Karlsruhe Institute of Technology

Bend Minimization in Planar Orthogonal Drawings

– Theory, Implementation and Evaluation –

Bachelor Thesis of

Sebastian Lehmann

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Thomas Bläsius
	Dr. Ignaz Rutter

Time Period: 1st September 2012 – 11th November 2012

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 11th November 2012

Abstract

Minimizing the number of bends in orthogonal drawings of planar graphs is one of the major steps for improving the readability of the drawing. While the general problem is $\mathcal{NP}$-hard if the embedding is not fixed, there exist multiple restrictions to the problem which make it solvable in polynomial time and thus relevant for actual use.

This thesis focuses on two such approaches.

1. The decision problem FlexDraw with positive flexibility: Given a graph $G = (V, E)$ and a function $\text{flex} : E \to N\backslash\{0\}$. Can G be drawn with at most $\text{flex}(e)$ bends per edge?
2. The bend minimization problem OptimalFlexDraw for series-parallel graphs: Given a series-parallel graph $G = (V, E)$ and for every edge $e \in E$ a cost function $\text{cost}_e : \mathbb{N}_0 \to \mathbb{R}$. Find a drawing minimizing $\sum_{e \in E} \text{cost}_e(\text{bends}(e))$.

In this thesis we implement and evaluate an algorithm solving the decision problem FlexDraw with positive flexibility. In the experimental evaluation we will discuss several questions, the most interesting of them being: How much bends do we need to allow in average, so that there exists a valid drawing? Since FlexDraw is a decision problem, its major disadvantage for practical use is that it will not generate any drawing if the instance does not admit one respecting the flexibility constraints given. One idea is to allow a flexibility of two on every edge, resulting in good but not optimal drawings. Thus, a demand for algorithms solving the *optimization problem* arise.

We present a polynomial-time algorithm for OptimalFlexDraw for *series-parallel graphs* minimizing the bends, giving us more control over the output.

Deutsche Zusammenfassung

Eine wichtige Eigenschaft übersichtlicher orthogonaler Zeichnungen von Graphen besteht in einer möglichst geringen Anzahl Knicke auf den Kanten. Das allgemeine Problem der Knickminimierung sowie die Entscheidung, ob sich ein Graph knickfrei orthogonal zeichnen lässt, ist NP-schwer. Es gibt jedoch Ansätze, das Problem mit leichten Einschränkungen in Polynomialzeit zu lösen, welche für einen praktischen Einsatz relevant sind.

Das Problem FlexDraw behandelt die Entscheidung, ob sich ein Graph mit einer pro Kante individuellen Maximalzahl an Knicken, welche echt positiv sein muss, zeichnen lässt. Das Optimierungsproblem OptimalFlexDraw minimiert in serienparallelen Graphen die Kosten verursacht durch Knicke, für welche jeweils eine individuelle Kostenfunktion angegeben wird.

Wir stellen in dieser Arbeit einen Algorithmus vor, welcher das Problem OptimalFlexDraw für serienparallele Graphen in Polynomialzeit löst. Für das Entscheidungsproblem FlexDraw für allgemeine Graphen und echt positiver Flexibilität wurde im Rahmen dieser Arbeit eine Implementierung angefertigt, deren Algorithmus zunächst skizziert wird und welche wir im Anschluss vorstellen und evaluieren.

Contents

1. Introduction

In the field of computer science, we often deal with data which we want to visualize. Depending on the nature of information this may be a *graph* with *nodes* interconnected by *edges*. *Graph visualization* is a field in computer science which is about *generating drawings* of graphs automatically. This field can be divided further by generating a lot of different *types of drawings* depending on the class of graphs or what we want to express with the drawings. In this work we focus on *planar graphs*, the class of graphs that can be drawn without any edge crossings. We further restrict the class of graphs to have a maximum degree of 4, meaning every node is connected to at most 4 nodes. These graphs are also called *4-planar graphs*.

Visualizing graphs is a complex task since in almost all use cases the drawings will be read by human beings and thus should be easily readable. Furthermore, in most scenarios we do not want to spend a lot of time generating the drawing, especially in interactive applications, so the running time matters, too. Given a graph to be visualized, we have to take three major steps in order to generate a drawing automatically.

First, we have to think about the *type of drawing* we want to generate. In this thesis we want to draw 4-planar graphs on a *grid* where every node is placed on a grid crossing and the edges are drawn on the grid lines as horizontal and vertical line segments without crossing or touching each other. Such a drawing is commonly called an *orthogonal grid drawing*; an example is given in Figure 1.1.

Secondly, we have to define *measurable properties* of these drawings in order to measure the overall *readability*. These might be the number of edge crossings, the number of bends

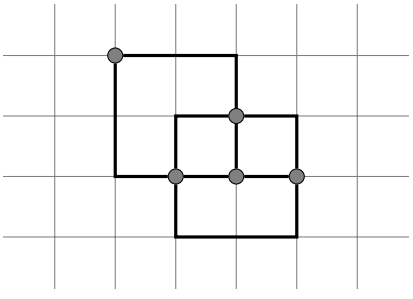


Figure 1.1.: An orthogonal grid drawing generated by our implementation of FlexDraw

on edges, the total length of the edges, the size of the drawing, etc. In this work we focus on *minimizing* the number of bends on the edges in a grid drawing.

Finally, we *find an algorithm* that optimizes the criteria we defined. A common problem of *optimization algorithms* is the enormous numbers of possible solutions from which we want to pick the best one; often the set is too large to be handled efficiently or even infinitely large. There are some well-known methods for efficient optimization, one of them being *dynamic programming* applicable in algorithms where we can construct the optimal solution of the problem instance by solving and combining smaller pieces of the instance. The algorithms introduced and implemented in this work are based on dynamic programming.

One very important property of planar drawings is their *combinatorial embedding*, which defines in which order the incident edges of a node are laid out. While some graphs have a fixed combinatorial embedding by nature (e.g., street networks, where the streets leading into a crossing have a fixed order), for most of them we can choose an arbitrary embedding, which we use to draw the graph. In the case of bend minimization in orthogonal drawings, the embedding is an essential "variable" we also want to optimize since some embedding may admit drawings with substantially fewer bends than other. However, in general the number of possible embeddings grows exponentially with the size of the graph. To deal with this issue, we make use of the SPQR-tree, a data structure enabling us to optimize the result over all possible embeddings efficiently using dynamic programming.

This thesis is divided into the following parts. We start with phrasing the problem in a couple of interesting variants and introduce some preliminaries we will need later. We introduce an algorithm solving the optimization problem OptimalFlexDraw for series-parallel graphs in polynomial time of the input graph size. For this, we write a dynamic program traversing the SPQR-tree in a bottom-up manner, generating a solution for the problem out of smaller pieces recursively. Then, we will cover the decision problem FlexDraw as introduced by Bläsius et al. [BKRW12], which has been implemented as part of this work. The algorithm is also based on a dynamic program traversing the SPQR-tree. We provide a technical description of the algorithm and some pseudo-code for the most interesting procedures. Furthermore, we cover some details of our implementation we have done in this work. Finally, we evaluate our implementation experimentally and answer a couple of interesting questions, also regarding the demand for better algorithms.

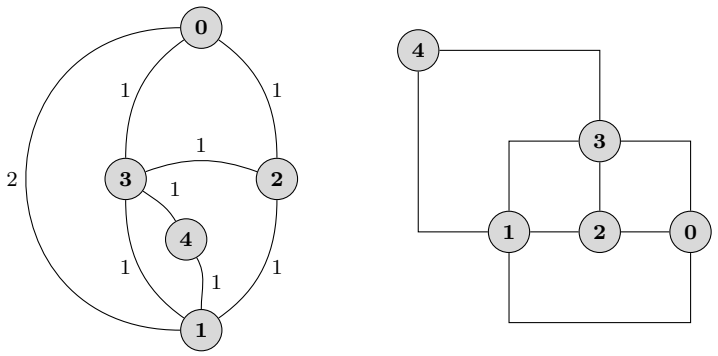
1.1. The Problem

We consider the following two general problems for bend-minimization in orthogonal drawings of 4-planar graphs whose combinatorial embedding is not fixed.

Problem 1 (General FlexDraw: drawing with flexibility constraints)**.** *Given a 4-planar graph $G = (V, E)$ and a flexibility function* $\text{flex} : E \to \mathbb{N}_0$*. Does G admit a planar embedding on the grid with at most* $\text{flex}(e)$ *bends for every edge?*

Problem 2 (General OptimalFlexDraw: bend minimization with cost functions)**.** *Given a 4-planar graph $G = (V, E)$ and for every edge $e \in E$ a cost function* $\text{cost}_e : \mathbb{N}_0 \to \mathbb{R}$*. Find a drawing minimizing* $\sum_{e \in E} \text{cost}_e(\text{bends}(e))$*.*

The problem to decide whether or not a graph is *0-embeddable*, (admitting an orthogonal drawing without any bends) has been proven to be $\mathcal{NP}$-hard by Garg and Tamassia [GT95]. Since an algorithm solving one of the two problems from above can be used to decide 0-embeddability of a graph by setting $\text{flex}(e) = 0$ or $\text{cost}_e(x) = x$ respectively for every edge e in the graph, they both are $\mathcal{NP}$-hard, too.



(a) (b)

Figure 1.2.: (a) An exemplary instance for FlexDraw with the flexibility constrained labeled on the edges and (b) an orthogonal drawing respecting this constraints

In a similar way the $\mathcal{NP}$-hardness of the following problems follows:

- We can test if a graph is *0-embeddable* by setting $\text{flex}(e) = 0$ for every edge $e \in E$ in the problem General FlexDraw.
- By setting $\text{cost}_e(n) = n$ for every edge $e \in E$ in the problem General OptimalFlexDraw we treat all bends on all edges equally and thus *minimize the total number* of bends.
- Given a weight function $w : E \to \mathbb{R}^+$ we can minimize the total bends *weighted per edge*. Therefore we set $\text{cost}_e(n) = n \cdot w(e)$ in the problem General OptimalFlexDraw.

Since all these problems are $\mathcal{NP}$-hard, we are interested in algorithms solving slightly modified versions of them in polynomial time in order to make them attractive for practical use.

Problem 3 (FlexDraw)**.** *Given a 4-planar graph $G = (V, E)$ and a function* $\text{flex} : E \to \mathbb{N}\backslash\{0\}$ *that gives each edge a positive flexibility. Does G admit a planar drawing on the grid such that each edge e has at most* $\text{flex}(e)$ *bends?*

Note that we do not allow $\text{flex}(e) = 0$ for any edge $e \in E$, otherwise this problem would be $\mathcal{NP}$-hard as pointed out above. An exemplary instance for FlexDraw and a valid drawing is shown in Figure 1.2. We will use this instance for illustration throughout this work.

For any instance of FlexDraw, we call a corresponding drawing of G a *flex-drawing* if it respects the flexibility constraints, i.e., it has no more than $\text{flex}(e)$ bends on each edge e. We say G can be *flex-drawn*.

Problem 4 (OptimalFlexDraw)**.** *Given a 4-planar graph $G = (V, E)$ and for each edge an increasing and convex cost function* $\text{cost}_e : \mathbb{N}_0 \to \mathbb{R}$ *with the restriction* $\text{cost}_e(0) = \text{cost}_e(1)$ *that defines the cost for a given number of bends on that edge. Find an orthogonal drawing minimizing* $\sum_{e \in E} \text{cost}_e(\text{bends}(e))$.

Note that we do not allow additional costs for the first bend on any edge $e \in E$. Without loss of generality, we can just subtract the *base cost* $\text{cost}_e(0)$ from the whole cost function for every edge, thus also $\text{cost}_e(1) = 0$. A cost function $\text{cost}_e : \mathbb{N}_0 \to \mathbb{R}$ is *convex* if and only if $\Delta\,\text{cost}_e(x+1) \geq \Delta\,\text{cost}_e(x)$ holds for all $x \in \mathbb{N}\backslash\{0\}$, where $\Delta\,\text{cost}_e(x) := \text{cost}_e(x) - \text{cost}_e(x-1)$ denotes the additional costs for the x-th bend on edge e.

Problem 5 (OptimalFlexDraw for series-parallel graphs)**.** *Given a 4-planar series-parallel graph $G = (V, E)$ and for each edge $e \in E$ a monotone cost function $\text{cost}_e : \mathbb{N}_0 \to \mathbb{R}$ defining the cost for a given number of bends on that edge. Find an orthogonal drawing minimizing $\sum_{e \in E} \text{cost}_e(\text{bends}(e))$.*

We call a cost function cost_e *monotone* if $\text{cost}_e(x+1) \geq \text{cost}_e(x)$ for all $x \in \mathbb{N}_0$, i.e. we never decrease the costs for an increasing number of bends. In other words, in contrast to the general OptimalFlexDraw we restrict the input graph to be series-parallel, while allowing a non-convex cost function and also costs for the first bend on any edge $e \in G$.

1.2. Related Work

As already addressed in the introduction, the *0-embeddability* of a graph, deciding whether it can be drawn without any bend, is shown to be $\mathcal{NP}$-hard by Garg and Tamassia [GT95]. The $\mathcal{NP}$-hardness of problems 1 and 2 from above follows directly.

Biedl and Kant [BK98] show that every 4-planar graph can be drawn with at most two bends on each edge (*2-embeddability*) except for the tetrahedron. Di Battista et al. [DLV98] introduce an algorithm minimizing the number of bends in an orthogonal drawing for *3-planar graphs* as well as for *series-parallel graphs* and thus decides *0-embeddability* for these special cases.

Tamassia [Tam87] introduces the following problem. Given a graph G with a *fixed* combinatorial embedding, compute a drawing with a minimum total number of bends and minimum size. Tamassia introduces the *orthogonal representation*, an abstract description of an orthogonal drawing not yet defining any metrics but shapes of edges, which we cover in the next chapter. It splits the problem into two parts: Finding an orthogonal representation with a minimum total number of bends and generating a drawing with minimum size based on this representation afterwards. For the former, Tamassia introduces a polynomial-time algorithm using a flow network. The latter has been proven to be $\mathcal{NP}$-complete by [Pat99]. However, Tamassia introduced an algorithm which generates a drawing with at most linear size in both dimensions, also based on a flow network. To emphasize the difference in Tamassia's problem and our OptimalFlexDraw problem from above you should recall that we do *not* fix the combinatorial embedding of the input graph, while Tamassia does.

Bläsius et al. [BKRW12] introduce the problem FlexDraw from above, deciding if a graph and a given positive flexibility admit an orthogonal drawing without fixing the combinatorial embedding. They provide an algorithm with a worst-case running time of $\mathcal{O}(n \cdot T_{\text{flow}}(n))$, where $\mathcal{O}(T_{\text{flow}}(n))$ denotes the worst-case runtime to calculate a minimum-cost flow in a planar network with multiple sources and sinks.

Bläsius et al. [BRW12] also cover the optimization problem OptimalFlexDraw minimizing bends with the combinatorial embedding *not* being fixed. For this, they introduce convex functions for each edge individually, not allowing any costs for the first bend. They provide an algorithm with a worst-case time of $\mathcal{O}(n \cdot T_{\text{flow}}(n))$ for biconnected and $\mathcal{O}(n^2 \cdot T_{\text{flow}}(n))$ for connected graphs.

Both algorithms can be used to decide *1-embeddability* for general graphs. The restriction of the cost function not to be able to assign costs for the first bend as well as its convexity motivated us to find better algorithms possibly restricted on the graph class.

Within this work, FlexDraw is implemented and evaluated. While being efficient, the major inconvenience of both problems introduced by Bläsius et al. is that they do not allow any restrictions on the first bend per edge. However, for some classes of graphs it is possible to still allow costs for the first bend; for *series-parallel graphs* an algorithm is introduced in this work.

2. Preliminaries

Before we start, we define some graph classes and introduce data structures as well as some notations.

A graph is *connected* if, for every pair of vertices $u, v \in V$, there exists a path $(u, \ldots, v)$ in G, otherwise it is *disconnected*. A vertex is called a *cut vertex* if its removal disconnects the graph into two or more components. We call the maximal subgraphs of G in which v is not a cut vertex the *cut components* with respect to v. A connected graph is *biconnected* if it does not contain a cut vertex. A pair of vertices is called a *separation pair* if its removal disconnects the graph. A biconnected graph is *triconnected* if it does not contain a separation pair. A maximal biconnected subgraph of a graph is called a *block*. Note that every vertex is part of at least one block and every cut vertex is part of at least two blocks.

Given an undirected graph $G = (V, E)$. For every edge $e = \{s, t\} \in E$, we define the undirected graph $G - e := (V, E \backslash \{e\})$ with the *poles* s and t.

In the following sections we define the class of *series-parallel graphs* used in the version of OptimalFlexDraw for which we provide an algorithm in Chapter 3. We will also briefly cover two data structures we use for decomposition of graphs, namely the *SPQR-tree* and the *block-cutvertex-tree*. As addressed in the introduction, in order to describe orthogonal drawings on an abstract level without providing any metrics we define the *orthogonal representation* as introduced by Tamassia [Tam87].

2.1. The Class of Series-Parallel Graphs

One of the algorithms in this work is restricted to the class of so-called *series-parallel graphs*. Every series-parallel graph G has two poles s and t. The class is recursively defined as follows.

- A graph $G = (V, E)$ with two nodes $V = \{s, t\}$ and a single edge $E = \{\{s, t\}\}$ is a series-parallel graph with the poles s and t.
- Let $G_i = (V_i, E_i), i = 1 \ldots k$ be series-parallel graphs with the poles s_i and t_i. Then
  - the parallel composition $G = (\bigcup V_i, \bigcup E_i)$ is a series-parallel graph with the poles s and t where all poles s_i are identified to s and all poles t_i to t.
  - the series composition $G = (\bigcup V_i, \bigcup E_i)$ is a series-parallel graph with the poles s_1 and t_k where t_i is identified with s_{i+1} for all $i = 1 \ldots k - 1$.

This definition directly implies that all series-parallel graphs are connected.

2.2. SPQR-Trees

The SPQR-tree is a data structure that can be used to handle the decomposition of biconnected graphs into triconnected components. A detailed description of the SPQR-tree can be found in the literature [DBT96, GM01].

The SPQR-tree $\mathcal{T}$ of a graph G is a tree with four different types of nodes, namely S-, P-, Q- and R-nodes (hence its name). For every edge $e \in E$ there exists a Q-node μ_e that represents e and we define $\text{orig}(\mu_e) := e$ as the edge represented by a Q-node. S-, P- and R-nodes define compositions of other subgraphs of G represented by child nodes in the tree.

The structure of $\mathcal{T}$ is determined by the *split pairs* of G. A pair of vertices s, t is called a *split pair* if it is either connected by an edge $s = \{s, t\}$ in G or a separation pair. Recall that a separation pair is a pair of vertices disconnecting G into two or more *split components* $\mathcal{S}_{s,t}$ of the split pair. A split component is a maximal subgraph of G not having s, t as a split pair. Note that a single edge $s = \{s, t\}$ in G also is a split component. A split pair defines a P-node if $|\mathcal{S}_{s,t}| \geq 3$. If $|\mathcal{S}_{s,t}| = 2$ and one split component is not biconnected, it defines an S-node. Otherwise, it is an R-node representing *rigid structures* in G.

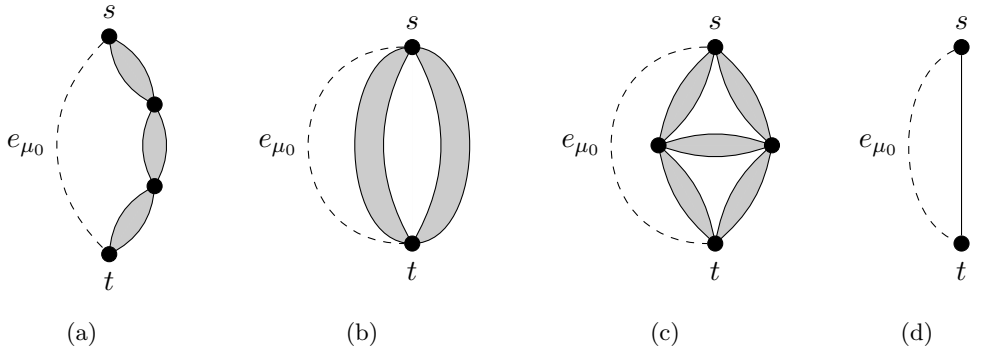


Figure 2.1.: Exemplary skeletons of the four different node types in an SPQR-tree: (a) an S-node with 3 children, (b) a P-node with 2 children, (c) an R-node with the structure of the K_4 and (d) a Q-node. e_{μ_0} represents the reference edge in the skeleton with the respect to the root of $\mathcal{T}$. The gray edges represent children nodes in $\mathcal{T}$ which for themselves represent subgraphs of G, while the edge of a Q-node represents a single edge in G.

We typically consider an SPQR-tree $\mathcal{T}$ of a graph $G = (V, E)$ to be rooted at a particular Q-node $\mu_e (e \in E)$ of $\mathcal{T}$. However, we can always *re-root* $\mathcal{T}$ at any other node, which does not affect the structure of the tree. We denote the adjacent nodes of μ as $\mu_0, \ldots, \mu_k$, where $k + 1$ is the number of adjacent nodes. If $\mathcal{T}$ is rooted, μ_0 is the *parent* node of μ and the other k adjacent nodes $\mu_1, \ldots, \mu_k$ are called *child* nodes. Note that re-rooting $\mathcal{T}$ in general changes those indices, as the parent node of a node μ might change.

In order to "describe" the structure of a node μ in $\mathcal{T}$ and where the child nodes in the SPQR-tree are located, each node has an associated planar multi-graph, its *skeleton* $\text{skel}(\mu)$, which can be seen as a simplified version of G. Its nodes correspond to nodes in G. The skeleton contains one edge e_{μ_i} for every adjacent node μ_i of μ in $\mathcal{T}$. When $\mathcal{T}$ is rooted at a particular root node, $\text{ref}(\mu) := e_{\mu_0}$ denotes the *reference edge* in $\text{skel}(\mu)$ with respect to this root. The skeleton of a Q-node only contains two edges: The reference edge and an edge representing an edge in the original graph. We denote this edge $\text{orig}(\mu)$. Figure 2.1 shows exemplary skeletons of the four node types.

Notation	Description
skel(μ)	the skeleton graph of μ in the SPQR-tree
pert(μ)	the pertinent graph of μ in the SPQR-tree
$\mu_i \ \ (i = 1..k)$	the child nodes of μ in the rooted SPQR-tree
μ_0	the parent node of μ in the rooted SPQR-tree
$e_{\mu_i} \ \ (i = 1..k)$	the edge in skel(μ) corresponding to μ_i
$e_{\mu_0} = \text{ref}(\mu)$	the reference edge in skel(μ)
orig(μ)	the original edge $e \in E$ in the original graph if μ is a Q-node
$\mu_e \ \ (e \in E)$	the Q-node in $\mathcal{T}$ corresponding to $e \in E$ in the original graph G
emb(μ)	all possible combinatorial embeddings of skel(μ)

Table 2.1.: Overview of notations regarding SPQR-trees used in this work

Since each skeleton is a planar graph, it has a certain combinatorial embedding. This allows us to change the embedding of skel(μ) which implicitly changes the combinatorial embedding of G. An embedding of G implies well-defined embeddings of all skeletons of $\mathcal{T}$ as well as embeddings of all skeletons define one well-defined embedding of G. On the other side, if we do *not* fix the embedding of G, the skeletons also don't have a fixed embedding. Very important is the fact that different combinations of all skeleton embeddings in $\mathcal{T}$ implicitly define *all* possible embeddings of the whole graph G. We will use this decomposition to optimize over all of them. Note that changing the embedding of an skeleton skel(μ) of a tree-node μ does not flip the whole embedding of its pertinent graph pert(μ) in G but only affects how the child nodes of μ are placed within the skeleton skel(μ). Flipping the embedding of a whole pertinent graph pert(μ) in G requires flipping of the skeletons of all nodes in the sub-tree rooted at μ.

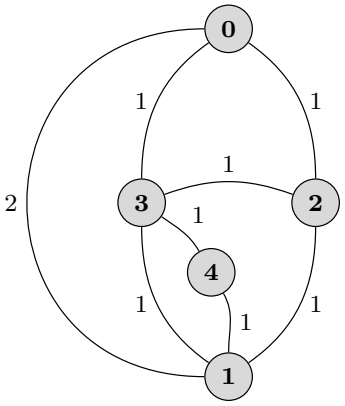
To illustrate how the embedding of the skeleton of a node in $\mathcal{T}$ affects the embedding of G, we have a look again at the exemplary instance as seen in Figure 2.2. The two operations shown in (c) to (e) illustrate the steps needed to fully flip the embedding of the whole *pertinent* graph of the R-node, resulting in the combinatorial embedding of G of the output graph as seen in Figure 1.2 from the previous chapter.

We denote the set of possible combinatorial embeddings of skel(μ) by emb(μ). The number of embeddings is 1 for Q- and S-nodes, 2 for R-nodes (they can only be flipped) and $k!$ for P-nodes (every possible permutation of the skeleton edges defines an embedding).

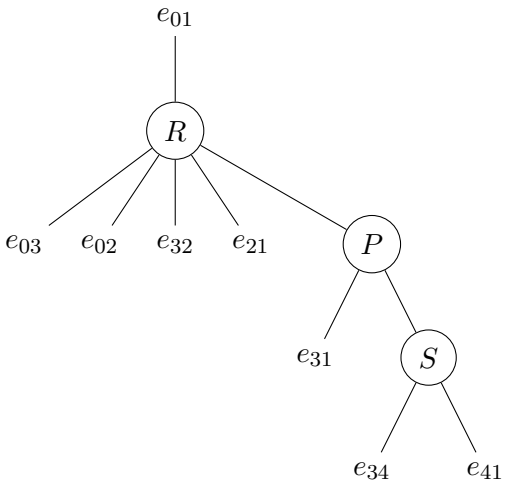
The subtree of $\mathcal{T}$ with root μ represents a subgraph in G called the *pertinent graph* of μ with respect to the root node of $\mathcal{T}$. We denote this pertinent graph as pert(μ) which is the single edge orig(μ) if μ is a Q-node. Otherwise, we construct the pertinent graph recursively by replacing every edge $e_{\mu_1}, \ldots, e_{\mu_k}$ in skel(μ) having k child nodes with pert(μ_i) and removing the reference edge e_{μ_0}. Note that if $\mathcal{T}$ is rooted at μ, its pertinent graph pert(μ) is $G - e$.

Note that for 4-planar graphs a P-node can have at most four adjacent nodes in $\mathcal{T}$ of which one is the parent node. Thus, the number of children of a P-node is limited to three, limiting the number of possible embeddings of its skeleton to six. This limit is very important for the time complexity in an algorithm handling all possible embeddings of a skeleton as we do.

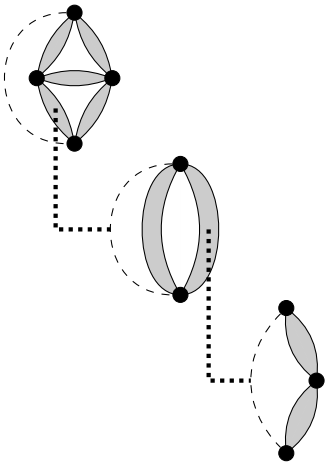
Table 2.1 shows an overview of all notations we defined for the SPQR-tree.



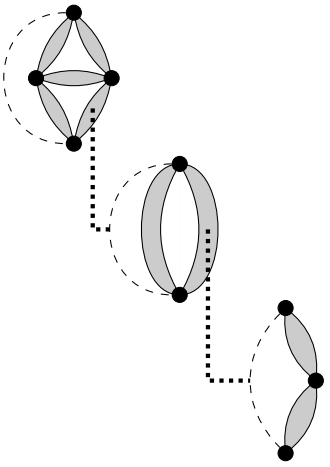
(a) The original graph



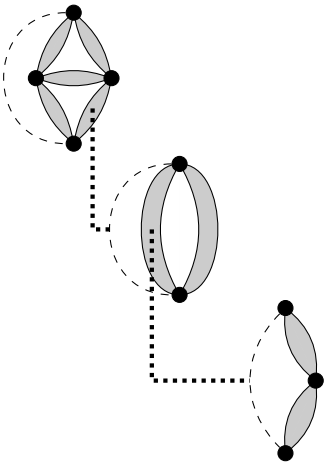
(b) The SPQR-tree only, without any skeletons being shown



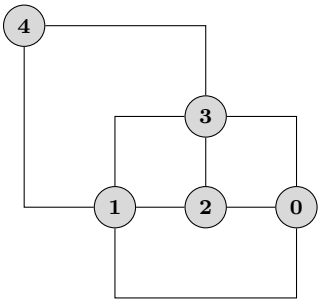
(c) Skeleton embeddings corresponding to the input



(d) Skeleton embeddings after flipping the R-node's skeleton



(e) Skeleton embeddings after also flipping the P-node's skeleton



(f) The output of FlexDraw after flipping the pertinent graph of the R-node

Figure 2.2.: The SPQR-tree of our exemplary instance from the previous chapter

2.3. Block-Cutvertex-Trees

The *block-cutvertex-tree* (BC-tree) is a data structure that can be used to handle the decomposition of a connected component G into blocks and cut vertices. The tree consists of B- and C-nodes. A B-node represents a block in G, a C-node represents a cut vertex in G. A C-node in the tree is adjacent to a B-node if and only if it is part of that block. Note that a cut vertex can be part of more than two blocks.

2.4. Orthogonal Representations and Rotations

Tamassia [Tam87] introduced the *orthogonal representation* of a graph based on a combinatorial embedding, describing the bends on edges and the angles of faces. We will use this representation as a combinatorial description of a grid drawing. Note that an orthogonal representation is not a complete description of a drawing, as it defines neither any metrics nor absolute orientations of edges.

A slightly simplified definition of orthogonal representations suitable for this work is defined as follows. For every face f_i of G there is a *face description* consisting of a *bend number* for every incident edge in clockwise order (counter-clockwise for the external face) and the *inner angle* for every incident node as a multiple of 90°. A bend number of an edge incident to a face f_i tells us the number of bends to the right ("convex" bends if f_i is an internal face). If this number is negative, the absolute value equals the number of bends to the left ("concave" bends if f_i is an internal face).

Note that every edge $e \in E$ has exactly two bend numbers since it is incident to two (not necessarily different) faces f_l, f_r. Every node $v \in V$ is incident to one to four faces. The following constraints have to be satisfied for every valid orthogonal representation.

1. For every edge $e \in E$ the two bend numbers b_l and b_r in its two incident faces f_l and f_r respectively are consistent, i.e., $b_l = -b_r$. We define $\text{bends}(e) := |b_l| = |b_r|$.
2. The angular sum of any face f_i is 4 if f_i is an internal face and -4 if f_i is the external face. The angular sum is the sum of inner angles of all incident nodes plus the sum of bend numbers of all incident edges.
3. The angles around each node sum up to 4.

Once we found an orthogonal representation of a graph, an orthogonal drawing, where the bends and angles at the nodes correspond to the orthogonal representation, can be computed efficiently [Tam87]. We define the number of bends $\text{bends}(e)$ of an edge e in an orthogonal drawing being the absolute bend number of that edge in the orthogonal representation.

Bläsius et al. [BKRW12] introduce the following notations to describe "how well" a graph can be bent in a flex-drawing. Given a graph $G = (V, E)$, two poles $s, t \in V$ and an orthogonal representation, we define the *rotation* $\text{rot}(\pi(s,t))$ describing the *path* $\pi(s,t)$ from s to t along the outer face in counter-clockwise direction as being the number of bends to the right minus the number of bends to the left. As an example, let us calculate $\text{rot}(\pi(4,0))$ in Figure 2.3: The edge $(4,1)$ has one bend to the left, the edge $(1,0)$ two bends to the left. The node (1) introduces one bend to the right on the path $\pi(4,0)$, thus the rotation of $\pi(4,0)$ is $1 - 3 = -2$ in this drawing.

The *maximum rotation of a given embedding* $\mathcal{E}$, $\text{maxrot}_{\mathcal{E}}(\pi(s,t))$, is the highest possible value $\text{rot}(\pi(s,t))$ can attain for any flex-drawing of this embedding $\mathcal{E}$. A further generalization is denoted by $\text{maxrot}(\pi(s,t))$, the *maximum rotation* of the path $\pi(s,t)$ in *any*

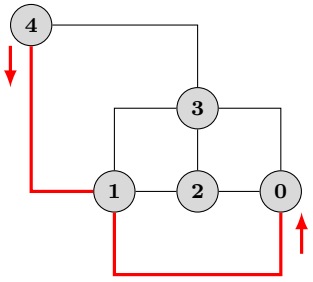


Figure 2.3.: Our exemplary instance for FlexDraw with the path $\pi(4,0)$ highlighted, on which we count one bend to the right $(+1)$ and three bends to the left (-3)

embedding of G, which is defined as the maximum value of $\text{maxrot}_{\mathcal{E}}(\pi(s,t))$ over all embeddings of G. If $e = \{s,t\}$ is an edge in G, $\text{maxrot}(\pi(s,t))$ in $G - e$ is also written as $\text{maxrot}(G-e)$. When not talking about chosen pole nodes but only a graph G, we define the *graph rotation* with respect to an edge $e \in E$ as $\text{graphrot}_e(G) := \text{maxrot}(G-e)+\text{flex}(e)-2$.

Given a graph $G = (V,E)$ and a positive flexibility function $\text{flex} : E \to \mathbb{N}\backslash\{0\}$ a graph can be *flex-drawn* if there exists a drawing with at most $\text{flex}(e)$ bends for every edge $e \in E$. Such a drawing is called a *flex-drawing*. We say *e admits a flex-drawing* if G can be flex-drawn with e being on the external face, which is true if and only if $\text{graphrot}_e(G) \geq 0$.

Tamassia [Tam87] introduces a flow network that can be used to calculate the orthogonal representation having the minimum number of bends for a given graph G with a fixed combinatorial embedding. The embedding is represented by the set of faces, $\mathcal{F}$, and an external face $f_0 \in \mathcal{F}$.

Definition 2.1. *The network $N(G) := ((W,A); l; u; d; \text{cost})$ of a graph $G = (V,E)$ with a fixed combinatorial embedding $\mathcal{F}$ is defined as follows.*

$$\begin{aligned} W &:= V \cup \mathcal{F} \\ A &:= \{(v,f) \in V \times \mathcal{F}, v \text{ incident to } f\} \cup \\ &\quad \{(f_a,f_b) \in \mathcal{F} \times \mathcal{F}, f_a \text{ and } f_b \text{ adjacent}\} \end{aligned}$$

The demand of a node is $d(v) = 4$ for nodes of $v \in V$ and

$$d(f) := \begin{cases} -2 \cdot (d_G(f)+2) & , \text{if } f = f_0 \\ -2 \cdot (d_G(f)-2) & , \text{if } f \neq f_0 \end{cases}$$

for face nodes $f \in \mathcal{F}$. The edge capacity bounds l, u and the edge costs cost *are defined as*

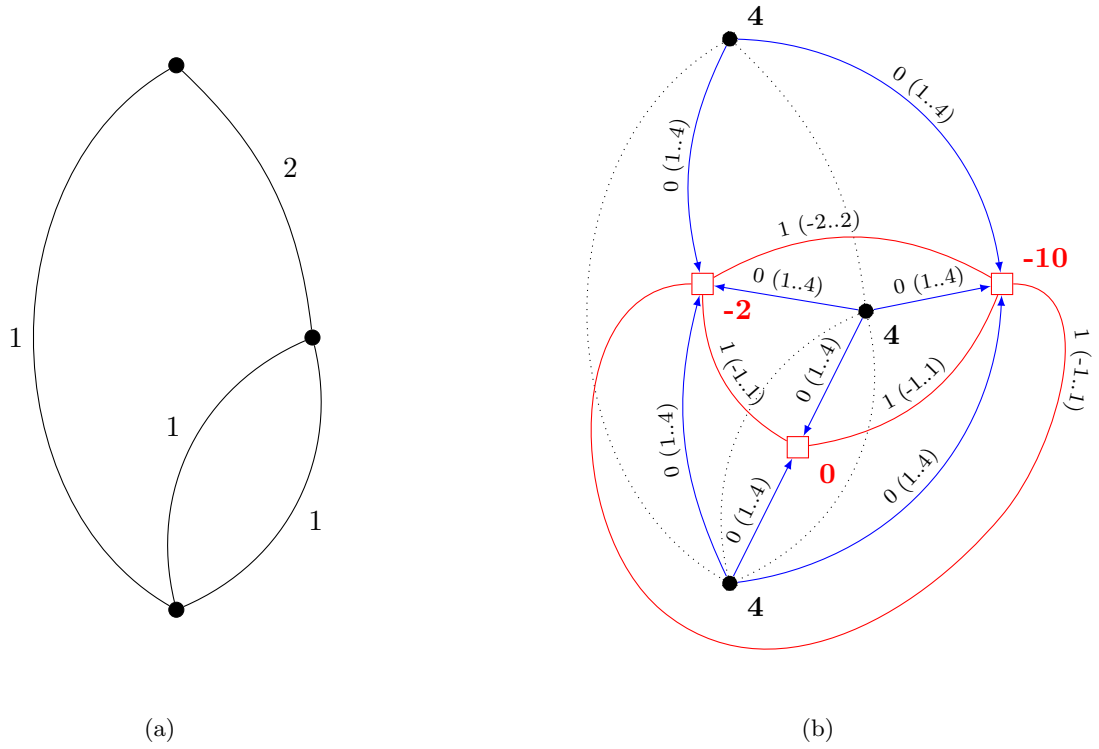
$$l(v,f) := 1, \quad u(v,f) := 4, \quad \text{cost}(v,f) := 0$$

for node-to-face-edges with $v \in V, f \in \mathcal{F}$ and

$$l(f_a,f_b) := 0, \quad u(f_a,f_b) := \infty, \quad \text{cost}(f_a,f_b) := 1$$

for face-to-face-edges with $f_a, f_b \in \mathcal{F}$.

This definition of the flow network $N(G)$ can be interpreted as follows. For every node $v \in V$ there are four 90° angles that can and have to be "assigned" to incident faces, which is done using the node-to-face-edges in A. Every face-to-face-edge in A corresponds to an edge e in G, which is incident to both faces. We call the face-to-face-edge the *dual edge* of e. The units of flow that pass through this edge represent one bend in the direction of the flow.



(a) (b)

Figure 2.4.: An example graph with flexibility constraints (a) and the corresponding network of our version of Tamassia's algorithm computing an orthogonal representation (b). Red edges and red square nodes represent the dual graph; the blue edges connect dual nodes with original nodes. The bold labels besides nodes are their demand, labels on edges denote their cost as well as the lower and upper capacity written in parentheses. Between two adjacent faces we have only drawn a single undirected edge denoting two directed edges in the flow network.

The flow network can be solved using a minimum-cost flow algorithm which moves exactly $d(v)$ units away from a node if $d(v)$ is positive and $-d(v)$ units into a node, if $d(v)$ is negative. The flow which passes through an edge e has to be at least $l(e)$ and at most $u(e)$. The flow algorithm minimizes the total cost of the flow. We can compute an orthogonal representation for a flex-drawing by slightly modifying $N(G)$. Since $u(f_a, f_b)$ represents the upper bound, we simply set it to $\text{flex}(e)$ of the corresponding edge $e \in E$ in the original graph. An example is given in Figure 2.4.

In the following we want to calculate the maximum rotation $\text{maxrot}(\pi(s,t))$ of the pertinent graph $\text{pert}(\mu)$ using the skeleton $\text{skel}(\mu)$ and already computed maximum rotations of the subgraphs represented by the edges in the skeleton. Bläsius et al. [BKRW12] show that each edge in the skeleton can be replaced by a gadget $G^{\rho}_{\deg(s),\deg(t)}$ with the same behavior so that the maximum rotation of this *gadget graph* equals $\text{maxrot}(\pi(s,t))$ of the pertinent graph $\text{pert}(\mu)$. For this, we have to know the maximum rotation $\rho := \text{maxrot}(\pi(s,t))$ and pole degrees $\deg(s), \deg(t)$ of the subgraph represented by each skeleton edge. The gadgets we will use are shown in Figure 2.5, which was taken from [BKRW12].

We introduce a network similar to Tamassia's, which we will use to calculate $\text{maxrot}(\pi(s,t))$ of a pertinent graph $\text{pert}(\mu)$. First, we replace all edges in $\text{skel}(\mu)$ with appropriate gadgets. For this, we have to recursively calculate the maximum rotation of the corresponding subgraph. Then we construct a network graph like Tamassia's but again with the flexibility

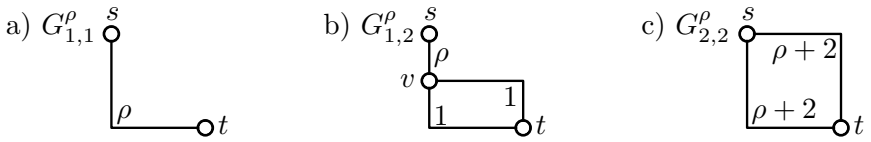


Figure 2.5.: Gadgets used to calculate the maximum rotation of pert(μ) using the skeleton skel(μ) in a dynamic program, depending on the pole degrees

constraint used as the upper edge capacity $u(e) := \text{flex}(e)$. We also have to give the face-to-face-edge in the network corresponding to ref(μ) a negative cost of $\text{cost}(\text{ref}(\mu)) := -1$ and all the other face-face-edges a zero cost. This will automatically pass as many units of flow as possible through the gadget graph, either from left to right or from right to left, depending on which results in a lower cost, and thus in a greater amount of flow passed through the reference edge's dual edge. This direction indicates whether or not the skeleton's embedding has to be flipped in order to achieve this rotation. The direction of the flow through each gadget determines if a child node has to flip the embedding of its *pertinent graph* in order to achieve the rotation we interpret into the flow. We make use of this flow network in Chapter 5 when discussing the pseudo-code for the algorithm solving FlexDraw.

3. An Algorithm Solving OptimalFlexDraw for Series-Parallel Graphs

In this chapter we introduce an algorithm solving the general problem OptimalFlexDraw for series-parallel graphs. Let us recall the problem definition.

Problem 5 (OptimalFlexDraw for series-parallel graphs)**.** *Given a 4-planar series-parallel graph $G = (V, E)$ and for each edge $e \in E$ a monotone cost function $\text{cost}_e : \mathbb{N}_0 \to \mathbb{R}$ defining the cost for a given number of bends on that edge. Find an orthogonal drawing minimizing $\sum_{e \in E} \text{cost}_e(\text{bends}(e))$.*

In the following we define the cost of a *combinatorial embedding* $\mathcal{E}$, the cost of an *orthogonal drawing* of a 4-planar series-parallel graph G having its poles embedded on the external face as well as the *graph cost function* of G. We need these definitions in the following section, where we initially restrict the problem to such embeddings.

Definition 3.1. *Given a 4-planar series-parallel graph $G = (V, E)$, for each edge $e \in E$ a monotone cost function $\text{cost}_e : \mathbb{N}_0 \to \mathbb{R}$ and an orthogonal drawing $\mathcal{R}$ of G. Let further for any edge $e \in E$, $\text{bends}_{\mathcal{R}}(e)$ denote the number of bends of e in $\mathcal{R}$. We define the cost $\text{cost}_{\mathcal{R}} : \mathbb{Z} \to \mathbb{R}$ of the orthogonal representation $\mathcal{R}$*

$$\text{cost}_{\mathcal{R}}(x) := \sum_{e \in E} \text{cost}_e(\text{bends}_{\mathcal{R}}(e)).$$

We call $\mathcal{R}$ the optimal representation *of G for rotation x if there is no other orthogonal representation $\mathcal{R}'$ with $cost_{\mathcal{R}'}(x) < cost_{\mathcal{R}}(x)$.*

Definition 3.2. *Given a 4-planar series-parallel graph $G = (V, E)$ with the poles $s, t \in V$, for each edge $e \in E$ a monotone cost function $\text{cost}_e : \mathbb{N}_0 \to \mathbb{R}$ and a combinatorial embedding $\mathcal{E}$ of G having s and t embedded on the external face. Let $\Omega^x_{\mathcal{E}}$ denote the set of orthogonal representations with the embedding $\mathcal{E}$ and a rotation of $\text{rot}_{\mathcal{E}}(\pi(s, t)) = x \in \mathbb{Z}$. We define the cost $\text{cost}_{\mathcal{E}} : \mathbb{Z} \to \mathbb{R}$ of the combinatorial embedding $\mathcal{E}$*

$$\text{cost}_{\mathcal{E}}(x) := \min_{\mathcal{R} \in \Omega^x_{\mathcal{E}}} \sum_{e \in E} \text{cost}_e(\text{bends}_{\mathcal{R}}(e)).$$

Definition 3.3. *Given a 4-planar series-parallel graph $G = (V, E)$ with the poles $s, t \in V$ and for each edge $e \in E$ a monotone cost function $\text{cost}_e : \mathbb{N}_0 \to \mathbb{R}$. Let $\mathcal{X}$ denote the set of all combinatorial embeddings of G having s and t embedded on the external face. We define the* graph cost function

$$\text{cost}_G(x) := \min_{\mathcal{E} \in \mathcal{X}} \text{cost}_{\mathcal{E}}(x).$$

If $\text{cost}_{\mathcal{E}}(x) = \text{cost}_G(x)$, then $\mathcal{E}$ is called the optimal embedding *of G for rotation x.*

Note that minimizing $\text{cost}_G(x)$ finds the rotation $x = \text{rot}(\pi(s, t))$ of the drawing with minimal costs. The idea of the following algorithm is, for every block in G, to traverse the SPQR-tree $\mathcal{T}$ of the block in a bottom-up manner and to compute the graph cost function of a node μ in $\mathcal{T}$ with the cost functions of its child nodes in a dynamic program. A drawing of G can be computed by generating and merging orthogonal representation of all blocks in G afterwards.

Before we can start describing and proving this dynamic program, we have to transform the edge cost functions in the problem instance into graph cost functions for every edge $e \in E$. Let us denote the subgraph in G having only the edge e by G_e.

Corollary 3.4. *Let $G = (V, E)$ be any graph, $e = \{v, w\} \in E$ any edge of G and cost_e its edge cost function. Then, the following holds.*

$$\text{cost}_{G_e}(x) = \text{cost}_e(|x|).$$

Proof. The correctness is easy to see, as we can bend e in both directions leading to the same costs and the bends on that edge will always equal the absolute value of its rotation. Recall that the rotation is, in contrast to the bends of an edge, *directed*. □

The algorithm in this chapter is based on an approach similar to FlexDraw as introduced by Bläsius et al. [BKRW12], which we cover in the next chapter. Basically it is divided into two parts: We first find the optimal representation for each block of G. Once found, we merge those orthogonal representations in the second step allowing us to easily calculate an optimal drawing using Tamassia's approach as seen in Section 2.4.

3.1. Biconnected Series-Parallel Graphs

We now show how to find an optimal representation of a *biconnected series-parallel graph* G. Given a biconnected series-parallel graph $G = (V, E)$, the graph $G - e$ is a series-parallel graph for any edge $e = \{s, t\} \in E$, where s and t are the poles of $G - e$. We use the SPQR-tree $\mathcal{T}$ of G in order to decompose G. Recall that we can root $\mathcal{T}$ at any Q-node $\mu_e \in \mathcal{T}$. When choosing an edge $e \in E$ and rooting $\mathcal{T}$ at the Q-node $\mu_e \in \mathcal{T}$, $\mathcal{T}$ represents a series-parallel decomposition of $G - e$. Note that not only the graph $G - e$ changes when choosing a different edge $e \in E$ to root $\mathcal{T}$ at, but also the parentship of the tree nodes.

Now we are ready to introduce the following modified version of the problem.

Problem 6 (OptimalFlexDraw for biconnected series-parallel graphs)**.** *Given a 4-planar, biconnected series-parallel graph $G = (V, E)$ and for each edge $e = \{v, w\} \in E$ a graph cost function $\text{cost}_{G_e} : \mathbb{Z} \to \mathbb{R}$ defining the cost for a given rotation of the subgraph $G_e = (\{v, w\}, \{\{v, w\}\})$. Find the drawing minimizing $\text{cost}_G(x)$, where x is the rotation of the orthogonal representation of this drawing.*

Problems 5 and 6 only differ in the type of input graphs: where the original problem allows any series-parallel graph, we further restricted them to being biconnected. Note that we also replaced the term *edge cost function* with *graph cost function*. As seen above, this transformation can easily be done. After solving this problem, we will later generalize it back to series-parallel not necessarily being biconnected.

3.1.1. Finding the Cost Function of a Subgraph

Once an SPQR-tree $\mathcal{T}$ has been constructed and rooted at an edge in the biconnected input graph, the nodes in $\mathcal{T}$ describe serial and parallel compositions of smaller subgraphs which can be single edges. We now write a dynamic program by expressing the cost function of a tree node μ using the cost functions of its k children $\mu_1, \dots, \mu_k$. We will also show that such a step in the dynamic program leads to the correct cost function and can be computed efficiently.

The following lemma is used to restrict the cost functions to rotations up to four times the graph size, which we will need to prove the efficiency (both time and space complexity) of the algorithm.

Lemma 3.5. *For every instance of* OptimalFlexDraw *for series-parallel graphs with n nodes there exists an optimal solution in which all series-parallel subgraphs have their rotation $\pi(s,t)$ in the range $\{-5n, \dots, 5n\}$, where s, t denote the poles of the subgraph.*

Proof. Recall the flow network introduced by Tamassia to calculate an orthogonal representation given a combinatorial embedding sets a supply of 4 units per node, resulting in a total supply of $4n$. The sum of the demands of all faces is also $4n$. On every edge of the flow network resulting from a dual edge in the original graph has cost 1, all other edges have cost 0.

Consider a solution to OptimalFlexDraw for series-parallel graphs having more than $4n$ bends in total, which is the case if we find a series-parallel subgraph having a rotation $\text{rot}(\pi(s,t))$ not being in the range $\{-5n, \dots, 5n\}$, since at most n angles of the rotation can be bends at vertices and at least $4n$ angles are bends on edges. We will show that a solution with such a number of bends can not be optimal.

If we construct a flow network according to Tamassia using the combinatorial embedding of the solution, we can show that the solution can not be optimal. The total supply of such a network is $4n$ and thus any valid flow with minimum costs does not exceed a total cost of $4n$, otherwise at least one unit would have been routed along a cycle, implying that the flow was not minimal. Since only the dual edges have any cost and they represent bends in the drawing, there are at most $4n$ bends per edge in any optimal drawing. □

Using this lemma, we will restrict the cost functions to the range $\{-5n, \dots, 5n\}$ in order to compute them efficiently.

In the following, we show how to compute $\text{cost}_{\text{pert}(\mu)}$ of a P- or S-node μ in $\mathcal{T}$ with k child nodes using their cost functions $\text{cost}_{\text{pert}(\mu_1)}, \dots, \text{cost}_{\text{pert}(\mu_k)}$. Note that a P-node can only have at most three children as seen in Section 2.2, so $k \leq 3$ in this case.

The degree of v in the whole graph G is denoted by $\deg(v)$. Note that the term "*pole degrees*" always refers to the degree of poles in a *subgraph* only, not in the whole graph.

Lemma 3.6. *Let G be a 4-planar, biconnected series-parallel graph, $\mathcal{T}$ the SPQR-tree of G and μ an S-node in $\mathcal{T}$ with two child nodes μ_1, μ_2. Let $\text{cost}_{\text{pert}(\mu_i)}$ be their cost functions. Then, the following holds.*

$$\text{cost}_{\text{pert}(\mu)}(x) = \min_{y \in \mathbb{Z}} \left(\text{cost}_{\text{pert}(\mu_1)}(y) + \min_{\alpha \in A} \left(\text{cost}_{\text{pert}(\mu_2)}(x - y - \alpha) \right) \right),$$

where $A = \{\deg(v) - 3, \dots, 1\}$. The cost function can be computed in $\mathcal{O}(n^2)$ time where n is the size of G.

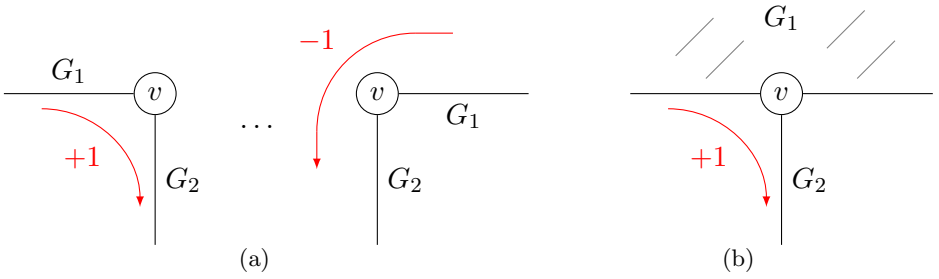


Figure 3.1.: When combining two graphs G_1, G_2 in a series with the cut vertex v, the rotation $\pi(s,t)$ via v can introduce different additional angles α at v. In (a) $\deg(v) = 2$ and α is between -1 and 1, in (b) $\deg(v) = 4$ and thus $\alpha = 1$.

Proof. For any rotation number x in $\text{pert}(\mu)$ we can choose an integer y defining the rotation of the first graph which implicitly chooses a very limited number of rotations for the second graph, since at the cut vertex v connecting these graphs we only have a limited number of possibilities for the angle α along the path $\pi(s,t)$ in the drawing of $\text{pert}(\mu)$. The smallest possible value for α is $\deg(v) - 3$, where $\deg(v)$ denotes the degree of the cut vertex v in G. If $\deg(v) = 4$, there is only one possibility how the rotation of the first graph (y) affects the rotation of the right graph ($x - y - 1$), since when adding the additional angle the path $\pi(s,t)$ has at v, this sums up to x. For every degree of $\deg(v)$ less than 4 the lower bound of α decreases by 1 accordingly. The maximum flexibility at v is achieved when it has degree 2. Figure 3.1 illustrates two different situations.

For a particular value y we can iterate through all values of A in constant time, since its cardinality is at most 3. For a particular value $x \in \{-5n, \dots, 5n\}$ we can calculate $\text{cost}_{\text{pert}(\mu)}$ in linear time since we only need to iterate over a linear number of values for y, as by Lemma 3.5, an optimal drawing of G has at most $5n$ bends on each edge. This results in time $\mathcal{O}(n^2)$ to calculate all values of the cost function for $x \in \{-5n, \dots, 5n\}$. □

Corollary 3.7. *For an S-node μ with $k > 2$ children, we can iteratively combine two children using Lemma 3.6 until we combined all children. We can interpret μ as a binary tree of serial compositions. In any case, this results in $k - 1$ applications of Lemma 3.6.*

Lemma 3.8. *Let G be a 4-planar, biconnected series-parallel graph, $\mathcal{T}$ the SPQR-tree of G and μ a P-node in $\mathcal{T}$ with two child nodes μ_1, μ_2. Let $\text{cost}_{\text{pert}(\mu_i)}$ be their cost functions and $d_{\text{pert}(\mu_i)}$ the sum of the pole degrees within the pertinent graphs of μ_i. Then, the following holds.*

$$\begin{aligned}\text{cost}_{\text{pert}(\mu)}(x) = \min\{\ &\text{cost}_{\text{pert}(\mu_1)}(x) + \min_{\alpha \in A_2} (\,\text{cost}_{\text{pert}(\mu_2)}(x+\alpha)\,),\\ &\text{cost}_{\text{pert}(\mu_2)}(x) + \min_{\alpha \in A_1} (\,\text{cost}_{\text{pert}(\mu_1)}(x+\alpha)\,)\ \}\,,\end{aligned}$$

where $A_i := \{d_{\text{pert}(\mu_i)}, \dots, \deg(s) + \deg(t) - 4\}$ for $i \in \{1, 2\}$. The cost function can be computed in $\mathcal{O}(n)$ time where n is the size of G.

Proof. The pole degrees of the pertinent graphs $d_{\text{pert}(\mu_i)}$ are at most 3 each, since otherwise their degree in G would be greater than 4 (recall that there is also the reference edge, so the sum of degrees in the subgraphs is at most 3). The skeleton of a P-node can be embedded in two ways, with either e_{μ_1} or e_{μ_2} being on the left.

The rotation x of $\text{pert}(\mu)$ in any drawing equals the rotation of the pertinent graph of the node being embedded on the left. For the right subgraph, a certain value α has to be

added to the rotation. This value $\alpha \in A_i$ is at least the degree of the left graph $G_{\text{pert}(\mu_i)}$. Its upper limit is in the range $\{2, \ldots, 4\}$, depending on the degrees of s and t in the whole graph G, which can be 3 or 4 each. If both are 4, the rotation of the parallel composition has no additional "flexibility"; if both are 3, it has an additional "flexibility" of 1 at both poles.

Thus, the set A_i describes all possible differences between the two rotations of the left and the right subgraph. We select the difference resulting in a minimal cost value. Given a certain rotation x of $\text{pert}(\mu)$ its cost is given by the minimum of the combined costs for the two possible embeddings.

Since for a particular value x we take the minimum value of $|A_1|+|A_2| \leq 6$ different values, we can compute all values of the cost function for $x \in \{-5n, \ldots, 5n\}$ in linear time. □

Now we can similarly combine three subgraphs as a parallel composition. Note that in this case, $\deg(s) = \deg(t) = 4$ and $d_{\text{pert}(\mu_i)} = 2$ for all $i \in \{1, 2, 3\}$. Thus, fixing the rotation x of the leftmost subgraph also fixes the rotation for the other, being $x + 2$ and $x + 4$ respectively. Note that we now have to consider six embeddings as represented by all permutations $\{i, j, k\} = \{1, 2, 3\}$.

Corollary 3.9. *For a P-node μ with three children $\mu_1, \ldots, \mu_3$, the following holds.*

$$\text{cost}_{\text{pert}(\mu)}(x) = \min_{\{i,j,k\}=\{1,2,3\}} \left(\text{cost}_{\text{pert}(\mu_i)}(x) + \text{cost}_{\text{pert}(\mu_j)}(x+2) + \text{cost}_{\text{pert}(\mu_k)}(x+4) \right)$$

Similar to Lemma 3.8, calculating the cost function of a P-node having three children can obviously be done in linear time, too. Let us sum things up.

Theorem 3.10. *Let G be a 4-planar, biconnected series-parallel graph with n nodes, $\mathcal{T}$ the SPQR-tree of G with root μ_e. The graph cost function $\text{cost}_{\text{pert}(\mu_e)}$ can be computed in time $\mathcal{O}(n^3)$.*

Proof. As the number of nodes in an SPQR-tree is limited to the graph size n, there are at most n steps in which we compute a new cost function using one of the lemmas from above, where the time complexity $\mathcal{O}(n^2)$ for S-nodes with two children clearly dominate the runtime over P-nodes. S-nodes with $k > 2$ children are processed using $k - 1$ times Lemma 3.6, resulting in time $\mathcal{O}(k \cdot n^2)$. However, since each vertex in G can only appear in exactly one skeleton as a vertex not being one of its poles, we use Lemma 3.6 only once per vertex in G, resulting in a worst-case runtime of $\mathcal{O}(n^3)$. □

3.1.2. Finding the Optimal Representation

So far, we can efficiently compute the cost function cost_G of a 4-planar, biconnected series-parallel graph $G = (V, E)$ with the poles $s, t \in V$. Recall that this function tells us the *minimum costs* of an orthogonal drawing of G with the two poles embedded on the *external face* having a certain rotation along the path from s to t. However, embedding them on the external face is not a restriction in the original problem. Recall also that traversing an SPQR-tree bottom up ends at the root node of $\mathcal{T}$ which is a Q-node as explained in Section 2.2, but we did not specify how to compute the graph cost function in this case; we can only compute it for the S- or P-node adjacent to the root Q-node.

The idea now is to iterate over all edges in G which we want to allow to be embedded on the external face. We call those edges the *candidate edges* for being external. To solve OptimalFlexDraw for biconnected series-parallel graphs, we want any edge to be a

candidate. We will see in the following section that we have to restrict the set of candidate edges to solve OptimalFlexDraw for connected and general series-parallel graphs.

Let us denote the set of candidate edges allowed to be embedded on the external face as $E^\star$. We iterate over all candidate edges $e^\star = \{s^\star, t^\star\} \in E^\star$ and root the SPQR-tree $\mathcal{T}$ at $\mu_{e^\star}$. We can then compute the cost function $\text{cost}_{G-e^\star}$ of the series-parallel graph $G - e^\star$ with the poles $s^\star$ and $s^\star$ efficiently by Theorem 3.10 using the following dynamic program.

We traverse the rooted SPQR-tree $\mathcal{T}$ of G in a bottom-up manner. For every Q-node μ_e representing an edge $e \in E$, its cost function is calculated by Corollary 3.4. For every S-node μ with k children $\mu_1, \ldots, \mu_k$, its cost function is calculated by Lemma 3.6 (if $k = 2$) or Corollary 3.7 (if $k \geq 3$). For every P-node μ with k children $\mu_1, \ldots, \mu_k$, its cost function is calculated by Lemma 3.8 (if $k = 2$) or Corollary 3.9 (if $k = 3$). Since we have shown that the cost function is only relevant on the range $\{-5n, \ldots, 5n\}$, we limit all computations on this range.

Finally, we add the edge $e^\star = \{s^\star, t^\star\}$ to the cost function $\text{cost}_{G-e^\star}$ as follows. Similar to a P-node having two children, adding the edge $e^\star$ to an orthogonal drawing of $G - e^\star$ may introduce a small flexibility for the number of bends on $e^\star$ depending on the degrees of $s^\star$ and $t^\star$ in G. For every rotation $\text{rot}(\pi(s^\star, t^\star)) =: x \in \{-5n, \ldots, 5n\}$, we add to the graph cost $\text{cost}_G(x)$ the minimum cost introduced by adding the edge in an optimal way to the drawing of $G - e^\star$. With d denoting the sum of the pole degrees in G, $\deg(s^\star) + \deg(t^\star)$, the allowed number of directed bends for $e^\star$ is in the range $A_{e^\star}(x) := \{x - (10 - d), \ldots, x - 2\}$. This implies no flexibility on the number of bends of $e^\star$ if both poles have degree 4. The cost introduced by $e^\star$ is then

$$c_{e^\star}(x) = \min_{\alpha \in A_{e^\star}(x)} (\text{cost}_{e^\star} |\alpha|)$$

which results in an overall cost c for an optimal drawing of G of

$$c := \min_{e^\star \in E^\star} \left(\min_{x \in \{-5n, \ldots, 5n\}} (\text{cost}_{G-e^\star}(x) + c_{e^\star}(x)) \right) .$$

Corollary 3.11. *The cost c of an optimal drawing of* OptimalFlexDraw *for biconnected series-parallel graphs can be found in time $\mathcal{O}(n^4)$.*

Proof. For a particular $e^\star \in E^\star$, the cost function $\text{cost}_{G-e^\star}$ can be calculated in $\mathcal{O}(n^3)$ time by Theorem 3.10. We can add the cost of $e^\star$ in constant time. Once calculated, we find the rotation of $\pi(s^\star, t^\star)$ in an optimal representation of G with $e^\star$ on the external face by finding the minimum of $\text{cost}_{G-e^\star}$ in linear time. To find the optimal drawing of G with any edge $e^\star$ on the external face by iterating over $E^\star$ and finding the minimum cost, which introduces an additional linear factor, resulting in a total running time of $\mathcal{O}(n^4)$ □

So far, we have only found *the cost* of an optimal representation of G. However, we still have to compute the representation itself. Recall that Tamassia [Tam87] introduces an algorithm computing a complete drawing once we have an orthogonal representation. Tamassia also introduces an algorithm computing an orthogonal representation minimizing the number of bends from a fixed combinatorial embedding, however, in our case we can not make use of this approach, since non-convex edge cost functions can not be expressed in an analogous flow network.

Recall that when computing the graph cost function of the pertinent graph $\text{pert}(\mu)$ of a node μ in $\mathcal{T}$ with the poles s and t for a particular rotation $\pi(s, t)$, we find the embedding and angles at the nodes of $\text{skel}(\mu)$ leading to the minimum cost induced by the child nodes

of μ. In the same step we choose a particular rotation for each child node μ_i. For every calculated value at $x \in \{-5n, \dots, 5n\}$ of the cost function of $\text{pert}(\mu)$, we store this choices at μ in the SPQR-tree. A final step finding the orthogonal representation is now easy to add to the algorithm.

For this, we root $\mathcal{T}$ at $\mu_{e^\star}$ for which we found the minimum cost c. We again traverse the SPQR-tree recursively, starting with the node $\mu'_{e^\star}$ adjacent to the Q-node $\mu_{e^\star}$. We pass the optimal rotation during traversation as stored before, starting with the optimal rotation x for $G - e^\star$ as found when c was computed. Each node μ in $\mathcal{T}$ now decides based on the stored information what the optimal embedding is for this rotation as well as the rotation for the child nodes, which it passes down the tree.

For each Q-node, we use this rotation as the directed bends in the orthogonal representation. For each S-node with the poles s and t, we only have to determine the angle at each cut vertices on the side of $\pi(s,t)$; the angle on the other side can be calculated at the very end since all angles of a node sum up to 4. For each P-node with k children and the poles s and t, we apply the best combinatorial embedding of the skeleton to G as well as denote the angles of s and t in the $k-1$ faces between each subgraph in the orthogonal representation as stored in the SPQR-tree. Finally, the angles in the face between the edge $e^\star$ chosen as the root of $\mathcal{T}$ and the rest of the graph, $G - e^\star$, are determined by the value of α we have chosen when finding $c_{e^\star}$ as well as the number of directed bends on that edge. Note that it does not matter for which of the two poles $s^\star$ and $t^\star$ we assign how many right angles, as long as they lead to a valid representation.

This leads to the following final theorem for solving Problem 6.

Theorem 3.12. *An optimal drawing to an instance of the problem* OptimalFlexDraw *for biconnected series-parallel graphs can be computed in time $\mathcal{O}(n^4)$ with a storage of polynomial size.*

Proof. The cost of an optimal drawing can be computed in time $\mathcal{O}(n^4)$. We extended this algorithm with storing information at every node in the SPQR-tree. This information has a maximum size of $\mathcal{O}(n)$ for each rotation value and tree node, leading to a total storage size of $\mathcal{O}(n^3)$. Generating and storing this information does not introduce an additional factor in the time complexity, as we needed them anyways to compute the minima.

We again traversed the SPQR-tree top-down and calculated a part of the optimal representation based on the information stored at each tree-node by looking up this information for a particular rotation of the pertinent graph represented by that node only. □

However, if we force a particular vertex to be incident to the external face, we can get better results which we use when combining blocks of a non-biconnected graph in the next section.

Lemma 3.13. *An optimal drawing to an instance of the problem* OptimalFlexDraw *for biconnected series-parallel graphs can be computed in time $\mathcal{O}(n^3)$ with a storage of polynomial size, if a particular vertex is forced to be incident to the external face.*

Proof. Similar to Theorem 3.12, we compute the orthogonal drawing by storing additional info and accumulating them afterwards. However, when we force a particular node v to be incident to the external face, we only have to root the SPQR-tree at all edges $e^\star \in E^\star$ incident to v. Since this set is limited to four edges, we use 3.10 only a constant number of times and thus save the additional factor of n when compared to Theorem 3.12. □

3.2. Connected Series-Parallel Graphs

In the previous section we focused on biconnected series-parallel graphs. Now we want to generalize our algorithm to solve the original problem OptimalFlexDraw for general series-parallel graphs.

A series-parallel graph G is obviously always connected. We compute its block-cutvertex-tree $\mathcal{B}$ as introduced in Section 2.3 which decomposes G into blocks and cut vertices connecting these. Note that similar to the SPQR-tree, $\mathcal{B}$ can be rooted at any node.

If $\mathcal{B}$ has only one block $B = G$, we can simply use the algorithm from the previous section to compute an optimal drawing of B and are done. Otherwise, we have to merge orthogonal representations, which we explain in the following. Recall that a cut vertex v cuts G into two or more cut components $C_1, \dots, C_k$ represented by the k subtrees rooted at the blocks $B_1, \dots, B_k$ incident to v in $\mathcal{B}$.

Lemma 3.14. *Let G be a 4-planar, series-parallel graph. Let v be a cut vertex of G cutting G into k cut components $C_1, \dots, C_k$ of G and $\mathcal{R}_1, \dots, \mathcal{R}_k$ orthogonal representations of these cut components. We can merge $\mathcal{R}_1, \dots, \mathcal{R}_k$ at their common cut vertex v in G into one orthogonal representation if and only if the following holds.*

(1) At most one of the representations $\mathcal{R}_1, \dots, \mathcal{R}_k$ has v not embedded incident to the external face.

(2) If $k = 2$ and v has degree 2 in both cut components, v has to have only one right angle at an incident internal face in the orthogonal representations. (In this case, both have three right angles in one face incident to v. In combination with (1), this face has to be external for one of them.)

Proof. It's easy to see that if (1) does not hold, the representations can not be merged, as more than one of them require all others to be drawn inside one of their internal faces. As illustrated in Figure 3.2, if (2) does not hold, the angles at the cut vertex within the two representations forbid merging those into one representation.

For the other direction, we show that if both constraints hold, we can actually construct an orthogonal drawing. Without loss of generality, let C_1 be the cut component having the cut vertex v on one of its internal faces in its orthogonal representation $\mathcal{R}_1$. We now show how to draw all other cut components within faces incident to v in $\mathcal{R}_1$. For this, we make a case distinction over the degree of v within $\mathcal{R}_1$.

(Case 1) v has degree 1 in $\mathcal{R}_1$. Let f_0 be the face incident to v in $\mathcal{R}_1$. If $k = 4$, we can draw all other cut components in any order at v within f_0. If $k = 2$ we can draw $\mathcal{R}_2$ inside of f_0. If $k = 3$ and $\deg(v)$ is 4 within G, one of the other cut components has to have degree 2 at v; without loss of generality let this be $\mathcal{R}_2$. If v is tight within $\mathcal{R}_2$, we can draw both $\mathcal{R}_2$ and $\mathcal{R}_3$ inside of f_0. If it is not, we draw $\mathcal{R}_2$ inside of f_0 and $\mathcal{R}_3$ inside of the face incident to v in $\mathcal{R}_2$ not being the external face.

(Case 2) v has degree 2 in $\mathcal{R}_1$. If $k = 2$ and $\deg(v)$ is 4, there has to be a face f_0 in $\mathcal{R}_1$ at which v has three right angles since (2) holds. Thus, we can draw $\mathcal{R}_2$ inside of f_0, since v is tight within $\mathcal{R}_2$. If $k = 2$ and $\deg(v)$ is 3, we can draw $\mathcal{R}_2$ within the face incident to v in $\mathcal{R}_1$ with the greater angle (within any of the two faces if the angles are the same). If $k = 3$, v has degree 1 in both other cut components, so they can easily be drawn without the faces incident to v in $\mathcal{R}_1$.

(Case 3) v has degree 3 in $\mathcal{R}_1$. As there is exactly one face f_0 in $\mathcal{R}_1$ where v has two right angles, we can draw $\mathcal{R}_2$ within f_0. □

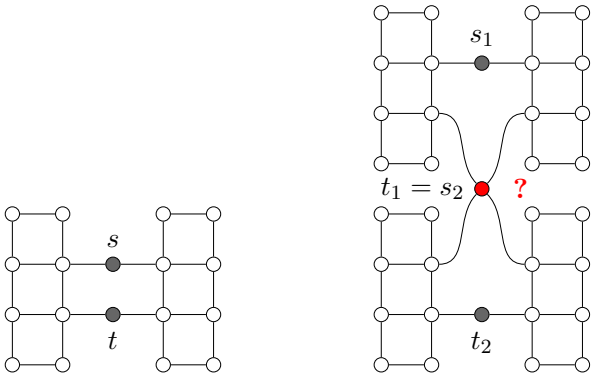


Figure 3.2.: Two blocks with orthogonal representations as in (a) do not allow merging them at the cut vertex $v = s_2 = t_1$. The embedding in (b) requires the orthogonal representations of both blocks to have three right angles at the cut vertex at the external face.

If a cut vertex v is incident to exactly two blocks and has degree 2 within both, we have to restrict the representations to those having an angle of 1 at v on its internal incident face in order to not violate (2). We put this as an additional constraint when computing the cost these two blocks.

At least all but one blocks incident to a cut vertex have to embed v on the external face in the orthogonal representation. We have to decide which of them *does not* have to, since this affects the overall costs of the drawing. The cost and the drawing of a block B having v embedded on the external face can be done by setting $E^\star$ in the algorithm from the previous section to the edges incident to v within B. For the block in which we do not force v to be embedded on the external face, we simply set $E^\star$ to all edges in the block.

Choosing a root node in $\mathcal{B}$ defines a reference block B_0 for each cut vertex v in $\mathcal{B}$ in which we do not force v to be drawn on the external face. This results in different total costs for the drawing of G when rooting $\mathcal{B}$ at different B-nodes. We want to calculate this cost for every B-node and choose the block B_{opt} with the minimum cost.

Theorem 3.15. *An optimal drawing to an instance of the problem* OptimalFlexDraw *for series-parallel graphs can be computed in time $\mathcal{O}(n^4)$ with a storage of polynomial size.*

Proof. The block-cutvertex-tree $\mathcal{B}$ of a series-parallel graph G of size n has $b \in \mathcal{O}(n)$ blocks. The costs and drawings of a block B can be computed in $\mathcal{O}(|B|^4)$ time by Theorem 3.12. The drawings for all blocks can be computed in $\sum_{B \text{ of } G} \mathcal{O}(|B|^4) = \mathcal{O}(n^4)$ time, since their *total* size is in $\mathcal{O}(n)$.

For every block B of G, we calculate the cost and optimal drawing without forcing any vertex on the external face in $\mathcal{O}(n^4)$ and denote this cost at the nodes in $\mathcal{B}$. In all drawings of G, there is only one block $B^\star$ for which we do not have to force any incident cut vertex to be embedded on the external face, since we can draw all cut components inside of its drawing. This can be done by rooting $\mathcal{B}$ at $B^\star$ and forcing for every other block B in $\mathcal{B}$ the parent cut vertex on the external face. We can calculate a drawing of all blocks in the tree in time $\mathcal{O}(n^3)$ by Lemma 3.13.

However, the overall cost of the drawing depends on the chosen tree root $B^\star$. Thus, we still have to find the root node leading to the minimum costs. For this, we root at all blocks once, calculate the costs and drawings for all other blocks in time $\mathcal{O}(n^3)$ for each root node and find the minimum in a total running-time of $\mathcal{O}(n^4)$. □

4. An Algorithm Solving FlexDraw

In this chapter we describe the algorithm introduced by Bläsius et al. [BRW12] solving an instance of the decision problem FlexDraw with positive flexibility in $\mathcal{O}(n \cdot T_{\text{flow}}(n))$ worst-case time, where $T_{\text{flow}}(n)$ denotes the time complexity to calculate a min-cost max-flow instance of size n. As seen before, the most challenging part is to find a feasible embedding of an input graph G out of the enormous number of possible embeddings. For this, we will use the following dynamic program on an SPQR-tree. The "best" embedding of a subgraph pert(μ) of G represented by any tree node μ is calculated by choosing the best embeddings of the subgraphs represented by its child nodes $\mu_1, \ldots, \mu_k$ in the SPQR-tree. We'll see that we also have to find which Q-node of the SPQR-tree has to be the root node where the program starts. In the worst case we have to try to embed G at most $m = |E|$ times (the number of Q-nodes), since the root will always be embedded on the external face and thus one single root node will not represent all embeddings of G.

First, we split the graph in its connected components and these into blocks connected via cut vertices using a BC-tree as seen in Section 2.3. Thus we only have to consider biconnected graphs and merge them afterwards, which puts some more constraints on the embeddings of the blocks. Consider the BC-tree of a connected component. In order to merge the blocks in the resulting drawing, all but the root block have to have the cut vertex corresponding to the parent node in the BC-tree embedded on the outer face. If there is exactly one block for which we can't hold this constraint this block has to be the root of the BC-tree. If all blocks can hold this constraint, we can choose an arbitrary root. If more than one block can't hold this constraint, it is not possible to merge the drawings and thus the whole algorithm can be canceled since G does not admit a valid representation.

In the following we only consider the sub-algorithm taking a *biconnected graph* and optionally a vertex to be drawn incident to the outer face as its input. The algorithm either generates a drawing satisfying the flexibility constraints or decides that such a drawing does not exist.

For this, the idea is to construct and traverse the SPQR-tree of the input graph G in a bottom-up manner and process only the skeleton graph of such a node in the SPQR-tree. We write a dynamic program in order to constructively calculate some characteristics of G by using characteristics of subgraphs of G and putting them together using the skeletons of the SPQR-tree. This makes it possible to efficiently solve the major difficulty of FlexDraw, namely the enormous number of possible embeddings that need to be

considered when trying to find an embedding with which G can be flex-drawn. Bläsius et al. found out that we only need to look at all possible embeddings of a particular skeleton in the SPQR tree and pass some piece of information to the parent node in order to virtually test the flexibility constraints against all possible embeddings.

We now look at the algorithm from a practical point of view. In order to implement FlexDraw, we first need to define what kind of information is required for each child node in order to process a node in the SPQR tree. This includes

- A binary value feasible(μ) $\in \{0, 1\}$ telling if μ admits a valid representation.
- The pole degrees $\deg(s)$ and $\deg(t)$ in the pertinent graph of μ.
- The maximum rotation maxrot(pert(μ)) of the pertinent graph of μ.

We now describe the main procedure of FlexDraw, namely `processSkelEdge`, which takes a skeleton and any virtual edge within this skeleton, representing an S-, P- or R-node μ in $\mathcal{T}$. It calculates the characteristic properties from above of the pertinent graph pert(μ).

When processing a node μ with k children we first call the procedure `processSkelEdge` recursively on the skeleton edges representing its child nodes $\mu_1, \dots, \mu_k$. If at least one of the child nodes does not admit a valid representation, i.e. feasible $= 0$, μ also does not and we return feasible $= 0$ immediately.

The pole degrees $\deg(s)$ and $\deg(t)$ in the pertinent graph are simply the sum of the pole degrees of the subgraphs whose skeleton edge in skel(μ) is incident to s or t respectively.

In order to find the maximum rotation maxrot(pert(μ)) of the pertinent graph of μ we have to make a case distinction based on the type of μ:

- μ is a **Q-node**:
$$\text{maxrot}(\text{pert}(\mu)) = \text{flex}(\text{original}(\mu)),$$
where original(μ) gives us the original edge e in G corresponding to the Q-node μ in the SPQR-tree.

- μ is an **R- or P-node**:
$$\text{maxrot}(\text{pert}(\mu)) = \max_{\mathcal{E} \in \text{emb}(\mu)} \text{maxrot}_{\mathcal{E}}(\mu),$$
$$\text{maxrot}_{\mathcal{E}}(\mu) = \text{flow}(\text{network}_{\mathcal{E}}(\mu)),$$
where flow(network$_{\mathcal{E}}(\mu)$) is the maximum possible flow through the gadget graph based on the combinatorial embedding $\mathcal{E}$ as introduced in Section 2.4. We calculate this for all possible embeddings $\mathcal{E} \in$ emb(μ) of the skeleton. Recall that an R-node only allows two embeddings while a P-node with k children allows $k!$ embeddings. Since k is either two or three, it has at most six possible embeddings, so this step requires a constantly limited number of max-flow instances to be calculated.

  We embed the skeleton graph according to the combinatorial embedding resulting in the highest maximum rotation value possible and return this as the maximum rotation of the pertinent graph.

- μ is an **S-node**:
$$\text{maxrot}(\text{pert}(\mu)) = \sum_{i=1}^{k} \text{maxrot}(\mu_i) + k - 1$$
We simply sum up the maxrot values of the child nodes and add one rectangular angle per cut vertex between the subgraphs. There are $k - 1$ cut vertices in the pertinent graph of an S-node with k children.

5. Implementation of FlexDraw

In this section, we provide an implementation of the algorithm solving FlexDraw from above. First, we will discuss the algorithm in a more technical way and provide some pseudo-code. Within this work, we implemented FlexDraw in C++ using OGDF for which we'll see an overview, some more technical details and finally an evaluation at the end of this section.

The pseudo-codes for the procedures referenced in this section are found in the appendix.

The algorithm can be divided into two steps. In the first step, we find an embedding providing the maximum graph rotation. This step takes the graph and the flexibility function as its input and applies the best combinatorial embedding on the input graph. If no feasible embedding could be found, the algorithm terminates and returns "infeasible".

In the second step, we minimize the bends in the drawing of the embedding found in the first step. For this we calculate an orthogonal representation using a flow network similar to Tamassia's [Tam87], while still restricting the number of bends per edge as defined by the given flexibility function. Based on this orthogonal representation the rest of the orthogonal grid layout algorithm in [Tam87] computes the final drawing.

When implementing FlexDraw within this work, care was taken to follow the object-oriented code style and naming convention of OGDF. Analogous to the two major steps as described above, we implemented the following two main classes and a helper class which can be found in the UML class diagram as seen in Figure 5.1:

- Class `EmbedderFlexDraw` (extending `ogdf::EmbedderModule`):
  Finds and applies the optimal embedding as described in Chapter 4.
- Class `FlexDrawLayout` (extending `ogdf::GridLayoutPlanRepModule`):
  Calls `EmbedderFlexDraw`, then calculates the final layout based on this embedding.
- Class `SubGraphInfo`:
  Characterizes a pertinent graph pert(μ) of a node μ in the SPQR tree.

OGDF provides some data structures for graph algorithms we needed to implement FlexDraw. The class `ogdf::StaticPlanarSPQRTree` implements a linear-time algorithm calculating the SPQR-tree of any biconnected graph. The data structure only uses S-, P- and R-nodes to represent the tree, while in the skeletons of the nodes non-virtual edges implicitly represent Q-nodes. Virtual edges always correspond to another tree node. In this thesis, the pseudo-code expects such an implementation of the SPQR-tree.

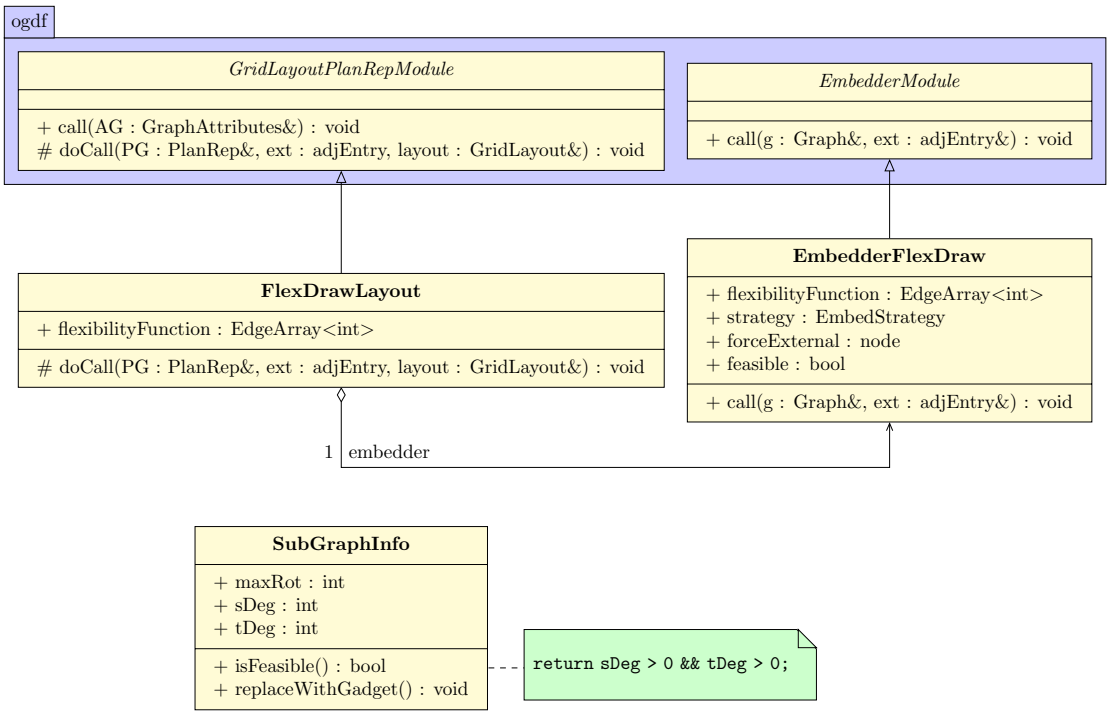


Figure 5.1.: UML class diagram of the FlexDraw implementation

5.1. The Class `SubGraphInfo`

The major concept of the algorithm FlexDraw is the dynamic program which first processes smaller subgraphs of G, collects pieces of information about them and combines these to a description of a greater subgraph. These pieces of information are collected and passed around using an instance of the class `SubGraphInfo` in order to realize the bottom-up traversation in the SPQR-tree.

In order to process a graph μ in $\mathcal{T}$ having k child nodes, we have seen in the previous chapter that we need to know the feasibility, the maximum rotation and pole degrees of every subgraph represented by the child nodes $\mu_1, \ldots, \mu_k$. Exactly this information is stored within this class. Whenever a node has been processed and the information has been stored in an instance of this class, it is returned to the caller which can then process the parent node by combining the collected information. If a node μ happens to be infeasible, i.e. it does not admit a flex-drawing, we know that the whole graph G can not be flex-drawn. In this case we want to terminate the whole algorithm. We stored this boolean value implicitly by just setting the pole degrees both to zero, as the pole degrees are at least one for valid subgraphs and we do not need the real pole degrees in the case a subgraph is infeasible.

Note that we do not have to explicitly store the feasibility, since if a subgraph is infeasible, the maximum rotation is undefined and we also do not need the pole degrees since the algorithm terminates. We indicate the infeasibility by setting both pole degrees to 0.

5.2. The Class `EmbedderFlexDraw`

Given a 4-planar graph $G = (V, E)$ and a positive flexibility function $\text{flex} : E \to \mathbb{N}\backslash\{0\}$, the class finds and applies a combinatorial embedding with that G can be flex-drawn. In this class, we implemented the major part of the algorithm introduced in the previous chapter, namely the dynamic program with which we traverse the SPQR-tree in a bottom-up manner in order to calculate an optimal embedding of a graph by first calculating the

optimal embeddings of its subgraphs. It is called by the class `FlexDrawLayout` which uses this embedding to generate a flex-drawing.

The class `EmbedderFlexDraw` inherits from the abstract class `ogdf::EmbedderModule` requiring to implement the pure virtual function `void call(Graph &g, adjEntry &adjExternal)`. Where the abstract class in OGDF does not return anything (since it is intended to always succeed), we want to return a boolean value whether or not the problem instance is feasible; this can be queried after calling `call` using the method `bool feasible() const`.

Let $E^\star$ denote the edges being candidates to be put on the outer face. We set $E^\star = E$ if we do not force any vertex to be embedded on the outer face and $E^\star = \{e \in E : v_{\text{ext}} \text{ incident to } e\}$ if v_{ext} is forced to be on the outer face. As in the original paper [BKRW12] we want to root the SPQR-tree $\mathcal{T}$ of G at every edge $e \in E^\star$ in order to find a feasible embedding with *any* of them on the external face.

While, by the problem definition, it is sufficient to find only one such edge (since then we know that G admits a valid drawing and we can flex-draw it with this edge being on the external face), we may obtain better results if we choose the edge with the maximum graph rotation. For this, `EmbedderFlexDraw` is configurable to choose either the first candidate found or the one providing the maximum graph rotation. This is done via the setter `void embedStrategy(EmbedStrategy strategy)`, passing either `EmbedFirstFeasibleExternalEdge` or `EmbedHighestGraphFlexibility`.

The implementation of the class `EmbedderFlexDraw` itself is split into two parts: In the first step we iterate over all candidate edges $e \in E^\star$ and test if G admits a valid representation with e on the outer face. After the best (or first, depending on the strategy) candidate was found, the embedding implied by e is applied on G in a second step. If no candidate satisfies the problem instance, the input graph remains unchanged. Algorithm 7.1 shows the pseudo-code which puts these steps together.

Testing if $e \in E^\star$ admits a flex-drawing: We root the SPQR-tree at the Q-node μ_e. Recall that e now is represented by the reference edge $\text{ref}(\mu'_e)$ in $\text{skel}(\mu'_e)$ and that $\text{ref}(\mu'_e) = \{s, t\}$ where s, t are the poles of the graph $\text{skel}(\mu'_e) - \text{ref}(\mu'_e)$.

We define the function `processSkelEdge` that calculates the `SubGraphInfo` of a given edge e of a skeleton $S := skel(\mu)$ in the SPQR-tree $\mathcal{T}$ as in Algorithm 7.3. Note that in SPQR-tree implementations with implicit Q-nodes (as the one in OGDF), μ might be implicit; thus we provide a skeleton edge e rather than μ. This method either directly returns a `SubGraphInfo` describing the Q-node (an edge in the original graph G) or calls a method that processes the node μ and returns its result. We implement separate routines to handle S-, P- and R-nodes in the Algorithms 7.5 to 7.7. Note that these methods recursively call `processSkelEdge` on all but the reference edge in the skeleton of μ. If any of the subgraphs happens to be infeasible, we also return "infeasible" immediately in order to terminate the whole algorithm with no further processing.

In the algorithms processing tree nodes, the pole degrees $\deg(s)$ and $\deg(t)$ in the pertinent graph can be calculated easily by summing up the corresponding pole degrees of the child nodes whose representative skeleton edge is incident to the pole in the skeleton. Note that in the SPQR-tree the skeleton edges of μ might be stored in arbitrary directions, regardless of the node type. Thus, s in μ might be identified with s or t in μ_i $(i = 1 \dots k)$. If μ is a Q-node, it obviously has degree 1 at both s and t since it represents a single edge in G.

The maximum rotation of a pertinent graph $\text{pert}(\mu)$ is calculated depending on the type of μ. As seen in the previous section, in the case of P- and R-nodes we use a flow network for every possible embedding (two or six exist) and return the maximum flow we found.

The procedures to handle P- and R-nodes are almost the same. However, for R-nodes we only have to construct one single flow network and testing the flow in both directions to simulate both possible embeddings. For P-nodes we have to construct one or three flow networks (if it has two or three child nodes respectively) and use the same technique as for R-nodes to calculate the flow in both directions. We immediately change the embedding of the skeleton graph such that its embedding allows this rotation. Since S-nodes only have one possible embedding, we don't have to change it in its procedure. We also don't need a flow network since the maximum rotation can directly be expressed with the maximum rotations of its subgraphs.

In order to calculate the `SubGraphInfo` of $G - e$ we call `processSkelEdge` on the root node μ'_e. We then calculate $\text{graphrot}_e(G) = \text{maxrot}(G - e) + \text{flex}(e) - 2$. The embedding is feasible if and only if $\text{graphrot}(G) \geq 0$.

Applying the embedding with $e \in E^\star$ on the external face: In the previous step, we only found out which edge e admits a valid flex-drawing. We have also embedded the skeletons in a way that, with respect to μ_e being the root of $\mathcal{T}$, they admit a maximum rotation on the *left* path $\pi(s, t)$. For this, we flipped the skeletons whenever the right path admitted better results. Note that we have to apply this embedding again as, in general, e has not been the last root edge at which we rooted $\mathcal{T}$. Thus we again root $\mathcal{T}$ at μ_e and re-run `processSubGraph` again.

Furthermore, it may be the case that in order to obtain the maximum rotation possible for a node μ, the skeleton edge e_{μ_i} for a child node μ_i has to be rotated in the other direction. Recall that the value of $\text{maxrot}(\mu_i)$ stands for the maximum rotation on the *left* path, where positive values mean that concave bends dominate on the external face. Whenever the algorithm processing μ (the parent of μ_i) decides that it would result in a better maximum rotation for itself if it bends the subgraph represented by the edge e_{μ_i} to the right instead to the left, we have to flip *the entire pertinent graph* represented by μ_i.

However, the SPQR-tree implementation provided by OGDF (as will probably most other implementations) only allows us to flip the embedding of a *skeleton graph* $\text{skel}(\mu)$ in constant time. In order to flip the embedding of the pertinent graph $\text{pert}(\mu)$ we are required to flip the embeddings of all nodes in the subtree below μ, which results in a linear time complexity handling only a single node in $\mathcal{T}$.

We found a better way to do this, only requiring linear time for the whole tree $\mathcal{T}$. For this, we introduce a boolean value for every node μ in $\mathcal{T}$ which we call the `flipSubGraph` mark. Its name has been derived from the fact that the procedure handling μ decides whether or not the subgraphs of G represented by its child nodes $\mu_1, \ldots, \mu_k$ have to be flipped in order to obtain the maximum rotation for μ which it returns to its caller in the `SubGraphInfo` instance. For all nodes μ in $\mathcal{T}$ we now want to flip the pertinent graph $\text{pert}(\mu)$ if $\texttt{flipSubGraph}[\mu] = \text{true}$. This can easily done by flipping the skeleton $\text{skel}(\mu)$ if the number of marked nodes on the path up to the root (including μ) with is even. This is because flipping a pertinent graph is equivalent to flipping all skeletons on the subtree and flipping a skeleton twice is equivalent to not flipping it at all.

Now let's see how this is done in the algorithm. In the first step as described above we do not need the array, thus we pass an empty set $\emptyset$ in line 9 of Algorithm 7.1 to `processVirtualNode`. In this second step the array is initialized to "false" for every node μ in $\mathcal{T}$. After invoking `processVirtualNode` on the root node μ_e again in line 18 of Algorithm 7.1, the subroutines have set the mark to "true" where the pertinent graphs have to be flipped. Finally, we can apply the embeddings by calling the subroutine `flipSkeletons` in line 20 providing this array.

5.3. The Class FlexDrawLayout

Given a 4-planar graph $G = (V, E)$ and a positive flexibility function flex : $E \to \mathbb{N}\backslash\{0\}$, this class finds a flex-drawing. For this, it first calls `EmbedderFlexDraw` to find and apply a combinatorial embedding on G that admits such a drawing.

Since we now have a fixed combinatorial embedding, we can apply Tamassia's classical approach by calculating the orthogonal representation using a flow network, but restricting the edge capacity to make sure that the resulting drawing has no more than flex(e) bends for any edge $e \in E$, as explained in Section 2.4.

Finally, we draw the graph on a grid by using the compaction step of Tamassia's grid layout algorithm [Tam87] which has already been implemented in OGDF.

5.4. Using the Implementation

Using the implementation developed in this work in a C++ application is very easy. The main class of the implementation is `FlexDrawLayout`. Given an instance of the class `GraphAttributes` describing the input graph and an instance of `EdgeArray<int>` describing the flexibility function, FlexDraw is invoked with the method `FlexDrawLayout::call`. The drawing is written into the `GraphAttributes` instance given by reference.

The source code in Listing 5.1 is enough to implement a standalone CLI application taking an input graph as the first argument and writing the results to the file that is given as the second argument. It uses the GML file format in which the flexibility constraints are encoded as edge labels (defaulting to one when missing).

```
1  #include <ogdf/basic/GraphAttributes.h>
2  #include "flexdrawlayout.h"                    // The FlexDrawLayout class
3  using namespace ogdf;
4
5  int main(int argc, char *argv[]) {
6      if(argc != 3)
7          return 1;
8      Graph G;
9      GraphAttributes GA(G, GraphAttributes::nodeGraphics |
10                           GraphAttributes::edgeGraphics |
11                           GraphAttributes::edgeLabel);
12     GA.readGML(G, argv[1]);                    // read input graph
13
14     EdgeArray<int> flex(G);
15     edge e;
16     forall_edges(e, G) {
17         String label = GA.labelEdge(e);        // edge labels denote flex(e)
18         flex[e] = (label.length() > 0) ? atoi(label.cstr()) : 1;
19     }
20     FlexDrawLayout flexdraw;
21     flexdraw.flexibilityFunction(flex);        // set flexibility function
22     flexdraw.call(GA);                         // run FlexDraw
23
24     if(flexdraw.feasible())
25         GA.writeGML(argv[2]);                  // write output graph
26     return 0;
27 }
```

Listing 5.1: Standalone application illustrating the usage of the `FlexDrawLayout` class

6. Experimental Evaluation of FlexDraw

In this chapter we will analyze some results and the efficiency of our implementation of FlexDraw. We will discuss how the input parameters (the graph size and the flexibility function) affect both the feasibility and the runtime. In Section 1.2 we have seen that all graphs except the tetrahedron are 2-embeddable. This introduces a couple of questions:

1. Will most graphs be 1-embeddable?
2. If we set the flexibility of only a couple of edges to two instead of one, can most graphs be flex-drawn? How many edges need a flexibility of two?
3. How fast is FlexDraw? Is the average runtime notably better than the theoretical worst-case time? Can we save time using the strategy selecting the first edge admitting a feasible drawing instead of testing all edges? Does the feasibility of the input affect the runtime?
4. Will FlexDraw produce satisfying results in practice or is there a demand for better algorithms like OptimalFlexDraw?

In order to evaluate these questions, we need a lot of graphs which we generated automatically. We implemented a simple graph generator which we briefly describe now, since the structure of the input graphs may be important for the results of the algorithm. Note that we only generated biconnected graphs, as this is the most interesting input.

To generate graphs with a size of at most n nodes, our graph generator starts with a triangulated planar graph $G = (V, E)$ with $|V| = n$, randomly generated by OGDF using the method `ogdf::planarTriconnectedGraph`[1]. Triangulated planar graphs are graphs with a maximum number of edges ($|E| = 3n - 6$). While these graphs are obviously not necessarily 4-planar, we successively remove edges until it is. This is achieved by iterating over all nodes $v \in V$ and removing incident edges randomly until $\deg(v) \leq 4$. Finally, we choose the biconnected component G_1 of G with the maximum number of nodes n_1. This results in a 4-planar graph G_1 of size $n_1 \leq n$ suitable for FlexDraw. The average node degree for a generated graph is about 3.26.

[1] We used this method in version 2012.07 of OGDF. The official documentation can be found at `http://www.ogdf.net/doc-ogdf/namespaceogdf.html#a181c070c885d02d1611cd83efd529611`

6.1. Experiments

We have done the following experiments for which we used the graph generator from above to create appropriate inputs:

(1) To measure the efficiency of our implementation of FlexDraw as well as test 1-embeddability depending on the graph size n, we generated each 500 graphs for every graph size $20 \leq n \leq 1000$ in steps of 10 with $\text{flex}(e) = 1$ for every edge e.

(2) To measure the efficiency of our implementation of FlexDraw as well as test 2-embeddability depending on the graph size n, we generated each 500 graphs for every graph size $20 \leq n \leq 1000$ in steps of 10 with $\text{flex}(e) = 2$ for every edge e. We expect that every instance is feasible.

(3) We want to know how the amount of edges with a flexibility of 2 instead of 1 affects the amount of feasible instances in average. For every probability $0 \leq p \leq 1$ in steps of 0.05 we generated 1000 graphs of arbitrary sizes $400 \leq n \leq 1000$ with $\text{flex}(e)$ randomly chosen from $\{1, 2\}$ with the probabilities $P(\text{flex}(e) = 1) = p - 1$, $P(\text{flex}(e) = 2) = p$. Note that for $p = 0$ and $p = 1$ we test for 1- and 2-embeddability respectively like above. We expect an increasing rate of feasible instances for an increasing p.

The experiments in which we measure the runtime have been evaluated on a machine with a Dual-Core AMD Opteron^{TM} Processor 2218 which has 128 KiB L1 and 1 MiB L2 cache per core and is clocked at 2600 MHz[2]. Our implementation of FlexDraw however only uses one thread. The machine has 16 GiB RAM and runs OpenSUSE Linux with the Linux kernel version `2.6.34.10`[3]. FlexDraw has been compiled and linked with `g++` in the version 4.5.0 using the optimization level `-O3`. We used OGDF in the version 2012.07. Running times have been measured with the POSIX function `getrusage` which returns the user CPU time used by the process in milliseconds granularity.

For the experiments, we implemented a simple standalone command line program. It takes an input graph in the GML file format, runs the algorithm and writes the drawing as a GML file. It also measures and prints the running time of the function `FlexDrawLayout::call` as well as some bend statistics (total number of bends in the drawing and the number of edges with the maximum allowed number of bends) if the instance was feasible.

To simplify debugging and verification of the results, some optional *debug output* can be enabled, describing the internal steps finding the optimal embedding as well as some information about the flow network to find the final bends on the edges. In addition to FlexDraw we also implemented a user interface which we equipped with a simple graph editor with very restricted functionalities and the option to run the FlexDraw standalone application on the edited graph.

To complete the evaluation environment, we also implemented a script invoking FlexDraw on a set of generated graph files and summarizing the results (timings and bends statistics) in a data file. Finally, the data files have been visualized using `gnuplot`.

[2]For details see the vendor's website at `http://products.amd.com/pages/OpteronCPUDetail.aspx?id=318`

[3]The FlexDraw implementation does not make use of any platform-dependent functionality and should run on every system OGDF is also running.

6.2. Results

We now present the results of the experiments we have made. In Figure 6.1 on the next page you see some diagrams which we generated from accumulated statistics.

In our first two experiments, we applied the FlexDraw algorithm on a set of generated graphs of up to 1000 nodes and measured the running time the implementation needed to both decide the feasibility as well as, if feasible, compute an orthogonal drawing for it. Reading and parsing the graph file as well as writing the results are not included in this running time as this is not needed when invoking a layout algorithm in real-world applications like interactive graph editors, resulting in experiments near realistic scenarios.

In experiment (1), we measure the runtimes for instances having a flexibility of 1 on every edge, which we also call *flex-1-instances*. Similarly, experiment (2) is about *flex-2-instances* which we all expect to be feasible. The two first plots in Figure 6.1 show the results.

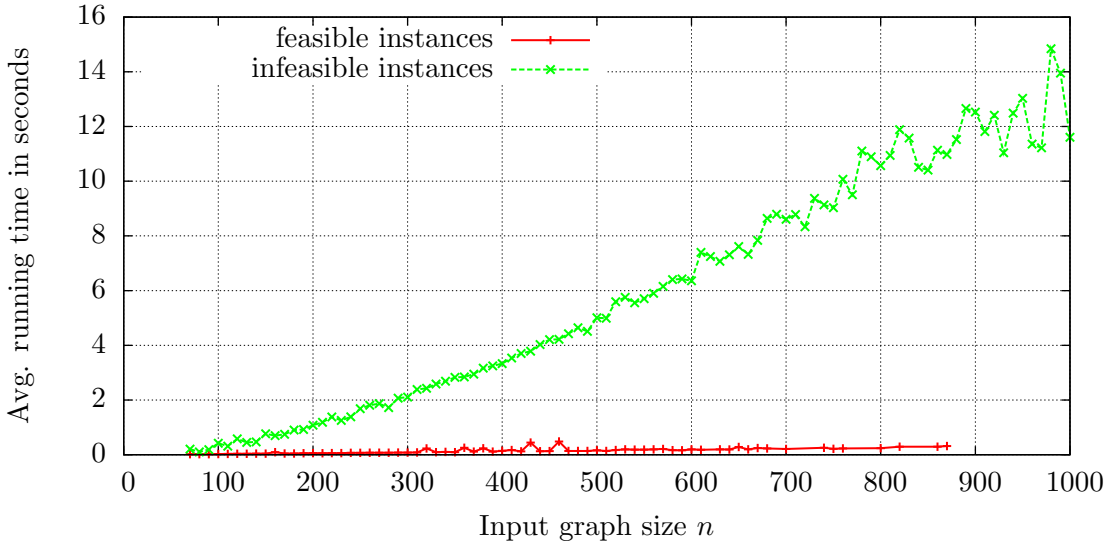
One can see, the running time of our implementation of FlexDraw in experiment (1) is in the range up to a couple of seconds and behaves roughly proportional to n^2 if the instance is infeasible. Since only a few instances have been feasible, the timing statistics for them is of no great significance. However, when we look at the small peaks for $n = 430$ and 460 we note an interesting fact about FlexDraw. Most feasible instances allow most of the edges being embedded on the external face. These peaks most probably are caused by instances for which this is not the case.

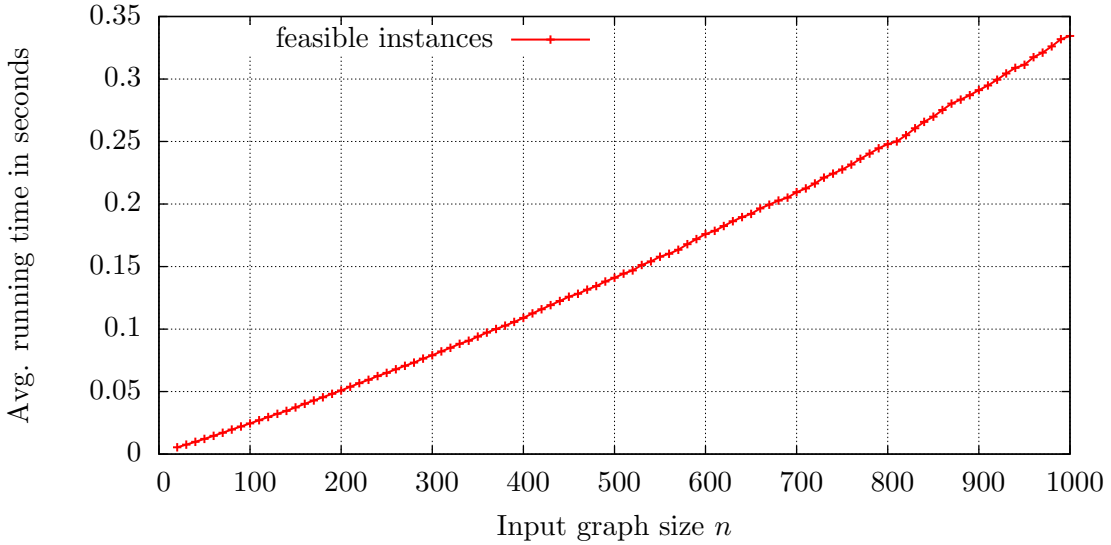
In the benchmark tests of experiment (2) we see a curve below quadratic complexity for the average running time of FlexDraw for flex-2-instances. This difference when compared to the previous experiment is easy to describe. For most feasible instances, not only one but a lot of edges where we root the SPQR-tree $\mathcal{T}$ at result in a feasible drawing, saving a factor of $|\mathcal{T}| \in \mathcal{O}(n)$. Note that, as predicted by theory, all instances in our second experiment have been feasible.

Experiment (3) evaluates the behavior of the feasibility rate when changing the average number of edges having a flexibility of 2, while the other edges have a flexibility of 1. The third plot in Figure 6.1 has the probability of any edge to have a flexibility of 2, that is $p := P(\text{flex}(e) = 2)$, on its axis to the right. We divided our tests in samples for p from 0 to 1 in steps of five percent. We plotted the number of feasible instances divided by the number of all instances for each sample as a point and connected them with a line. One can see a curve increasing slowly for small values of p, while increasing very fast around $p = 0.5$. For $p \geq 0.8$, almost all instances have been feasible. Again, all instances for $p = 1$, where we test for 2-embeddability, have been feasible.

Interpretating the results of our third experiment leads to interesting conclusions. While we know that every 4-planar graph is 2-embeddable, almost no graph has been 1-embeddable in our experiment, where nodes have an average degree of 3.26. As expected, an increasing number of edges with a flexibility of two leads to a higher rate of feasible instances. Among the sample graphs in which we uniformly choose a flexibility of 1 or 2 randomly ($p = 0.5$), about one third is still infeasible. When we want to draw a graph without having a flexibility function and still want to minimize the number of bends in a certain way, one idea would be to set the flexibility to 1 for every edge and to try to generate a drawing. Then, we could successively set the flexibility to 2 for a couple of edges until the instance becomes feasible.

To sum things up, the worst-case running time of FlexDraw of $\mathcal{O}(n^{5/2})$ in theory do not promise very fast results at a first glance. However, since the algorithm has to assume an unlikely worst-case scenario like a linear number of edges we have to root the SPQR-tree at in order to find a feasible drawing, in the practice we achieve much better results. A big disadvantage of FlexDraw is that we do not obtain *any* drawing for infeasible instances.





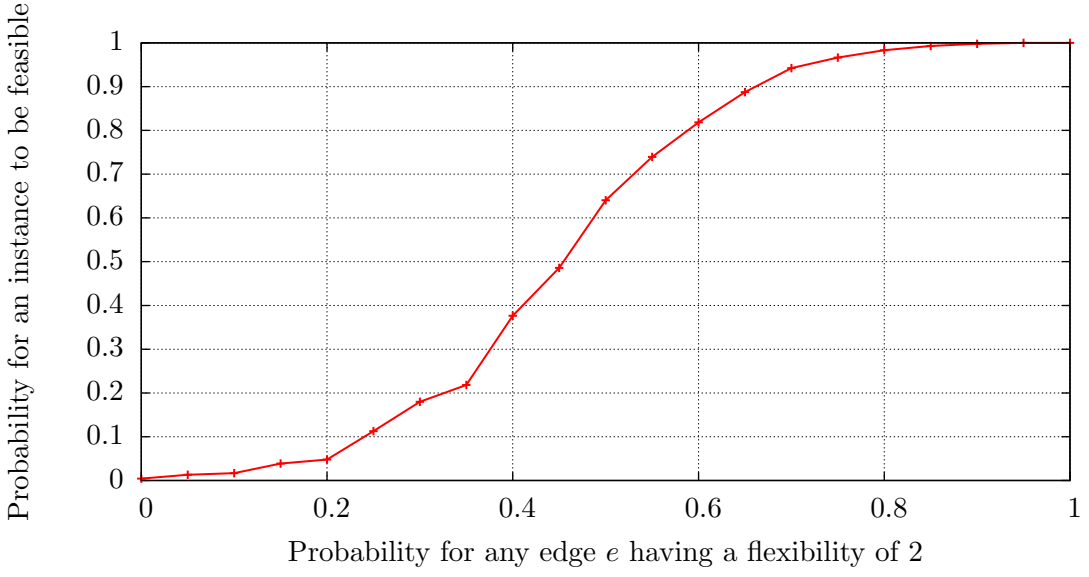


Figure 6.1.: Plots showing the average running time of FlexDraw as well as the rate of feasible instances with an increasing amount of edges having a flexibility of 2

7. Conclusion

In this work, we covered two algorithms for bend minimization in orthogonal drawings of 4-planar graphs.

We introduced an algorithm solving the optimization problem OptimalFlexDraw for series-parallel graphs of size n in time $\mathcal{O}(n^4)$ with the only restriction that the cost function for each edge has to be monotone. A demand for solving the problem with non-monotone cost functions will hardly arise. However, if we want to optimize bend costs in general graphs, we need another algorithm. Bläsius et al. [BRW12] introduce an algorithm taking convex cost functions for every edge individually, not allowing any costs for the first bend. While the convexity of the cost functions is not a very problematic restriction, the other forbids us to minimize the overall number of bends, which is proven to be NP-hard as such an algorithm could decide the *0-embeddability*.

The other algorithm we focused in this work, which solves the decision problem FlexDraw in polynomial time, has been implemented and evaluated in practice. We demonstrated briefly how such an implementation might work by providing some pseudo-code as well as descriptions of some of the most important procedures. In our evaluation we answered a couple questions about the efficiency of our implementation as well as the feasibility of instances of FlexDraw. To sum things up, we demonstrated that, if we assign a flexibility of two for every edge in an instance, we gain short running-times suitable for interactive applications like graph editors. However, we do not solve the bend minimization problem for arbitrary embeddings but only those allowing 2-embeddability.

We can imagine our implementation to be integrated in OGDF easily. However, in the future an implementation of the optimization problem as introduced by Bläsius et al. would promise even better results, as does an implementation of the algorithm for series-parallel graphs.

We did not answer some questions which are interesting for future works. Can FlexDraw with positive flexibility be modified such that we can force *some* of the edges to have no bends while still being solvable in polynomial time? For which other graph classes than series-parallel graphs is the 0-embeddability as well as bend minimization problem solvable in polynomial time?

Bibliography

[BK98] Therese Biedl and Goos Kant. A Better Heuristic for Orthogonal Graph Drawings. *Comput. Geom.*, 9(3):159–180, 1998.

[BKRW12] Thomas Bläsius, Marcus Krug, Ignaz Rutter, and Dorothea Wagner. Orthogonal graph drawing with flexibility constraints. *Algorithmica*, pages 1–27, 2012.

[BRW12] T. Bläsius, I. Rutter, and D. Wagner. Optimal Orthogonal Graph Drawing with Convex Bend Costs. *ArXiv e-prints*, April 2012.

[DBT96] Giuseppe Di Battista and Roberto Tamassia. On-Line Maintenance of Triconnected Components with SPQR-Trees. *Algorithmica*, 15:302–318, 1996.

[DLV98] G. Di Battista, G. Liotta, and F. Vargiu. Spirality and Optimal Orthogonal Drawings. *SIAM J. Comput.*, 27(6):1764–1811, 1998.

[GM01] Carsten Gutwenger and Petra Mutzel. A linear time implementation of SPQR-trees. In *Proceedings of the 8th International Symposium on Graph Drawing*, GD '00, pages 77–90, London, UK, UK, 2001. Springer-Verlag.

[GT95] Ashim Garg and Roberto Tamassia. On the Computational Complexity of Upward and Rectilinear Planarity Testing. In *DIAMCS International Workshop*, volume 894 of *Lecture Notes in Computer Science*, pages 286–297. Springer, January 1995.

[Pat99] Maurizio Patrignani. On the complexity of orthogonal compaction. *Computational Geometry: Theory and Applications*, 19:56–61, 1999.

[Tam87] Roberto Tamassia. On Embedding a Graph in the Grid with the Minimum Number of Bends. *SIAM Journal on Computing*, 16(3):421–444, 1987.

Appendix

A. Algorithms

Algorithm 7.1: The main algorithm of FlexDraw: Find and apply a combinatorial embedding of G, optionally with v_{ext} on the external face

input : *(by ref.)* $G = (V, E)$, flex : $V \to \mathbb{N}\backslash\{0\}$, *(optional)* $v_{\text{ext}} \in V$, strategy[1]
output: "feasible" if G admits a valid drawing, "infeasible" otherwise

```
1  if Optional parameter v_ext was given then
2      E* ← {e ∈ E : e incident to v_ext}
3  else
4      E* ← E
5  T = (V_T, E_T) ← SPQR-tree of G
6  for e ∈ E* do
7      Root T at μ_e
8      μ'_e ← (unique) node in T adjacent to μ_e
9      info ← processVirtualNode(G, T, μ'_e, ∅)
10     if info is feasible then
11         graphRot[e] ← info.maxRot + flex(e) − 2
12         if graphRot[e] ≥ 0 ∧ strategy is "first feasible external edge" then
13             break
14     else
15         graphRot[e] ← −∞
16 Choose e ∈ E* with maximum graphRot[e]
17 Initialize flipSubGraph[μ] ← false ∀μ ∈ T
18 processVirtualNode(G, T, μ'_e, flipSubGraph)
19 if graphRot[e] ≥ 0 then
20     flipSkeletons(T, μ'_e, flipSubGraph)
21     return "feasible"
22 else
23     return "infeasible"
```

[1] Either "first feasible external edge" or "highest graph rotation"

Algorithm 7.2: Helper routine `calculateDegrees`

input : subGraphInfo : Set of `SubGraphInfo`, s, t
output: sDeg, tDeg $\in \{1, \ldots, 4\}$

(sDeg, tDeg) $\leftarrow$ (0, 0)
for $i = 1 \ldots$ size(subGraphInfo) **do**
  if s = source node of subGraphInfo[i] **then**
    sDeg $\leftarrow$ sDeg + subGraphInfo.sDeg
  if s = target node of subGraphInfo[i] **then**
    sDeg $\leftarrow$ sDeg + subGraphInfo.tDeg
  if t = source node of subGraphInfo[i] **then**
    tDeg $\leftarrow$ tDeg + subGraphInfo.sDeg
  if t = target node of subGraphInfo[i] **then**
    tDeg $\leftarrow$ tDeg + subGraphInfo.tDeg
  return (sDeg, tDeg)

Algorithm 7.3: Routine `processSkelEdge` processing the edge e in a skeleton S

input : $G = (V, E)$, $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$, S, $e \in S$, (*by ref.*) flipSubGraphs : $V_\mathcal{T} \to \{0, 1\}$
output: An instance of `SubGraphInfo` describing $e \in S$

if e is virtual **then**
  $\mu_i \leftarrow$ the corresponding tree node in $\mathcal{T}$ of the skeleton edge e
  return `processVirtualNode`(G, $\mathcal{T}$, μ_i, flipSubGraphs)
else
  $e' \leftarrow$ the original edge in E corresponding to e in S
  $f \leftarrow$ flex(e')
  return `SubGraphInfo`(feasible $\leftarrow$ **true**, maxRot $\leftarrow f$, sDeg $\leftarrow$ 1, tDeg $\leftarrow$ 1)

Algorithm 7.4: Routine `processVirtualNode` processing an S-, P- or R-node in $\mathcal{T}$

input : $G = (V, E)$, $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$, $\mu \in V_\mathcal{T}$, (*by ref.*) flipSubGraphs : $V_\mathcal{T} \to \{0, 1\}$
output: An instance of `SubGraphInfo` describing pert(μ)

if μ is an S-node **then**
  return `processSNode`(G, $\mathcal{T}$, μ, flipSubGraphs);
if μ is an P-node **then**
  return `processPNode`(G, $\mathcal{T}$, μ, flipSubGraphs);
if μ is an R-node **then**
  return `processRNode`(G, $\mathcal{T}$, μ, flipSubGraphs);

Algorithm 7.5: Routine `processSNode` processing the skeleton of an S-node in $\mathcal{T}$

input : $G = (V, E)$, $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$, $\mu \in V_\mathcal{T}$, (*by ref.*) flipSubGraphs : $V_\mathcal{T} \to \{0, 1\}$
output: An instance of `SubGraphInfo` describing pert(μ)

```
1  info ← SubGraphInfo(feasible ← true, maxRot ← 0, sDeg ← 0, tDeg ← 0)
2  (s, t) ← the poles of skel(μ) = the nodes of ref(μ)
3  k ← number of child nodes of μ
4  for i = 1 ... k do
5      e ← edge in skel(μ) corresponding to μ_i
6      subGraphInfo[i] ← processSkelEdge(G, T, skel(μ), e, flipSubGraphs)
7      if subGraphInfo[i] is infeasible then
8          return "infeasible"
9  info.maxRot ← k − 1 + Σ_{i=1}^{k} subGraphInfo[i].maxRot
10 info.sDeg, info.tDeg ← calculateDegrees(subGraphInfo, s, t)
```

Algorithm 7.6: Routine `processPNode` processing the skeleton of a P-node in $\mathcal{T}$

input : $G = (V, E)$, $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$, $\mu \in V_\mathcal{T}$, (*by ref.*) flipSubGraphs : $V_\mathcal{T} \to \{0, 1\}$
output: An instance of `SubGraphInfo` describing pert(μ)

```
1  info ← SubGraphInfo(feasible ← true, maxRot ← 0, sDeg ← 0, tDeg ← 0)
2  (s, t) ← the poles of skel(μ) = the nodes of ref(μ)
3  for e ∈ skel(μ), e ≠ ref(μ) do
4      μ_i ← the corresponding SPQR-node of the skeleton edge e
5      subGraphInfo[i] ← processSkelEdge(G, T, skel(μ), e, flipSubGraphs)
6      if subGraphInfo[i] is infeasible then
7          return "infeasible"
8  info.sDeg, info.tDeg ← calculateDegrees(subGraphInfo, s, t)
9  for ε ∈ emb′(μ) do
10     G_gadget ← copy of skel_ε(μ)
11     for i = 1 ... k do
12         e_gadget ← edge in G_gadget corresponding to μ_i
13         subGraphInfo[i].replaceWithGadget(G_gadget, flex_gadget, e_gadget)
14     e′ ← edge in G_gadget corresponding to ref(μ)
15     (maxRot[ε], feasible[ε], flip[ε], flipChild[ε]) ← gadgetFlow(G_gadget, flex_gadget,
       e′)
16 ε_0 ← embedding in emb′(μ) having the maximal maxRot[ε]
17 Apply the embedding ε_0 on skel(μ)
18 if flip[ε_0] then
19     Flip the embedding of skel(μ)
20 for e ∈ skel(μ), e ≠ ref(μ) and e is virtual do
21     flipSubGraphs(e) ← flipChild[ε_0](e)
```

Algorithm 7.7: Routine `processRNode` processing the skeleton of an R-node in $\mathcal{T}$

input : $G = (V, E)$, $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$, $\mu \in V_\mathcal{T}$, (*by ref.*) flipSubGraphs : $V_\mathcal{T} \to \{0, 1\}$
output: An instance of `SubGraphInfo` describing pert(μ)

```
1  info ← SubGraphInfo(feasible ← true, maxRot ← 0, sDeg ← 0, tDeg ← 0)
2  (s, t) ← the poles of skel(μ) = the nodes of ref(μ)
3  G_gadget ← copy of skel(μ)
4  for e = (s_e, t_e) ∈ skel(μ) do
5      μ_i ← the corresponding SPQR-node of the skeleton edge e
6      e_gadget ← the corresponding gadget edge in G_gadget of the skeleton edge e
7      if e = ref(μ) then
8          flex_gadget(e) ← 0
9      else
10         subGraphInfo[i] ← processSkelEdge(G, T, skel(μ), e, flipSubGraphs)
11         if subGraphInfo[e] is infeasible then
12             return "infeasible"
13         subGraphInfo[i].replaceWithGadget(G_gadget, flex_gadget, e_gadget)
14 info.sDeg, info.tDeg ← calculateDegrees(subGraphInfo, s, t)
15 e' ← edge in G_gadget corresponding to ref(μ)
16 (info.maxRot, info.feasible, flip, flipChild) ← gadgetFlow(G_gadget, flex_gadget, e')
17 if flip then
18     Flip the skeleton embedding of μ in T
19 for e ∈ skel(μ), e ≠ ref(μ) and e is virtual do
20     flipSubGraphs(e) ← flipChild(e)
```

Algorithm 7.8: Routine `flipSkeletons` flipping the embedding of pert(μ)

input : $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$, $\mu \in V_\mathcal{T}$, flipSubgraphs : $V_\mathcal{T} \to \{0, 1\}$, flipThisSubgraph $\in \{0, 1\}$

```
1  if flipThisSubgraph = 1 then
2      Flip the embedding of skel(μ) in T
3  for e : skel(μ), e ≠ ref(μ) and e is virtual do
4      μ_i ← the corresponding SPQR-node of the skeleton edge e
5      flipChild ← flipSubgraphs[μ_i] xor flipThisSubgraph
6      flipSkeletons(T, μ_i, flipSubgraphs, flipChild)
```

Glossary

GML Graph Markup Language. A human-readable file format describing graphs (including their layout, style information and more) used in OGDF as one of the exchange formats. A lot of other programs can also handle this file format. Within this work, GML was used to store the input graphs as well as their drawings. Not to be confused with GraphML (sometimes also abbreviated "GML"), an XML based graph description format.

OGDF Open Graph Drawing Framework. An open source C++ graph library mainly for graph drawing, maintained by Carsten Gutwenger at the time of writing. Within this work, OGDF was used to implement FlexDraw. The official website can be found at `http://www.ogdf.net`.