rightmost cycle in $\overrightarrow{\text{skel}}(\mu_p)$ as then $\overrightarrow{\text{skel}}(\mu_p) \setminus \{e\}$ would be acyclic and e would be not cyclic in μ. Based on the **RUP** embedding of $\overrightarrow{\text{skel}}(\mu_p) = (V_{\mu_p}, E_{\mu_p})$ and its dual, these tests are carried out in time $\mathcal{O}(|V_{\mu_p}| + |E_{\mu_p}|)$.

- If μ_p contains two cyclic edges e'_p and e''_p (cf. Fig. 3.61(b)), then either e'_p is cyclic-L and e''_p is cyclic-R or vice versa since the alignment of μ_p is feasible. In this case, the alignment of e in μ is LR.
- If μ_p contains a single cyclic edge e'_p, we obtain the cases depicted in Figs. 3.61(c) and 3.61(d). In case e'_p is cyclic-L (cyclic-R) and e is part of the rightmost (leftmost) cycle in $\overrightarrow{\text{skel}}(\mu_p)$, e must be cyclic-L (cyclic-R) in μ (Fig. 3.61(c)). If e'_p is cyclic-LR (Fig. 3.61(c)) or e is neither part of the left- nor of the rightmost cycle in $\overrightarrow{\text{skel}}(\mu_p)$ (see Fig. 3.61(d)), then e becomes a cyclic-LR edge in μ. Again, testing if e is part of the right- or leftmost cycle is done in $\mathcal{O}(|V_{\mu_p}| + |E_{\mu_p}|)$.

Now we assume that e is cyclic in μ_p (Figs. 3.61(e) and 3.61(f)). Then, node μ must contain an L- or R-cycle depending on the alignment of e in μ_p. Observe that at this point we still have no embedding of $\overrightarrow{\text{skel}}(\mu)$, however, from the alignment of e in μ_p we can infer how the embedding of $\overrightarrow{\text{skel}}(\mu)$ must look like. Also note that e cannot be cyclic-LR in μ_p since e is cyclic in μ and, therefore, $\overrightarrow{\text{skel}}(\mu_p) \setminus \{e\}$ must either contain a cyclic edge or a cycle of directed edges. If μ_p contains a cycle of directed edges, which is either left- or rightmost and of which e is not part of, edge e is aligned to cyclic-L or cyclic-R in μ accordingly (Fig. 3.61(e)). If μ_p contains a cyclic edge e'_p that is cyclic-R or -L, then e is aligned according to e'_p (Fig. 3.61(f)).

- **μ contains a single cyclic edge** $e' \neq e$ (line 10)
  Since e' is the single cyclic edge in μ, e is directed in μ and cyclic in μ_p. This also implies that μ_p contains no cyclic edge besides e. First assume that e is cyclic-L or -R in μ_p. The case shown in Fig. 3.62(a) applies and e' adopts the alignment of e in μ_p. Suppose



(a) μ_p contains an L-/R-/LR-cycle and e is directed: e in μ is aligned to L/R/LR.

(b) μ_p contains two cyclic edges and e is directed: e in μ is aligned to LR.

(c) μ_p contains one cyclic edge e'_p and e is directed in μ_p: e in μ adopts the alignment of e'_p in μ_p.

(d) μ_p contains a cyclic-L/-R edge, an R-/L-cycle and e is directed in μ_p. Edge e in μ is aligned to LR.

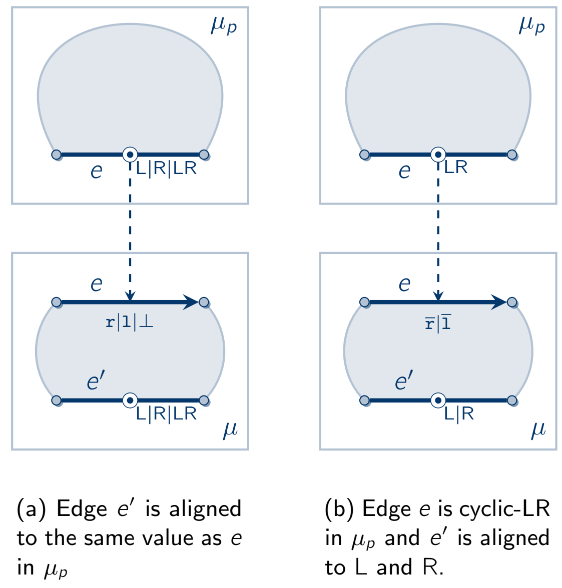
(e) μ_p contains a R-/L-cycle and e is cyclic in μ_p: Edge e in μ is aligned to R/L and contains an L-/R-cycle.

(f) μ_p contains a cyclic R-/L-cycle and e is cyclic in μ_p: Edge e in μ is aligned to R/L and contains an L-/R-cycle.

Figure 3.61: μ contains exactly one cyclic edge e that is refined by μ_p.

that e is cyclic-L in μ_p. In the LR-feasible **RUP** embedding of $\overrightarrow{\text{skel}}(\mu)$, edge e must be rightmost since otherwise $\text{expg}_{\mu_p}(e)$ contains a rightmost cycle of which e is no part of and, thus, e would be cyclic-LR in μ_p. Therefore, we obtain embedding constraint $\mathtt{r}$ for e. Also note that in principal e must not be leftmost in the **RUP** embedding of $\overrightarrow{\text{skel}}(\mu)$. However, $\overrightarrow{\text{skel}}(\mu)$ contains a cyclic-L edge and, therefore, e cannot be leftmost anyway. Symmetrically, if e is cyclic-R, the embedding constraint is $\mathtt{r}$. If e is cyclic-LR in μ_p, e' must be cyclic-LR in μ and no embedding constraint applies.

If e is cyclic-LR in μ_p, we obtain two additional alignment candidates (Fig. 3.62(b)): For the first candidate, e' is cyclic-L and e must not be rightmost in $\overrightarrow{\text{skel}}(\mu)$ since e is cyclic-LR in μ_p. In this case, we obtain the embedding constraint $\overline{\mathtt{r}}$ for e. The second candidate is the symmetric case, i. e., e' is cyclic-R and e must not be leftmost.



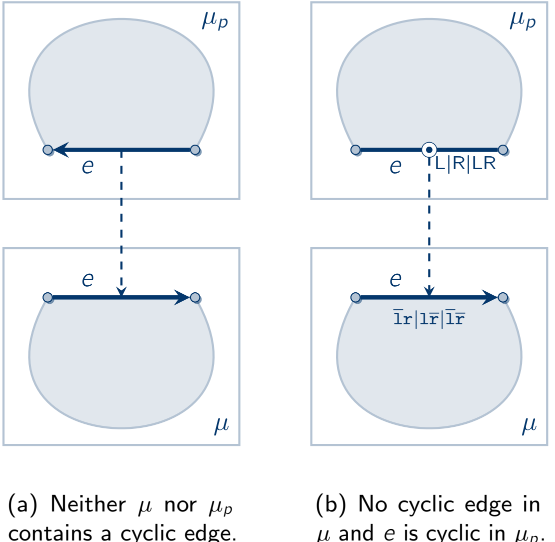
(a) Edge e' is aligned to the same value as e in μ_p

(b) Edge e is cyclic-LR in μ_p and e' is aligned to L and R.

Figure 3.62: μ contains exactly one cyclic edge e' that is not refined by μ_p.

- ▶ **μ contains no cyclic edge** (line 15)
  In this case, e is directed in μ. If e is also directed in μ_p, then the skeletons of μ and μ_p are both acyclic dipoles. In this case, the candidate alignment of μ_p is $\oslash$ with no embedding constraint (Fig. 3.63(a)). Note that in this case, e in μ_p and e in μ are antiparallel as the compound is strongly connected. If e is cyclic in μ_p, the candidate alignment of μ is also $\oslash$ (Fig. 3.63(b)). Additionally, we get an embedding constraint: If e is cyclic-L in μ_p, then $\text{expg}_{\mu_p}(e)$ contains a leftmost cycle and, hence, e must not be leftmost in $\overrightarrow{\text{skel}}(\mu)$'s embedding and we get the embedding constraint $\overline{\mathtt{l}}$. Likewise, if e is cyclic-R or cyclic-LR, the embedding constraint is $\overline{\mathtt{l}}$ or $\overline{\mathtt{l}}\overline{\mathtt{r}}$, respectively.

For all of these cases, the running time is in $\mathcal{O}(|V_\mu|+|E_\mu|+|V_{\mu_p}|+|E_{\mu_p}|)$. In the worst case **T** contains three elements if μ contains a cyclic edge $e' \neq e$ and e is cyclic-LR in μ_p (Figs. 3.62(a) and 3.62(b)). Hence, the number of iterations in line 21 is at most three. Moreover, the



(a) Neither μ nor μ_p contains a cyclic edge.

(b) No cyclic edge in μ and e is cyclic in μ_p.

Figure 3.63: μ contains no cyclic edges.

number of iterations in line 2 is at most five as the number c_μ of vertices in G_Σ belonging to a particular node μ is at most five: For $\overline{\mu}$, i. e., the root, we have already seen that $c_{\overline{\mu}} \leq 3$. For all other nodes, we bound the number of possibilities for candidate alignments and embedding constraints as determined in `ComputeAlignments` as follows. If μ contains two cyclic edges, there are two possible alignment candidates and, hence, $c_\mu \leq 2$. In case e is the single cyclic edge in μ, there are at most three alignment candidates ($c_\mu \leq 3$). If there is a single cyclic edge $e' \neq e$, then e' can be cyclic-L, -R, or -LR with embedding constraints $\mathtt{l}$, $\mathtt{r}$, or $\perp$ for e, respectively, which makes three possibilities (Fig. 3.62(a)). Additionally, if e is cyclic in μ_p, we obtain two further possibilities (Fig. 3.62(b)) which yields a total number of five and, hence, $c_\mu \leq 5$. Finally, if μ contains no cyclic edge, there is either only one candidate (Fig. 3.63(a)) or three (Fig. 3.63(b)), which yields $c_\mu \leq 3$. Hence, the maximum number of candidate alignments and embedding constraints is five and, thus, $c_\mu \leq 5$ for all nodes. Therefore, the number of vertices in G_Σ belonging to a node μ is at most five in total and the number of vertices in G_Σ is linear in the number of nodes in the pre-dSPQR tree. Moreover, between the vertices belonging to two nodes that are adjacent in the pre-dSPQR tree there are at most 25 edges in G_Σ. Hence, the number of edges in G_Σ is linear in the number of nodes in the pre-dSPQR tree, and, thus, linear in the number of vertices in the compound.

For each alignment candidate σ_μ and embedding constraint $\mathcal{C}$ in $\mathbf{T}$, the cyclic edges in $\widetilde{\text{skel}}(\mu)$ are aligned to obtain $\overrightarrow{\text{skel}}(\mu, \sigma_\mu)$ (line 22) and `EmbedSkeleton` is used to find out whether the alignment is feasible (line 23). If this is the case we obtain $\mathcal{E}(\sigma_\mu)$ as an LR-feasible **RUP** embedding and $(\mu, \sigma_\mu, \mathcal{E}(\sigma_\mu))$ is inserted to G_Σ as a new vertex. Additionally, the edge between $(\mu, \sigma_\mu, \mathcal{E}(\sigma_\mu))$ and $(\mu_p, \sigma_{\mu_p}, \mathcal{E}(\sigma_{\mu_p}))$ is inserted as σ_{μ_p} and σ_μ are compatible.

Since the cardinality of $\mathbf{T}$ and the number of vertices in G_Σ belonging to μ_p are bounded by constants, and since `EmbedSkeleton` runs in $\mathcal{O}(|V_\mu| + |E_\mu|)$, the steps in lines 21 to 26 run in $\mathcal{O}(|V_\mu| + |E_\mu|)$. In the loop in line 27, `ComputeAlignments` is called recursively for all of μ's children in $\widetilde{\mathcal{T}}$. As the running time of `ComputeAlignments` is in $\mathcal{O}(|V_\mu| + |E_\mu| + |V_{\mu_p}| + |E_{\mu_p}|)$

without the recursion, the overall running time of all recursion steps is in $\mathcal{O}(|V|)$ as the overall size of all skeletons is in $\mathcal{O}(|E|)$ by Prop. 3.5.

□

Computation of LR-feasible RUP Embeddings `ComputeCompoundEmbedding` and `ComputeAlignments` use `EmbedSkeleton` (cf. Alg. 3.12) which we discuss now. Recall, as input parameters, `EmbedSkeleton` receives a skeleton $\overrightarrow{\mathsf{skel}}(\mu)$ of a node μ and, if the virtual edge e refined by its parent is directed, an embedding constraint $\mathcal{C}$ for e. `EmbedSkeleton` finds an LR-feasible **RUP** embedding of $\overrightarrow{\mathsf{skel}}(\mu)$ that adheres the embedding constraint for e, if one is given, or it returns $\perp$ if no such embedding exists.

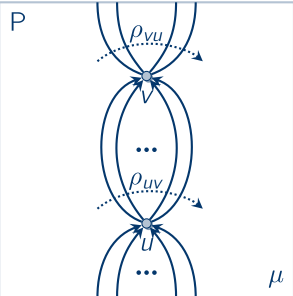


Figure 3.64: RUP embedding of a P node skeleton.

A high-level description of `EmbedSkeleton` is as follows: To find a **RUP** embedding of the skeleton, `EmbedSkeleton` uses the auxiliary skeleton for which it tries to find a LR feasible **RUP** embedding. The first step is to derive up to two embedding candidates (lines 6 to 25). The method to derive such an embedding candidate depends on the type of the node: For a P node with vertices u and v, `EmbedSkeleton` constructs an embedding as shown in Fig. 3.64 where all outgoing edges are consecutive in u and in v, and so are all incoming edges. Such an embedding is **RUP**. In addition, `EmbedSkeleton` ensures that the L-cycle is leftmost, if existent, and the R-cycle is rightmost, if existent, and that all the other embedding constraints are fulfilled. The embedding of an S node is unique and so is the embedding of an R node up to inversion, i. e., there are two embedding candidates. As the last step, `EmbedSkeleton` tests for each embedding candidate whether it fulfills all properties to be an LR-feasible **RUP** embedding (lines 26 to 33). If one such embedding is found, it is returned. Otherwise, $\perp$ is returned.

As described before, `EmbedSkeleton` operates on the auxiliary of the input skeleton for which it tries to find an LR-feasible **RUP** embedding. Remember that the embedding of an auxiliary skeleton $\mathsf{aux}(\overrightarrow{\mathsf{skel}}(\mu))$ is feasible if all auxiliary edges that belong to a cyclic edge are consecutive in the rotation system and if all L-cycles are left- and all R-cycles are rightmost. Further, a feasible embedding is LR-feasible if it fulfills the properties of Lem. 3.30, i. e., for each edge $e^{\mathtt{l}} \in \mathbf{E}^{\mathtt{l}}(\mu)$ and $e^{\mathtt{r}} \in \mathbf{E}^{\mathtt{l}}(\mu)$ at least one of $\mathsf{aux}(e^{\mathtt{l}})$ must be leftmost and at least one $\mathsf{aux}(e^{\mathtt{r}})$ must be rightmost. From an LR-feasible **RUP** embedding of $\mathsf{aux}(\overrightarrow{\mathsf{skel}}(\mu))$, we obtain an LR-feasible **RUP** embedding of the input skeleton $\overrightarrow{\mathsf{skel}}(\mu)$ by applying the function aux^{-1} (cf. Fig. 3.48 on page 182).

`EmbedSkeleton` uses a slightly modified version of aux^{-1}: First and foremost, `Embed-Skeleton` ensures that each L-cycle is left- and each R-cycle is rightmost. Afterwards, it passes the resulting embedding to aux^{-1} which returns $\perp$ if the auxiliary edges that belong to a cyclic

Algorithm 3.12. `EmbedSkeleton`

Input: directed skeleton $\overrightarrow{\text{skel}}(\mu) = (V_\mu, E_\mu)$, directed virtual edge e in μ, embedding constraint $\mathcal{C}$ for e

Output: LR-feasible **RUP** embedding of $\overrightarrow{\text{skel}}(\mu)$, which adheres to the embedding constraint of e; if no such embedding exists, $\bot$ is returned

1 $E^{\mathtt{l}} \leftarrow \mathbf{E}^{\mathtt{l}}(\mu)$; $E^{\mathtt{r}} \leftarrow \mathbf{E}^{\mathtt{r}}(\mu)$; $V^{\mathtt{l}} \leftarrow V^{\mathtt{l}} \cap V_\mu$; $V^{\mathtt{r}} \leftarrow V^{\mathtt{r}} \cap V_\mu$
2 $\overline{\mathtt{l}}$, $(\overline{\mathtt{r}}) \leftarrow$ `true` if and only if e must not be leftmost (rightmost) according to $\mathcal{C}$
3 **if** $(\overline{\mathtt{l}} \wedge e \in E^{\mathtt{l}}) \vee (\overline{\mathtt{r}} \wedge e \in E^{\mathtt{r}})$ **then return** $\bot$

4 **if** $\mathtt{l}$ appears in $\mathcal{C}$ **then** $E^{\mathtt{l}} \leftarrow E^{\mathtt{l}} \cup \{e\}$
5 **if** $\mathtt{r}$ appears in $\mathcal{C}$ **then** $E^{\mathtt{r}} \leftarrow E^{\mathtt{r}} \cup \{e\}$

6 **if** μ is P node with vertices u and v **then**
7 $\quad E^{\mathtt{l}}_{uv}$, $(E^{\mathtt{r}}_{uv}) \leftarrow$ non-cyclic edges from u to v in $E^{\mathtt{l}}$ $(E^{\mathtt{r}})$
8 $\quad E^{\sim}_{uv} \leftarrow$ non-cyclic edges from u to v not in $E^{\mathtt{l}}_{uv}$ and $E^{\mathtt{r}}_{uv}$
9 $\quad \rho_{uv} \leftarrow []$
10 $\quad$ **if** $\overrightarrow{\text{skel}}(\mu)$ contains cyclic-L or cyclic-LR edge $e^{\mathtt{l}}$ **then**
11 $\quad\quad \rho_{uv} \leftarrow \rho_{uv} \cdot [e^{\mathtt{l}}_{uv}]$, where $e^{\mathtt{l}}_{uv}$ is the auxiliary of $e^{\mathtt{l}}$ that points from u to v

12 $\quad \rho_{uv} \leftarrow \rho_{uv} \cdot E^{\mathtt{l}}_{uv} \cdot E^{\sim}_{uv} \cdot E^{\mathtt{r}}_{uv}$
13 $\quad$ **if** $\overrightarrow{\text{skel}}(\mu)$ contains cyclic-R or cyclic-LR edge $e^{\mathtt{r}}$ **then**
14 $\quad\quad \rho_{uv} \leftarrow \rho_{uv} \cdot [e^{\mathtt{r}}_{uv}]$, where $e^{\mathtt{r}}_{uv}$ is the auxiliary of $e^{\mathtt{r}}$ that points from u to v

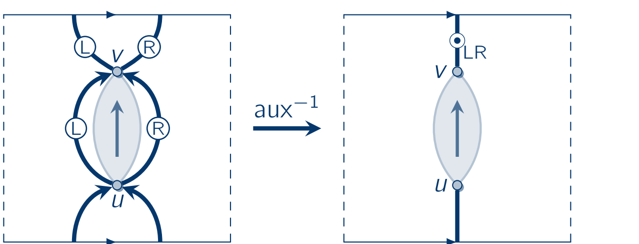
15 $\quad$ **if** $(\overline{\mathtt{l}} \vee \overline{\mathtt{r}}) \wedge e = (u, v)$ **then** position e such that the embedding constraint is fulfilled; if not possible, **return** $\bot$
16 $\quad \rho_{vu} \leftarrow$ analogous total order for all edges from v to u
17 $\quad \mathcal{E}^{\text{aux}} \leftarrow$ embedding, where u's rotation system is $\rho_{vu} \cdot \overleftarrow{\rho_{uv}}$ and $\rho_{uv} \cdot \overleftarrow{\rho_{vu}}$ for v
18 $\quad \overline{\mathcal{E}}^{\text{aux}} \leftarrow \{\mathcal{E}^{\text{aux}}\}$
19 **else** μ is an S or R node
20 $\quad$ **if** μ is an S node **then** $\overline{\mathcal{E}} \leftarrow$ set containing single unique embedding of $\overrightarrow{\text{skel}}(\mu)$
21 $\quad$ **else** $\overline{\mathcal{E}} \leftarrow$ set containing unique embedding of $\overrightarrow{\text{skel}}(\mu)$ and its inversion
22 $\quad \overline{\mathcal{E}}^{\text{aux}} \leftarrow \emptyset$
23 $\quad$ **foreach** $\mathcal{E} \in \overline{\mathcal{E}}$ **do**
24 $\quad\quad \mathcal{E}^{\text{aux}} \leftarrow$ embedding of $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ according to $\mathcal{E}$, where each cyclic-L, cyclic-R, and cyclic-LR edge is replaced by an L-, R , and L-/R-cycle, respectively
25 $\quad\quad \overline{\mathcal{E}}^{\text{aux}} \leftarrow \overline{\mathcal{E}} \cup \{\mathcal{E}^{\text{aux}}\}$

26 **foreach** $\mathcal{E}^{\text{aux}} \in \overline{\mathcal{E}}^{\text{aux}}$ **do**
27 $\quad$ Embed $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ according to $\mathcal{E}^{\text{aux}}$
28 $\quad \text{aux}(\overrightarrow{\text{skel}}(\mu))^* \leftarrow$ `ComputeDual`$(\text{aux}(\overrightarrow{\text{skel}}(\mu)))$
29 $\quad$ **if** $\text{aux}(\overrightarrow{\text{skel}}(\mu))^*$ is no acyclic dipole **then continue**
30 $\quad$ **if** $\exists v \in V^{\mathtt{l}}$ not leftmost $\vee$ $\exists v \in V^{\mathtt{r}}$ not rightmost **then continue**
31 $\quad$ **if** $\exists e \in E^{\mathtt{l}}$ not leftmost $\vee$ $\exists e \in E^{\mathtt{r}}$ not rightmost **then continue**
32 $\quad$ **if** $(\overline{\mathtt{l}} \wedge e$ is leftmost$)$ $\vee$ $(\overline{\mathtt{r}} \wedge e$ is rightmost$)$ **then continue**
33 $\quad$ **return** $\text{aux}^{-1}(\text{aux}(\overrightarrow{\text{skel}}(\mu))$

34 **return** $\bot$

edge $e = \{u, v\}$ are not consecutive in the rotation systems of u and v. Otherwise, aux^{-1} returns the embedded skeleton by replacing the bundle of auxiliary edges by the original edge. If there is a cyclic-L edge in the input skeleton, its auxiliary edges are always consecutive since they are leftmost. Likewise, the auxiliaries of cyclic-R edges are consecutive. Suppose that there is a cyclic-LR edge in the skeleton. In the auxiliary skeleton, this edge is replaced by an L- and an R-cycle. Since `EmbedSkeleton` only ensures that the L-cycle is left- and the R-cycle is rightmost, the corresponding auxiliary edges may not be consecutive in the rotation system of the auxiliary. However, if this is the case, we can show that whole compound is not **RUP**. For this, remember Lem. 3.29 which states that, in a **RUP**-embedded compound, for a node μ with a cyclic-LR edge e, the remaining skeleton $\overrightarrow{\text{skel}}(\mu) \setminus \{e\}$ is an acyclic dipole. The proof of Lem. 3.29 argues that if $\overrightarrow{\text{skel}}(\mu) \setminus \{e\}$ would contain a cycle, then the node μ' which refines e contains either too many L- or two many R-cycles and, hence, μ' cannot be **RUP**-embedded and so the whole compound is not **RUP**.

Corollary 3.28. *Let μ be a node of the dSPQR tree of a compound. If μ contains a cyclic-LR edge such that $\overrightarrow{\text{skel}}(\mu) \setminus \{e\}$ is no acyclic dipole, then the node that refines e cannot be* **RUP***-embedded. In this case, the whole compound is not* **RUP***-embedded.*



(a) For an LR-cycle, the remainder of the skeleton is acyclic and, hence, all edges of the LR-cycle are consecutive in the rotation system.

(b) Situation obtained in the proof of Lem. 3.36.

Figure 3.65: In a node with a cyclic-LR, the remainder of the skeleton must be an acyclic dipole if the compound is **RUP**.

Consider Fig. 3.65(a). On the left side, the **RUP**-embedded auxiliary skeleton of a node with a cyclic-LR edge $e = \{u, v\}$ is displayed. In order for the auxiliary edges to be consecutive in the rotation system, there cannot be edges from v to u in the auxiliary skeleton. In other words, the skeleton without e must be an acyclic dipole. We get the following lemma.

Lemma 3.36. *Let μ be a node of the dSPQR tree of a compound such that μ contains a cyclic-LR edge e. Assume that $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ is* **RUP***-embedded, where the L-cycle is left- and the R-cycle is rightmost. $\overrightarrow{\text{skel}}(\mu) \setminus \{e\}$ is an acyclic dipole if and only if all auxiliary edges belonging to e are consecutive in the embedding of $\text{aux}(\overrightarrow{\text{skel}}(\mu))$.*

Proof. $\Rightarrow$: Since the auxiliary edges of the L-cycle are leftmost, they are consecutive in the rotation system and so are the edges of the R-cycle. Assume w. l. o. g. that u is the single source and v the single sink of $\overrightarrow{\text{skel}}(\mu) \setminus \{e\}$ with $e = \{u, v\}$. In $\text{aux}(\overrightarrow{\text{skel}}(\mu))$, v has exactly two outgoing edges which are auxiliary edges of e and u has two incoming auxiliary edges.

Moreover, since $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu))$ is **RUP**-embedded, in the rotation systems of u and v all incoming edges are consecutive and so are all outgoing edges. Hence, at u and v all auxiliary edges are consecutive; see left hand side of Fig. 3.65(a).

$\Leftarrow$: Due to the **RUP** embedding of $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu))$, all outgoing edges of u and v are consecutive in their rotation systems, and so are all incoming edges. By assumption, also all auxiliary edges $\mathrm{aux}(e)$ are consecutive in the rotation systems of u and v. Therefore, either all incoming edges of u and all outgoing edges of v are in $\mathrm{aux}(e)$, or vice versa. W. l. o. g., we assume the former. Then, in $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu)) \setminus \mathrm{aux}(e)$, u is a source as all incoming edges have been removed from u. Analogously, v a sink. As the compound is strongly connected, no vertex distinct from u and v is a terminal. Hence, $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu)) \setminus \mathrm{aux}(e)$ contains the single source u and the single sink v.

What is left to show is that $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu)) \setminus \mathrm{aux}(e)$ is acyclic. The situation we obtain is illustrated in Fig. 3.65(b). Remember, all incoming edges of u are from $\mathrm{aux}(e)$ and so are all outgoing edges of v. Since these edges are consecutive in the rotation systems of u and v, the incoming edges of u and the outgoing edges of v enclose a face h. By assumption, the dual $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu))^*$ of $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu))$ is an acyclic dipole with source s and sink t. Remember, the two pairs of antiparallel edges in $\mathrm{aux}(e)$ form the L- and the R-cycle and, hence, are incident to s and t, respectively. Face h is adjacent to s and t as one of the incoming auxiliary edges of u is incident to s and one is incident to t. We remove the two incoming auxiliary edges of u from $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu))$ to obtain the digraph G_{aux}. For the dual G^*_{aux} of G_{aux}, this corresponds to contracting the duals of these edges which identifies the faces s, t and h. By this operation, G^*_{aux} becomes strongly connected. Therefore, G_{aux} is acyclic and, as $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu)) \setminus \mathrm{aux}(e)$ is a subgraph of G_{aux}, $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu)) \setminus \mathrm{aux}(e)$ is also acyclic. Altogether, $\mathrm{aux}(\overrightarrow{\mathrm{skel}}(\mu)) \setminus \mathrm{aux}(e)$ is an acyclic dipole with source u and sink v and so is $\overrightarrow{\mathrm{skel}}(\mu) \setminus \{e\}$ ☐

By Lem. 3.36, we only have to make sure that an L-cycle is leftmost and an R-cycle is rightmost. If in the resulting rotation system, the auxiliary edges of an LR-edge are not consecutive, i. e., aux^{-1} returns $\perp$, the whole compound cannot be **RUP**-embedded (at least if e is cyclic-LR). We will use this fact in `EmbedSkeleton` which we analyze now.

Lemma 3.37. *For a directed skeleton $\overrightarrow{\mathrm{skel}}(\mu) = (V_\mu, E_\mu)$, `EmbedSkeleton` (Alg. 3.12) returns an LR-feasible* **RUP** *embedding that adheres an embedding constraint $\mathcal{C}$ of a directed virtual edge e in $\overrightarrow{\mathrm{skel}}(\mu)$ (if one is given). If no such embedding exists, $\perp$ is returned. The running time is in $\mathcal{O}(|V_\mu| + ||E_\mu|)$*

Proof. As guaranteed by `ComputeCompoundEmbedding` and `ComputeAlignments`, we assume that all cyclic edges are aligned and $\overrightarrow{\mathrm{skel}}(\mu)$ contains at most one cyclic-L and at most one cyclic-R edge, or a single cyclic-LR edge. Let $E^{\mathtt{l}} = \mathbf{E}^{\mathtt{l}}(\mu)$, $E^{\mathtt{r}} = \mathbf{E}^{\mathtt{r}}(\mu)$ and denote by $V^{\mathtt{l}}$ and $V^{\mathtt{r}}$ the vertices in $\overrightarrow{\mathrm{skel}}(\mu)$ that must be left- and rightmost, respectively (line 1). Further, we introduce two Boolean variables $\bar{\mathtt{l}}$ and $\bar{\mathtt{r}}$ that are true if and only if $\bar{\mathtt{l}}$ and $\bar{\mathtt{r}}$ appear in $\mathcal{C}$, respectively (line 2). If e must not be left- or rightmost and, at the same time, e is in $E^{\mathtt{l}}$ or $E^{\mathtt{r}}$, respectively, we can immediately return $\perp$ (line 3). In lines 4 to 5, e is inserted to $E^{\mathtt{l}}$ and $E^{\mathtt{r}}$ if $\mathcal{C}$ contains the respective values. We distinguish between the cases that node μ is a P, an R, or an S node. In all cases, `EmbedSkeleton` computes a set of up to two embedding candidates of which one that is LR-feasible is returned. If no embedding candidate is LR-feasible, $\perp$ is returned.

First, let node μ be a P node with vertices u and v (line 6). A **RUP** embedding of the P node's auxiliary skeleton always has the structure as displayed in Fig. 3.64, i. e., all edges from

u to v are consecutive in the rotation system and so are the edges from v to u. In the following, we compute the list ρ_{uv} which defines a left-to-right order on the set of edges pointing from u to v (see Fig. 3.64). We maintain three sets of non-cyclic edges: $E^{\mathtt{l}}_{uv}$ contains all non-cyclic edges from u to v that must be leftmost, and $E^{\mathtt{r}}_{uv}$ contains all non-cyclic edges from v to u that must be rightmost (line 7). All remaining non-cyclic edges from u to v are in $E^{\sim}_{uv}$.

Before we analyze how `EmbedSkeleton` computes ρ_{uv}, it should be noted that during the computation of ρ_{uv} in lines 6 to 18 some tests are not performed that, under certain circumstances, would immediately imply that $\overrightarrow{\text{skel}}(\mu)$ has no LR-feasible **RUP** embedding. For instance, it is not tested whether there is a cyclic-L edge and a non-cyclic edge in $E^{\mathtt{l}}_{uv}$, which would imply that no LR-feasible **RUP** embedding exists. Instead, these tests are performed at the end of `EmbedSkeleton` in lines 26 to 33.

We denote by $(\cdot)$ the concatenation of two lists. If there is a cyclic-L or cyclic-LR edge $e^{\mathtt{l}}$ in $\overrightarrow{\text{skel}}(\mu)$, the respective auxiliary edges must be the first in ρ_{uv} (line 10), where $e^{\mathtt{l}}_{uv}$ denotes the auxiliary edge of $e^{\mathtt{l}}$ that points from u to v. In line 12, the edges from the sets $E^{\mathtt{l}}_{uv}$, $E^{\sim}_{uv}$, and $E^{\mathtt{r}}_{uv}$ are append to ρ_{uv} in order. Hence, edges that must be leftmost are appended first and edges that must be rightmost are appended last. Note that the order of the vertices within one of these sets in ρ_{uv} is arbitrary. Finally, if there is a cyclic-R or cyclic-LR edge, the respective auxiliary edge is last in ρ_{uv}. If e must be not leftmost and points from u to v, `EmbedSkeleton` moves e to a position in ρ_{uv} such that it is not first. Likewise, if must not be rightmost and $e = (u, v)$, e is moved to a position such that it is not last. If this is not possible, for instance, if e is the single edge from u to v, $\perp$ is returned (line 15).

By the same method as described before, we obtain the list ρ_{vu} for all edges from v to u (line 16). In line 17, the embedding of the auxiliary $\mathcal{E}^{\text{aux}}$ is obtained by defining $\rho_{vu} \cdot \overleftarrow{\rho}_{uv}$ as the rotation system of u, where $\overleftarrow{\rho}_{uv}$ is the reversal of ρ_{uv}. Likewise, the rotation system of v is $\rho_{uv} \cdot \overleftarrow{\rho}_{vu}$. The so obtained embedding is the embedding candidate in $\overline{\mathcal{E}}^{\text{aux}}$. Note that by construction, the embedding candidate is **RUP** such that an L-cyclic is leftmost and an R-cycle is rightmost. Overall, computing this embedding takes time $\mathcal{O}(|V_\mu| + |E_\mu|)$.

If μ is an S node, the embedding of $\overrightarrow{\text{skel}}(\mu)$ is unique and stored in $\overline{\mathcal{E}}$ (line 20). For an R node, the unique embedding and its inversion is also stored in $\overline{\mathcal{E}}$ (line 21). Note that the embedding of an R node and its inversion is computed in time $\mathcal{O}(|V_\mu|)$ [KW01]. For each embedding in $\overline{\mathcal{E}}$ (line 23), we obtain an embedding $\mathcal{E}^{\text{aux}}$ of the auxiliary skeleton $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ by adopting the embedding $\mathcal{E}$ and by replacing each cyclic-L edge by a leftmost cycle, each cyclic-R edge by a rightmost cycle, and a cyclic-LR edge by both. Remember, a cycle is leftmost (rightmost) if it encloses a source (sink) in the dual. The so obtained embeddings are the candidates in $\overline{\mathcal{E}}^{\text{aux}}$. All of the aforementioned steps are carried out in $\mathcal{O}(|V_\mu|)$.

Finally, `EmbedSkeleton` tests for each embedding candidate $\mathcal{E}^{\text{aux}}$ of the auxiliary whether it is LR-feasible and **RUP** in lines 26 to 33. Note that by construction, an L-cycle is left- and an R-cycle is rightmost in embedding $\mathcal{E}^{\text{aux}}$. First, the dual skeleton $\text{aux}(\overrightarrow{\text{skel}}(\mu))^* = (F_\mu, E^*_\mu)$ is computed in time $\mathcal{O}(|V_\mu| + |E_\mu|)$. The dual must be an acyclic dipole which takes $\mathcal{O}(|F_\mu| + |E^*_\mu|) \subseteq \mathcal{O}(|V_\mu| + |E_\mu|)$ time to test. If it is an acyclic dipole, we obtain the information whether all vertices $V^{\mathtt{l}}$ are leftmost and all vertices $V^{\mathtt{r}}$ are rightmost. Furthermore, for each virtual edge $e \in E^{\mathtt{l}}$ ($e \in E^{\mathtt{r}}$), at least one of $\text{aux}(e)$ must be leftmost (rightmost) according to Lem. 3.30. The last test ensures that if e must not be leftmost (rightmost), it is indeed not leftmost (rightmost). If one of these tests fails, the candidate is rejected. Otherwise, an LR-feasible **RUP** embedding of $\overrightarrow{\text{skel}}(\mu)$ is obtained by applying the function aux^{-1} to $\text{aux}(\overrightarrow{\text{skel}}(\mu))$ endowed with the embedding $\mathcal{E}^{\text{aux}}$. Function aux^{-1} first ensures that all auxiliary

edges belonging to a single cyclic edge are consecutive in the rotation system. If this is not the case, aux^{-1} returns $\perp$. Note that this can only happen if μ contains an LR-edge e' and $\overrightarrow{\text{skel}}(\mu) \setminus \{e'\}$ is no acyclic dipole by Lem. 3.36. Consequently, e' is refined by another node μ' which cannot be **RUP** by Cor. 3.28 and the whole compound cannot be **RUP** (at least with the alignment as defined in $\overrightarrow{\text{skel}}(\mu)$). Hence, `EmbedSkeleton` returns $\perp$. Otherwise, the embedding of $\overrightarrow{\text{skel}}(\mu)$ is obtained from aux^{-1} as shown in Fig. 3.48 on page 182. These last steps need a running time of $\mathcal{O}(|V_\mu| + |E_\mu|)$ and, hence, the overall running time of `EmbedSkeleton` is in $\mathcal{O}(|V_\mu| + |E_\mu|)$. □

This concludes the proof of Theorem 3.2.

3.6 wSUP Digraphs

In this section, we study **wSUP** digraphs. Remember that a digraph is **SUP** if it is embeddable on the standing cylinder such that all edge curves are monotonically increasing in y-direction. These digraphs are characterized as spanning subgraphs of planar, acyclic dipoles [Han06, LMS06, Has01]. Here, we turn our attention to weak upward drawings: A digraph is **wSUP** if it is embeddable on the standing cylinder such that all edge curves are non-decreasing in y-direction. In particular, an edge curve may entirely or at least partially stay on the same y-coordinate. An example of a **wSUP**-embedded digraph is shown in Fig. 3.66(a). In this section, we derive a characterization of **wSUP** digraphs by means of their primals and duals. For this we use compound digraphs (Sect. 3.4.1).

First, we investigate cycles in **wSUP** digraphs. Whereas **SUP** digraphs are always acyclic, a **wSUP** digraph may have horizontal cycles (shaded regions in Fig. 3.66(a)). These cycles are always (vertex-)disjoint.

Lemma 3.38. *All cycles in a* **wSUP** *digraph are disjoint.*

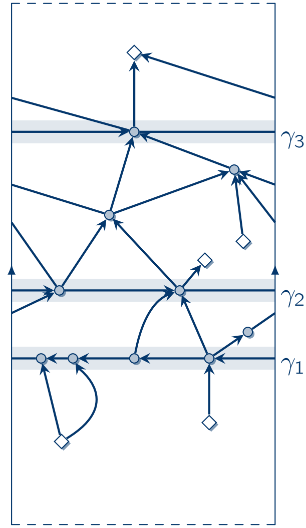
Proof. Let Γ be a **wSUP** drawing of a digraph G. For contradiction, suppose that G has two non-disjoint cycles C_1 and C_2 with vertex v in common. Each of C_1 and C_2 wind around the cylinder horizontally exactly once. All vertices in C_1 have the same y-coordinate and so have all vertices in C_2. Since v is in both cycles, all vertices of C_1 and C_2 have the same y-coordinate and, hence, there is an edge in C_1 and an edge in C_2 with common points in Γ which contradicts planarity. Hence, C_1 and C_2 must be vertex-disjoint and, thus, also edge-disjoint. □

We call a cycle *edge-simple* if it contains no multiple edges.

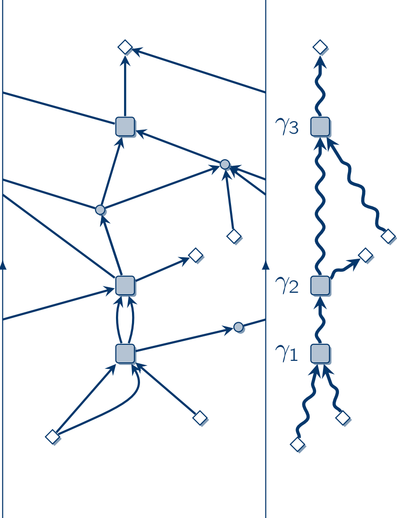
Corollary 3.29. *Each compound in a* **wSUP** *digraph is an edge-simple cycle.*

To characterize **wSUP** digraphs, we use the same strategy as for **RUP** digraphs in Sect. 3.4. Remember that for **RUP** digraphs, we first applied compound digraphs to closed digraphs and their duals, and then showed that each **RUP** digraph is a spanning subgraph of a closed **RUP** digraph. Here, we show that every **wSUP** digraph is a subgraph of a **wSUP** dipole. Note that in contrast to **RUP** digraphs, the supergraph is not spanning for **wSUP** digraphs. For instance, we can introduce new edges to the **wSUP** digraph in Fig. 3.66(a), which contains multiple sources and sinks, until we obtain its **wSUP** supergraph in Fig. 3.66(c). For our proof, we extend techniques for **SUP** digraphs from [Has01].

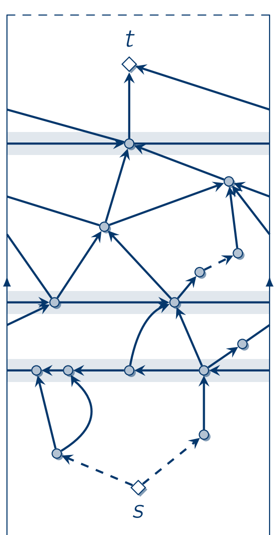
Lemma 3.39. *A digraph is* **wSUP** *if and only if it is a subgraph of a* **wSUP** *dipole.*



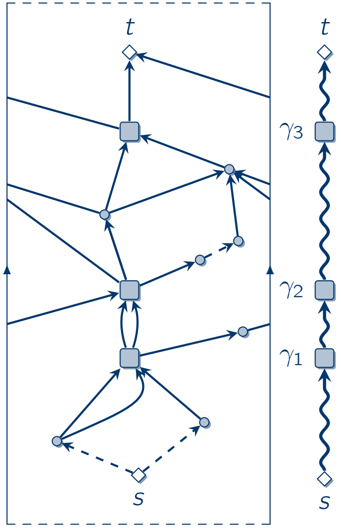
(a) A **wSUP**-embedded digraph.



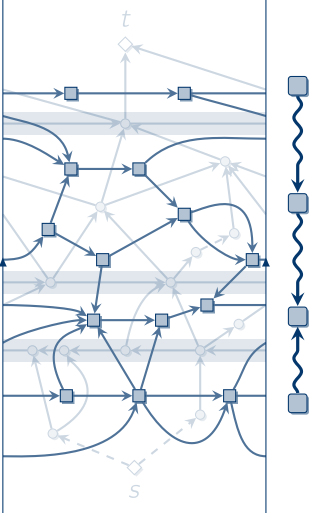
(b) Component and compound digraph of the digraph from Fig. 3.66(a).



(c) The source s and the dashed edges have been introduced to obtain a **wSUP** digraph with exactly one source and one sink.



(d) The component and compound digraph, where the latter is a dipath from s to t and, hence, the whole digraph is a dipole.



(e) The dual of the digraph in Fig. 3.66(c) with its compound digraph. The dual is a **RUP**-embedded closed digraph, where the transits consist of edges only.

Figure 3.66: A **wSUP** digraph and its augmentation to a **wSUP** dipole. Fig. 3.66(e) shows the dual of the augmented digraph.

Proof. ⇒: Let G be a **wSUP**-embedded digraph and let $\mathbb{G}$ and $\overline{\mathbb{G}}$ be its component and compound digraph, respectively. According to Cor. 3.29, all compounds in $\mathbb{G}$ are edge-simple cycles. For our example, its component and compound digraph are shown in Fig. 3.66(b). The compounds subdivide $\mathbb{G}$ into *sections*, i. e., the union of transits whose source is the lower compound or whose sink is the upper compound or both. For instance, the section between γ_2 and γ_3 is the union of three transits. As G is planar, no edge can span multiple sections (see also proof of Lem. 3.5). As a compound contains neither sources nor sinks, we focus on the sections.

Let σ be an *intermediate section*, i. e., a section bounded by a lower compound γ_i and an upper compound γ_{i+1}. As σ is acyclic and **wSUP** it is also **SUP** [Has01]. Hence, it is the subgraph of a planar and acyclic dipole $\overline{\sigma}$. Due to the construction of $\overline{\sigma}$ in [Has01], $\overline{\sigma}$ respects the **wSUP** embedding of σ, i. e., removing all vertices and edges from $\overline{\sigma}$ to obtain σ yields the original embedding of σ. Further, γ_i is the single source and γ_{i+1} the single sink of $\overline{\sigma}$. If σ is *extremal*, i. e., it is the lowermost or the uppermost section (or both) and not bounded by a compound on the lower or upper end, then a source or sink, respectively, is chosen according to the construction given in [Has01]. Thereby, we obtain a planar, acyclic dipole for every section. Now, we expand the compounds to horizontal cycles again by adjusting the embedding accordingly: Let $e_1, \dots, e_r$ be the rotation system of the source in a section and assume that this source is a compound γ_i. Denote by C_i the cycle that is γ_i. If due to the construction from before outgoing edges have been added to the source, these edges are attached to vertices of γ_i such that the contraction of C_i results in the rotation system $e_1, \dots, e_r$ of the source γ_i. This construction assures planarity and does not introduce any new cycles. In the end, we obtain a **wSUP** digraph with at most one source, situated in the lowermost section, and at most one sink in the uppermost section. If there is no uppermost (lowermost) section, we add a vertex and an edge from (to) one of the vertices of the uppermost (lowermost) compound. The so obtained supergraph H of G is **wSUP** and contains exactly one source s and one sink t. Note that H is not necessarily a spanning supergraph of G as we may have introduced a new source or a new sink.

What is left to show is that H fulfills the properties of a dipole for which we use Lem. 3.1 on page 108. Since s is the single source and t the single sink, there is a dipath from s to every vertex v and from v to t which implies (i) of Lem. 3.1. By planarity, each transit in H is situated between two cycles, i. e., compounds. In other words, the compounds can be totally ordered by $\gamma_1, \dots, \gamma_k$ such that each transit either points from γ_i to γ_{i+1} for $1 \leq i < k$, or from s to γ_1, or from γ_k to t. Thereby and by the fact that each compound winds around the cylinder, a dipath from s to t contains a vertex from each compound. Hence, (ii) follows and we can conclude that H is a dipole.

⇐: Any subgraph of a **wSUP** digraph is also **wSUP**. □

With the help of Lem. 3.39 and Cor. 3.29, we readily arrive at a characterization of **wSUP**.

Theorem 3.7. *A digraph is* **wSUP** *if and only if it is a subgraph of a planar dipole whose compounds are edge-simple cycles.*

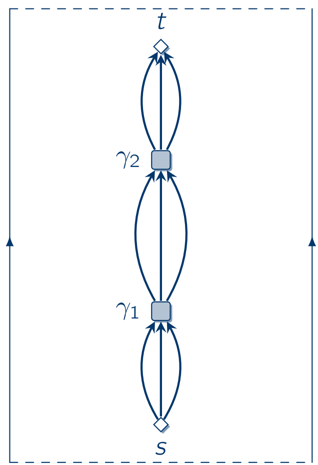
Proof. ⇒: Let G be a **wSUP** digraph. Then, the **wSUP** supergraph H constructed according to Lem. 3.39 is a dipole and by Lem. 3.38 all cycles of H are disjoint.

⇐: Let H be an embedded dipole with edge-simple cycles only such that G is a subgraph of H. Consider the component digraph $\mathbb{H}$ of H. As H is a dipole with source s and sink t, its compound digraph is a dipath $(s, \tau_1, \gamma_1, \dots, \gamma_{k-1}, \tau_k, t)$ for $k \geq 0$. This implies that the

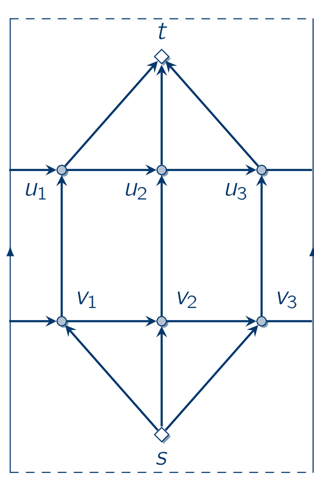
component digraph $\mathbb{H}$ itself is an acyclic dipole and, hence, its embedding is **SUP**. Thereby, we obtain a **SUP** drawing $\Gamma_{\mathbb{H}}$ of $\mathbb{H}$. By assumption, each compound γ_i is an edge-simple cycle C_i. In $\Gamma_{\mathbb{H}}$, we expand each of these cycles C_i according to the embedding of H such that the vertices of C_i all have the same y-coordinate as γ_i in $\Gamma_{\mathbb{H}}$.

At this point it is important to make a subtle and yet crucial observation: In a **wSUP** drawing, the edges of a cycle C_i either all point to the right side or to the left side. Expanding any of the cycles such that it has the wrong orientation, can lead to crossings between edges from the compound to the neighboring transit. Consider, for instance, the component digraph in Fig. 3.67(a) that contains two compounds of which each is a cycle of three vertices. Fig. 3.67(b) shows that expanding both cycles such that both point to the right leads to a plane drawing. In contrast, if the edges of γ_1 point to the left and the edges of γ_2 point to the right, we inevitably get a crossing as shown in Fig. 3.67(c). In our proof, we assure planarity by expanding the compounds according to H's embedding.

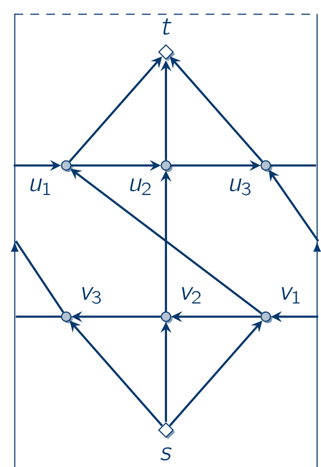
Further, the compound digraph of H is a path from s to t and, therefore, there is no transit that "overleaps" a compound, i. e., each transit points from γ_i to γ_{i+1}, from s to γ_1, or from γ_{k-1} to t. The obtained drawing of H is **wSUP** and, therefore, H and G are **wSUP**. □



(a) Component digraph of an embedded dipole.



(b) Both cycles point to the right and the drawing is plane.



(c) The cycle of γ_1 points to the left and the cycle of γ_2 in opposite direction which inevitably leads to a crossing.

Figure 3.67: Expanding a cycle of a **wSUP** digraph such that it has the wrong orientation can lead to crossings.

In Sect. 3.4, we have characterized **RUP** digraphs by means of their duals. We do the same for **wSUP** digraphs. For the example in Fig. 3.66(c), consider its dual in Fig. 3.66(e). In Sect. 3.4, we have proved that a closed digraph is **RUP**-embedded if and only if its dual is a dipole (follows from Thm. 3.1). As every **wSUP** digraph is the subgraph of a **wSUP** dipole, we obtain:

Corollary 3.30. *A* **wSUP** *digraph is the subgraph of an embedded digraph whose dual is* **RUP**.

The converse is not true as the compounds in a **wSUP** digraph have a very particular and simple structure which, in turn, implies a special structure of the transits of the dual. Let G be a **wSUP**-embedded dipole. Then, each compound of G is an edge-simple cycle. In the dual G^*, an edge-simple cycle becomes a transit which is a "bundle" of parallel edges from its source to its sink. In particular, each transit in G^* contains no face besides its source and sink. This also implies that all faces belong to compounds of G^*. Remember that a (strongly-connected component) is called trivial if it contains no edges (Sect. 3.4.1). We arrive at the following characterization of **wSUP** digraphs.

Theorem 3.8. *A digraph is* **wSUP** *if and only if it is the subgraph of an embedded digraph H such that H^* is* **RUP***-embedded and contains no trivial components.*

Proof. $\Rightarrow$: Let G be **wSUP**, then by Lem. 3.39 it is the subgraph of a **wSUP**-embedded dipole H. Thm. 3.1 and Prop. 3.1 imply that H^* is **RUP**.

If H is acyclic, then H^* is strongly connected and, hence, H^* has no trivial components. Otherwise, let γ be a compound of H. γ is an edge-simple cycle and, hence, its dual is an acyclic dipole with a single source and a single sink, and one or more edges in between them. Cor. 3.2 and the proof of Lem. 3.5 assert that the dual of γ is a transit of H^*. Hence, each face in H^* belongs to a compound since the transits consist of edges only. In other words, H^* contains no trivial components.

$\Leftarrow$: Suppose that G is a subgraph of an embedded dipole H whose dual H^* is **RUP**-embedded and which contains no trivial components. By Thm. 3.1, H is a dipole. Further, each face in H^* belongs to a compound and, thereby, none of the transits of H^* contains a face besides its source and sink. Hence, the primal of each transit of H^* is an edge-simple cycle in H and, consequently, all cycles in H are edge-simple. By Thm. 3.7, we can conclude that G is **wSUP**. $\square$

3.7 Summary, Further Remarks, and Future Work

At the end of this chapter, we take the opportunity to summarize, make further remarks, and give possible directions for future work.

First, we have characterized **RUP** digraphs by means of their duals (Sect. 3.4.3). The key insight, which paved the way to this characterization, is that a compound is **RUP**-embedded if and only if its dual is an acyclic dipole (Lem. 3.4). We extended this result to closed **RUP** digraphs by introducing the compound digraph and by generalizing acyclic dipoles to dipoles: a closed digraph is **RUP**-embedded if and only if its dual is a dipole (Thm. 3.1). Finally, we have obtained a characterization of all **RUP** digraphs by showing that a digraph is **RUP** if and only if it is the spanning subgraph of a closed **RUP** digraph (Lem. 3.7). Compound digraphs and the insights about the duals of **RUP** digraphs have also proven to be valuable for characterizing **wSUP** digraphs and their duals (Sect. 3.6).

Tab. 3.2 gives an overview of known and new (bold font) characterizations of different classes of upward planar digraphs (first column). "SC" abbreviates "strongly connected". Each digraph of one of the classes is the spanning subgraph of a characterizing planar supergraph (second column), e. g., a digraph is **SUP** if and only if it is the spanning subgraph of a planar acyclic dipole. The third column displays the principle structure of the supergraph by its compound digraph, e. g., two terminals connected by a transit in case of an acyclic dipole.

	Primal		Dual	
	Characterizing Supergraph	Compound Digraph	Structure	Compound Digraph
UP	planar acyclic dipole with *st*-edge [Kel87, DBT88]		**SC & RUP (Lem. 3.4)**	
SUP	planar acyclic dipole [Has01]		**SC & RUP (Lem. 3.4)**	
wSUP	**planar dipole, edge-simple cycles (Thm. 3.7)**	**compounds are edge-simple cycles**	**closed & RUP only edges in transits (Thm. 3.8)**	**no trivial components**
RUP	**planar closed digraphs, dual is dipole (Thm. 3.1)**		**dipole (Thm. 3.1)**	

Table 3.2: Overview of the different classes of upward planar digraphs, their characterizing supergraphs, compound digraphs, duals and compound digraphs of the duals.

The fourth column lists the properties of the dual of the characterizing supergraph and the compound digraph of the dual is shown in the last column.

As testing whether a digraph is **RUP** is $\mathcal{NP}$-hard in general [Bra14], we focused on closed digraphs for which we developed a linear-time algorithm consisting of three parts (Sect. 3.5): The first part considers compounds and transits (almost) separately, the second part deals with compounds, and the last part with biconnected compounds. For each part, we used and extended a high-level description, i. e., the compound digraph, the block-cut-tree, and the SPQR tree. Tab. 3.3 shows an overview, where the first column lists the classes of digraphs, corresponding to the input of the respective part of the algorithm, and the second column shows the used high-level description. For each of part of the algorithm, we derived a **RUP** characterization of the primal and dual by means of the respective high-level description. The properties of the primal and dual are shown in the third and fourth column, respectively, where characterizing properties are displayed in bold font, e. g., a compound is **RUP** if and only if its directed block-cut tree contains a Eulerian dipath.

Most notably, we have extended SPQR trees to incorporate dual graphs by defining dual SPQR trees in which the skeletons of the nodes are replaced by their duals. In this context, one of our main results is that the dual SPQR tree is the SPQR tree of the dual (Thm. 3.4). In order to deal with directed edges, we introduced dSPQR trees of acyclic digraphs and of compounds, which we in turn combined with dual SPQR trees to obtain dual dSPQR trees. As with dual SPQR trees, we have proved that the dual dSPQR tree is the dSPQR tree of the dual (Thm. 3.5). By using these extensions, we have derived a neat characterization: a biconnected compound is **RUP**-embedded if and only if each node of its dSPQR tree is **RUP**-embedded, if

	High-Level Description	Primal	Dual
closed digraph	compound digraph	path (Lem. 3.8)	dipath (Thm. 3.1)
compound	block-cut tree	caterpillar (Cor. 3.17)	path (Lem. 3.14)
	directed block-cut tree	**contains Eulerian dipath (Thm. 3.3)**	
biconnected compound	(dual) dSPQR tree	**nodes are RUP (Thm. 3.6)**	**nodes are acyclic dipoles (Thm. 3.6)**

Table 3.3: Used high-level description of the algorithm in Sect. 3.5 and their properties for **RUP** digraphs and their duals.

and only if each node of its dual dSPQR tree is an acyclic dipole (Thm. 3.6). In particular, the property of being **RUP**-embedded is visible in each node of the dSPQR tree and, conversely, if each node is **RUP** then so is the whole compound. All these extensions are defined in general in order to be applicable in a wider range, potentially encompassing other cases of upward planarity.

At the beginning of this chapter, our goal was to extend the study of upward planarity to rolling upward planarity. It is justified to claim that we have reached the first two major milestones towards this goal: a characterization and a decision algorithm for closed digraphs. On our way, the dual digraph has turned out to be a most helpful traveling companion and its value is twofold: First, the study of duals has led to the first combinatorial characterization of **RUP** and to new insights to **SUP** and **wSUP** (see Tab. 3.2). Second, this characterization is "algorithmically effective", that is, it can be turned into a decision algorithm. The latter aspect justly raises the hope that the developed characterizations and algorithmic tools are useful in an even wider range, in particular, in cases of upward planarity that allow for a characterization by means of duals. In this sense, we close this chapter by discussing possible future research.

3.7.1 Minors of Non-RUP Compounds

A celebrated result by Robertson and Seymour is that a graph class that is closed under taking minors is characterized by a set of finitely many forbidden minors [RS04]. That is, a graph is in the respective class if and only if none of its minors is in the set of forbidden minors. Remember, a graph G is a minor of another graph H if G can be obtained from H by contracting edges and removing vertices and edges. For instance, the planar graphs are closed under taking minors and, indeed, by Wagner's theorem [Wag37], their forbidden minors are the complete graph K_5 with five vertices and the complete bipartite graph $K_{3,3}$ with three vertices in each partition. Forbidden minors are in so far interesting as they are minimal archetypes of the graphs that not fulfill a certain property, e. g., planarity. In the context of **RUP**, we can also ask for forbidden minors, where we concentrate on compounds.

As all **RUP** compounds are planar, the K_5 and the $K_{3,3}$ are forbidden minors of **RUP** compounds. Note that the K_5 and the $K_{3,3}$ are forbidden minors of the *underlying undirected graphs* of **RUP** compounds. In the following, we will derive directed minors by using the same

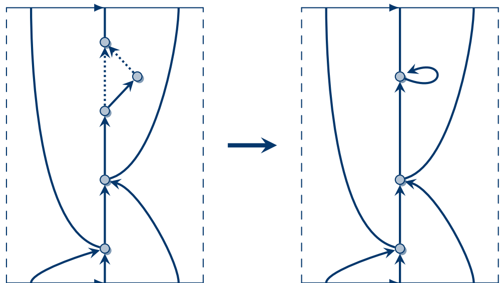


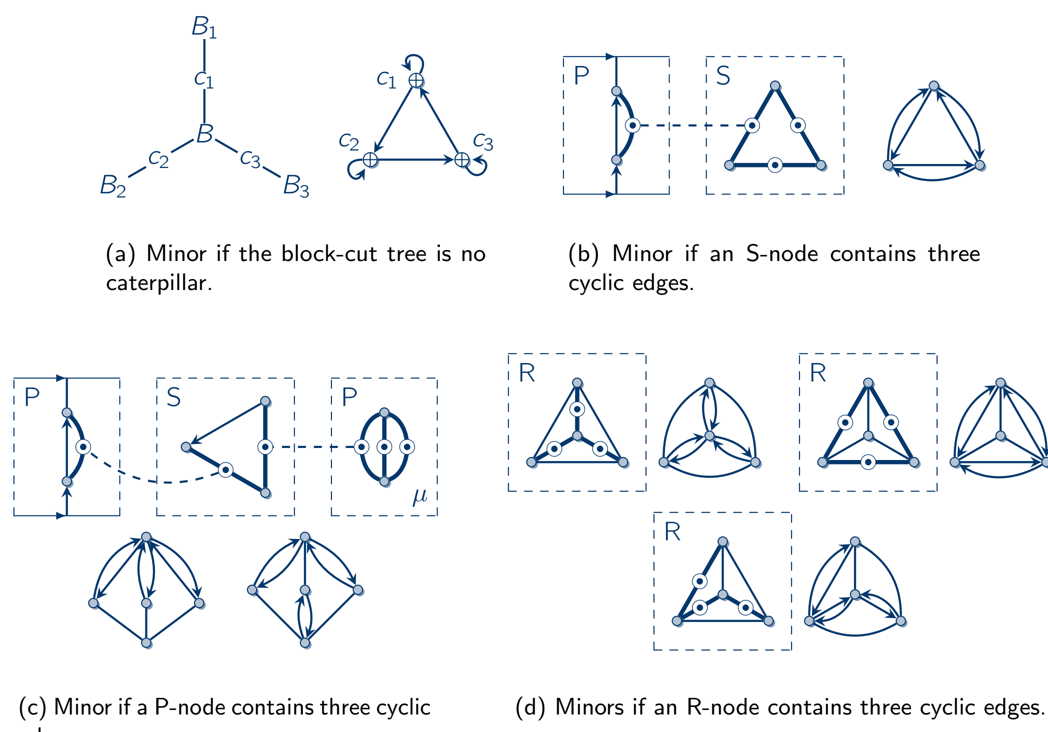
Figure 3.68: Contracting the dotted edges in the **RUP** compound results in a non-**RUP** compound.

operations as with undirected graphs. For an example, consider the **RUP** compound on the left side of Fig. 3.68. When we contract the dotted edges, we obtain the directed minor on the right side which is also a compound. Since we only deal with directed minors in the following, we omit the attribute "directed". Also note that the minor on the right side of Fig. 3.68 is not **RUP** as the loop cannot wind around the cylinder. Hence, **RUP** digraphs are not closed under taking minors and we cannot apply the theorem of Robertson and Seymour.

Nevertheless, we can use our testing algorithm from Sects. 3.5.2 and 3.5.3 to derive minors of non-**RUP** compounds. The idea is that whenever `false` is returned, we identify a non-**RUP** minor that causes the algorithm to abort. We will also ensure that the non-**RUP** minor is minimal in the sense that removing a vertex or an edge, or contracting an edge results in a **RUP** digraph. For instance, if the block-cut tree of a compound is no caterpillar, `TestCompound` returns `false` (line 3 in Alg. 3.3). A tree is no caterpillar if and only if it contains a graph isomorphic to the one shown on the left side of Fig. 3.69(a) as subgraph [HS71]. In case of a block-cut tree, we have a block B which shares three cut vertices c_1, c_2, and c_3 with three blocks B_1, B_2, and B_3, respectively. If the block-cut tree of a compound γ contains this configuration, we can remove all edges and vertices from γ such that we obtain another compound γ' whose block-cut tree looks as on the left side of Fig. 3.69(a). As B is strongly connected, it contains a cycle through c_1, c_2, c_3. Likewise, each of B_1, B_2, and B_3, contains a cycle through c_1, c_2, and c_3, respectively. Removing all edges and vertices from γ' that belong not to these cycles and contracting the cycles afterwards such that they only contain c_1, c_2, and c_3, results in the compound γ'' on the right side of Fig. 3.69(a). Note that each of c_1, c_2, and c_3 with the loop is a block. γ'' is a minor of γ and not **RUP** as its block-cut tree is no caterpillar.

There are **RUP** digraphs that contain γ'' as minor. For instance, by contracting the dotted edges in the compound on the left side of Fig. 3.68, we obtain the compound on the right side which is isomorphic to γ'' and is not **RUP**. Alas, this destroys any hope of finding a characterizing set of forbidden **RUP** minors: if the compound is not **RUP** we may identify a culpable minor but finding such a minor does not imply that the compound is not **RUP**.

Another situation in which a (biconnected) compound γ is not **RUP** is if one of the nodes in its dSPQR tree contains more than two cyclic virtual edges (line 1 in `ComputeCompound-Embedding`; Alg. 3.9). Assume that this is the case for an S node. The expansion digraph of each of these cyclic virtual edges contains a cycle. Then, γ has a minor γ' whose dSPQR tree contains an S node whose skeleton consists of three cyclic virtual edges. Each of these cyclic virtual edges is refined by a P node that contains an antiparallel pair of directed non-virtual



(a) Minor if the block-cut tree is no caterpillar.

(b) Minor if an S-node contains three cyclic edges.

(c) Minor if a P-node contains three cyclic edges.

(d) Minors if an R-node contains three cyclic edges.

Figure 3.69: Minors of non-**RUP** compounds.

edges. On the left side of Fig. 3.69(b), the S node is displayed and only one of the P nodes is shown as the other two are equal. The corresponding non-**RUP** compound is shown at the right side of Fig. 3.69(b). Note, the minor is minimal in the sense that removing or contracting any of the edges results in a **RUP** digraph.

Likewise, a P node can contain three or more cyclic virtual edges. In this case, we can find a minor where the P node μ has exactly three cyclic virtual edges (see top of Fig. 3.69(c)) of which each is refined by an S node. Each of these S nodes consists of a directed non-virtual edge and two virtual edges of which one is refined by μ and the other one by a P node that contains an antiparallel pair of directed non-virtual edges. All other virtual edges of μ are refined in the same way. The directed edge in each S node must be oriented such that the whole digraph is strongly connected. At the bottom of Fig. 3.69(c) the possible non-**RUP** compounds are shown where the undirected edges must be directed such that the digraph is strongly connected.

If an R node contains three cyclic virtual edges, then we can derive a minor such that the skeleton of the R node is a complete graph with four vertices with three cyclic virtual edges and three directed non-virtual edges. Each of the virtual edges is refined by a P node which contains an antiparallel pair of directed non-virtual edges. Fig. 3.69(d) illustrates the three principle cases that can occur, where the P nodes are omitted: either all three virtual edges are incident to one vertex (top left), they form a circle (top right), or a path (bottom). The corresponding non-**RUP** compounds are shown to the right of each R node, where the undirected edges can be directed arbitrarily as long as the digraph is strongly connected. Note

that the non-**RUP** compound in the top right corner of Fig. 3.69(d) is not minimal as it has as minor the digraph at the right side of Fig. 3.69(b).

By investigating the other cases where `false` is returned, we can derive further minors of non-**RUP** digraphs and, thereby, better understand the structure of (non-)**RUP** compounds.

Open Problem 3.1. *What are other minors of non-***RUP** *compounds? Are there finitely many?*

3.7.2 Upward Toroidal Digraphs

A digraph is *upward toroidal* or **TUP** if it has a crossing-free drawing that is upward on the surface of the torus endowed with the homogeneous field (cf. Tab. 3.1 on page 100), i. e., all edge curves are monotonically increasing in y-direction. An example of a closed **TUP** digraph is shown in Fig. 3.70, where the corresponding compound digraph is shown beneath on the standing cylinder. Similar to **RUP**, cycles can wind around in vertical direction. The digraph in Fig. 3.70 is not **RUP** by Cor. 3.3 as the underlying undirected graph of its compound digraph is a circle. Observe how the digraph winds around the torus in horizontal direction which is reflected in the circular structure of its compound digraph. Assume now that we remove any of the transits, e. g., transit τ_4. Thereby, we obtain a closed digraph which is **RUP** and for which our testing algorithm `TestClosedDigraph` in Sect. 3.5 returns a **RUP** embedding. By using this observation, we can use `TestClosedDigraph` for a **TUP** test for closed digraphs: In a nutshell, we remove a transit from the digraph if its compound digraph is a circle. Then, we test if the remaining closed digraph is **RUP** and, if this the case, we test if we can reinsert the transit into the **RUP** embedding to obtain a **TUP** embedding. In the following, we see how this works in detail. For the correctness of the approach, we will make some assumptions whose proofs we leave for future work.

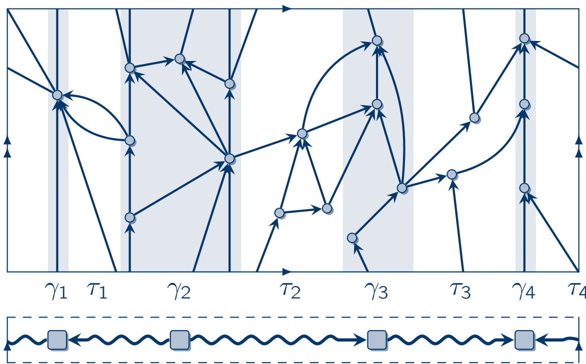


Figure 3.70: Example of a **TUP** digraph.

Consider `TUPTestClosedDigraph` in Alg. 3.13. Let G be the closed input digraph. For reasons discussed later, we assume that G is not strongly connected. If G is **RUP**, it is certainly also **TUP** (line 1). Hence, we assume that G is not **RUP**. `TUPTestClosed-Digraph` uses `ComputeCompoundDigraph` (Alg. 3.1) to compute the compound digraph $\overline{\mathbb{G}}$ of G. `ComputeCompoundDigraph` returns $\perp$ if the transits are not independent. Further, if the underlying undirected graph of $\overline{\mathbb{G}}$ is no circle, $\perp$ is returned (line 4). Note that as G is not strongly connected, it contains at least one transit.

Assumption 3.1. *If G is* **TUP** *and not* **RUP**, *then the underlying undirected graph of its compound digraph is a circle and all its transits are independent.*

Let $\gamma_1, \tau_1, \gamma_2, \dots, \tau_{k-1}, \gamma_k, \tau_k, \gamma_1$ be the compounds and transits of the compound digraph in order of a traversal of the circle. We obtain a closed digraph $\tilde{G}$ by removing all trivial components in τ_k from G (line 6). `TestClosedDigraph` is used to test whether $\tilde{G}$ is **RUP**.

Assumption 3.2. *If G is* **TUP**, *then $\tilde{G}$ is* **RUP**.

Denote by $\mathcal{E}_{\tilde{G}}$ the **RUP** embedding of $\tilde{G}$. Similar to `TestClosedDigraph` in Alg. 3.2, τ_k is attached to the leftmost cycle C^{l} and the rightmost cycle C^{r} of $\tilde{G}$. For this, a partial rotation system $\mathcal{E}'_{\tau_k}$ at the source and sink of τ_k is derived in line 11 (see Sect. 3.5.1.1 for details). The routine `PlanarConstraint` from [GKM07] is used to test whether τ_k can be embedded according to $\mathcal{E}'_{\tau_k}$.

Assumption 3.3. *If G is* **TUP**, *then τ_k has an embedding that respects $\mathcal{E}'_{\tau_k}$.*

Let $\mathcal{E}_{\tau_k}$ be the embedding obtained from `PlanarConstraint`. By assembling $\mathcal{E}_{\tilde{G}}$ with the embedding $\mathcal{E}'_{\tau_k}$, we obtain an embedding $\mathcal{E}$ on the torus which is upward.

Assumption 3.4. *The embedding $\mathcal{E}$ returned in line 15 is a* **TUP** *embedding.*

Note that `TUPTestClosedDigraph` uses `TestClosedDigraph` and its subroutines, and the overall running time of `TUPTestClosedDigraph` is in $\mathcal{O}(|V|)$ (Thm. 3.2).

Open Problem 3.2. *Prove Assumptions 3.1 to 3.4.*

Algorithm 3.13. `TUPTestClosedDigraph`

Input: closed digraph $G = (V, E)$ which is not strongly connected
Output: **TUP** embedding; or $\perp$ if G is not **TUP**

1 **if** `TestClosedDigraph`$(G) \neq \perp$ **then return RUP** embedding of G
2 $\overline{\mathbb{G}} \leftarrow$ `ComputeCompoundDigraph`(G)
3 **if** $\overline{\mathbb{G}} = \perp$ **then return** $\perp$
4 **if** underlying undirected graph of $\overline{\mathbb{G}}$ is no circle **then return** $\perp$
5 Compound digraph is circle $\gamma_1, \tau_1, \gamma_2, \dots, \tau_{k-1}, \gamma_k, \tau_k, \gamma_1$
6 $\tilde{G} \leftarrow G$ without trivial components in τ_k
7 $\mathcal{E}_{\tilde{G}} \leftarrow$ `TestClosedDigraph`$(\tilde{G})$
8 **if** $\mathcal{E}_{\tilde{G}} = \perp$ **then return** $\perp$
9 $C^{\mathrm{l}} \leftarrow$ leftmost cycle of $\tilde{G}$ according to $\mathcal{E}_{\tilde{G}}$
10 $C^{\mathrm{r}} \leftarrow$ rightmost cycle of $\tilde{G}$ according to $\mathcal{E}_{\tilde{G}}$
11 $\mathcal{E}'_{\tau_k} \leftarrow$ partial rotation system at the source and sink of τ_k according to C^{l} and C^{r}
12 $\mathcal{E}_{\tau_k} \leftarrow$ `PlanarConstraint`$(V_{\tau_k}, E_{\tau_k}, \mathcal{E}'_{\tau_k})$
13 **if** $\mathcal{E}_{\tau_k} = \perp$ **then return** $\perp$
14 $\mathcal{E} \leftarrow$ embedding of G obtained by assembling the embedding $\mathcal{E}_{\tilde{G}}$ of $\tilde{G}$ with the embedding $\mathcal{E}_{\tau_k}$ of τ_k
15 **return** $\mathcal{E}$

Remember that we assume that the input digraph G is not strongly connected. If G is strongly connected, then we obtain a single compound as compound digraph. Hence, the

trick of removing a transit from G does not work as there is no transit. However, if we could identify acyclic components within G, similar to transits, such that removing such a component would result in a closed digraph that is **RUP** if G is **TUP**, then we could extend `TUPTestClosedDigraph` to work for closed digraphs.

Open Problem 3.3. *Which extensions to the definitions of compounds, transits, and the compound digraph are needed to identify acyclic and non-trivial strongly connected components in strongly connected digraphs? Based on these extensions, is it possible to extend* `TUPTest-ClosedDigraph` *to work for closed digraphs?*

3.7.3 Drawing RUP Digraphs

In Sect. 3.1, we have motivated the study of **RUP** by their usefulness in graph drawing, in particular, when visualizing periodic processes or recurrent hierarchies [Bru10]. Computing a **RUP** embedding, though, is only half the way to a concise drawing that exhibits the cyclic structures of the **RUP** digraph.

Drawings of **RUP** digraphs have been investigated in [Bra14]: Given a **RUP** drawing of a digraph, the drawing can be simplified such that its edge curves are polylines, i. e., geodesics on the rolling cylinder, with at most two bends per edge and such that each edge winds at most once fully around the cylinder. Further, the vertices and bends are placed on an integer grid of cubic size. This simplification process is constructive and can be carried out in time $\mathcal{O}(\tau n^3)$, where n is the number of vertices and τ the time to compute the intersection of an edge curve with a horizontal line. However, this method presumes that we are already given a **RUP** drawing, leading to a chicken-and-egg situation.

The extension of the Sugiyama framework to recurrent hierarchies in [Bru10, BBBF12, BBBL09, BBBH11] produces cyclic drawings which can be used to aptly display cycles (see Fig. 3.3(c) on page 98 for an example) . The recurrent hierarchy algorithm adapts the four phases of the classical Sugiyama framework [STT81], where phase 1, the removal of cycles, is omitted. The other three phases are the positioning of the vertices on cyclic levels, which are half-lines extending from the origin (see Fig. 3.3(c)), along with the introduction of dummy vertices (phase 2; [Bru10, BBBL09]), the minimization of crossings (phase 3; [Bru10, BBBH11]), and the assignment of coordinates (phase 4; [Bru10, BBBF12]). The result is a cyclic drawing with an area consumption that, in the worst case, is cubic in the number of vertices. In its current form, the recurrent hierarchy algorithm ignores any given embedding of the input digraph, which raises the question for a suitable adaption to incorporate a **RUP** embedding. In particular, the positioning of the vertices on the cyclic levels (phase 2) needs to respect the **RUP** embedding and the upward direction of the edges. Phase 3 can be omitted due to planarity and phase 4 needs no adaption.

Open Problem 3.4. *Is it possible to adapt phase 2 of the recurrent hierarchy approach from [BBBL09] to incorporate a given* **RUP** *embedding?*

For strongly connected **RUP** digraphs, the following approach could be viable: In the proof of Lem. 3.4, we have shown how to construct a **RUP** drawing for a given strongly connected **RUP**-embedded digraph. As the aim of the proof was to show that such a drawing exists, the construction neither produces a "nice" drawing nor are the vertices placed on integer coordinates. Nevertheless, such a drawing could be used as a starting point for the simplification process in [Bra14]. Even more, from the construction in [Bra14], we obtain a leveling for the recurrent

hierarchy approach as follows: In the drawing, we insert a horizontal line through each vertex and, thereby, obtain the leveling and the dummy vertices, i. e., the points where the horizontal line crosses an edge curve, and the order of the (dummy) vertices on each level. However, this construction produces a level for each vertex and is cumbersome as a drawing is generated in order to obtain another "nicer" drawing. Nevertheless, the idea itself could be the starting point for a more stringent approach.

Open Problem 3.5. *Can the construction from the proof of Lem. 3.4 be adapted to directly obtain a leveling with few levels?*

From a theoretical viewpoint, it is also interesting how large a **RUP** drawing might get. As discussed before, an area consumption cubic in the number of vertices can always be achieved if bends are allowed [Bra14]. The picture changes drastically if we forbid bends: there are **UP** digraphs that require an exponential area when drawn straight-line and upward without bends [DBTT92]. For the proof, Di Battista et al. construct a triconnected acyclic dipole with n vertices and show that the minimal area consumption of a straight-line **UP** drawing is in $\Omega(2^n)$. Note that, in principle, this acyclic dipole may require asymptotically less area on the rolling cylinder as it can wind around the cylinder multiple times to reduce area consumption. Let G be the triconnected and acyclic dipole as constructed in [DBTT92] with source s and sink t. We can insert G into a face of a triconnected and **RUP**-embedded compound γ as shown in Fig. 3.71. Now, G cannot wind around the rolling cylinder as otherwise the leftmost or rightmost cycle of γ would wind around the cylinder multiple times which is not possible due to planarity. Hence, we have "trapped" G in a face that is homeomorphic to the plane and, hence, its drawing must be **UP**, leading to an exponential area consumption.

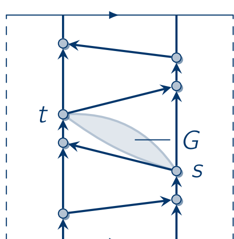


Figure 3.71: A **RUP**-embedded compound into which an acyclic dipole is inserted that requires exponential area when drawn straight-line.

3.7.4 Further Complexity Considerations

3.7.4.1 Other Digraph Classes

In Sect. 3.5, we have developed a rich set of tools to decide whether a closed digraph is **RUP**. This raises the question whether these tools can be used for other classes of digraphs or when additional information is given. For instance, deciding whether an embedding is **UP** is efficiently solvable [BDBLM94] and so is the respective **SUP** decision problem for single-source digraphs [DH08]. For **RUP**, we obtain the following corollary.

Corollary 3.31. *Let $G = (V, E)$ be a closed and embedded digraph. Deciding whether the embedding of G is* **RUP** *can be done in time $\mathcal{O}(|V|)$.*

Proof. By Lemmas 3.3 and 3.5, the embedding is **RUP** if and only if its dual is a dipole. Remember, a digraph is a dipole if it has exactly one source s and exactly one sink t and the compound digraph is a dipath $s \rightsquigarrow t$. From the embedding of G, the dual $G^* = (F, E^*)$ is obtained in $\mathcal{O}(|V|)$. Testing whether G^* contains only a single source s and a single sink t takes $\mathcal{O}(|F|) \subseteq \mathcal{O}(|V|)$ many time steps. `ComputeCompoundDigraph` in Alg. 3.1 is used to compute the compound digraph $\overline{\mathbb{G}}^*$ of G^* in time $\mathcal{O}(|V|)$. If `ComputeCompoundDigraph` returns $\perp$, the transits of G^* are not independent. In this case, G and also its embedding are not **RUP** as the underlying undirected graph of $\overline{\mathbb{G}}^*$ is no path (cf. proof of Cor. 3.3). Finally, testing whether $\overline{\mathbb{G}}^*$ is a dipath $s \rightsquigarrow t$ takes $\mathcal{O}(|V|)$ time steps. □

We leave the general case for future research.

Open Problem 3.6. *Given an embedded digraph, is there an efficient algorithm to decide whether the embedding is* **RUP***?*

Deciding whether a digraph is **UP** becomes efficiently solvable for certain classes of planar digraphs, e. g., outerplanar digraphs [Pap95], which raises the hope that there are classes for which testing **RUP** becomes efficiently solvable.

Open Problem 3.7. *Are there classes of planar digraphs for which testing whether they are* **RUP** *is solvable in polynomial time?*

3.7.4.2 Fixed-Parameter Tractability

Fixed-parameter tractability has already been discussed in the context of deque layouts in Sect. 2.6.1.2: Given the input to an ($\mathcal{NP}$-hard) problem and a parameter k that depends on the input, the problem is fixed-parameter tractable (FPT) if there is an algorithm that decides the problem and has a running time in $\mathcal{O}(f(k) \cdot n^p)$, where $f(k)$ is a function that depends on k, n is the length of the input and p is an integer constant. As deciding whether a digraph is **RUP** is $\mathcal{NP}$-hard in general [Bra14], we can ask for parameters that make the problem FPT.

For instance, the number of terminals, i. e., sources and sinks, is a possible choice for a parameter k. By the proof of Lem. 3.7 we know that a digraph is **RUP** if and only if we can introduce an edge from each sink to some other vertex and to each source from another vertex such that the resulting closed digraph is **RUP**. Given a digraph with n vertices, we try out all k^n possibilities to obtain a closed digraph and for each we use `TestClosedDigraph` (Alg. 3.2) to test whether it is **RUP** in time $\mathcal{O}(n)$. This leads to an overall running time of $\mathcal{O}(k^n \cdot n)$ and, unfortunately, this is no FPT algorithm. Still, by applying the tools from Sects. 3.5.2 and 3.5.3, the principle idea is still of interest. For instance, connecting a terminal to any vertex makes no sense if this destroys planarity. If, however, a terminal is situated within the skeleton of an R node, we can connect it to a vertex that is incident to a face to which the terminal is also incident.

Open Problem 3.8. *Is deciding whether a digraph G is* **RUP** *FPT if the number of terminals in G is the parameter?*

Deciding whether a digraph is **UP** is FPT, where the parameter is the number of triconnected components or, alternatively, the difference between the number of edges and vertices in the digraph [HL06].

Open Problem 3.9. *Are there other parameters, e. g., the number of triconnected components, that make the* **RUP** *decision problem FPT?*

3.7.5 Acyclic RUP Digraphs

It comes as no surprise that **SUP** is a proper subset of **RUP** as **SUP** digraphs are acyclic. But even when restricted to acyclic digraphs, **RUP** is a proper extension of **SUP** [ABBG11, Bra14]. The example from [ABBG11] of an acyclic **RUP** digraph that is not **SUP** is shown in Fig. 3.5 on page 103. Note that it is triconnected and, thus, its embedding is unique. The digraph cannot be augmented to a planar acyclic dipole: attaching an outgoing edge to any of the two sinks would either introduce a cycle or destroy planarity. Acyclic **RUP** digraphs lie between **SUP** and **RUP** and combine the property of **SUP** digraphs to be acyclic with the ability to wind around the rolling cylinder, which makes them an interesting class to study. Especially, separating acyclic **RUP** digraphs from **SUP** digraphs may reveal new properties of **RUP** digraphs as well as **SUP** digraphs. Possible questions that could be tackled are as follows:

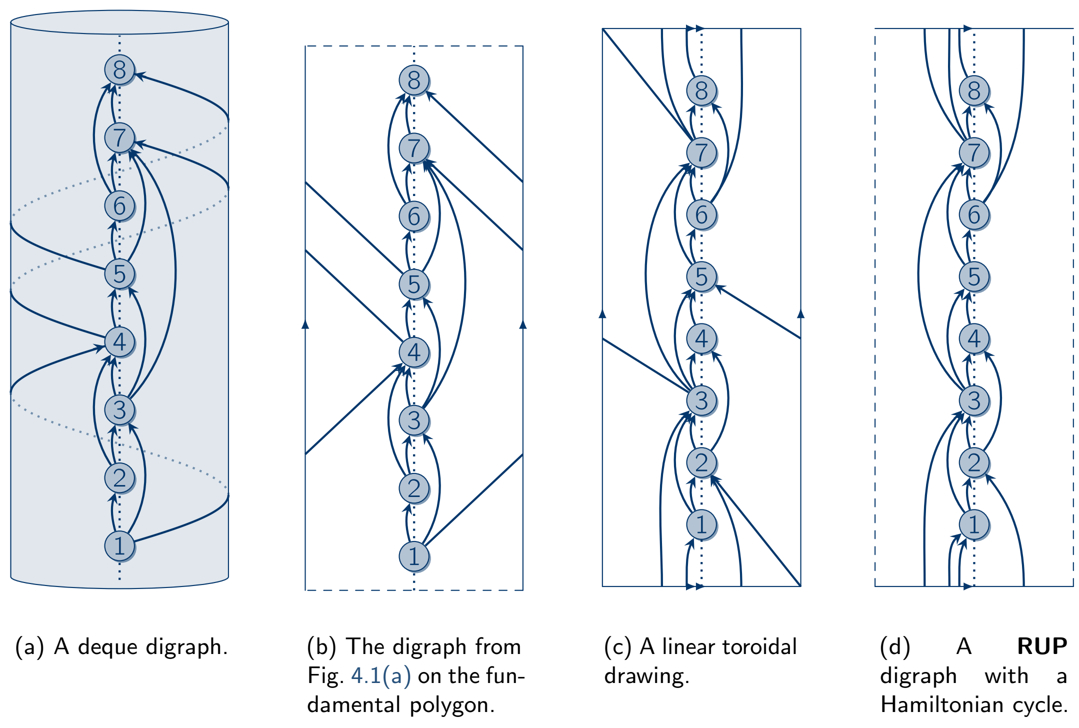
Open Problem 3.10. *How can acyclic* **RUP** *digraphs or their duals be characterized? Given an acyclic* **RUP** *digraph, possibly with a* **RUP** *embedding, how hard is it to decide whether the digraph is* **SUP***?*

Note that deciding whether an acyclic digraph is **RUP** is also $\mathcal{NP}$-hard [Bra14].

Chapter 4

Bringing Chapters 2 and 3 Together

Now that we have reached the end of the thesis, a question still remains: what is the relationship between linear cylindric drawings from Chapter 2 and upward planarity on cylinder surfaces discussed in Chapter 3? In this last chapter, we propose an answer to this question.



(a) A deque digraph.

(b) The digraph from Fig. 4.1(a) on the fundamental polygon.

(c) A linear toroidal drawing.

(d) A **RUP** digraph with a Hamiltonian cycle.

Figure 4.1: A deque digraph and a cyclic two-stack digraph.

Graph layouts can also be defined for digraphs where each edge is inserted to the data structure at its source and removed at its target [HPT99a, HPT99b]. We call an acyclic digraph G *deque digraph* if it has a *directed deque layout* $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$ where $\prec$ must be a topological ordering of G. Analogously, we call an acyclic digraph *linear cylindric planar* if

it has a linear cylindric plane drawing where the order of the vertices on the front line is a topological ordering of G. By Thm. 2.1, we get:

Corollary 4.1. *An acyclic digraph is a deque digraph if and only if it is linear cylindric planar.*

An example of a deque digraph and its linear cylindric drawing on the standing cylinder is shown in Fig. 4.1(a) where, this time, the front line (dotted) is vertical. Fig. 4.1(b) shows the representation on the fundamental polygon. Note that the digraph contains a Hamiltonian dipath. Similar to the undirected case, it is always possible to introduce directed edges to a linear cylindric planar digraph to obtain a Hamiltonian dipath (cf. Thm. 2.2).

Corollary 4.2. *An acyclic digraph is a deque digraph if and only if it is a spanning subgraph of an acyclic planar digraph with a Hamiltonian dipath.*

An acyclic digraph with a Hamiltonian dipath contains a single source and a single sink and is, thus, an acyclic dipole.

Corollary 4.3. *An acyclic digraph is a deque digraph if and only if it is the spanning subgraph of a* **SUP** *digraph with a Hamiltonian dipath.*

In particular, a linear cylindric planar digraph always has a linear cylindric plane drawing that is **SUP**. We call the drawing and the digraph *linear* **SUP** .

Corollary 4.4. *For an acyclic digraph G, the following statements are equivalent:*

(i) G is a deque digraph.

(ii) G is linear **SUP***.*

(iii) G is the subgraph of a **SUP** *digraph with a Hamiltonian dipath.*

Cor. 4.4 relates upward planarity with deque layouts and is our first step towards the goal of this chapter.

The queue edges in a deque layout are the ones that wind around the cylinder. Such edges are possible in **SUP** but not in **UP**. Similar to linear **SUP** drawings, we call a linear cylindric drawing that is **UP** *linear* **UP** . A digraph that has a linear **UP** drawing is also called *linear* **UP**. This leads to the following characterization of two-stack digraphs:

Corollary 4.5. *For an acyclic digraph, the following statements are equivalent:*

(i) G is a two-stack digraph.

(ii) G is linear **UP***.*

(iii) G is the subgraph of a **UP** *digraph with a Hamiltonian dipath.*

Proof. (i) $\Rightarrow$ (ii): Let G be a two-stack digraph with directed two-stack layout $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}})$, i. e., Σ is a directed deque layout with no queue edges. As in Sect. 2.2.4, we define the acyclic $\prec$-augmentation $G_\prec$ of G which contains the Hamiltonian dipath $p = (v_1, \dots, v_n)$ with $v_i \prec v_j$ if and only if $i < j$ for all $1 \leq i, j \leq n$. The corresponding $\prec$-augmentation of Σ is $\Sigma_\prec$. Remember that all edges on p are processed in $\Sigma_\prec$ as stack edges and, thus, $\Sigma_\prec$ is a two-stack layout. We introduce the edge (v_1, v_n) to $G_\prec$, if not existent, to obtain the digraph $G_\prec^+$ and extend $\Sigma_\prec$ to $\Sigma_\prec^+$ such that (v_1, v_n) is inserted at v_1 before all other edges and removed at v_n

after all other edges. Since there are no queue edges in $\Sigma_\prec$, $\Sigma_\prec^+$ is also a two-stack layout. By Cor. 4.4, $G_\prec^+$ has a linear **SUP** drawing which is also **UP** as $G_\prec^+$ contains the edge from the single source v_1 to the single sink v_n.

(ii) ⇒ *(iii)*: Assume that G is linear **UP** where the order of the vertices on the front line is $v_1, \dots, v_n$. We introduce the edge (v_1, v_n) to G, if not existent, to obtain the digraph G^+. G^+ is linear **UP** since G is linear **UP**. By applying Cor. 4.4, G^+ is a spanning subgraph of a **SUP** digraph $\overline{G}^+$ with a Hamiltonian dipath. Since $\overline{G}^+$ also contains the edge from v_1, which is the single source, to v_n, which is the single sink, we can conclude that $\overline{G}^+$ is **UP**.

(iii) ⇒ *(i)*: Let G^+ be the spanning supergraph of G that is a planar acyclic dipole with source s and sink t, and the edge from s to t. Similar to the proof of Cor. 2.4, G^+ is linear cylindric planar and, thus, a deque digraph. In the respective directed deque layout of G^+, we define that edge (s, t) is a stack edge which is inserted at s first and is removed at t last. Thereby, no edge can be a queue edge, and G^+ and G are two-stack digraphs. □

Remember, in the undirected case, a graph is a two-stack graph if and only if it is the spanning subgraph of a planar graph with a Hamiltonian circle [BK79]. The deque is characterized by the Hamiltonian path (Thm. 2.2). In the directed case, the difference between deque and two-stack layouts corresponds to the difference between upward planarity on the standing cylinder and the plane. Still, the characterizations of the undirected case "shine through": Every two-stack digraph is the spanning digraph of a **UP** digraph with a Hamiltonian dipath p and the edge from the first vertex of p to the last vertex. In particular, the underlying undirected graph contains a Hamiltonian circle.

The standing cylinder allows for queue edges, which raises the question what we obtain for the rolling cylinder. For an answer, we generalize deque layouts to *cyclic deque layouts*: Remember, in a deque layout, the input to the first vertex is empty and so is the output of the last vertex. By loosening this restriction, we define *cyclic deque layouts*: instead of an empty input, the first vertex receives a deque with content $\mathcal{C}_0$ as input and the last vertex must produce $\mathcal{C}_0$ as output, where the elements in $\mathcal{C}_0$ are edges of the digraph. A *cyclic deque schedule* $\Sigma^\circ = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}}, \mathcal{C}_0)$ is called *cyclic deque layout* if `IsCyclicDequeLayout` in Alg. 4.1 returns `true`. As before, we assume that each directed edge must be inserted at its source and must be removed at its target. Note that in comparison to `IsDequeLayout` in Alg. 2.2, `IsCyclicDequeLayout` initializes $\mathcal{C}$ to $\mathcal{C}_0$ instead of the empty content, and returns `true` only if the last vertex produces $\mathcal{C}_0$ as output. A digraph is a *cyclic deque digraph* if it has a cyclic deque layout.

Algorithm 4.1. `IsCyclicDequeLayout`

Input: digraph $G = (V, E)$ and cyclic deque schedule $\Sigma = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}}, \mathcal{C}_0)$
Output: `true` if cyclic deque schedule is a cyclic deque layout and `false` otherwise

```
1 C ← C_0
2 foreach v_i = v_1, ..., v_n in order of ≺ do
3 |  C ← ProcessVertex(C, ≺, E^h(v_i), E^t(v_i))
4 |  if C = ⊥ then return false
5 if C = C_0 then return true
6 else return false
```

Corollary 4.6. *A digraph is a cyclic deque digraph if and only if it is a spanning subgraph of a cyclic deque digraph with a Hamiltonian cycle.*

Proof. $\Rightarrow$: Let G be a cyclic deque digraph with cyclic deque layout $\Sigma^\circ = (\prec, E^{\mathtt{h}}, E^{\mathtt{t}}, \mathcal{C}_0)$. W. l. o. g., $v_1, \dots, v_n$ are the vertices ordered according to $\prec$. Similar to deque layouts, we augment G and Σ° by introducing edges (v_i, v_{i+1}) for all $1 \leq i < n$ such that all these edges are processed as stack edges at the head of the deque (cf. Cor. 2.3). Additionally, we introduce the edge (v_n, v_1) which is inserted at the head at v_n after all other edges incident to v_n are processed. Likewise, (v_n, v_1) is removed at v_1 before all other edges of v_1 are processed. Furthermore, we modify $\mathcal{C}_0$ by inserting (v_n, v_1) at its head. The so obtained digraph contains a Hamiltonian cycle and the modified cyclic deque schedule is a cyclic deque layout.

$\Leftarrow$: A subgraph of a cyclic deque digraph is also a cyclic deque digraph. ☐

Just as we defined linear cylindric drawings for deque layouts, we define *linear toric drawings* for cyclic deque layouts. Remember, the fundamental polygon of the torus is defined by $\mathbf{T} = I_\circ \times I_\circ$, where $I_\circ$ is obtained from the unit interval $I = (-1, 1)$ by identifying its boundaries. The set of points $h = \{(x, y) \in \mathbf{T} \mid x = 0\}$ form a closed curve on $\mathbf{T}$, where h is called *front circle*. In a linear toric drawing, the vertices are placed on distinct points of h and the edge curves must not share a point with h except for its endpoints. A linear toric drawing without edge crossings is called *linear toroidal* and so is a digraph that admits a linear toroidal drawing. An example of a linear toroidal drawing is shown in Fig. 4.1(c) where the front circle is dotted and winds around the torus in vertical direction.

Claim 4.1. *A digraph is a cyclic deque digraph if and only if it is linear toroidal.*

The outline of the proof is as follows: By adopting the definition of linear cylindric rotation systems (Def. 2.3), a linear toric drawing defines a linear toric rotation system which, in turn, defines a cyclic deque schedule. The edges in $\mathcal{C}_0$ are the ones that "overleap" the region between the last and the first vertex in the linear toric drawing, e. g., edges $(6, 2)$, $(7, 3)$, $(7, 2)$, and $(8, 1)$ in Fig. 4.1(c). By a proof along the lines of the proof of Lem. 2.1, there are no edge crossings in the linear toric rotation system if and only if the cyclic deque schedule is a cyclic deque layout. Note that there are non-planar linear toroidal digraphs, for instance, the K_5 with an appropriate orientation of the edges.

Remember, every deque digraph is linear **SUP** (cf. Cor. 4.4). The edge curves of the linear toric drawing in Fig. 4.1(c) are (rolling) upward and, thus, the drawing is **TUP** (toroidal upward plane). In this case, we call the drawing and the digraph *linear* **TUP** .

Claim 4.2. *For a digraph G, the following statements are equivalent:*

(i) G is a cyclic deque digraph.

(ii) G is linear **TUP***.*

(iii) G is the subgraph of a linear **TUP** *digraph with a Hamiltonian cycle.*

Linear cyclic deque layouts and **TUP** drawings combine the ability of queue edges to wind around in horizontal direction, as in **SUP**, with the ability of edges in $\mathcal{C}_0$ to wind around in vertical direction, as in **RUP**. Hence, the standing cylinder corresponds to the deque's queue mode whereas the rolling cylinder reflects the non-empty input $\mathcal{C}_0$. If we forbid queue edges, we obtain cyclic two-stack digraphs with the following characterization by Cor. 4.6 and Claims 4.1 and 4.2:

Claim 4.3. *For a digraph G, the following statements are equivalent:*

(i) G *is a cyclic two-stack digraph.*

(ii) G *is linear* **RUP**, *i. e.,* G *has a linear toric drawing that is* **RUP**.

(iii) G *is the subgraph of a linear* **RUP** *digraph with a Hamiltonian cycle.*

For an example of a linear **RUP** drawing, see Fig. 4.1(d), where the digraph contains a Hamiltonian cycle.

In the undirected case, we have used the following informal equation in Sect. 2.2.4 to illustrate the difference between deque and two-stack layouts:

$$\frac{\text{deque layout}}{\text{two-stack layout}} = \frac{\text{planar \& Hamiltonian path}}{\text{planar \& Hamiltonian circle}}.$$

In a similar manner, we summarize the results and claims of this chapter as follows:

$$\frac{\text{directed deque layout}}{\text{cyclic two-stack layout}} = \frac{\text{linear } \mathbf{SUP}}{\text{linear } \mathbf{RUP}} = \frac{\mathbf{SUP} \text{ \& Hamiltonian dipath}}{\mathbf{RUP} \text{ \& Hamiltonian cycle}}.$$

Open Problem 4.1. *Prove Claims 4.1 and 4.2.*

Open Problem 4.2. *Are there more relationships between graph layouts and upward planarity?*

Bibliography

[ABB+10] Christopher Auer, Christian Bachmaier, Franz J. Brandenburg, Wolfgang Brunner, and Andreas Gleißner. Plane drawings of queue and deque graphs. In *Proc. 18th International Symposium on Graph Drawing, GD 2010*, volume 6502 of *LNCS*, Heidelberg, Germany, 2010. Springer Verlag.

[ABB+12a] Christopher Auer, Christian Bachmaier, Franz J. Brandenburg, Andreas Gleißner, and Kathrin Hanauer. The duals of upward planar graphs on cylinders. Technical Report MIP-1204, Faculty of Informatics and Mathematics, University of Passau, 2012. http://www.fim.uni-passau.de/en/research/forschungsberichte/mip-1204.html.

[ABB+12b] Christopher Auer, Christian Bachmaier, Franz J. Brandenburg, Andreas Gleißner, and Kathrin Hanauer. The duals of upward planar graphs on cylinders. In Martin Charles Golumbic, Michal Stern, Avivit Levy, and Gila Morgenstern, editors, *Proc. 38th Workshop on Graph-Theoretic Concepts in Computer Science, WG 2012*, volume 7551 of *LNCS*, pages 103–113. Springer Verlag, 2012.

[ABBG11] Christopher Auer, Christian Bachmaier, Franz J. Brandenburg, and Andreas Gleißner. Classification of planar upward embedding. In Marc van Kreveld and Bettina Speckmann, editors, *Proc. 19th International Symposium on Graph Drawing, GD 2011*, volume 7034 of *LNCS*, pages 415–426, Heidelberg, Germany, 2011. Springer Verlag.

[ABBH13] Christopher Auer, Christian Bachmaier, Franz J. Brandenburg, and Kathrin Hanauer. Rolling upward planarity testing of strongly connected graphs. In Andreas Brandstädt, Klaus Jansen, and Rüdiger Reischuk, editors, *Proc. 39th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2013*, volume 8165 of *LNCS*, pages 38–49, Heidelberg, Germany, 2013. Springer Verlag.

[ABGH13] Christopher Auer, Franz J. Brandenburg, Andreas Gleißner, and Kathrin Hanauer. Characterizing planarity by the splittable deque. In Stephen Wismath and Alexander Wolff, editors, *Proc. 21st International Symposium on Graph Drawing, GD 2013*, LNCS, Heidelberg, Germany, 2013. Springer Verlag. (to appear).

[ABR13] Patrizio Angelini, Thomas Bläsius, and Ignaz Rutter. Testing mutual duality of planar graphs. In Leizhen Cai, Siu-Wing Cheng, and Tak-Wah Lam, editors, *Algorithms and Computation*, volume 8283 of *LNCS*, pages 350–360. Springer Verlag, 2013.

[AG11] Christopher Auer and Andreas Gleißner. Characterizations of deque and queue graphs. In Petr Kolman and Jan Kratochvíl, editors, *Proc. 37th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2011*, volume 6986 of *LNCS*, Heidelberg, Germany, June 2011. Springer Verlag.

[AGHV12] Christopher Auer, Andreas Gleißner, Kathrin Hanauer, and Sebastian Vetter. Testing planarity by switching trains. In Walter Didimo Didimo and Maurizio Patrignani, editors, *Proc. 20th International Symposium on Graph Drawing, GD 2012*, volume 7704, pages 557–558, Heidelberg, Germany, September 2012. Springer Verlag.

[Atn68] G.Ã. Atneosen. *On the Embeddability of Compacta in N-books: Intrinsic and Extrinsic Properties*. Michigan State University, Department of Mathematics, 1968.

[Aye85] Kathleen Ayers. Deque automata and a subfamily of context-sensitive languages which contains all semilinear bounded languages. *Theor. Comput. Sci.*, 40(0):163–174, 1985.

[Bac07] Christian Bachmaier. A radial adaption of the Sugiyama framework for visualizing hierarchical information. *IEEE Trans. Vis. Comput. Graphics*, 13(3):583–594, 2007.

[BB80] Ronald V. Book and Franz J. Brandenburg. Equality sets and complexity classes. *SIAM J. Comput.*, 9(4):729–743, 1980.

[BB08] Christian Bachmaier and Wolfgang Brunner. Linear time planarity testing and embedding of strongly connected cyclic level graphs. In Dan Halperin and Kurt Mehlhorn, editors, *Proc. European Symposium on Algorithms, ESA 2008*, volume 5193 of *LNCS*, pages 136–147, Heidelberg, Germany, 2008. Springer Verlag.

[BBBF12] Christian Bachmaier, Franz J. Brandenburg, Wolfgang Brunner, and Raymund Fülöp. Drawing recurrent hierarchies. *J. Graph Alg. App.*, 16(2):151–198, 2012.

[BBBH11] Christian Bachmaier, Franz J. Brandenburg, Wolfgang Brunner, and Ferdinand Hübner. Global k-level crossing reduction. *J. Graph Alg. App.*, 15(5):631–659, 2011.

[BBBL09] Christian Bachmaier, Franz J. Brandenburg, Wolfgang Brunner, and Gergö Lovász. Cyclic leveling of directed graphs. In Ioannis G. Tollis and Maurizio Patrignani, editors, *Proc. 17st International Symposium on Graph Drawing, GD 2009*, volume 5417 of *LNCS*, pages 348–359. Springer Verlag, Heidelberg, Germany, 2009.

[BBF05] Christian Bachmaier, Franz J. Brandenburg, and Michael Forster. Radial level planarity testing and embedding in linear time. *J. Graph Alg. App.*, 9(1):53–97, 2005.

[BCDB+92] Paola Bertolazzi, R.F̃. Cohen, G. Di Battista, Roberto Tamassia, and I.G̃. Tollis. How to draw a series-parallel digraph. In Otto Nurmi and Esko Ukkonen, editors, *Proc. 3rd Scandinavian Workshop on Algorithm Theory, SWAT 1992*, volume 621 of *LNCS*, pages 272–283. Springer Verlag, Heidelberg, Germany, 1992.

[BDBD02] Paola Bertolazzi, Giuseppe Di Battista, and Walter Didimo. Quasi-upward planarity. *Algorithmica*, 32(3):474–506, 2002.

[BDBLM94] Paola Bertolazzi, Giuseppe Di Battista, Giuseppe Liotta, and Carlo Mannino. Upward drawings of triconnected digraphs. *Algorithmica*, 12(6):476–497, 1994.

[BDBMT98] Paola Bertolazzi, Giuseppe Di Battista, Carlo Mannino, and Roberto Tamassia. Optimal upward planarity testing of single-source digraphs. *SIAM J. Comput.*, 27(1):132–169, 1998.

[BES13] Michael J. Bannister, David Eppstein, and Joseph A. Simons. Fixed parameter tractability of crossing minimization of almost-trees. In Stephen Wismath and Alexander Wolff, editors, *Proc. 21st International Symposium on Graph Drawing, GD 2013*, LNCS, Heidelberg, Germany, 2013. Springer Verlag. (to appear).

[Bil92] T. Bilski. Embedding graphs in books: a survey. *Computers and Digital Techniques, IEEE Proceedings*, 139(2):134–138, 1992.

[BJC00] Jørgen Bang-Jensen and Gregory Cutin. *Digraphs: Theory, Algorithms and Applications*. Springer Verlag, Heidelberg, Germany, 1st edition, 2000.

[BK79] Frank Bernhart and Paul Chester Kainen. The book thickness of a graph. *J. Combin. Theory, Ser. B*, 27(3):320–331, 1979.

[BKW99] Ulrik Brandes, Patrick Kenis, and Dorothea Wagner. Centrality in policy network drawings. In Jan Kratochvíl, editor, *Proc. 7th International Symposium on Graph Drawing, GD 1999*, volume 1731 of *LNCS*, pages 250–258. Springer Verlag, 1999.

[Bó02] Miklós Bó. A survey of stack-sorting disciplines. *Electr. J. Comb.*, 9(2), 2002.

[Bra80] Franz J. Brandenburg. Multiple equality sets and post machines. *J. Comput. Syst. Sci.*, 21(3):292–316, 1980.

[Bra87] Franz J. Brandenburg. A note on: 'Deque Automata and a Subfamily of Context-Sensitive Languages which Contains all Semilinear Bounded Languages' (by K. Ayers): Theoretical computer science 40(2,3) (1985) 163–174. *Theor. Comput. Sci.*, 52(3):341–342, 1987.

[Bra09] Ulrik Brandes. The left-right planarity test, 2009. http://www.inf.uni-konstanz.de/algo/publications/b-lrpt-sub.pdf (last accessed 2013-12-09).

[Bra14] Franz J. Brandenburg. Upward planar drawings on the standing and the rolling cylinders. *Computational Geometry*, 47(1):25–41, 2014.

[Bru10] Wolfgang Brunner. *Cyclic Level Drawings of Directed Graphs*. Dissertation, Lehrstuhl für Informatik mit Schwerpunkt Theoretische Informatik, Universität Passau, Germany, January 2010.

[BS84] Jonathan F. Buss and Peter W. Shor. On the pagenumber of planar graphs. In *Proc. 16th Annual ACM Symposium on Theory of Computing, STOC 1984*, pages 98–100, New York, NY, USA, 1984. Association for Computing Machinery (ACM).

[CDS07] Sabine Cornelsen and Gabriele Di Stefano. Track assignment. *J. Discrete Alg.*, 5(2):250–261, 2007.

[Cet13] Matthias Cetto. *Offene Gauß-Codes*. Bachelor's thesis, Lehrstuhl für Informatik mit Schwerpunkt Theoretische Informatik, Universität Passau, Germany, November 2013.

[Chv85] Vašek Chvátal. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley and Sons, New York, NY, USA, 1985.

[CLR87] Fan R. K. Chung, Frank Leighton, Thomson, and Arnold L. Rosenberg. Embedding graphs in books: A layout problem with applications to VLSI design. *SIAM J. Algebra. Discr. Meth.*, 8(1):33–58, 1987.

[Cou90] Bruno Courcelle. The monadic second-order logic of graphs. I. recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990.

[Cou00] Bruno Courcelle. The monadic second-order logic of graphs XII: Planar graphs and planar maps. *Theor. Comput. Sci.*, 237(1–2):1–32, 2000.

[CR01] Derek G. Corneil and Udi Rotics. On the relationship between clique-width and treewidth. In Andreas Brandstädt and Van Bang Le, editors, *Proc. 27th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2001*, volume 2204 of *LNCS*, pages 78–90, Heidelberg, Germany, 2001. Springer Verlag.

[CS63] Noam Chomsky and Marcel-Paul Schützenberger. The algebraic theory of context-free languages. *Computer Programming and Formal Systems*, pages 118–161, 1963.

[dBCvKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer Verlag, Heidelberg, Germany, 3rd ed. edition, 2008.

[DBETT99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[DBF13] Giuseppe Di Battista and Fabrizio Frati. Drawing trees, outerplanar graphs, series-parallel graphs, and planar graphs in a small area. In János Pach, editor, *Thirty Essays on Geometric Graph Theory*, pages 121–165. Springer Verlag, Heidelberg, Germany, 2013.

[DBFP10] Giuseppe Di Battista, Fabrizio Frati, and János Pach. On the queue number of planar graphs. In *Proc. 51st Annual IEEE Symposium on the Foundations of Computer Science, FOCS 2010*, pages 365–374, Washington, DC, USA, 2010. IEEE Computer Society.

[DBLR90] Giuseppe Di Battista, Wei-Ping Liu, and Ivan Rival. Bipartite graphs, upward drawings, and planarity. *Information Processing Letters*, 36(6):317–322, 1990.

[DBN88] Giuseppe Di Battista and Enrico Nardelli. Hierarchies and planarity theory. *IEEE Trans. Syst., Man, Cybern.*, 18(6):1035–1046, 1988.

[DBT88] Giuseppe Di Battista and Roberto Tamassia. Algorithms for plane representations of acyclic digraphs. *Theor. Comput. Sci.*, 61(2–3):175–198, 1988.

[DBT96] Giuseppe Di Battista and Roberto Tamassia. On-line planarity testing. *SIAM J. Comput.*, 25(5):956–997, 1996.

[DBTT92] Giuseppe Di Battista, Roberto Tamassia, and Ioannis G. Tollis. Area requirement and symmetry display of planar upward drawings. *Discrete & Computational Geometry*, 7(1):381–401, 1992.

[DF99] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer Verlag, Heidelberg, Germany, 1999.

[dFdMR06] Hubert de Fraysseix, Patrice Ossona de Mendez, and Pierre Rosenstiehl. Trémaux trees and planarity. *Int. J. Found. Comput. Sci.*, 17(5):1017–1030, 2006.

[dFPP90] Hubert de Fraysseix, János Pach, and Richard Pollack. How to draw a planar graph on a grid. *Combinat.*, pages 40–51, 1990.

[dFR82] Hubert de Fraysseix and Pierre Rosenstiehl. A depth-first-search characterization of planarity. In *Ann. Discrete Math.*, volume 13, pages 75–80. North-Holland Publishing Company, Amsterdam, Netherlands, 1982.

[DGL06] Walter Didimo, Francesco Giordano, and Giuseppe Liotta. Upward spirality and upward planarity testing. In Patrick Healy and Nikola S. Nikolov, editors, *Proc. 14th International Symposium on Graph Drawing, GD 2006*, volume 3843 of *LNCS*, pages 117–128. Springer Verlag, 2006.

[DH08] Ardeshir Dolati and S. Mehdi Hashemi. On the sphericity testing of single source digraphs. *Discrete Math.*, 308(11):2175–2181, 2008.

[DHK08] Ardeshir Dolati, S. Mehdi Hashemi, and Masoud Kosravani. On the upward embedding on the torus. *Rocky Mt. J. Math.*, 38(1):107–121, 2008.

[DM03] Andrea Donafee and Carsten Maple. Planarity testing for graphs represented by a rotation scheme. In Ebad Banissi, Katy Börner, Chaomei Chen, Gordon Clapworthy, Carsten Maple, Amy Lobben, Christopher J. Moore, Jonathan C. Roberts, Anna Ursyn, and Jian Zhang, editors, *Proc. 7th International Conference on Information Visualization, IV 2013*, pages 491–497, Washington, DC, USA, 2003. IEEE Computer Society.

[DMW05] Vida Dujmović, Pat Morin, and David R. Wood. Layout of graphs with bounded tree-width. *SIAM J. Comput.*, 34(3):553–579, 2005.

[Dol08] Ardeshir Dolati. Digraph embedding on t_h. In *Proc. Seventh Cologne Twente Workshop on Graphs and Combinatorial Optimization, CTW 2008*, pages 11–14. University of Milan, May 2008.

[DW04a] Vida Dujmović and David R. Wood. On linear layouts of graphs. *Discrete Math. Theor. Comput. Sci.*, 6(2):339–358, 2004.

[DW04b] Vida Dujmović and David R. Wood. Three-dimensional grid drawings with subquadratic volume. In Giuseppe Liotta, editor, *Proc. 12th Symposium on Graph Drawing, GD 2004*, volume 2912 of *LNCS*, pages 190–201. Springer Verlag, Heidelberg, Germany, 2004.

[DW05] Vida Dujmovic and David R. Wood. Stacks, queues and tracks: Layouts of graph subdivisions. *Discrete Math. Theor. Comput. Sci.*, 7(1):155–202, 2005.

[EGW01a] Wolfgang Espelage, Frank Gurski, and Egon Wanke. Deciding clique-width for graphs of bounded tree-width. In Frank Dehne, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Proc. 7th International Workshop on Algorithms and Data Structures, WADS 2001*, volume 2125 of *LNCS*, pages 87–98, Heidelberg, Germany, 2001. Springer Verlag.

[EGW01b] Wolfgang Espelage, Frank Gurski, and Egon Wanke. How to solve NP-hard graph problems on clique-width bounded graphs in polynomial time. In Andreas Brandstädt and Van Bang Le, editors, *Proc. 27th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2001*, volume 2204 of *LNCS*, pages 117–128, Heidelberg, Germany, 2001. Springer Verlag.

[ENO97] Hikoe Enomoto, Tomoki Nakamigawa, and Katsuhiro Ota. On the pagenumber of complete bipartite graphs. *J. Combin. Theory, Ser. B*, 71(1):111–120, 1997.

[EPL72] S. Even, A. Pnueli, and A. Lempel. Permutation graphs and transitive graphs. *J. ACM*, 19(3):400–410, July 1972.

[Eve71] Itai Even, Shimon und Alon. Queues, stacks, and graphs. In Zvi Kohavi and Azaria Paz, editors, *Proc. International Symposium on the Theory of Machines and Computations*, pages 71–86, New York, NY, USA, August 1971. Academic Press.

[Eve12] Shimon Even. *Graph Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2012.

[FFRV11] Fabrizio Frati, Radoslav Fulek, and Andres J. Ruiz-Vargas. On the page number of upward planar directed acyclic graphs. In Marc van Kreveld and Bettina Speckmann, editors, *Proc. 19th International Symposium on Graph Drawing, GD 2011*, volume 7034 of *LNCS*, pages 391–402, Heidelberg, Germany, 2011. Springer Verlag.

[Fra07] Fabrizio Frati. On minimum area planar upward drawings of directed trees and other families of directed acyclic graphs. In Andreas Brandstädt, Dieter Kratsch, and Haiko Müller, editors, *Proc. 33rd Workshop on Graph-Theoretic Concepts in Computer Science, WG 2007*, volume 4769 of *LNCS*, pages 133–144, Heidelberg, Germany, 2007. Springer Verlag.

[FRU92] Stephan Foldes, Ivan Rival, and Jorge Urrutia. Light sources, obstructions and spherical orders. *Discrete Math.*, 102(1):13–23, 1992.

[Gau00] Carl Friedrich Gauß. *Werke*. Teubner, Leipzig, Germany, 1900.

[GH75] A. Goldner and Frank Harary. Note on a smallest nonhamiltonian maximal planar graph. *Bull. Malaysian Math. Soc.*, 6(1):41–42, 1975.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GKM07] Carsten Gutwenger, Karsten Klein, and Petra Mutzel. Planarity testing and optimal edge insertion with embedding constraints. In Michael Kaufmann and Wagner Dorothea, editors, *Proc. 14th International Symposium on Graph Drawing, GD 2007*, volume 4372 of *LNCS*, Heidelberg, Germany, 2007. Springer Verlag.

[GM01] Carsten Gutwenger and Petra Mutzel. A linear time implementation of SPQR-trees. In Joe Marks, editor, *Proc. 8th International Symposium on Graph Drawing, GD 2000*, volume 1984 of *LNCS*, pages 77–90, Heidelberg, Germany, 2001. Springer Verlag.

[Gol04] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, volume 57 of *Annals of Discrete Mathematics*. North-Holland Publishing Company, Amsterdam, Netherlands, 2004.

[GS99] Petra M. Gleiss and Peter F. Stadler. Relevant cycles in biopolymers and random graphs. In *Proc. 4th Slovene International Conference in Graph Theory*, June 1999.

[GT01a] Ashim Garg and Roberto Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM J. Comput.*, 31(2):601–625, 2001.

[GT01b] Jonathan L. Gross and Thomas W. Tucker. *Topological Graph Theory*. Dover Books on Mathematics. Dover Publications, reprint edition, 2001.

[Han06] Kristoffer A. Hansen. Constant width planar computation characterizes ACC^0. *Theor. Comput. Sci.*, 39(1):79–92, 2006.

[Has01] S. Mehdi Hashemi. Digraph embedding. *Discrete Math.*, 233(1–3):321–328, 2001.

[Haz87] M. Hazewinkel. *Cylinder Coordinates*. Encyclopaedia of Mathematics: An Updated and Annotated Translation of the Soviet 'Mathematical Encyclopaedia'. Springer Verlag, Heidelberg, Germany, 1987. D.D. Sokolov (originator), http://www.encyclopediaofmath.org/index.php?title=Cylinder_coordinates&oldid=16846.

[Hea84] Lenwood Heath. Embedding planar graphs in seven pages. In *Proc. 25th Annual Symposium on Foundations of Computer Science, SFCS 1984*, pages 74–83, Washington, DC, USA, 1984. IEEE Computer Society.

[Hel07] Guido Helden. *Hamiltonicity of Maximal Planar Graphs and Planar Triangulations*. Dissertation, Rheinisch-Westfälische Technische Hochschule Aachen, 2007. http://darwin.bth.rwth-aachen.de/opus3/volltexte/2007/1949/.

[HI87] Lenwood Heath and S. Istrail. The page number of genus g graphs is $\mathcal{O}(g)$. In Alfred V. Aho, editor, *Proc. 19th Annual ACM Symposium on Theory of Computing, STOC 1987*, pages 388–397, New York, NY, USA, 1987. Association for Computing Machinery (ACM).

[HL96] M. Hutton and A. Lubiw. Upward planar drawing of single-source acyclic digraphs. *SIAM J. Comput.*, 25(2):291–311, 1996.

[HL06] Patrick Healy and Karol Lynch. Two fixed-parameter tractable algorithms for testing upward planarity. *Int. J. Found. Comput. Sci.*, 17(05):1095–1114, 2006.

[HLP02] Nicholas G. Hall, Tae-Eog Lee, and Marc E. Posner. The complexity of cyclic shop scheduling problems. *Journal of Scheduling*, 5(4):307–327, 2002.

[HLR92] Lenwood S. Heath, Frank Thomson Leighton, and Arnold L. Rosenberg. Comparing queues and stacks as mechanisms for laying out graphs. *SIAM J. Discret. Math.*, 5(3):398–412, 1992.

[HMU03] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2nd edition, 2003.

[HN09] Seok-Hee Hong and H. Nagamochi. Two-page book embedding and clustered graph planarity. Technical Report TR [2009-004], Department of Applied Mathematics and Physics, Graduate School of Informatics, University of Kyoto, Japan, 2009. http://www-or.amp.i.kyoto-u.ac.jp/members/nag/Technical_report/TR2009-004.pdf.

[HPT99a] Lenwood Scott Heath, Sriram V. Pemmaraju, and Ann N. Trenk. Stack and queue layouts of directed acyclic graphs: Part I. *SIAM J. Comput.*, 28(4):1510–1539, 1999.

[HPT99b] Lenwood Scott Heath, Sriram V. Pemmaraju, and Ann N. Trenk. Stack and queue layouts of directed acyclic graphs: Part II. *SIAM J. Comput.*, 28(5):1588–1626, 1999.

[HR92] Lenwood Scott Heath and Arnold L. Rosenberg. Laying out graphs using queues. *SIAM J. Comput.*, 21(5):927–958, 1992.

[HRK98] S. Medi Hashemi, Ivan Rival, and Andrzej Kisielewicz. The complexity of upward drawings on spheres. *Order*, 14:327–363, 1998.

[HS71] Frank Harary and Allen J. Schwenk. Trees with hamiltonian square. *Mathematika*, 18:138–140, June 1971.

[HS73] Frank Harary and Allen J. Schwenk. The number of caterpillars. *Discrete Math.*, 6(4):359–365, 1973.

[HT73] John Hopcroft and Robert Endre Tarjan. Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[Hun16] Edward V. Huntington. A set of independent postulates for cyclic order. In *Proc. National National Academy of Sciences of the United States of America*, volume 2, pages 630–631. National Academy of Sciences, November 1916.

[JL02] Michael Jünger and Sebastian Leipert. Level planar embedding in linear time. *J. Graph Alg. App.*, 6(1):67–113, 2002.

[Jor87] Camille Jordan. *Cours d'Analyze de l'École Polytechnique*. 1887.

[Kai74] Paul Chester Kainen. Some recent results in topological graph theory. In Ruth A. Bari and Frank Harary, editors, *Graphs and Combinatorics*, volume 406 of *Lecture Notes in Mathematics*, pages 76–108. Springer Verlag, Heidelberg, Germany, 1974.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Kel87] David Kelly. Fundamentals of planar ordered sets. *Discrete Math.*, 63:197–216, 1987.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, MA, USA, 3rd edition, 1997.

[Kos79] S. Rao Kosaraju. Real-time simulation of concatenable double-ended queues by double-ended queues (preliminary version). In *Proc. 11th Annual ACM Symposium on Theory of Computing, STOC 1979*, pages 346–351, New York, NY, USA, 1979. Association for Computing Machinery (ACM).

[Kre12] Stephan Kreutzer. On the parameterized intractability of monadic second-order logic. *Logical Methods in Computer Science*, 8(1), 2012.

[KW01] Michael Kaufmann and Dorothea Wagner. *Drawing Graphs*, volume 2025 of *LNCS*. Springer Verlag, Heidelberg, Germany, 2001.

[LEC67] A. Lempel, Shimon Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In Pierre Rosenstiehl, editor, *Theory of Graphs, International Symposium, Rome*, pages 215–232. Gordon and Breach, 1967.

[Li88] Ming Li. Simulating two pushdown stores by one tape in $\mathcal{O}(n^{1.5}\sqrt{\log n})$ time. *J. Comput. Syst. Sci.*, 37(1):101–116, 1988.

[LMS06] Nutan Limaye, Meena Mahajan, and Jayalal M. N. Sarma. Evaluating monotone circuits on cylinders, planes and tori. In Bruno Durand and Wolfgang Thomas, editors, *STACS 2006*, volume 3884 of *LNCS*, pages 660–671, Heidelberg, Germany, 2006. Springer Verlag.

[LMS09] Nutan Limaye, Meena Mahajan, and Jayalal M. N. Sarma. Upper bounds for monotone planar circuit value and variants. *Comput. Complex.*, 18(3):377–412, 2009.

[LY78] Claudio Leonardo Lucchesi and Daniel Haven Younger. A minimax theorem for directed graphs. *J. London Math. Soc.*, s2-17(3):369–374, 1978.

[Mal94a] Seth M. Malitz. Genus g graphs have pagenumber $\mathcal{O}(\sqrt{g})$. *J. Alg.*, 17(1):85–109, July 1994.

[Mal94b] Seth M. Malitz. Graphs with E edges have pagenumber $\mathcal{O}(\sqrt{E})$. *J. Alg.*, 17(1):81–84, July 1994.

[Mas67] William S. Massey. *Algebraic Topology: An Introduction*. Springer Verlag, Heidelberg, Germany, 1967.

[Mic93] Gerhard Michal. *Biochemical Pathways (Poster)*. Boehringer Mannheim, Penzberg, 1993.

[Miy06] Miki Miyauchi. Topological book embedding of bipartite graphs. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E89-A(5):1223–1226, May 2006.

[MRA01] Jerrold E. Marsen, Tudor Ratiu, and Ralph Abraham. *Manifolds, Tensor Analysis, and Applications*. Springer Verlag, Heidelberg, Germany, 3rd edition, 2001.

[MT01] Bojan Mohar and Carsten Thomassen. *Graphs on Surfaces*. Johns Hopkins Series in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, USA, 2001.

[Nie06] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, Heidelberg, Germany, 2006.

[Oll73] L.T̃. Ollmann. On the book thicknesses of various graphs. In *Proc. 4th Southeastern Conference on Combinatorics, Graph Theory and Computing*, volume 8 of *Congressus Numerantium*, page 459, 1973.

[Pap95] Achilleas Papakostas. Upward planarity testing of outerplanar dags (extended abstract). In Roberto Tamassia and Ioannis G. Tollis, editors, *Proc. DIMACS International Workshop, GD 1994*, volume 894 of *LNCS*, pages 298–306, Heidelberg, Germany, 1995. Springer Verlag.

[Pem92] Sriram Venkata Pemmarju. *Exploring the Powers of Stacks and Queues via Graph Layouts*. PhD thesis, Virginia Polytechnic Institute & State University, Blacksburg, VA, USA, 1992.

[Pet01] Holger Petersen. Stacks versus deques. In Jie Wang, editor, *Proc. 7th Annual International Conference on Computing and Combinatorics, COCOON 2001*, volume 2108 of *LNCS*, pages 218–227, Heidelberg, Germany, 2001. Springer Verlag.

[Pra73] Vaughan R. Pratt. Computing permutations with double-ended queues, parallel stacks and parallel queues. In *Proc. of the 5th Annual ACM Symposium on Theory of Computing, STOC 1973*, pages 268–277, New York, NY, USA, 1973. Association for Computing Machinery (ACM).

[Pur97] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In Giuseppe Di Battista, editor, *Proc. 5th International Symposium on Graph Drawing, GD 1997*, volume 1353 of *LNCS*, pages 248–261, Heidelberg, Germany, 1997. Springer Verlag.

[Ros76] Pierre Rosenstiehl. Solution algébrique du problème de Gauss sur la permutation des points d'intersection d'une ou plusieurs courbes fermées du plan. *C.R. Académie des Sciences de Paris*, 283:551–553, October 1976.

[Ros83] A. L. Rosenberg. The diogenes approach to testable fault-tolerant arrays of processors. *IEEE Trans. Comput.*, 32(10):902–910, October 1983.

[Ros93] Burton Rosenberg. Simulating a stack by queues. In *Proc. XIX Latinamerican Conference on Computer Science*, volume 1, pages 3–13, 1993.

[RS84] Neil Robertson and P.D̃. Seymour. Graph minors. III: Planar tree-width. *J. Combin. Theory, Ser. B*, 36(1):49–64, 1984.

[RS04] Neil Robertson and P.D̃. Seymour. Graph minors. XX. Wagner's conjecture. *J. Combin. Theory, Ser. B*, 92(2):325–357, 2004.

[RT84] Pierre Rosenstiehl and Robert Endre Tarjan. Gauss codes, planar hamiltonian graphs, and stack-sortable permutations. *J. Algorithms*, 5:375–390, September 1984.

[SH99] Wei-Kuan Shih and Wen-Lian Hsu. A new planarity test. *Theor. Comput. Sci.*, 223(1-2):179–191, 1999.

[SP83] Maciej M. Sysło and Andrzej Proskurowski. On halin graphs. In M. Borowiecki, John W. Kennedy, and Maciej M. Sysło, editors, *Graph Theory*, volume 1018 of *Lecture Notes in Mathematics*, pages 248–256, Heidelberg, Germany, 1983. Springer Verlag.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Syst., Man, Cybern.*, 11(2):109–125, 1981.

[SU89] Paolo Serafini and Walter Ukovich. A mathematical model for periodic scheduling problems. *SIAM J. Discret. Math.*, 2(4):550–581, 1989.

[Tar72a] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[Tar72b] Robert Endre Tarjan. Sorting using networks of queues and stacks. *J. ACM*, 19(2):341–346, April 1972.

[Tho89] Carsten Thomassen. Planar acyclic oriented graphs. *Order*, 5(1):349–361, 1989.

[Tuf01] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001.

[Tut63] William Thomas Tutte. How to draw a graph. *Proc. Lond. Math. Soc.*, 13:743–767, 1963.

[Ung88] Walter Unger. On the k-colouring of circle-graphs. In Robert Cori and Martin Wirsing, editors, *Proc. 5th Annual Symposium on Theoretical Aspects of Computer Science, STACS 1988*, volume 294 of *LNCS*, pages 61–72, Heidelberg, Germany, 1988. Springer Verlag.

[Ung92] Walter Unger. The complexity of colouring circle graphs (extended abstract). In *Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science, STACS 1992*, volume 577 of *LNCS*, pages 389–400, Heidelberg, Germany, February 1992. Springer Verlag.

[Vol70] R. Vollmar. Über einen Automaten mit Pufferspeicherung. *Computing*, 5(1):57–70, 1970.

[Wag37] Klaus Wagner. Über eine Eigenschaft der ebenen Komplexe. *Math. Annalen*, 114(1):570–590, 1937.

[Weg05] Ingo Wegener. *Complexity Theory — Exploring the Limits of Efficient Algorithms*. Springer, 2005.

[Whi33] Hassler Whitney. Non-separable and planar graphs. *Trans. Amer. Math. Soc.*, 21:73–84, 1933.

[Wie87] Manfred Wiegers. Recognizing outerplanar graphs in linear time. In *Proc. 12th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 1986*, volume 246 of *LNCS*, pages 165–176. Springer, 1987.

[Wig82] Avi Wigderson. The complexity of the Hamiltonian circuit problem for maximal planar graphs. Technical report, Department of EECS, Princeton University, 1982.

[Woo02] David R. Wood. Queue layouts, tree-width, and three-dimensional graph drawing. In Manindra Agrawal and Anil Seth, editors, *Proc. Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2002*, volume 2556 of *LNCS*, pages 348–359, Heidelberg, Germany, December 2002. Springer Verlag.

[Yan86] Mihalis Yannakakis. Four pages are necessary and sufficient for planar graphs. In *Proc. 18th Annual ACM Symposium on Theory of Computing, STOC 1986*, pages 104–108, New York, NY, USA, 1986. Association for Computing Machinery (ACM).

[Yan89] Mihalis Yannakakis. Embedding planar graphs in four pages. *J. Comput. Syst. Sci.*, 38(1):36–67, 1989.

Index

acyclic digraph, 5
acyclic dipole (directed SPQR tree), 171
acyclic dipole (upward planarity), 101
algorithms
  ComputeAlignments, 207
  ComputeCompoundDigraph, 133
  ComputeCompoundEmbedding, 205
  ComputeDBCT&fRUPEmbeddings, 157
  ComputeLREdges, 201
  ComputePre-dSPQRTree, 199
  DFS, 67
  EmbedSkeleton, 214
  FindCompatibleAlignments, 205
  FindBlock, 155
  IsCyclicDequeLayout, 237
  IsDequeLayout, 32
  IsSDLayout, 72
  ProcessVertex, 32
  TestBiconnected, 195
  TestClosedDigraph, 136
  TestCompound, 154
  Split, 73
  TUPTestClosedDigraph, 228
Ampère's Law for Duals, 114
ancestor (depth-first search tree), 66
arched leveled-planar (queue graph), 54
arches (queue layout), 19, 54
$\prec$-augmentation of a deque schedule (deque layout), 41
$\prec$-augmentation of a graph (deque layout), 41
augmented queue embedding (queue layout), 58
(auxiliary) L-cycle (directed SPQR tree), 182
(auxiliary) R-cycle (directed SPQR tree), 182
auxiliary sink (directed SPQR tree), 171
auxiliary skeleton (acyclic digraph) (directed SPQR tree), 171
auxiliary skeleton (compound) (directed SPQR tree), 182
auxiliary source (directed SPQR tree), 171
azimuth (cylinder), 11

biconnected undirected graph, 7
bimodal rotation system (upward planarity), 105
block (undirected graph), 7
block chain, 145
block sequence, 139
block series, 139
block-cut tree (undirected graph), 7
book embedding, 17
book thickness, 18

Catalan numbers, 24
caterpillar, 148
  spine, 149
circle
  $\rightarrow$ *definition*, 5
  Eulerian, 5
  Hamiltonian, 5
  simple, 5
closed digraph, 109
closed Gauß code (Gauß code), 91
compatible alignments (directed SPQR tree), 203
complete undirected graph, 5
component, *see* strongly connected component
component digraph (digraph), 7
compound (digraph), 107
compound digraph (digraph), 107

ComputeAlignments (algorithm), 207
ComputeCompoundDigraph (algorithm), 133
ComputeCompoundEmbedding (algorithm), 205
ComputeDBCT&fRUPEmbeddings (algorithm), 157
ComputeLREdges (algorithm), 201
ComputePre-dSPQRTree (algorithm), 199
concatenation (splittable deque layout), 67
connected component undirected graph, 7
connected undirected graph, 7
content (deque), 30
context-free language, 25
cut (dual graph), 10
cut face (planar graph), 137
cut vertex (undirected graph), 7
cut-set (dual graph), 10
cycle
  edge-simple, 218
  Eulerian, 5
  Hamiltonian, 5
  simple, 5
cycle (digraph), 5
cyclic deque digraph (deque layout), 237
cyclic deque layout (deque layout), 237
cyclic deque schedule (deque layout), 237
cyclic drawings (upward planarity), 103
cyclic edge (directed SPQR tree), 177
cyclic order, 8
cyclic-L edge (directed SPQR tree), 177
cyclic-R edge (directed SPQR tree), 177
cyclomatic number, 91
cylinder
  azimuth, 11
  cylindrical coordinates, 11
  drawing of a graph, 12
  fundamental polygon, 12
  plane drawing of a graph, 12
  $\mathbb{R}^2$-representation, 12
  rolling, 11
  standing, 11
cylindrical coordinates (cylinder), 11

d^-, *see* indegree
d^+, *see* outdegree
DCEL, 10
De Morgan's Law for One-Clique Sums, 140
De Morgan's Law for Two-Clique Sums, 166
degree of a vertex, *see* undirected graph
depth-first search tree, 65
  ancestor, 66
  descendant, 66
  DFS number, 66
  forward edge, 66
  leaf, 66
  linear layout induced by DFS tree, 66
  root-to-leaf dipath, 66
  subtree of a vertex, 66
  tree edge, 65
deque, 15, 30
  content, 30
  head, 15, 30
  tail, 15, 30
deque digraph (deque layout), 235
deque graph (deque layout), 32
deque layout
  $\prec$-augmentation of a deque schedule, 41
  $\prec$-augmentation of a graph, 41
  cyclic deque digraph, 237
  cyclic deque layout, 237
  cyclic deque schedule, 237
  $\to$ *definition*, 32
  deque digraph, 235
  deque graph, 32
  deque schedule, 31
  deque schedule induced by Hamiltonian path, 43
  deque-reducible, 52
  directed deque layout, 235
  exuberant $\prec$-augmentation of a graph, 57
  input-restricted deque, 52
  linear cylindric planar digraph, 235
  output-restricted deque, 52
  queue edge, 31
  stack edge, 31
deque schedule (deque layout), 31
deque schedule induced by Hamiltonian path (deque layout), 43
deque-reducible (deque layout), 52
Derail, 84
descendant (depth-first search tree), 66

de Fraysseix-Rosenstiehl planarity criterion, 82
DFS (algorithm), 67
DFS number (depth-first search tree), 66
DFS traversal, 65
DFS tree, *see* depth-first search tree
dicut (digraph), 110
dicut-set (digraph), 110
digraph
  acyclic, 5
  biconnected, *see* undirected graph
  block-cut tree, *see* undirected graph
  closed, 109
  component digraph, 7
  compound, 107
  compound digraph, 107
  connected, *see* undirected graph
  cycle, 5
  → *definition*, 5
  dicut, 110
  dicut-set, 110
  digraph of strongly connected components, 7
  dipole, 108
  dual graph, 10
  forest, *see* undirected graph
  indegree, 5
  isomorphic digraphs, 6
  outdegree, 5
  simple, 5
  sink, 5
  source, 5
  strongly connected, 7
  strongly connected component, 6
  terminal, 107
  transit, 108
  tree, *see* undirected graph
  triconnected, *see* undirected graph
  trivial component, 107
  trivial strongly connected component, 7
  underlying undirected graph, *see* undirected graph
digraph of strongly connected components (digraph), 7
dipath
  → *definition*, 5
  Eulerian, 5
  Hamiltonian, 5
  length of, 5
  simple, 5
dipole (digraph), 108
directed block-cut tree, 149
directed deque layout (deque layout), 235
directed dual graph, *see* planar graph
directed graph, *see* digraph
directed loop, 5
directed skeleton (directed SPQR tree), 169
directed SPQR tree
  acyclic dipole, 171
  (auxiliary) L-cycle, 182
  (auxiliary) R-cycle, 182
  auxiliary sink, 171
  auxiliary skeleton (acyclic digraph), 171
  auxiliary skeleton (compound), 182
  auxiliary source, 171
  compatible alignments, 203
  cyclic edge, 177
  cyclic-L edge, 177
  cyclic-R edge, 177
  directed skeleton, 169
  dual dSPQR tree, 181
  expansion digraph, 168
  feasible alignment, 202
  feasible embedding of an auxiliary skeleton, 183
  LR-feasible **RUP** embedding of a skeleton, 193
  LR-feasible **RUP** embedding of an auxiliary skeleton, 193
  LR-feasible alignment, 202
  of a compound, 177
  of acyclic digraph, 168
  pre-dSPQR tree, 194, 195
  RUP(-embedded) node, 182
  RUP(-embedded) skeleton, 182
  RUP-embedded node, 182
  sink edge, 169
  source edge, 169
  terminal edge, 169
dSPQR tree, 159
doubly-connected edge list, 10
drawing of a graph
  → *definition*, 7
  edge curve, 7

inner part of an edge curve, 7
Jordan arc, 7
plane, 7
upward, 97
vertex position, 7
drawing of a graph (cylinder), 12
dual dSPQR tree (directed SPQR tree), 181
dual edge (planar graph), 10
dual graph, *see* digraph
cut, 10
cut-set, 10
dual graph (planar graph), 10
dual layout
dual queue layout, 64
dual node (dual SPQR tree), 164
dual queue layout (dual layout), 64
dual skeleton (dual SPQR tree), 164
dual SPQR tree
$\rightarrow$ *definition*, 164
dual node, 164
dual skeleton, 164

edge curve (drawing of a graph), 7
edge-simple cycle, 218
embedding (planar graph), 9
embedding constraint, 203
`EmbedSkeleton` (algorithm), 214
Euclidean plane, 7
Eulerian circle, 5
Eulerian cycle, 5
Eulerian dipath, 5
Eulerian path, 5
expansion digraph (directed SPQR tree), 168
expansion graph (SPQR tree), 160
exuberant $\prec$-augmentation of a graph (deque layout), 57

face (planar graph), 9
FAS (feedback arc set), 196
feasible **RUP** embedding (of a block), 147
feasible alignment (directed SPQR tree), 202
feasible embedding of an auxiliary skeleton (directed SPQR tree), 183
feedback arc set, 196
FAS, 196
feedback arc set problem, 196
minimum, 196
feedback arc set problem (feedback arc set), 196
field respecting edge curve (upward planarity), 99
FIFO, *see* first-in-first-out
`FindCompatibleAlignments` (algorithm), 205
`FindBlock` (algorithm), 155
finite automata, 25
first in, first out, 15
fixed-parameter tractability, 89
fixed-parameter tractable, 231
flip (SPQR tree), 163
fork (LR planarity criterion), 83
forward edge (depth-first search tree), 66
FPT, *see* fixed-parameter tractability
front circle (linear toric), 238
front line (linear cylindric), 27
fundamental circle (LR planarity criterion), 83
fundamental polygon (cylinder), 12

Γ, *see* drawing of a graph
Gauß code
closed Gauß code, 91
open Gauß code, 92
pile of twin-stacks, 92
sequence, 91
tangent point, 91
graph
directed, *see* digraph
drawing of a, *see* drawing of a graph
planar, *see* planar graph
undirected, *see* undirected graph

Hamiltonian circle, 5
Hamiltonian cycle, 5
Hamiltonian dipath, 5
Hamiltonian path, 5
head (deque), 15, 30
hierarchy, *see* acyclic digraph

indegree (digraph), 5
independent transits, 125
induced subgraph, 6
inner part of an edge curve (drawing of a graph), 7

input-restricted deque (deque layout), 52
IsCyclicDequeLayout (algorithm), 237
IsDequeLayout (algorithm), 32
isomorphic digraphs (digraph), 6
isomorphic graphs (undirected graph), 6
isomorphic multigraphs (multigraph), 6
IsSDLayout (algorithm), 72

Jordan arc (drawing of a graph), 7
Jordan's curve theorem, 8

last in, first out, 15
LC, *see* linear cylindric
LC drawing, *see* linear cylindric drawing
  plane, 27
LC embedding, *see* linear cylindric embedding
LC planar, *see* linear cylindric planar
LC rotation system (linear cylindric embedding), 29
LC rotation system induced by a Hamiltonian path (linear cylindric), 43
leaf (depth-first search tree), 66
leaf block, 149
left-right partition (LR planarity criterion), 83
leftmost cycle (planar graph), 109
leftmost vertex/edge (planar graph), 109
length of dipath, 5
LIFO, *see* last in, first out
linear **SUP** (linear cylindric), 236
linear **TUP** (linear toric), 238
linear **UP** (linear cylindric), 236
linear cylindric
  → *definition*, 27
  front line, 27
  LC rotation system induced by a Hamiltonian path, 43
  linear **SUP**, 236
  linear **UP**, 236
  plane LC drawing, 27
linear cylindric embedding, 30
  LC rotation system, 29
  linear cylindric rotation system, 29
  planar LC rotation system, 29
linear cylindric planar, 27
linear cylindric planar digraph (deque layout), 235
linear cylindric rotation system (linear cylindric embedding), 29
linear drawing (queue layout), 54
linear layout, 16, 26
linear layout induced by DFS tree (depth-first search tree), 66
linear toric
  front circle, 238
  linear **TUP**, 238
  linear toric drawing, 238
  linear toroidal digraph, 238
  linear toroidal drawing, 238
linear toric drawing (linear toric), 238
linear toroidal digraph (linear toric), 238
linear toroidal drawing (linear toric), 238
loop
  directed, 5
  undirected, 5
LR partition (LR planarity criterion), 83
LR planarity criterion, 82
  fork, 83
  fundamental circle, 83
  left-right partition, 83
  LR partition, 83
  return edges, 83
LR planarity criterion (planar graph), 83
LR-feasible **RUP** embedding of a skeleton (directed SPQR tree), 193
LR-feasible **RUP** embedding of an auxiliary skeleton (directed SPQR tree), 193
LR-feasible alignment (directed SPQR tree), 202

MD, *see* mergeable deque
mergeable deque (splittable deque layout), 84
minimum feedback arc set, 196
mixed layout, 53
monadic second-order logic, 90
MSOL, 90
multigraph
  → *definition*, 5
  isomorphic multigraphs, 6

node (SPQR tree), 159

node sequence (SPQR tree), 162
node series (SPQR tree), 162

of a compound (directed SPQR tree), 177
of acyclic digraph (directed SPQR tree), 168
one-clique sum, 139
open Gauß code (Gauß code), 92
outdegree (digraph), 5
outer face (planar graph), 9
outerplanar (planar graph), 17
output-restricted deque, 25
output-restricted deque (deque layout), 52

page number, 18
partial rotation system, 127
path
- → *definition*, 5
- Eulerian, 5
- Hamiltonian, 5
- simple, 5

PDA, 25
permutation network, 22
pile of twin-stacks (Gauß code), 92
planar
- linear cylindric, 27

planar graph
- cut face, 137
- → *definition*, 8
- directed dual graph, 10
- dual edge, 10
- dual graph, 10
- embedding, 9
- face, 9
- leftmost cycle, 109
- leftmost vertex/edge, 109
- LR planarity criterion, 83
- outer face, 9
- outerplanar, 17
- primal edge, 10
- primal graph, 10
- rightmost cycle, 109
- rightmost vertex/edge, 109

planar LC rotation system (linear cylindric embedding), 29
planar rotation system, 9
plane drawing of a graph, 7
plane drawing of a graph (cylinder), 12
plane LC drawing, 27
plane LC drawing (linear cylindric), 27
Post machine, 25
pre-dSPQR tree (directed SPQR tree), 194, 195
primal edge (planar graph), 10
primal graph (planar graph), 10
`ProcessVertex` (algorithm), 32
proper leveled-planar graphs, 55
push-down automaton, 25
P node (SPQR tree), 160

quasi-upward plane drawing (upward planarity), 103
queue, 15
queue edge (deque layout), 31
queue graph
- arched leveled-planar, 54
- → *definition*, 53

queue layout
- arches, 19, 54
- augmented queue embedding, 58
- linear drawing, 54
- queue layout, 53
- queue schedule, 53
- twist, 19, 55

queue layout (queue layout), 53
queue schedule (queue layout), 53
Q node (SPQR tree), 160

$\mathbb{R}^2$-representation (cylinder), 12
$\mathcal{R}$, *see* rotation system
radial drawing (upward planarity), 102
real-time deque automaton, 25
recurrent hierarchies (upward planarity), 103
recursively enumerable language, 25
regular language, 25
return edges (LR planarity criterion), 83
rightmost cycle (planar graph), 109
rightmost vertex/edge (planar graph), 109
rolling cylinder, 11
rolling upward planar (upward planarity), 102
root-to-leaf dipath (depth-first search tree), 66
rotation system
- → *definition*, 8
- planar, 9

RUP (upward planarity), 102
RUP(-embedded) node (directed SPQR tree), 182
RUP(-embedded) skeleton (directed SPQR tree), 182
RUP-embedded node (directed SPQR tree), 182
`TestBiconnected` (algorithm), 195
`TestClosedDigraph` (algorithm), 136
`TestCompound` (algorithm), 154
R node (SPQR tree), 160

SD, *see* splittabe deque
SD graph, *see* splittable deque graph
sequence (Gauß code), 91
simple circle, 5
simple cycle, 5
simple digraph, 5
simple dipath, 5
simple path, 5
simple undirected graph, 5
sink (digraph), 5
sink edge (directed SPQR tree), 169
skeleton (SPQR tree), 159
source (digraph), 5
source edge (directed SPQR tree), 169
spanning subgraph, 6
spanning supergraph, 6
spherical digraph (upward planarity), 101
spine (caterpillar), 149
spine block, 149
`Split` (algorithm), 73
split operation (splittable deque layout), 67
split pair (undirected graph), 7
splittable deque (splittable deque layout), 67
splittable deque graph (splittable deque layout), 69
splittable deque layout
  concatenation, 67
  → *definition*, 70
  mergeable deque, 84
  split operation, 67
  splittable deque, 67
  splittable deque graph, 69
  splittable deque schedule, 68
  tree layout, 68
splittable deque schedule (splittable deque layout), 68
SPQR tree, 159
  expansion graph, 160
  flip, 163
  node, 159
  node sequence, 162
  node series, 162
  P node, 160
  Q node, 160
  R node, 160
  skeleton, 159
  swap, 163
  S node, 160
  virtual edge, 160
st-digraphs (upward planarity), 100
stack, 15
stack edge (deque layout), 31
stack number, 18
stack subdivision of a graph, 18
standing cylinder, 11
standing upward planar digraph (upward planarity), 98
strict upward planarity (upward planarity), 103
strongly connected (digraph), 7
strongly connected component (digraph), 6
subgraph
  → *definition*, 6
  induced, 6
  spanning, 6
subtree of a vertex (depth-first search tree), 66
SUP (upward planarity), 98
supergraph
  → *definition*, 6
  spanning, 6
swap (SPQR tree), 163
S node (SPQR tree), 160

tail (deque), 15, 30
tangent point (Gauß code), 91
terminal (digraph), 107
terminal edge (directed SPQR tree), 169
topological book embedding, 18
train switching problem, 84
transit (digraph), 108

tree (undirected graph), 7
tree decomposition, 90
 treewidth, 90
 width of a tree decomposition, 90
tree edge (depth-first search tree), 65
tree layout (splittable deque layout), 68
treewidth, 19
treewidth (tree decomposition), 90
triconnected undirected graph, 7
trivial component (digraph), 107
trivial strongly connected component (digraph), 7
TUP (upward planarity), 227
`TUPTestClosedDigraph` (algorithm), 228
Turing machine, 25
twist (queue layout), 19, 55
two-clique summation, 162

unconnected undirected graph, 7
underlying undirected graph, 5
undirected graph
 biconnected, 7
 block, 7
 block-cut tree, 7
 complete, 5
 connected, 7
 connected component, 7
 cut vertex, 7
 → *definition*, 5
 isomorphic graphs, 6
 simple, 5
 split pair, 7
 tree, 7
 triconnected, 7
 unconnected, 7
 underlying, 5
undirected loop, 5

UP (upward planarity), 97
upward drawing of a graph, 97
upward planar digraph (upward planarity), 97
upward planarity
 acyclic dipole, 101
 bimodal rotation system, 105
 cyclic drawings, 103
 field respecting edge curve, 99
 quasi-upward plane drawing, 103
 radial drawing, 102
 recurrent hierarchies, 103, 229
 rolling upward planar, 102
 RUP, 102
 spherical digraph, 101
 st-digraphs, 100
 standing upward planar digraph, 98
 strict upward planarity, 103
 SUP, 98
 TUP, 227
 UP, 97
 upward planar digraph, 97
 upward toroidal digraph, 227
 weakly field respecting edge curve, 99
 wSUP, 101, 218
upward toroidal digraph (upward planarity), 227
uv-digraph, 168

vertex position (drawing of a graph), 7
virtual edge (SPQR tree), 160
VLSI, 18

weakly field respecting edge curve (upward planarity), 99
width of a tree decomposition (tree decomposition), 90
wSUP (upward planarity), 101