TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

DIPLOMARBEIT

BEND–MINIMAL ORTHOGONAL DRAWING OF NON–PLANAR GRAPHS

Ausgeführt am
INSTITUT FÜR COMPUTERGRAPHIK UND ALGORITHMEN
der Technischen Universität Wien

Unter Anleitung von
UNIV.PROF. DR. PETRA MUTZEL
und
UNIV.ASS. DR. GUNNAR W. KLAU &
UNIV.ASS. DR. RENÉ WEISKIRCHER

durch
MARKUS CHIMANI
Nusswaldgasse 28/3
1190 Wien

März 2004 _____

Acknowledgment

Warum sagt man eigentlich dankbar?
Weil der Mensch gewöhnlich keinen Dank bar ausdrückt...
Moritz Gottlieb Saphir

The reader may forgive me for writing this in German:

Ich danke an dieser Stelle all jenen die mich bei der Erstellung dieser Arbeit auf die unterschiedlichsten Weisen unterstützt haben. Dies sind meine Eltern für die Ermöglichung des Studiums sowie ihre umfassende Unterstützung währenddessen, meine Freunde und Kollegen und besonders auch die Korrekturleser, die sich durch den englischen Beweisurwald kämpfen mussten.
Ich danke auch meinen beiden Betreuern René Weiskircher und Gunnar Klau, sowie Prof. Petra Mutzel, die mir ein angenehmes, kompetentes und hilfreiches Arbeitsumfeld boten. Ebenso danke ich Prof. Günther Raidl für seine sofortige Bereitschaft, Prof. Mutzel zu vertreten.

Short Abstract

This thesis belongs to the field of graph drawing research. It presents a new procedure for calculating the bend minimal shape of non–planar graphs with given topology.
This method is an extension of the Simple–Podevsnef drawing standard. Simple–Podevsnef is a simplification of the more complex Podevsnef – also known as Kandinsky – standard. Both models guarantee bend minimality for planar graphs with given topology. They generate orthogonal drawings with equal vertex size where multiple edges can be attached to a single side of a node. In contrast to Kandinsky, Simple–Podevsnef has certain restrictions on the split up of such bundles.
The algorithm presented in this thesis expands the drawing standard for non–planar graphs. It treats crossing points of edges in a special way, and enables them to share identical grid points where appropriate. Hence it allows crossings of whole bundles of edges instead of single edges only.
Furthermore, we show a sharp upper bound of the bend count for the heuristic use of Simple–Podevsnef for non–planar graphs; we also present an extension of the new method that is able to draw non–planar clustergraphs. Clustergraphs are an extension of graphs, where there exists a hierarchical structure of clusters, in which the nodes of the graph are organized.

Kurzfassung

Diese Diplomarbeit ist in den Forschungsbereich des Graphenzeichnens einzuordnen. Sie stellt ein neues Verfahren zur Berechnung der knickminimalen Form nicht–planarer Graphen mit gegebener Topologie vor.
Dieses ist eine Erweiterung des Simple–Podevsnef Zeichenstandards, welcher eine Vereinfachung des komplexeren Podevsnef – auch bekannt als Kandinsky – Standards ist. Beide garantieren Knickminimalität für planare Graphen mit gegebener Topologie. Sie erzeugen orthogonale Zeichnungen mit gleichgrossen Knoten bei denen mehrere Kanten gemeinsam an der selben Seite eines Knoten Platz finden. Im Gegensatz zu Kandinsky, enthält Simple–Podevsnef bestimmte Einschränkungen bei der Aufspaltung eines solchen Kantenbündels.
Der Algorithmus der in dieser Arbeit präsentiert wird, erweitert den Zeichenstandard auf nicht–planare Graphen. Es werden Kantenkreuzungspunkte besonders berücksichtigt, sodass sie gemeinsam auf idente Gitterpunkte abgebildet werden können. Dadurch werden Kreuzungen von kompletten Kantenbündeln möglich, anstatt wie bisher nur von einzelnen Kanten.
Diese Arbeit zeigt darüberhinaus eine scharfe obere Schranke für die Knickanzahl bei der heuristischen Anwendung von Simple–Podevsnef auf nicht–planare Graphen, sowie eine Erweiterung der neuen Lösungsmethode auf das Feld der nicht–planaren Clustergraphen. Clustergraphen sind eine Erweiterung von Graphen, in der die Knoten in einer Hierarchie aus Clustern organisiert sind.

Contents

Chapter 1

Introduction

Alles, was man für einen guten Krimi braucht, ist ein guter Anfang.
Georges Simenon

This chapter gives an overview on the field of graphs and graph drawing in Section 1.1. While Section 1.2 introduces the topic of the thesis, Section 1.3 explains the structure of the remainder of this work.

1.1 Graph Theory and Visualization

Graphs are the mathematical representations of relations between objects, thus making complex systems describable. It is often not only easier, but also shorter and more exact, to define relations as graphs instead of any other representation. By drawing such a graph, it is much simpler to realize certain connections and implications in many cases. Objects are called *nodes* in graph theory, and are normally drawn as – possibly labeled – circles, squares, or other simple geometric figures. A relation – called *edge* – between objects is normally represented by a line between them. These lines may be augmented with arrows and other symbols.
Graphs are used in a wide area of different subjects; not only in computer science but also in other sciences, scientific research, management, and many more domains.
Graphical representations of graphs are of special importance in areas like:

- *Software Development:*
  As projects grow, diagrams like class hierarchies, structure diagrams, flow diagrams and alike become more and more important
- *Database Design:*
  Entity–Relationship–diagrams are the de–facto standard of planning and visualizing databases
- *VLSI (Very Large Scale Integration) Design:*
  Chip designer use graphs and their visualizations as wiring schemes
- *Automation Engineering:*
  Engineers visualize the logic needed in controllers as graphs during the construction phase and for documenting purposes
- *Project Management:*
  Both the staff and the decision makers enjoy the benefits of diagrams visualizing tasks and time requirements.

Furthermore, graphs – as well as their visualization – occur inherently in finite state machines, neuronal nets, molecule visualization, chemical reaction diagrams, sociograms, genealogical trees, maps, location based services, and many more.

1.2 Context

Most currently available algorithms for graph drawing focus on drawing planar graphs – graphs that can be drawn without intersecting edges. One way to draw non–planar graphs is by *planarizing* them: after precomputing the necessary edge crossings, we insert additional nodes instead of these crossing points.
After this step, we draw the modified graph using planar drawing methods. These methods of course ignore that the newly inserted nodes are different from the original nodes (e.g. they need less space than the latter ones).
There exist various orthogonal grid drawing standards, but the perhaps best of them is *Podevsnef* [12]. Since this standard is quite complex and complicated, Bertolazzi, Di Battista, and Didimo [4] created a similar but simpler drawing standard called *Simple–Podevsnef*. Both compute bend minimal shapes for planar graphs only.
The algorithm presented in this thesis is an extension to Simple–Podevsnef that is able to draw non–planar graphs with the minimum number of bends possible (for a given topology).
Unlike other algorithms, it uses the knowledge of node types (original vs. dummy nodes) to generate drawings where nodes representing crossings can share the same grid point.

1.3 Structure

While Chapter 2 will point out the necessary mathematical fundamentals, the third chapter will focus on the topic of graph drawing and its related algorithms. Chapter 4 will give a detailed view on the Simple–Podevsnef algorithm, which will be used as a foundation in Chapter 5. There we will present an algorithm called *SPED*, which is the solution for the main topic. Additional thoughts and findings related to the subject are collected in Chapter 6; Chapter 7 sums up the results of the thesis and tries to give a short outlook on the future evolvement of the topic.
Appendix A discusses the approach for non–planar graphs by Fößmeier – the inventor of the Podevsnef network; Appendix B presents examples and statistics measuring the performance of SPED.

Chapter 2

Preliminaries

Unvorbereitetes Wegeilen bringt unglückliche Widerkehr
Wilhelm Meisters Wanderjahre,
J. W. v. Goethe

This chapter describes the basic fundamentals needed throughout the thesis. We describe the notion of graphs first, whereas Section 2.2 gives an introduction to min–cost–flow networks. These are used by various graph drawing algorithms that define a 1–1 mapping between shape–calculation problems and such networks (see Chapter 4 for an example). The third section describes the powerful concepts of linear programs and their integer extensions. They cannot only be used to simulate min–cost–flow networks, but even enable us to make extensions that would not be possible in the aforementioned network type. We have to use such constructs for SPED – the main algorithm presented in this thesis.

2.1 Graphs

The following definitions are based on Diestel [9].

Definition 2.1. Graph: *A tuple $G = (V, E)$ is considered a graph, if V is a set of* nodes (vertices) *and E a multiset of* edges (arcs). *An edge is a tuple of two nodes $v, w \in V$. There exist different types of graphs due to differences in the exact definitions of E and the type of edges.*

Let e be an edge as a tuple of the nodes v and w. Both nodes are *adjacent* to each other, and *incident* to e.
The number of edges incident to node v is called *degree* ($\deg(v)$). We define the *maximum degree* of a graph G simply as

$$\text{maxdeg}(G) = \max_{v \in V} \deg(v)$$

Definition 2.2. (Un)Directed Graph: *A graph is considered directed, if its edges are directed. A directed edge is defined as an* ordered *tuple of two nodes. If the tuples are* unordered *the graph is said to be undirected.*

The node v (w) is the *source (target)* of a directed edge $e = (v, w)$. Furthermore, we define the *out–degree* $\deg^-(v)$ *(in–degree* $\deg^+(v)$*)* as the number of edges that have v as their source (target).
A path P between two nodes u and v is a sequence of edges linking these two nodes: $P = \{(x_0, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n)\}$, with $u = x_0$, $v = x_n$, and $n \in \mathbb{N}$.

In the context of this thesis, some properties of graphs are of special interest:

Definition 2.3. Simple Graph: *A graph is considered simple if and only if E is a set, instead of a multiset: two nodes can only either be adjacent to each other by a single edge or not adjacent at all.*

Definition 2.4. Self–loop–free Graph: *A graph is self–loop–free if the following constraint holds: $\{e = (n, n) \mid e \in E\} = \emptyset$. That is, the source and the target node cannot be identical for any edge.*

Definition 2.5. Connected Graph: *An undirected graph is connected if for any two nodes v and w there exists at least one path leading from v to w. A directed graph with this property is considered to be* strongly connected*; it is* weakly connected *if this constraint holds only for its underlying undirected graph.*

Definition 2.6. Complete Graph: *A simple graph is* complete*, if it contains every possible edge. Thus all nodes are adjacent to each other.*

We denote a complete graph with n nodes by Kn, e.g. K17 is the complete graph with 17 nodes.

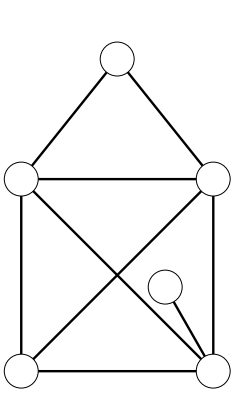
Definition 2.7. Planar Embedding: *A planar embedding Γ of a graph $G = (V, E)$ is a mapping of all nodes to points and all edges to lines on a two-dimensional plane. This mapping has to satisfy the following constraints:*

1. *No two different nodes are mapped on the same point.*
2. *The endpoints of the mapping of an edge $e = (u, v)$ coincide with $\Gamma(v)$ and $\Gamma(w)$.*
3. *No two lines $\Gamma(e_1)$ and $\Gamma(e_2)$ cross each other, except in their endpoints where necessary.*

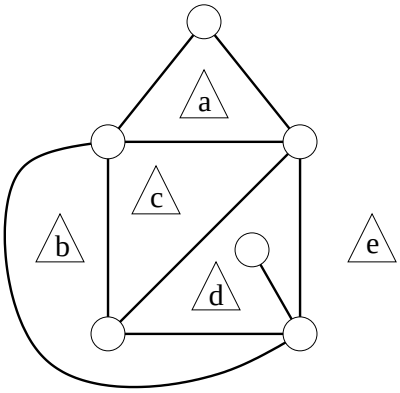
Definition 2.8. Topology: *A topology T of a graph $G = (V, E)$ defines an order of the edges around their nodes. Thus for every node we know the clockwise order of its outgoing edges.*

Definition 2.9. Planar Topology: *A topology T is* planar*, if there are planar embeddings that obey T. A planar topology inherently defines a set F of* faces*. Faces are the regions surrounded by edges.*

If not stated otherwise, we will only refer to planar topologies. Every planar embedding is an instantiation of a planar topology. There exists exactly one *outer face* in every planar embedding; that is the only region not completely bounded by edges, but the unbounded region outside of the drawing of the graph. Note that the topology does not define an outer face; a topology itself is therefore like an embedding on a sphere.
We define the degree $\deg(f)$ of a face $f \in F$ as the number of "sides" of edges touching f. If an edge is bounding the face, it counts once; but it counts twice, if it is completely inside of f (since both sides touch the face; cf. Fig. 2.1).
Furthermore, edges and nodes inside or on the border of a face, are *incident* to that face. Two faces are *adjacent* to each other if there exists at least one edge incident to both of them. We may specify these relations as follows: Two faces are adjacent to each other *with respect to* e, if the edge e is incident to both of them. A node n is incident to a face f *with respect to* e, if the edge e follows n on a counterclockwise walk around the incident nodes and edges of f.



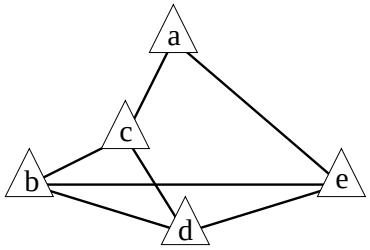
(a) planar graph G



(b) embedding Γ

Face f	$\deg(f)$
a	3
b	3
c	3
d	5
e	4

(c) face degrees in Γ



(d) dual graph G^D

Figure 2.1: Example of a planar embedding

Definition 2.10. Planar Graph: *A graph is planar if and only if it has at least one planar embedding.*

The *4–planar graphs* often play a decisive role. These are planar graphs with a maximum degree of no more than 4.

Definition 2.11. Dual Graph: *For every graph $G = (V, E)$ and its planar topology T (inducing the face set F), we can construct an undirected* dual graph $G^D = (V^D, E^D)$ *as follows (cf. Fig. 2.1):*

$$\begin{aligned} V^D &= F \\ E^D &= \{(f_1, f_2) \mid f_1, f_2 \in F, f_1 \text{ adjacent to } f_2 \text{ in } T\} \end{aligned}$$

2.2 Min–Cost–Flow Networks

A network is a directed connected graph $N = (V, E)$ with certain additional properties. For every node $v \in V$ we know its supply (demand). We denote this value $\sup(v)$ and identify the type by its sign:

- $\sup(v) > 0$: v has a supply of $\sup(v)$ units that have to be emitted into the network. Such a node is called *source*.
- $\sup(v) < 0$: v has a demand of $-\sup(v)$ units, hence it will retrieve just as many units in the solution. The node v is said to be a *sink*.

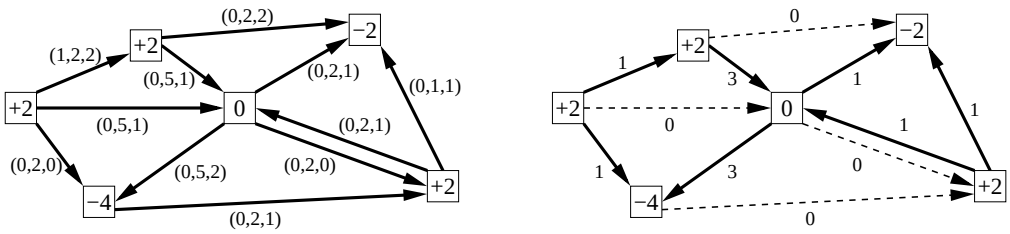


Figure 2.2: A min–cost–flow network (left) and its optimal solution (right). The parameters of the edges are given as (min, cap, cost)–triplets. The solution describes the resulting flow (unused arcs are dashed) and has an over–all cost of 13 units.

- $\sup(v) = 0$: v has neither a supply nor a demand. It will retrieve just as many units as it will emit.

Remark 2.1. *The sum of all supplies has to be identical to the sum of all demands. Hence the following constraint has to hold:* $\sum_{v \in V} \sup(v) = 0$.

The edge $e = (v, w)$ in such a network offers the possibility of transporting units from v to w. We can set the following parameters for such an edge:

- Capacity $\text{cap}(e)$: Defines how many units can be transported through e.
- Lower Bound $\min(e)$: Defines how many units have to be transported though e.
- Cost $\text{cost}(e)$: Defines the cost per unit that is being transported through e.

If not stated otherwise, we will always assume $\min(e) = 0 \ \forall \ e \in E$.
When solving such a network, a flow $\text{fl}(e)$ is assigned to each edge e, indicating the number of units to be transported through that edge. The over–all cost of the solution has to be as low as possible (see Figure 2.2 for an example). Such a problem is solvable in polynomial time [2][10].

Remark 2.2. *A network as described above is defined to transport only integral units, hence* sup(), cap(), min() *and* fl() *are only integers. In most cases, even* cost() *will only use this domain*[1]*. Note that there cannot exist a non–integer solution that is better than the best integral solution [2].*

2.3 Linear Programs

2.3.1 LP

A *Linear Program* (LP) is an optimization problem that consists of an objective function and several constraints formulated as inequalities. As the name implies, all expressions are only linear combinations of the variables, hence variables may only be added to each other and multiplied by a scalar. We look for an optimal assignment – in terms of the objective function – for the variables involved.

[1] Note that we can scale rational numbers to retrieve integer values.

Such a program is usually written as:

$$\begin{aligned} \max \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \end{aligned}$$

The first line of the program is the *objective function*, the second denotes the *constraints*. The abbreviations have the following meanings:

c	$\in \mathbb{R}^n$	coefficients of the objective function
A	$\in \mathbb{R}^{m \times n}$	coefficients of the constraints
b	$\in \mathbb{R}^m$	constant right–hand side
x	$\in \mathbb{R}^n$	variables

We look for a vector x^* for which the following constraint holds:

$$c^T x^* = \max\{c^T x \mid Ax \leq b\}$$

Though it seems quite restricted at first glance, we can easily reformulate the above model to solve minimization problems. Furthermore, we are not restricted to less–or–equal constraints, but can also use equal- and greater–or–equal constraints. Obviously this enables us to define upper and lower bounds for variables.
Note that all variables and constants involved are contained in $\mathbb{R}$. Much scientific effort has already been put into algorithms solving such programs. The most widely known are the Simplex-, the Ellipsoid- and several Interior–Point methods. The latter two can solve any linear program in polynomial time [14][2].

2.3.2 ILP

An *Integer Linear Program* (ILP) is an extension of the classical LP. In such programs the domain $\mathbb{Z}$ is demanded for at least one variable.
In general such an extension renders a problem NP-complete [13][28] (cf. Section 6.1), meaning there cannot exist any polynomial algorithm finding the optimal solution (assuming $P \neq NP$).
If, however, the matrix A of the relaxed problem (the problem without the integer-constraints) is *totally unimodular*[3] and the vector b contains only integer constants, all optimal solutions are (automatically) integers, hence computable in polynomial time [24].
Note that every min–cost–flow network can easily be transformed into an ILP. Such ILPs always satisfy the above constraint [24], and are therefore solvable in polynomial time.

[2] to be more exact: they can arbitrarily approximate in polynomial time [16].

[3] An $m \times n$ integral matrix A is totally unimodular if the determinant of each square submatrix of A is equal 0, 1, or –1 [24].

Chapter 3

Orthogonal Graph Drawing

Um ein guter Maler zu sein, braucht es vier Dinge:
weiches Herz, feines Auge, leichte Hand und immer frischgewaschene Pinsel.
Anselm Feuerbach

This chapter gives an overview of some common methods used to draw graphs orthogonally. We describe the general aims and the Topology–Shape–Metrics–Model. The latter is based on breaking the drawing problem down into three subproblems and solving these successively.
Since the SPED–algorithm presented in this thesis is a part of this model – it solves the orthogonalization subproblem – Section 3.3 gives a brief overview of the most common algorithms used for this step today.

3.1 General

When drawing a graph, it is of great importance to accomplish certain quality criteria in order to satisfy particular demands that improve the clarity of the drawing.
These include the following:

- Edges should follow orthogonal paths
- Edges should cross each other only very rarely, or even never at all
- Edges should have a small number of bends
- The drawing should use as little space as possible
- The graph should be evenly distributed on the drawing area
- The drawing should look as symmetric as possible
- The hierarchical structure of the graph (if present) should be emphasized

Obviously, it is impossible to satisfy all these criteria at the same time, so we have to establish a certain order on their priorities.
There have been many different approaches to address this issue, but one of the most common today is the Topology–Shape–Metrics–Model *(TSM–Model)*, which will be discussed in the next section. Although it has some inherent drawbacks, these are accepted due to its simple structure and applicability on large graphs.

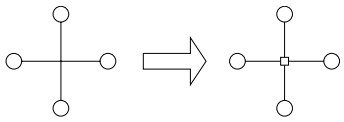


Figure 3.1: For each crossing (left), a dummy node is generated (right); (dummy nodes are represented by a square)

3.2 Topology–Shape–Metrics Model (TSM–Model)

In this three–phased model (see [3][6][32]), the three most important drawing criteria were selected and ordered by their significance. These are:

1. Minimum edge crossings
2. Minimum bend count
3. Minimum space requirement

We attempt to satisfy these requests stepwise, which leads to a scheme where unfortunate decisions in an earlier phase are irreversible and cannot be undone in a later stage of the algorithm. Nevertheless, this approach is currently one of the best known methods for calculating graph drawings. We can draw a graph in polynomial time by executing the steps in polynomial time using heuristics if necessary.

3.2.1 Topology – Planarization/Embedding

In the first phase, we planarize the input graph G. The output is a planar topology T of the planarized graph. If edge crossings should be necessary, we insert a *dummy node* instead of the crossing (see Fig. 3.1). This node will remain in the graph during the next two steps, and will be removed directly prior to drawing the graph. Such a node has a degree of four, since crossings of multiple edges in a single point are not allowed due to clarity reasons.
Furthermore, this step chooses a face to become the outer face. A standard heuristics is to select the face with the largest degree.
During this stage, the main objective is to generate as few crossings as possible. There exist efficient heuristics for this step [17][33].

3.2.2 Shape – Orthogonalization

The second phase starts with the graph G and its previously generated and now fixed planar topology T. In this step the *(orthogonal) shape* H will be calculated, trying to guarantee a minimum bend count.
An (orthogonal) shape is an in–between of the planar topology T and a planar embedding: in addition to the edge order, H includes both the description of the bends on the edges and the exact angles between the different edges. As its name implies all angles are multiples of 90°. The main difference between an embedding and a shape is that the latter does not include any information on the lengths of the various line segments.
One problem in this stage is the exact definition of valid orthogonal embeddings, since if using the strictest, only 4–planar graphs would be drawable. Details on these considerations are summarized in Section 3.3. For the time being it is sufficient to note that

there exist polynomial algorithms that achieve provable optimality for simple 4–planar graphs [31], as well as for a wide range of extended definitions of orthogonal embeddings (cf. Sec. 3.3).

3.2.3 Metrics – Compaction

In the last step of the TSM–model, we will calculate the dimensions of the (fixed) shape H; nodes are mapped on two–dimensional points, and edges on line–segments. This planar embedding Γ should need as little area as possible. This last property is often simplified to minimizing its perimeter, the sum of all edge lengths, etc.
Although this may not seem too complicated at first glance, this problem is – in general – NP–hard [21], but there exist efficient heuristics [19].

3.3 Orthogonalization Variations

We will only look at *grid drawings*, where nodes and edge bends are only allowed to be mapped on integer coordinates. Following the definition of embeddings, the (orthogonal) edges are not allowed to touch each other except in their endpoints. This renders the drawing of nodes with a degree greater than four impossible, since every node only has room for four incident edges leaving it on its four different sides (top, bottom, left, right). There are many different approaches on how to solve the problem with graphs having a maximum degree greater than four. The most common methods to circumvent these limitations can be categorized as follows – the most established convention of a category is given in brackets and shown in Figure 3.2:

1. Nodes can become larger than one grid point (*Giotto*, Fig. 3.2(a))
2. Edges use a finer grid than the nodes do (*Podevsnef*, Fig. 3.2(b))
3. Edges can leave a node non–orthogonally (*Quasi–Orthogonal*, Fig. 3.2(c))

Giotto [32]

Nodes in Giotto–like drawing standards are not represented by a single grid point. A single node can stretch over several of them (normally the nodes remain rectangular). Thus several edges can be incident to the side of one node.
The basic idea behind this mechanism is to replace high degree nodes (nodes with degree greater than four) with *cages* – a circularly connected node set representing the border of the node. The number of nodes of such a cage is identical to the degree of the original node (see Fig. 3.3). Since every cage node has a degree of three, this transformation produces a 4–planar graph. When drawing this modified graph, we demand that the cages have a rectangular shape by forbidding concave bends on them.
This simple concept has the drawback of irregular looking nodes. This is further aggravated by the fact that the size of a node may grow independently of its degree.

Quasi–Orthogonal [18][20]

This method is closely related to the Giotto–method. Essentially, it also generates bigger nodes, but finally maps the according node on a single grid point “in the middle” of the

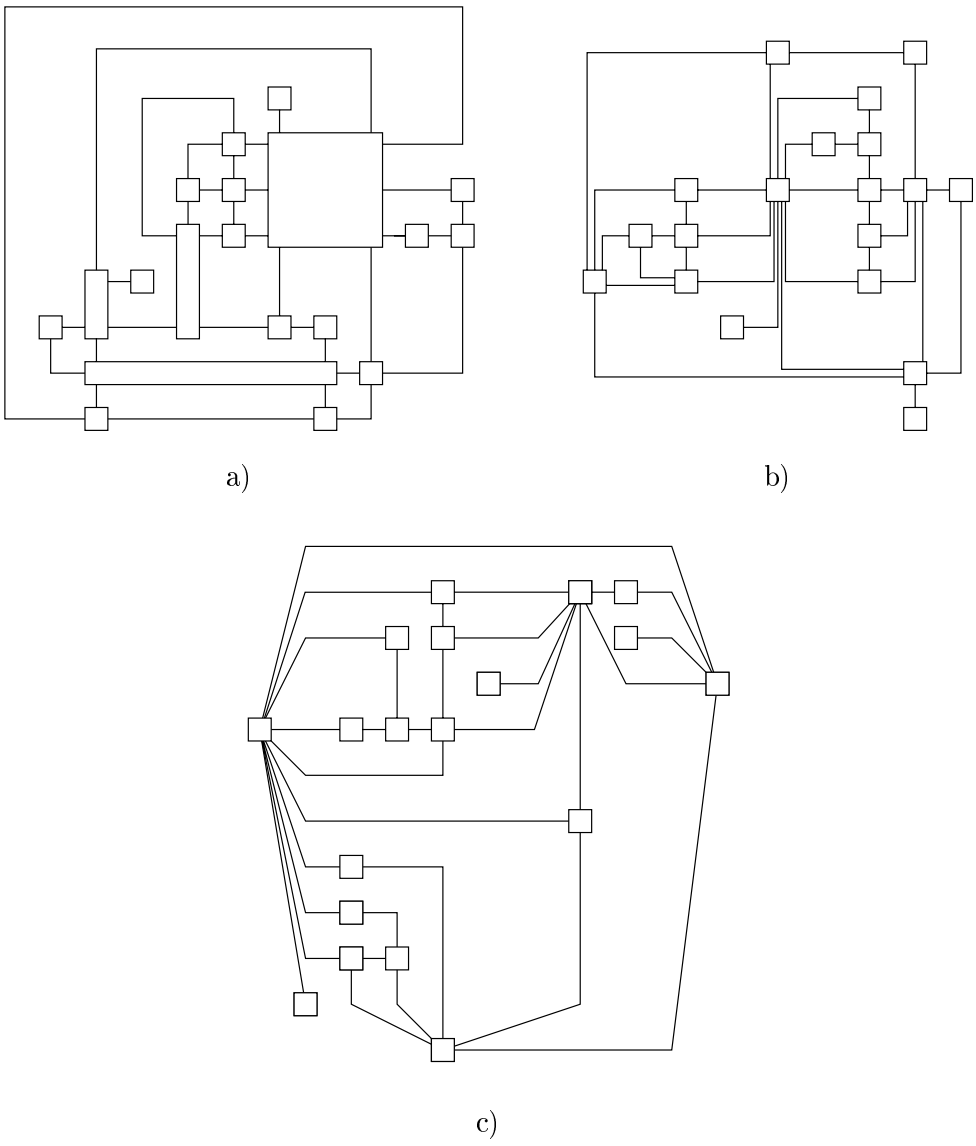


Figure 3.2: Orthogonalization Variations: a) Giotto, b) Podevsnef, c) Quasi–Orthogonal

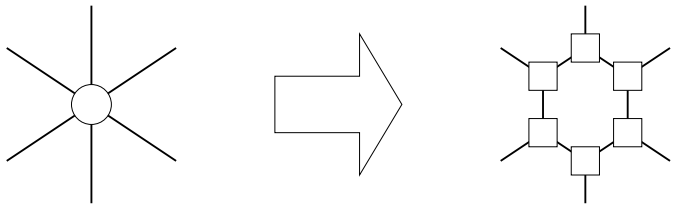
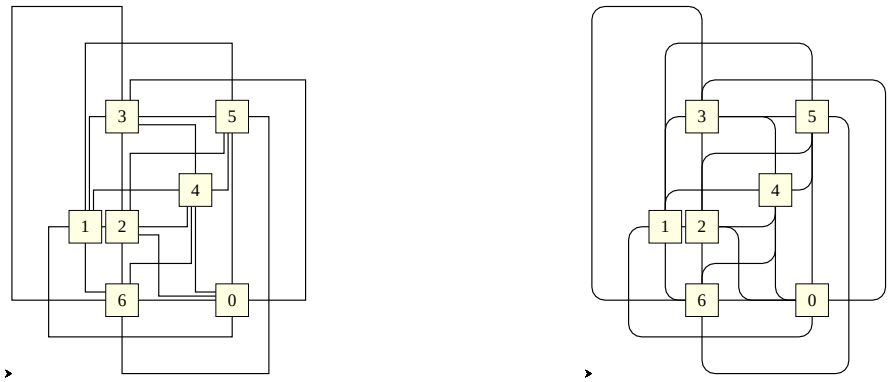


Figure 3.3: High degree nodes (left) are replaced by cages (right)



(a) (b)

Figure 3.4: (a) classic drawing style; (b) bus drawing style; (K7, Simple–Podevsnef)

larger node. This leads to bends and non–orthogonal edges between the border of the bigger node and the shrunken node itself.
Clearly, the benefit of this method is the uniform node size. The drawbacks are the often undesirable non–orthogonal edges, as well as the increased bend count.

Podevsnef [11][12]

This technique allows several edges to emit from one side of a node without the need of growing. The edges are embedded on a finer grid than the nodes. It requires much more complex methods to prohibit invalid drawings (like overlaps).
In contrast to the methods mentioned above, it is not necessary to generate a temporary 4–planar graph simulating the high degree nodes. Details on this method – as well as on the meaning of its name – can be found in the following chapter.

Remark 3.1. *In contrast to the drawing style used throughout this thesis there exists an equivalent* bus drawing style.[4] *As Fig. 3.4 demonstrates, multiple edges are only shown as one until they demerge. To make it possible to follow single edges, each such split has to be drawn with rounded corners or similar visual aids. Sharp corners would introduce ambiguities and suggest nonexistent edges. Note that this style is only applicable for undirected graphs, since the arrowheads would introduce ambiguities.*
To create a bus style drawing, we simply have to ignore the offsets of the edges on the finer grid, and transform each bend into a curved corner.

[4]The implementation of SPED is able to generate both types of drawing styles.

Chapter 4

Simple–Podevsnef

So einfach wie möglich. Aber nicht einfacher!
Albert Einstein

This chapter focuses on the Simple–Podevsnef drawing standard for planar graphs. It is used for calculating the shape in the TSM–model. We use it as a foundation for SPED, which basically is Simple–Podevsnef for non–planar graphs.
After a brief description of the more powerful Podevsnef standard in the first section, Section 4.2 explains the original approach to the simplification of Podevsnef. The algorithm will be called *Italian model* because of its origin. It is based on a min–cost–flow network and can be seen as a true extension to Tamassia's network for 4–planar graphs [31].
We modify this model into an equivalent solving scheme in Section 4.3, in order to allow extensions for non–planar graphs. We will refer to this changed version as the *alternative model*.

4.1 Podevsnef

The Simple–Podevsnef drawing convention that will be explained throughout the next sections, is based on the much more complicated Podevsnef standard by Fößmeier and Kaufmann [11][12]. Both schemes deal with the problem of high degree nodes in a very similar way (see Section 3.3).
By using a finer grid for placing the edges, and a coarser grid for placing the nodes, it is possible to have more than one edge incident to each side of a node. Edges that emit from the same side of a node are called *bundle* as long as they stay together (i.e. their distance is notably smaller than the resolution of the node grid).
The name Podevsnef is an acronym and stands for *Planar Orthogonal Drawing with Equal Vertex Size and Non–Empty Faces*. This implies that only drawings are generated in which all nodes have the same size and every face is big enough to hold a hypothetical square with the area of $1 - \varepsilon$. The variable ε represents a number much smaller than 1; to be exact, $\varepsilon \leq 2d - d^2$, where d denotes the maximal width an edge bundle can have. Note that d is by definition much smaller than the resolution of the node grid ($d \ll 1$) (see Fig. 4.1).
Podevsnef is also known by the name *Kandinsky*. A drawing in this convention is computed using a quite complex network flow that cannot be solved with classic min–cost–flow algorithms. Hence another similar standard was developed, featuring a simpler network: Simple–Podevsnef.

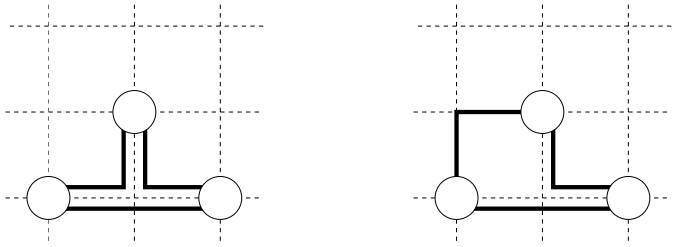


Figure 4.1: The drawing on the left contains an *empty face*, whereas you can fit in a square with the area of $1 - \varepsilon$ in the right one.

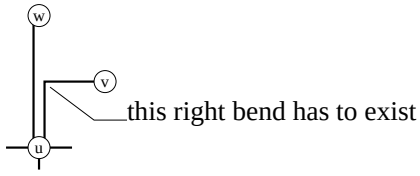


Figure 4.2: The right bend in Simple–Podevsnef

4.2 Simple–Podevsnef (Italian Model)

4.2.1 General

This simplification was first presented by Bertolazzi, Di Battista and Didimo [4]. It uses a much simpler network which is very similar to the classic approach by Tamassia [31]. It follows the rules of Podevsnef, with two additional properties:

1. Each node with a degree greater than 4 has at least one incident edge on each side
2. Let u be a node with $\deg(u) > 4$, and the edge (u, w) following (u, v) in the circular counterclockwise ordering. If there is a 0° angle between them, (u, v) has to have at least one right bend (see Fig. 4.2). This bend is the first when following the edge from u to v.

While this section focuses on the original model, Section 4.3 presents an alternative network solving the same problem, but – although it needs some more nodes and edges – has the advantage of being easier to understand and to extend. This formulation will then be used throughout the description of SPED, the new algorithm presented in Chapter 5. Section 4.2.4 supplements the technical description of the next two subsections by giving a stepwise example.

4.2.2 Network Construction

Let $G = (V, E)$ be a planar, simple, self–loop–free graph. Let T be its planar topology inducing the face set F. Let $f_0 \in F$ be the outer face. Let $L(V)$ be the subset of V that contains all nodes with a maximal degree of four, and $H(V)$ the subset that contains the nodes with higher degrees.

In order to compute an orthogonalization, we define a network $N = (V_N, E_N)$ as follows:

$$\begin{aligned}
V_N &= V \cup F \\
E_N &= E_{ff} \cup E_{vf} \cup E_{fv} \\
E_{ff} &= \{(f_1, f_2) \mid f_1, f_2 \in F, \exists e \in E : f_1 \text{ and } f_2 \text{ are adjacent, with respect to } e\} \\
E_{vf} &= \{\{(v, f) \mid v \in L(V), f \in F, \exists e \in E : v \text{ is incident to } f, \text{ w.r.t. } e\}\} \\
E_{fv} &= \{\{(f, v) \mid v \in H(V), f \in F, \exists e \in E : v \text{ is incident to } f, \text{ w.r.t. } e\}\}
\end{aligned}$$

Note that the sets E_N, E_{ff}, E_{vf} and E_{fv} are multisets, thus the same element may be included several times (once for each different edge e).
Furthermore, we assign the following flow parameters:

$$\begin{aligned}
\text{cap}(e) &= \infty && \forall\ e \in E_{ff} \\
\text{cap}(e) &= 4 - \deg(v) && \forall\ e = (v, f) \in E_{vf} \\
\text{cap}(e) &= 1 && \forall\ e \in E_{fv} \\
\\
\text{cost}(e) &= 1 && \forall\ e \in E_{ff} \cup E_{fv} \\
\text{cost}(e) &= 0 && \forall\ e \in E_{vf} \\
\\
\sup(x) &= 4 - \deg(x) && \forall\ x \in V_N - \{f_0\} \\
\sup(f_0) &= -4 - \deg(f_0)
\end{aligned}$$

Remark 4.1. *We show that the sum of all supplies and demands cancel each other out, as necessary for a valid network:*

- *The supply of all nodes is*

$$\sum_{v \in V} \Big(4 - \deg(v)\Big)$$

which can be simplified to

$$= \sum_{v \in V} 4 - \sum_{v \in V} \deg(v) = 4|V| - 2|E|$$

since each edge is incident to exactly two nodes.

- *The supply of all faces is calculated by*

$$\Big(-4 - \deg(f_0)\Big) + \sum_{f \in F - \{f_0\}} \Big(4 - \deg(f)\Big)$$

and may be reformulated to

$$= -4 - \deg(f_0) + \sum_{f \in F - \{f_0\}} 4 - \sum_{f \in F - \{f_0\}} \deg(f) =$$

$$= 4(|F| - 2) - \sum_{f \in F} \deg(f)$$

Because any edge is incident to exactly two faces, this results in

$$= 4(|F| - 2) - 2|E|$$

So this adds up to an overall supply of

$$4|V| - 2|E| + 4(|F| - 2) - 2|E| =$$

$$= 4|V| - 4|E| + 4(|F| - 2)$$

We want to proof that this is equal to 0, and divide the resulting equation by 4:

$$|V| - |E| + |F| - 2 = 0$$

Bringing the 2 to the right side, we get the Euler formula for planar graphs:

$$|V| - |E| + |F| = 2$$

□

4.2.3 Reverse Transformation

This section describes how to extract the orthogonal representation from the minimum flow (see Section 4.2.4 for an example).

Remark 4.2. *Note that Tamassia's original network [31] only addresses 4–planar graphs. When applying the Simple–Podevsnef construction on such a graph, $H(V)$ and E_{fv} are empty sets. The remainder is basically a classic Tamassia network.*

In general, a flow on an edge $a_{f_1,f_2} \in E_{ff}$ induces the bends on the edge $e = (u, v) \in E$, where e is incident to f_1 and f_2. Let – without loss of generality – f_1 be left, and f_2 right of e, and v above u. Each unit that is transported from f_1 to f_2 causes a left bend when following the edge e from u to v. A flow in the inverse direction causes a right bend.
In contrast to Tamassia's construction, this flow does not completely define all bends on an edge of G. An additional bend may be added to either end, as will become obvious soon.

Remark 4.3. *Note that due to the cost minimization either a_{f_1,f_2} or a_{f_2,f_1} will not transport flow (for all pairs of incident faces f_1 and f_2).*

The flow on the edge $a_{v,f} \in E_{vf}$ defines the angle $\alpha_{v,f}$ between the edges e_1 and e_2 incident to v and f. As $\deg(v) \leq 4$, v is a source of $4 - \deg(v)$ units, so $\alpha_{v,f}$ is calculated by

$$\alpha_{v,f} = 90° \cdot (1 + \mathrm{fl}(a_{v,f}))$$

It is obvious that the angles around each node sum up to 360°.
The flow on the edges $a_{f,v} \in E_{fv}$ also induces the angle $\alpha_{v,f}$ at the node v ($\deg(v) > 4$): if $\mathrm{fl}(a_{f,v}) = 0$, the angle between the edges incident to v and f should be 90°; a flow of 1 stands for 0°.
A 0° angle means that the two edges in question ought to be parallel (Fig. 4.3(a)). It is easy to see that this can cause problems: several nodes may be mapped onto the same point (Fig. 4.3(b)), or an edge may run through a node that is not incident to it (Fig. 4.3(c)).
The solution is shown in Figure 4.3(d): by adding an additional bend the desired parallelism is accomplished. This bend is being constructed by enlarging the angle counterclockwise from 0 to 90 degrees and adding a right bend to the affected edge of G. This bend has to take place before any other on this edge.

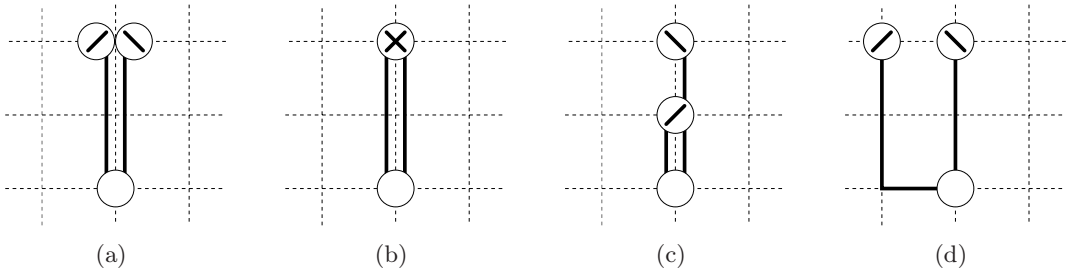


Figure 4.3: The problem of parallelism

The generation of a bend for any occurring 0° angle explains the costs for the according network arc $a_{f,v}$.

Proofs for the validity of the network and the applicability of the reverse transformation can be found in the initially mentioned paper [4] and in Weiskircher's PhD thesis [33].

4.2.4 Example

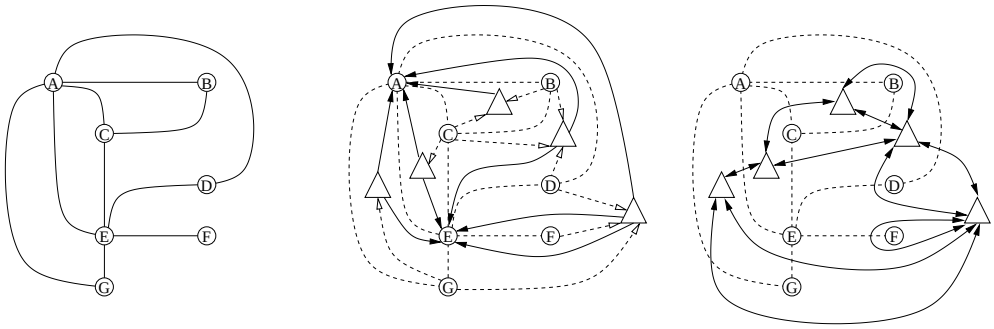
Figure 4.4 shows two solution for a graph drawn by Simple–Podevsnef: the solution (d)(e) is suboptimal (9 bends), whereas solution (f)(g) requires the minimum number of bends possible for this planarization (6 bends). Only arcs with flow greater than 0 are shown in subfigures (d) and (f); the flow of these arcs equal 1 if not labeled otherwise.
The supplies are:

Node	Supply
A	-1
B	2
C	1
D	2
E	-1
F	3
G	2

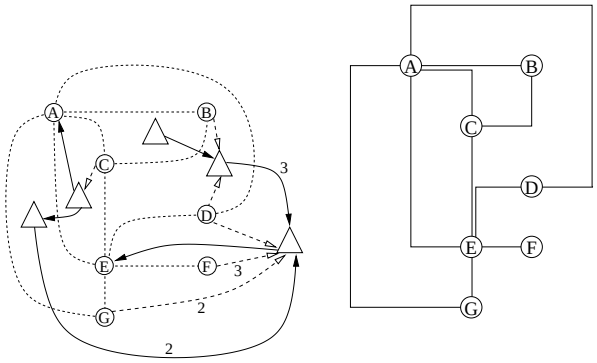
Face	Supply
AGE	1
AEC	1
ACB	1
ABCED	-1
ADEFEG (outer face)	-10

4.2.5 Preparing for Compaction

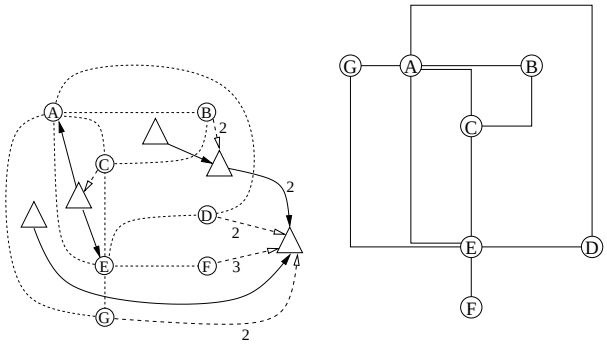
Before starting the compaction phase using the computed orthogonalization, the graph has to be prepared [4]: We modify the graph in such a way that all segments of parallel running edges will be represented by a single edge. When such a bundle demerges (aka. splits up), a dummy node (with a degree of three) will be added (see Fig. 4.5).
It is easy to see that the resulting graph is still planar, may easily be transformed back into the original, and is able to satisfy the shape criteria of the primary graph. This transformation is of great importance, since it enables us to use traditional compaction algorithms.
After the compaction stage, we restore the original graph and draw the edge bundles accordingly.



(a) Graph to draw (b) Node arcs ($E_{vf} \cup E_{fv}$) (c) Face arcs (E_{ff})



(d) Possible flow (e) Shape based on (d)



(f) Optimal flow (g) Shape based on (f)

Figure 4.4: Calculating a Simple–Podevsnef drawing; (circle denote nodes, triangles denote faces, arcs without costs are dashed vectors)

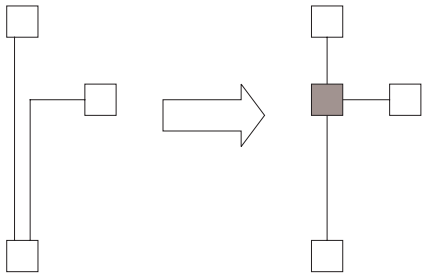


Figure 4.5: Modification of edge bundles prior to the compaction

4.3 Simple–Podevsnef (Alternative Model)

As mentioned earlier, this section will present an alternative network for computing a shape following the Simple–Podevsnef scheme.
This new model is even closer related to Tamassia's construction for 4–planar graphs than the Italian model. Besides of being easier to extend, it is also simpler to understand and offers a more homogeneous representation. Furthermore, the reverse transformation from the network flow to the shape of the graph is vastly simplified. Even though it needs some more nodes and edges in absolute numbers, its demand is of the same magnitude as the Italian Model, differing only by a constant factor.

As before, we define a network $N = (V_N, E_N)$, with $V_N = V \cup F$. This time, we will start – as in Tamassia's network – with only two types of edges, defined identical to E_{vf} and E_{ff}, with the only difference that E_{vf} is not restricted to $L(V)$–nodes anymore.
The logic applied for the flows in the network is as follows:

- Any flow unit in an edge $a_{v,f}$ induces a 90° angle. Thus the resulting angle can be calculated just by $90° \cdot \mathrm{fl}(a_{v,f})$.
- Any flow unit in an edge a_{f_1,f_2} induces a bend on the edge incident to both f_1 and f_2, just as described for the Italian model. Note that no other flow will add any more bends, since there is no set E_{fv}.

By setting the parameters correctly, we ensure that $L(V)$-nodes will not generate any 0° degree angles, and that $H(V)$-nodes distribute their incident edges over all their sides (simply by prohibiting angles greater than 90°). We also have to adjust the demands of the faces.

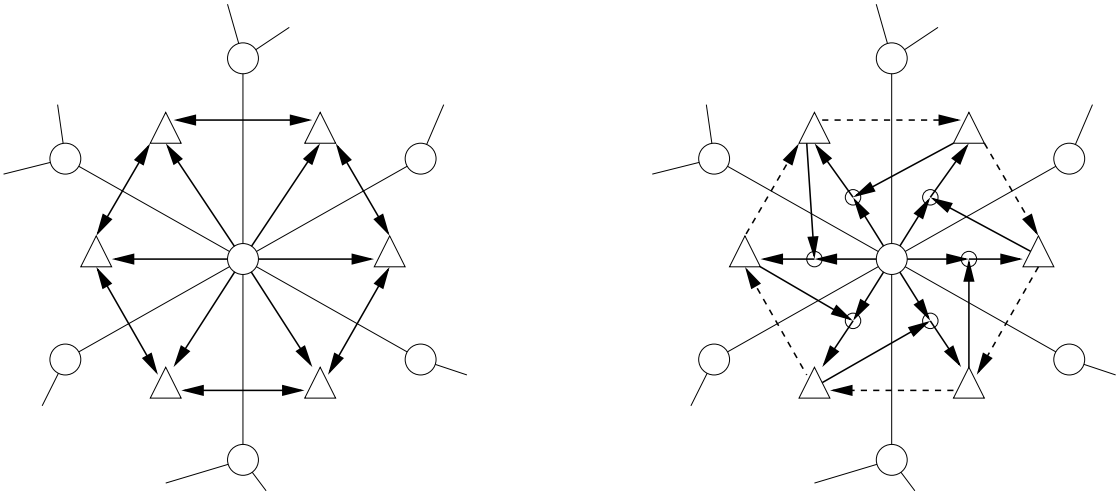


Figure 4.6: left: underlying network construction, right: correctly augmented network; (nodes are circles, faces are triangles, irrelevant arcs are not shown)

$$
\begin{array}{rcl}
\mathrm{cap}(e) = & \infty & \forall\ e \in E_{ff} \\
\mathrm{cap}(e) = & 5 - \deg(v) & \forall\ e = (v, f) \in E_{vf}, v \in L(V) \\
\mathrm{cap}(e) = & 1 & \forall\ e = (v, f) \in E_{vf}, v \in H(V) \\
 & & \\
\min(e) = & 0 & \forall\ e \in E_{ff} \\
\min(e) = & 1 & \forall\ e = (v, f) \in E_{vf}, v \in L(V) \\
\min(e) = & 0 & \forall\ e = (v, f) \in E_{vf}, v \in H(V) \\
 & & \\
\mathrm{cost}(e) = & 1 & \forall\ e \in E_{ff} \\
\mathrm{cost}(e) = & 0 & \forall\ e \in E_{vf} \\
 & & \\
\sup(v) = & 4 & \forall\ v \in V \\
\sup(f) = & 4 - 2\deg(f) & \forall\ f \in F - \{f_0\} \\
\sup(f_0) = & -4 - 2\deg(f_0) &
\end{array}
$$

This simple construction does not include any solution for the parallelism–problem (cf. Fig. 4.3) yet. This will be achieved by a little modification and augmentation of the network that will be applied to all nodes of $H(V)$ (see Fig. 4.6):
If we have two edges running directly parallel, we want to force the one that follows the other in clockwise order to make a right bend. This means that the face f between these two edges has to retrieve (at least) one flow unit from either the node (inducing a 90° angle in the node, thus there is no parallelism problem at all) or from its neighboring face in clockwise order. To achieve this, we simply reroute these two edges of the network into a dummy vertex v' (with $\sup(v') = 0$), and add a network edge e' from v' to f with

$\min(e') = 1$. (The capacity of this edge is unlimited and there are no costs charged for using it.)

Remark 4.4. *This construction cannot be easily transformed into a similar Podevsnef network by mirroring this technique for left bends. This would lead to invalid shapes and empty faces (see Section 4.1).*

Remark 4.5. *The proof for the balance of supplies/demands in this network is similar to the one given for the Italian model (see page 15), or even simpler, so it is only sketched briefly here:*

- *Supply in nodes:* $\sum_{v \in V} 4 = 4|V|$
- *Demand in faces:*

$$\Big(-4 - 2\deg(f_0)\Big) + \sum_{f \in F - \{f_0\}} \Big(4 - 2\deg(f)\Big) =$$

$$= 4(|F| - 2) - 2\sum_{f \in F} \deg(f) =$$

$$= 4(|F| - 2) - 2 \cdot 2|E|$$

Supply and demand should sum up to 0:

$$4|V| - 4|E| + 4|F| - 8 = 0$$

which leads to Euler's formula, again. □

Having this valid Simple–Podevsnef algorithm at hand, we can extend it for dummies.

Chapter 5

SPED – Simple Podevsnef Extended for Dummies

Ein Hündchen wird gesucht,
Das weder knurrt noch beißt,
Zerbrochne Gläser frißt
Und Diamanten ...
Annonce, J. W. v. Goethe

This is the main chapter of this thesis and explains an algorithm for drawing non–planar graphs following the Simple–Podevsnef standard. The first section points out why we have to extend Simple–Podevsnef at all.
Section 5.2 explains the big picture of the algorithm and its various steps. It is designed to be as non–technical as possible without being over simplifying. It introduces various concepts and notions only informally; their exact definitions are given in the sections thereafter.
The Sections 5.3–5.8 give the exact description of SPED and the appertaining proofs. Section 5.3 focuses on fundamental definitions and abbreviations and Section 5.4 describes the transformation of Simple–Podevsnef (alternative model) into a linear program. Sections 5.5–5.7 are the quite technical main part of this chapter, where the extension itself is developed and proven.
After Section 5.8 summarizes the resulting SPED algorithm mathematically, the last section of the chapter describes the necessary modifications to the precompaction algorithm that prepares the shape for the compaction step.

5.1 The Problem

When we want to draw a non–planar graph G', we first have to planarize it by adding dummy nodes where edge crossings happen, as described in Section 3.2.1. If the resulting graph G is then used for the next steps without special considerations, every such crossing will need a complete grid point in the final drawing (see Fig. 5.1).
The task is to define an orthogonalization algorithm sensitive to such dummy nodes that is able to merge them into a single grid point (if possible). Thus it is possible that whole edge bundles, as they frequently occur in Simple–Podevsnef, can cross another edge collectively, instead of splitting up first. Nevertheless, the prime objective remains to generate a shape with the minimum number of bends.
In this section, we present SPED – an algorithm that uses the (alternative) Simple–Podevsnef network as a basis and extends it. To allow these enhancements, we use an integer linear program.
Additional topics related to SPED are collected in Chapter 6. This chapter refers to them where necessary.

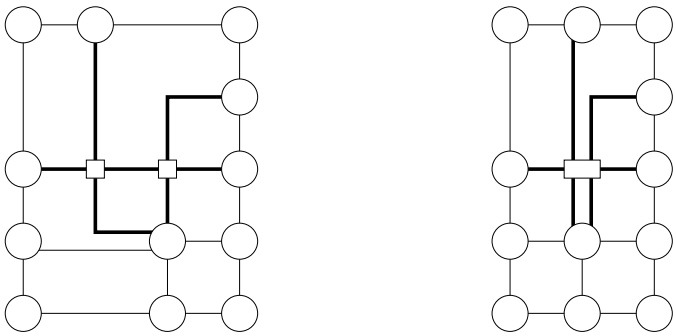


Figure 5.1: left: Simple Podevsnef, right: SPED; (squares denote dummy nodes)

5.2 Overview of SPED

This section gives a non–technical overview of the concepts and structure of SPED. It outlines the techniques of the proofs and describes the results. This overview does not try to give exact mathematical explanations or exhaustive analyses. For details and proofs see Sections 5.4–5.7. The mathematical results are summarized in Section 5.8.
Since there is a considerable number of lemmata that depend on each other, Section 5.2.1 gives an overview of their relations.

As its name implies, SPED is an extension to Simple–Podevsnef. Hence it is based on the network described in Section 4.3. We transform this network into an ILP (Sec. 5.4) to make all necessary additions possible.
The first step towards SPED is to analyze the situations where bundle crossing – and thus dummy merging – is applicable (Fig. 5.2): the main necessity is a bundle, hence we need a high degree node and a 0° angle between two edges adjacent to this node. As described earlier, we want to be able to merge dummy nodes into a single grid point if they result from an edge crossing the bundle. Thus we can deduce that we need to have a triangular face adjacent to the high degree node. The other two nodes adjacent to this face have to be dummy nodes. We call such a face *coltri (collapsible triangle)*.

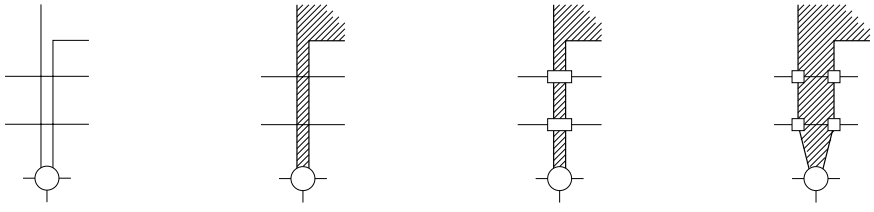


Figure 5.2: To generate a drawing like (a), we have to look at the hyperface (b). Such a drawing includes merged dummies (c). The structure of the hyperface is shown in (d). (Circles denote original nodes, squares denote dummy nodes.)

To enable multiple bundle crossings per bundle, this coltri may be followed by any number of *colquods (collapsible quods)*. A colquod is a face with exactly four incident nodes and all of them are dummy nodes.

Based on this analysis, we know the structure necessary for dummy merging (Sec. 5.5): a *hyperface* is a list that starts with a coltri, may contain colquods and ends with any other face; a bundle has to split up only somewhere on its hyperface and not necessarily on the coltri – as would be demanded by the Podevsnef model.
All faces between the coltri and the face where such a bundle splits up are said to be *collapsed*.

Simple–Podevsnef requires a right bend on the right edge if two edges leave a node with a 0° opening angle. To combine this with the knowledge of hyperfaces, we simply fine–tune that demand: SPED requires a right bend somewhere on the right side – called *hyperedge* – of each hyperface (Sec. 5.6.1). Thus, Fig. 5.2 represents a validly drawn hyperface.
This extension introduces two problems:

1. We cannot formulate the demand for such a right bend as a classic min–cost–flow network.

2. There exist solutions of this modified network that cannot be transformed into a valid shape of the graph.

The first drawback is the reason why SPED needs an ILP instead of a classic min–cost–flow network; in the former, the formulation of such a demand does not constitute any difficulty. Although most simple cases introduce no problems, more complex graphs show the necessity of the explicit demand for integer solutions. Thus, we have to have intractability in mind (see Section 6.1).
The second drawback is tackled by certain additional restrictions to the ILP that prohibit all invalid (integer) solutions. These restrictions are constructed in such a way that they do not introduce any additional non–integer solutions and can be applied in polynomial time. SPED needs the three listed kinds of additional error circumventions, which are defined and explained in the next few paragraphs:

1. Prohibiting most *downflows*

2. Prohibiting certain *left–FRFs*

3. Mitigating invalid *updown–FRFs* by the use of a repair–function

The first two parts are implemented by means of additional inequalities, whereas the third is a function that modifies the solution of the ILP after the optimization is finished. To explain these tasks, we have to analyze the possible errors first; invalid shapes may only be generated inside of the collapsed part of such a hyperface (Sec. 5.6.2).
It can be shown that flow that runs inside the collapsed part of a hyperface downwards to the coltri generates undrawable shapes (Fig. 5.3). We call such a flow – in whichever part of the hyperface it may occur – a *downflow*. Furthermore, it can be shown that nearly all downflows in the non–collapsed part of the hyperface can be forbidden without decreasing the quality of the solution. (Thus there always exists a related solution without most of these downflows). The only downflows that have to be allowed in the non–collapsed (aka. *demerged*) part of a hyperface are so–called *updown–FRFs*.
An *FRF (FlowReFlow)* is the situation where there is both a flow from f_1 to an incident face f_2 and vice versa. If one of these flows is a downflow, we call the FRF *updown–FRF*.

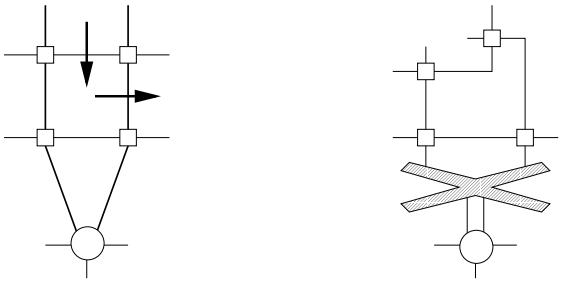


Figure 5.3: The above downflow (left) would generate an invalid shape (right): the 0° angle on the high degree node does not offer enough space for the bends above.

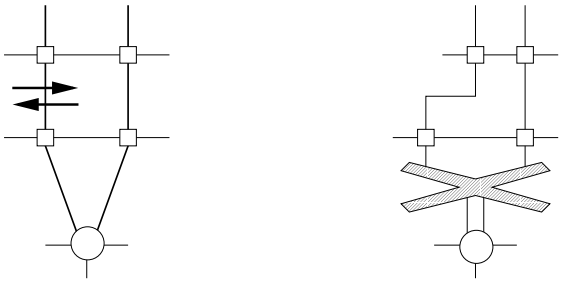


Figure 5.4: The above left–FRF (left) would generate an invalid shape (right): the 0° angle on the high degree node does not offer enough space for the bends above. (Note that Simple–Podevsnef forces right bends to precede left bends)

Hence the first additional extension for the ILP of SPED is the prohibition of all downflows except for updown–FRFs (Sec. 5.6.3, Constraint (5.10)). Note that we still allow updown–FRFs in the collapsed parts of hyperfaces, although they generate invalid shapes. The only source of error other than a downflow is an invalid *left–FRF* (Fig. 5.4; Sec. 5.6.4). A left–FRF is an FRF that happens on the left side of a hyperface. If this FRF happens on the collapsed part of a hyperface it generates an invalid shape. Due to certain hyperface properties, the positions where we have to allow the FRF in order to sustain the optimality of the solution can be further specified: there exists a simple inequality that enables exactly these necessary positions and prohibits all others (Constraint (5.11)).
These first two extensions remove all errors except for updown–FRFs in the collapsed parts of hyperfaces. It can be shown that there exists a polynomial function that shifts the invalid flows in such a way that all remaining errors are removed (Sec. 5.7). Since this function does not change the objective value, the optimality of the solution is still guaranteed.

Thus, after solving the ILP and modifying its result by the repair–function, we get a valid and optimal flow. It can be transformed into a shape just as we know it from the alternative Simple–Podevsnef formulation. The merged dummies are implicitly given by 0° opening angles and no demerging bends (i.e. bends on the hyperface that happen prior to the considered dummynodes). We just have to slightly refine the precompaction–step (Sec. 5.9).

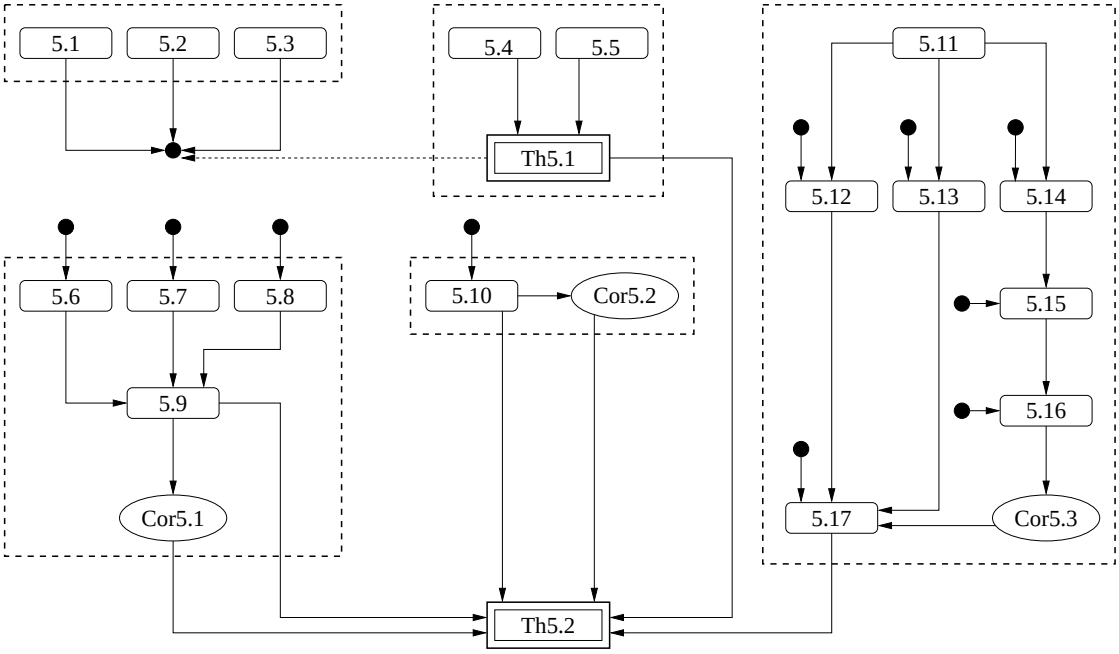


Figure 5.5: Dependencies of the upcoming lemmata

5.2.1 Dependecies of the Upcoming Lemmata

Fig. 5.5 shows the dependencies of the lemmata presented throughout the remainder of this chapter. Note that dependencies on the lemmata located in the top left cluster are visualized as little black dots, to keep the visualization readable.
The tables give a short description of the issues tackled by the lemmata, corollaries, and theorems in this chapter.

Cluster	Topic	Chapter
top left	General Prerequisites	5.5.1
top center	Invalid Shapes	5.6.2
bottom left	Downflows	5.6.3
bottom center	Left FRFs	5.6.4
right	Repairing the Solution	5.7

5.3 Important Definitions and Abbreviations

If not stated otherwise, the next sections will refer to a planarized graph $G = (V, E)$ with the dummy nodes $D(V)$, a planar topology T, and the face set F. G is based on the non–planar graph $G' = (V', E')$.
In order to make G and T more usable, we transform G into a *planar map* (cf. [26]).

Definition 5.1. Planar Map: *A planar map is a planar graph G with the following properties based on its planar topology T:*

- *Instead of any undirected edge $e_u = (u, v)$, there exist two directed* halfedges *$e_{d1} = (u, v)$ and $e_{d2} = (v, u)$. e_{d1} is said to be the* reversal *of e_{d2} and vice versa.*
- *For any node v there exists a counterclockwise ordering of all out-edges of v.*
- *A halfedge e_d is incident to a face f if and only if f lies to the left of e_d (considering the direction of the halfedge)*
- *For any face f there exists a counterclockwise ordering of all incident halfedges.*

We denote the set of halfedges of G by E_h for simpler distinction, and define the following functions:

source(e_d)	Source node of halfedge e_d
target(e_d)	Target node of halfedge e_d
rev(e_d)	Reversal of halfedge e_d
face(e_d)	Face to the left of halfedge e_d
nodesucc(e_d)	counterclockwise successor of e_d in source(e_d)
nodepred(e_d)	counterclockwise predecessor of e_d in source(e_d)
facesucc(e_d)	counterclockwise successor of e_d in face(e_d)
facepred(e_d)	counterclockwise predecessor of e_d in face(e_d)
IE(v)	$\{e_d \in E_h \mid \text{source}(e_d) = v\}$
IE(f)	$\{e_d \in E_h \mid \text{face}(e_d) = f\}$

The name IE for the latter two sets stands for *incident edges*.

Remark 5.1. *The node- and face–successor functions are related to each other by simple formulas:*

$$\text{facesucc}(e_d) = \text{nodepred}(\text{rev}(e_d))$$
$$\text{facepred}(e_d) = \text{rev}(\text{nodesucc}(e_d))$$

5.4 Simple–Podevsnef as LP

We first need to transform the alternative network of Simple–Podevsnef into an LP. After this transformation, we will be able to use it as a foundation for the upcoming enhancements.
The first step is to arrange the variables in two classes: the flow on edges from E_{vf} and E_{ff} is represented by the variable classes a and b, respectively. The single variables of each class are indexed by a halfedge (cf. Figure 5.6).

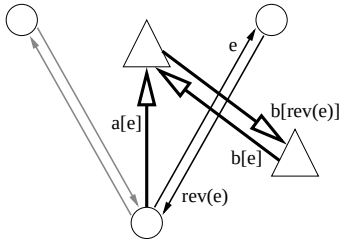


Figure 5.6: We use halfedges as an index for the flow variables; (circles denote nodes, triangles denote faces)

This leads to the objective function:

$$\min \sum_{e \in E_h} b_e \tag{5.1}$$

and the following constraints for the balance in the nodes:

$$\forall n \in V : \sum_{e \in \text{IE}(n)} a_e = 4 \tag{5.2}$$

We will ignore the described augmentation for high degree nodes for now – this will be handled by additional inequalities later on – and get a straight forward equality for the balances in the faces:

$$\forall f \in F: \sum_{e \in \mathrm{IE}(f)} \left(a_e + b_e - b_{\mathrm{rev}(e)} \right) = 2\deg(f) + B(f) \tag{5.3}$$

with

$$B(f) = \begin{cases} +4 & \text{if } f = f_0 \\ -4 & \text{otherwise} \end{cases}.$$

Instead of using the dummy nodes and additional edges of the augmentation, we simply add an inequality forcing the sum of the two otherwise joint edges to be greater or equal than 1:

$$\forall n \in H(V), \forall e \in \mathrm{IE}(n): \; a_e + b_e \geq 1 \tag{5.4}$$

The bounds are just as in the network:

$$\begin{aligned} a_e &\in \begin{cases} [1, 5 - \deg(\mathrm{source}(e))] & \text{if source}(e) \in L(V) \\ [0, 1] & \text{if source}(e) \in H(V) \end{cases} \\ b_e &\in [0, \infty] \end{aligned}$$

5.5 Collapsepaths, Hyperfaces, and Hyperedges

SPED will be able to let dummy nodes share the same grid point if applicable; this section analyzes the valid cases.
Therefore we will define a set of concepts and notions that will be used in the upcoming sections. This section will simultaneously proof that the presented concepts cover all possibilities for valid grid point sharing.
Section 5.5.1 will define certain layout notations and prove properties of these that will be needed in Section 5.6 and 5.7.

A *dummy merge* is the situation where we create a shape that forces two diverse dummy nodes to get mapped onto the same grid point. A bundle is a set of edges leaving a node on a common side; when such a bundle splits up – e.g. the right–most edge has a right bend whereas the others continue straight ahead – we call the situation a *demerge*. If a face is completely inside a bundle it is an empty face (cf. Sec. 4.1) and said to be *collapsed*; a face *demerges* if it is between two demerging edges (see Fig. 5.7).
We want bundles to be able to cross other edges and bundles without a demerge.

Definition 5.2. Metaedge: *The edge e' in the non–planar graph G' that has been split up into the edges $e_1, e_2, \ldots, e_k$ in the planarized graph G by adding dummy nodes during the planarization process, is called* metaedge *of these subedges.*

We know that the following properties hold:

1. Edge bundles are bundles of metaedges.
2. Edge bundles start in a common node – the source node s – and leave it in parallel.
3. Bundled edges cannot have a common target node.

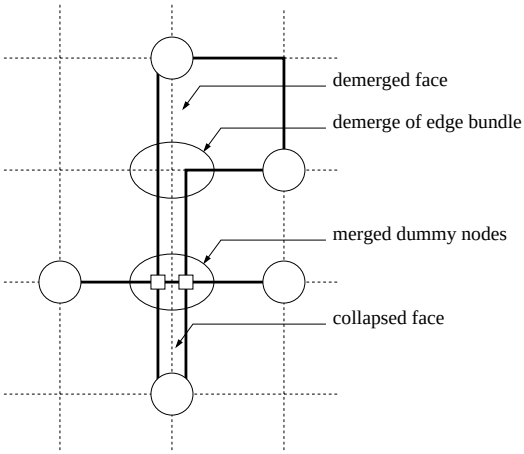


Figure 5.7: Example for a dummy merge, a demerge, and a collapsed face

4. The degree of the source node is greater than four, more precisely $s \in H(V) \Leftrightarrow \deg(s) > 4$

5. Dummy nodes have a degree of four, that is, $v \in D(V) \Rightarrow \deg(v) = 4$ (see Section 3.2.1)

Ad 3) Let there be an edge bundle consisting of the metaedges $e'_1 = (s, u)$ and $e'_2 = (s, v)$. If the bundle had a common target node t, we would have $u = v = t$. Thus essentially we would have two metaedges (s, t), which would be a contradiction to the assumption that G' is a simple graph (see the preconditions in Section 4.2.2).
Ad 4) According to the rules of Simple–Podevsnef, the outgoing edges are distributed amongst the four orthogonal directions. The network allows 0° angles only if the degree of a node exceeds 4 (see Section 4.2.2). But as property *2)* points out, bundled edges leave their source node in a common direction, which directly leads to property *4)*.

On the basis of these properties we can derive the following two structures of faces that potentially may collapse:

1. *collapsible triangle ("coltri"):* The face has exactly one incident node with a degree greater than four (the source node of a bundle), and two incident dummy nodes.

2. *collapsible quod ("colquod"):* The face has exactly four incident dummy nodes.

These observations lead to the following definitions (cf. Figure 5.8):

Definition 5.3. Collapsepath: *A connected sequence of faces, starting with a coltri (node v as its non–dummy node, with its out–edges e_1, e_2), containing several (or no) colquods and ending with an arbitrary face (not a colquod), is called* potential collapsepath in the dual graph*, or simply* collapsepath.
All faces in such a path have to be incident to the metaedges m_1 and m_2 of e_1 and e_2, respectively.

Faces of a collapsepath that are neither collapsed nor demerging are *demerged*. Each validly drawn collapsepath can be separated into three continuous parts: it starts with

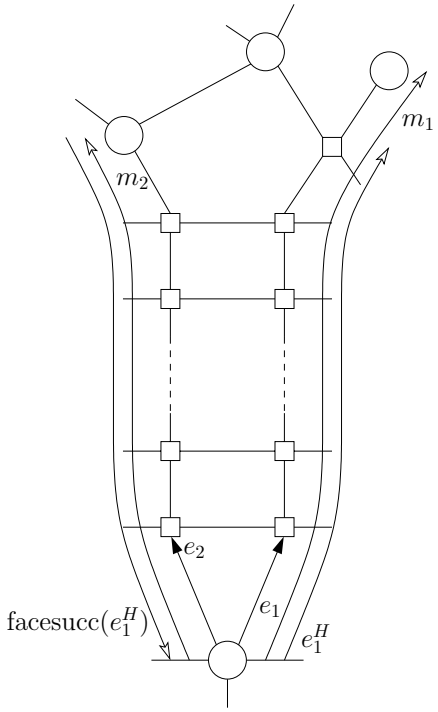


Figure 5.8: Metaedges (m_1, m_2), collapsepath, hyperface, and hyperedge (e_1^H); (circles denote original nodes, squares are dummy nodes)

collapsed faces (if any), followed by exactly one demerging face; the rest of the collapsepath consists of demerged faces (if any). A coltri with an opening degree of 90° is also called *demerging*.

Remark 5.2. *The end face of a collapsepath cannot be another coltri, since an edge bundle cannot have a common target node.*
A triangular end face consisting of three dummy nodes is theoretically possible but highly implausible for any reasonable planarization, since this constitutes an unnecessary crossing (cf. Fig. 5.9).

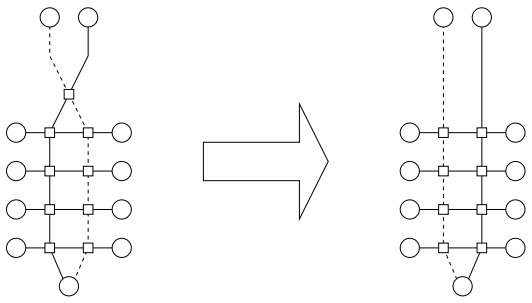


Figure 5.9: A triangular end face is based on a bad planarization that includes an unnecessary crossing.

Definition 5.4. Hyperface: *Let f be a coltri. We define the set of all faces in the collapsepath starting at f as the* hyperface f^H.

The right side of such a hyperface is of special interest; it will be used for the (modified) right–bend demand inherited from Simple–Podevsnef (see Section 5.6.1). We define this right side more exact as follows:

Definition 5.5. Hyperedge: *Let e be a halfedge with* $\deg(\text{source}(e)) > 4$, *and* $\text{face}(e)$ *being a coltri (thus being the starting point of a collapsepath). We denote the set of halfedges e' for which the following constraints hold the* hyperedge e^H*:*

- *The metaedge of the halfedge e' is identical to the metaedge of e*
- *The halfedge e' is incident to a face of the collapsepath*

For the sake of consistency, the symbol e^H describes a set with the single element e for all e that do not satisfy the properties demanded by the definition. We can formulate this more mathematically using a recursive definition:

$$\begin{aligned} e^H &= \{e\} \cup \{e_2 \mid \exists e_1 \in e^H, \text{rev}(\text{nodesucc}(\text{nodesucc}(e_2))) = e_1, \\ &\quad \text{target}(e_1) \in D(V), \text{facesucc}^4(e_1) = e_1, \text{target}(\text{facesucc}(e_1)) \in D(V)\} \\ f^H &= \{f' \mid \exists e' \in e^H, \text{face}(e') = f'\} \end{aligned}$$

Remark 5.3. *Thanks to the mathematical definition it is easy to see that $|f^H| = |e^H|$. Furthermore we obviously have a 1-1 mapping between faces in f^H and halfedges in e^H. To show that this of course also holds for the textual Definitions 5.4 and 5.5, we have to recognize that (after any reasonable planarization) there always exists only one single halfedge in the last face of the collapspath that has the same metaedge as e.*
Even though it is a contradiction to the TSM–model as described in Section 3.2, it may be interesting to deal with unorthodox planarization methods that do not try to minimize the number of crossings. In that case, the exact and unambiguous mathematical definition is to be preferred over the textual.

Furthermore we define the *facepred*–function for hyperedges (cf. Fig. 5.8), as well as their source and target nodes.

$$\begin{aligned} \text{facepred}(e^H) &= \Big\{e_1 \mid \exists e_2 \in e^H, \Big(\text{source}(e_2) \in H(V) \ \wedge \ \text{facepred}(e_1) = e_2\Big) \\ &\quad \vee \Big(\text{source}(e_2) \notin H(V) \ \wedge \ \text{facepred}^2(e_1) = e_2\Big)\Big\} \end{aligned}$$

$$\begin{aligned} \text{source}(e^H) &= \text{source}(e) \\ \text{target}(e^H) &\in \{\text{target}(t) \mid t \in e^H, \nexists r \in e^H : \ \text{target}(r) = \text{source}(t)\} \end{aligned}$$

$$\begin{aligned} \text{source}(\text{facepred}(e^H)) &\in \{\text{source}(t) \mid t \in e^H, \nexists r \in e^H : \ \text{source}(r) = \text{target}(t)\} \\ \text{target}(\text{facepred}(e^H)) &= \text{source}(e^H) \end{aligned}$$

Note that $\text{target}(e^H)$ and $\text{source}(\text{facepred}(e^H))$ cannot be identical for any e^H.
To ease the textual descriptions, the left side of a hyperface will be called *bundle partner* of the respective hyperedge and is defined as:

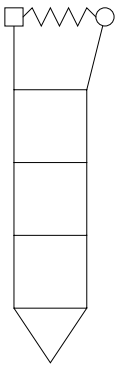


Figure 5.10: Symbolic notation for hyperfaces

Definition 5.6. Bundle Partner of a Hyperedge: *The set that contains the reversals of the elements of* facepred(e^H) *is considered the* bundle partner *of* e^H*:*

$$partner(e^H) = \{\text{rev}(e_1) \mid e_1 \in \text{facepred}(e^H)\}$$

We also define a *partner*–function that gives the corresponding subedge on the bundle partner for any subedge $e' \in e^H$:

$$\text{partner}(e') = \begin{cases} \text{rev}(\text{facepred}(e')) & \text{if source}(e') \in H(V) \\ \text{rev}(\text{facepred}^2(e')) & \text{otherwise} \end{cases}$$

Remark 5.4. *Alternatively to the formula given above, a bundle partner can also be described by this equivalent definition:*

$$\text{partner}(e^H) = \{\text{partner}(e_1) \mid e_1 \in e^H\}$$

Remark 5.5. *Note that there is an alternative method to calculate the partner of a subedge for all subedges of a hyperedge except the last one:*

$$\forall e' \in e^H \text{ with } \text{target}(e') \neq \text{target}(e^H) : \text{ partner}(e') = \text{rev}(\text{facesucc}^2(e'))$$

Remark 5.6. *Note that hyperfaces may overlap each other. In particular, such an overlap in a colquod is a potential crossing of two edge bundles.*

Since the next chapters include various figures containing hyperfaces, we will use a simplified drawing method to symbolize them (Fig. 5.10): If a hyperface contains an arbitrarily complex end face, this face is drawn using a zigzag line. If necessary, the vertices on the border are marked according to their type: a square symbolizes a dummy node, a circle is an original node and a diamond stands for an arbitrary/unknown type of node.

5.5.1 Definitions and Lemmata Regarding Relations Between Hyperfaces

This section presents some important lemmata that will be needed as a background for the next sections. Many proofs and methods will rely on shifting flow to the left or to the right of a hyperface. Hence the first lemma will assure that such shifts cannot cycle around any high degree node. This will be further clearified in Section 5.6.

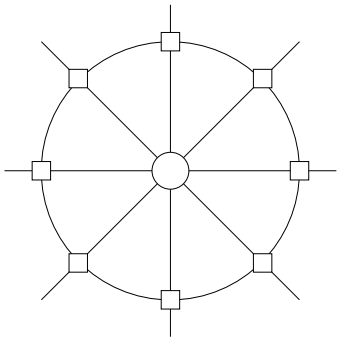


Figure 5.11: A node surrounded by coltris

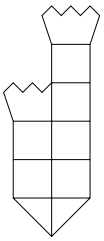


Figure 5.12: Example of neighboring hyperfaces

The definitions and lemmata afterwards will define certain relations between hyperfaces based on their layout. We will need definitions like *neighboring hyperfaces* to effectively enumerate all possibilities of certain error–prone situations. And we have to assure that, e.g., each hyperface can only have one neighbor to its left.

Lemma 5.1. *A high degree node cannot be completely surrounded by coltris.*

Proof. (Indirect) Let us assume there would be a node v with $\deg(v) > 4$, completely surrounded by coltris (Fig. 5.11). We know that a coltri consists of one high degree node (v) and two dummy nodes. Thus all adjacent nodes to v would be dummy nodes. Since the faces are coltris, all these dummy nodes would be connected to one single circle around v.
Considering their order, every second edge around a dummy node belongs to the same metaedge. Thus it is obvious that the edges between the dummy nodes of two adjoining coltris are subedges of the same original edge. It follows that the described circle of edges around v is merely one single metaedge. But as this edge has neither a source nor a target node, it represents a contradiction to the definition of edges of graphs. □

Definition 5.7. Neighboring Hyperfaces: *Two hyperfaces f_1^H, f_2^H are* neighbors*, iff the intersection of the corresponding hyperedge e_1^H with the bundle partner of the respective hyperedge e_2^H is not an empty set, or vice versa:*

$$\exists e' \in e_1^H : e' \in \text{partner}(e_2^H) \vee \exists e' \in e_2^H : e' \in \text{partner}(e_1^H)$$

Figure 5.12 gives an example of neighboring hyperfaces.
The following lemma ensures that all neighboring hyperfaces share a common source node, each hyperface can only have one neighboring hyperface on each side, and the neighborship cannot be interrupted or discontinued until one of the hyperfaces ends.

Lemma 5.2. *Neighboring hyperfaces always share a common source node and are neighbors during the entire length of the shorter hyperface of the two.*

Proof. Let e_1^H and e_2^H be two hyperedges with $\exists e \in e_1^H : p = \text{partner}(e) \in e_2^H$. Especially, let e be the halfedge farthest aways from the coltri of e_1^H meeting this requirement.
The proof of the Lemma is a direct result of the following sub–proofs:

1. Either e or p are subedges incident to the end face of their respective hyperface.
2. If p is the partner edge of e, this also holds for their respective predecessors p' and e' in their hyperedges. (This sub–proof may be applied several times by renaming e' into e and p' into p until the next point applies.)
3. If $e' = e_1$, the source nodes of e' and p' are identical.

These sub–proofs represent an iterative proof: (1) is the initial property, (2) is the invariant of the implicit loop, (3) is the termination condition. The validity of these three steps induces the validity of the lemma.
Sub–Proof 1 – Initial Property: (Indirect) Let us assume that both e and p have a successor in their respective hyperedges. Thus both target nodes of e and p are dummy nodes. We can also deduce that e is either incident to a coltri or to a colquod; hence we can calculate the partner p by the formula $\text{rev}(\text{facesucc}^2(e))$ as described in Remark 5.5. The successors can be calculated by

$$\bar{e} = \text{nodepred}^2(\text{rev}(e)) \tag{5.5}$$

$$\bar{p} = \text{nodesucc}^2(\text{rev}(p))$$

And we can deduce from the latter:

$$\bar{p} = \text{nodesucc}^2(\text{rev}(\text{rev}(\text{facesucc}^2(e))))$$

Since we know that $\text{rev}(\text{rev}(x)) = x$ this leads to

$$\bar{p} = \text{nodesucc}^2(\text{facesucc}^2(e)) \tag{5.6}$$

By transforming (5.5), we get

$$\text{nodesucc}^2(\bar{e}) = \text{rev}(e)$$

$$e = \text{rev}(\text{nodesucc}^2(\bar{e}))$$

which we can apply to (5.6)

$$\bar{p} = \text{nodesucc}^2(\text{facesucc}^2(\text{rev}(\text{nodesucc}^2(\bar{e}))))$$

By the definition of $\text{facesucc}(x) = \text{nodepred}(\text{rev}(x))$ we know

$$\bar{p} = \text{nodesucc}^2(\text{nodepred}(\text{rev}(\text{nodepred}(\text{rev}(\text{rev}(\text{nodesucc}^2(\bar{e})))))))$$

Due to $\text{nodesucc}(\text{nodepred}(x)) = x$ and $\text{rev}^2(x) = x$, this can be simplified to

$$\bar{p} = \text{nodesucc}(\text{rev}(\text{nodepred}(\text{nodesucc}^2(\bar{e}))))$$

$$\bar{p} = \text{nodesucc}(\text{rev}(\text{nodesucc}(\bar{e})))$$

This can be extended to

$$\bar{p} = \text{rev}(\text{rev}(\text{nodesucc}(\text{rev}(\text{nodesucc}(\bar{e})))))$$

so that we can apply the definition $\text{facepred}(x) = \text{rev}(\text{nodesucc}(x))$ two times:

$$\bar{p} = \text{rev}(\text{facepred}^2(\bar{e}))$$

Since $\bar{e} \neq e_1$, this resolves into the definition of partners

$$\bar{p} = \text{partner}(\bar{e})$$

which is a contradiction to the assumption of e being the edge farthest away from its coltri that neighbors e_2^H.

Sub–Proof 2 – Invariant of the Loop: The predecessors e' and p' can naturally be defined as:

$$e' = \text{rev}(\text{nodesucc}^2(e))$$
$$p' = \text{rev}(\text{nodepred}^2(p))$$

which can be transformed into:

$$\text{nodepred}^2(\text{rev}(e')) = e \tag{5.7}$$
$$\text{nodesucc}^2(\text{rev}(p')) = p \tag{5.8}$$

Since we know that $p = \text{partner}(e)$ and $e \neq e_1$, we can write

$$p = \text{rev}(\text{facepred}^2(e))$$

and apply (5.7) and (5.8).

$$\text{nodesucc}^2(\text{rev}(p')) = \text{rev}(\text{facepred}^2(\text{nodepred}^2(\text{rev}(e'))))$$

By the definition $\text{facepred}(x) = \text{rev}(\text{nodesucc}(x))$ we get

$$\text{nodesucc}^2(\text{rev}(p')) = \text{rev}(\text{rev}(\text{nodesucc}(\text{rev}(\text{nodesucc}(\text{nodepred}^2(\text{rev}(e'))))))))$$

which we simplify to

$$\text{nodesucc}^2(\text{rev}(p')) = \text{nodesucc}(\text{rev}(\text{nodepred}(\text{rev}(e'))))$$
$$p' = \text{rev}(\text{nodepred}^2(\text{nodesucc}(\text{rev}(\text{nodepred}(\text{rev}(e'))))))$$
$$p' = \text{rev}(\text{nodepred}(\text{rev}(\text{nodepred}(\text{rev}(e')))))$$

Since $\text{nodepred}(\text{rev}(x)) = \text{facesucc}(x)$ we get

$$p' = \text{rev}(\text{facesucc}^2(e'))$$

Because we know that e' – being the predecessor of e on the hyperedge – cannot be the last subedge of the hyperedge, this is identical to the alternative definition of the *partner*–function given in Remark 5.5:

$$p' = \text{partner}(e')$$

Sub–Proof 3 – Termination Condition: If $e' = e_1$, $p' = \text{partner}(e')$ will resolve into $\text{rev}(\text{facepred}(e'))$. Since the *facepred*–function returns an edge whose target node is identical to the source node of the parameter and the *rev*–function basically flips source and target, it is obvious that p' will have the same source node as e'. □

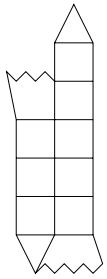


Figure 5.13: Example of opposing hyperfaces

Definition 5.8. Opposing Hyperfaces: *Two hyperfaces f_1^H, f_2^H are* opposing*, iff the intersection of the according hyperedge e_1^H with the reversals of the respective hyperedge e_2^H is not an empty set:*

$$\exists e' \in e_1^H : \mathrm{rev}(e') \in e_2^H$$

Figure 5.13 gives an example of opposing hyperfaces.
The following lemma ensures that all hyperfaces can only have one opposing hyperface on each side, and that their opposing status can not be interrupted. E.g., it is impossible that the end face and some face in the middle of a hyperface f_1^H are opposing the hyperface f_2^H, but the subface directly prior to the end face is not.

Lemma 5.3. *The set of common edges of two opposing hyperfaces is always the smallest continuous subset that includes the edges adjacent to the two end faces.*

Proof. Let e_1^H and e_2^H be two hyperedges with $\exists e_A \in e_1^H : r_A = \mathrm{rev}(e_A) \in e_2^H$ and $\exists e_B \in e_2^H : r_B = \mathrm{rev}(e_B) \in e_1^H$. Especially, let e_A and e_B be the halfedges farthest away from the coltris of their respective hyperfaces meeting this requirement.
The proof of the lemma is a direct result of the following sub–proofs:

1. The subedges e_A and e_B are adjacent to the end face of their respective hyperface.
2. If a halfedge e is an element of one of the above hyperedges and its reversal r is part of the other hyperedge, this also holds true for the predecessor of e $(= e')$ and the successor of r $(= r')$ on their respective hyperedges, if r' exists.[5]

These sub–proofs represent again an iterative proof: (1) is the initial property, (2) is the invariant of the implicit loop; the latter includes the termination condition. The validity of these two steps induces the validity of the lemma.
Sub–Proof 1 – Initial Property: (The proof will only consider e_A since the proof for e_B is analogous.)
(Indirect) Assume e_A would not be the last subedge of its hyperedge. Thus we know $\mathrm{target}(e_A) \in D(V)$ and we can calculate the successor $\bar{e} = \mathrm{nodesucc}^2(\mathrm{rev}(e_A))$. Since e_A is defined to be the subedge farthest away from its corresponding coltri satisfying the demanded property, we know: $\bar{r} = \mathrm{rev}(\bar{e}) \notin e_2^H$. Thus $\bar{r}$ cannot be the predecessor of r_A. That could either happen because of the non–existence of such a predecessor – thus $r_A = e_2$ – or by $\hat{r} \neq \bar{r}$ ($\hat{r}$ being the predecessor of r_B). Let us consider both cases:

[5]Note that the existence of e' based on the existence of r' follows from the first sub–proof.

- $r_A = e_2$: Thus source$(r_A) \in H(V)$, in particular source$(r_A) \notin D(V)$. Since rev$(r_A) = e_A$, this would lead to target$(e_A) \notin D(V)$, which would be a contradiction to target$(e_A) \in D(V)$.
- $\hat{r} \neq \bar{r}$: Since we can calculate

  $$\hat{r} = \text{rev}(\text{nodesucc}^2(r_A))$$

  and we know

  $$\bar{r} = \text{rev}(\bar{e}) = \text{rev}(\text{nodesucc}^2(\text{rev}(e_A)))$$

  we can apply that to the given inequality:

  $$\text{rev}(\text{nodesucc}^2(r_A)) \neq \text{rev}(\text{nodesucc}^2(\text{rev}(e_A)))$$

  Because of $r_A = \text{rev}(e_A)$, this can be expressed as

  $$\text{rev}(\text{nodesucc}^2(\text{rev}(e_A))) \neq \text{rev}(\text{nodesucc}^2(\text{rev}(e_A)))$$

  Since this is obviously wrong, the assumption that e_A would not be the last subedge of the hyperface is invalid.

Sub–Proof 2 – Invariant of the Loop, Termination Condition: Since we assume the existence of r' we calculate it – as well as e' – by

$$r' = \text{nodesucc}^2(\text{rev}(r)) \Rightarrow r = \text{rev}(\text{nodepred}^2(r'))$$

$$e' = \text{rev}(\text{nodesucc}^2(e)) \Rightarrow e = \text{nodepred}^2(\text{rev}(e'))$$

and since we know that $\text{rev}(e) = r$ we can join these as

$$\text{rev}(\text{nodepred}^2(\text{rev}(e'))) = \text{rev}(\text{nodepred}^2(r'))$$

$$\text{nodepred}^2(\text{rev}(e')) = \text{rev}(\text{rev}(\text{nodepred}^2(r')))$$

$$\text{nodepred}^2(\text{rev}(e')) = \text{nodepred}^2(r')$$

$$\text{rev}(e') = \text{nodesucc}^2(\text{nodepred}^2(r'))$$

$$\text{rev}(e') = r'$$

Thus e' and r' are reversals of each other. □

Definition 5.9. Orthogonal Hyperfaces: *Two hyperfaces f_1^H and f_2^H are said to be* orthogonal*, iff their intersection contains a colquod:*

$$\exists f' \in f_1^H \cap f_2^H : f' \textit{ is a colquod}$$

Figure 5.14 gives an example of such orthogonal hyperfaces.

Remark 5.7. *The cardinality of the intersection of two hyperfaces f_1^H and f_2^H never exceeds 1 for any reasonable planarization ($|f_1^H \cap f_2^H| \leq 1$).*
Figure 5.15 shows a situation with the cardinality of 2: each of the two metaedges of the hyperface to the right would cross the metaedges of the left hyperface twice. This is a suboptimal planarization that normally will not be produced.

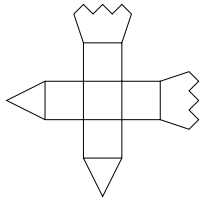


Figure 5.14: Example of orthogonal hyperfaces

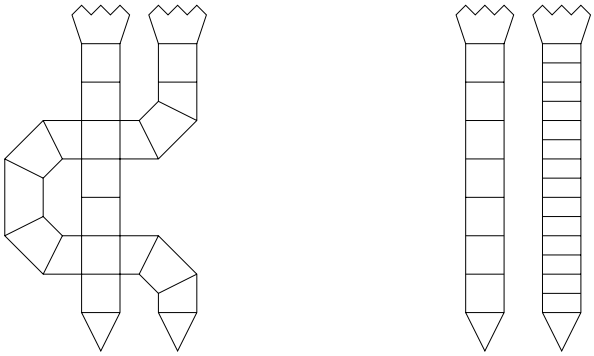


Figure 5.15: Example of a quite cumbersome planarization, and its usual counterpart

5.6 Constructing a Solution

We can now look at the solution for the task given in Section 5.1. As initially explained, we will extend the LP of Simple–Podevsnef with further inequalities. These add–ons always produce a "nearly correct" solution; there exists a repair–function with polynomial time complexity that is able to generate a valid solution with the same objective value based on it.
We know the faces that may collapse, and that these faces are organized as hyperfaces. Thus we can now start with the extension:

5.6.1 Right Bend on Hyperedges

Alternatively to its mathematical definition as a set, a hyperface can be thought of as one homogeneous face consisting of its subfaces (with the incident edges of its subfaces, not considering the edges that are shared between these subfaces).
Naturally, the outline of this hyperface has to be a valid drawing according to the classic Simple–Podevsnef standard. Thus, if the start face of the according collapsepath has a 0° angle that forces the adequate metaedges to leave the source node in parallel, the according hyperedge has to have at least one right bend.
Such an extension is not hard to integrate into the given ILP. We just have to modify Constraint (5.4) of the LP given in Section 5.4: Let e be the halfedge that is both element of the hyperedge and incident to the start face of the collapsepath. If there is a 0° angle to the left of e on its source node, Simple–Podevsnef would force a right bend on e. Instead of this we now force a right bend on its hyperedge.

So instead of the original (5.4)

$$\forall n \in H(V), \forall e \in \mathrm{IE}(n) : \; a_e + b_e \geq 1$$

we now get

$$\forall n \in H(V), \forall e \in \mathrm{IE}(n) : \; a_e + \sum_{t \in e^H} b_t \geq 1 \tag{5.9}$$

Unfortunately, this extension is not sufficient since it does not exclude invalid shapes inside the hyperfaces. We have to add some more constraints to ensure correct drawings of them. Furthermore, we cannot integrate this extension into a normal min–cost–flow network. See Section 6.1 for details.

5.6.2 Possible Flaws in Hyperfaces

Dummy nodes have nearly no influence in guaranteeing valid shapes of hyperfaces. Hence various situations that cause incorrect drawings can happen. We will show the number of these possibilities by induction on the structure of a hyperface. The text will refer to this induction and summarize it in Theorem 5.1.
Let a hyperface f^H consist of a coltri f_0, $n-1$ colquods $f_1, ..., f_{n-1}$, and a final face f_n. Let $e_0, ..., e_n$ be the respective subedges of f^H and $p_0, ..., p_n$ their partner edges.
An undrawable flow solution may only occur on a stage of the hyperface where it is still collapsed. So if the opening angle of a coltri is 90°, the face is trivially always drawable (just as in Simple–Podevsnef). Errors may only occur in case of 0° angles.
We have to consider all stages until a hyperface is demerged. A split–up can be achieved by two different situations:

- If there is a right bend on the halfedge e_i, the face f_i demerges, and $f_{i+1}...f_n$ are demerged. Hence those faces are always drawable.
- If there is a left bend on the edge p_i, we know that the faces $f_{i+1}...f_n$ are demerged. But the face f_i is still *partially collapsed*:
  As in Simple–Podevsnef, we demand that right bends happen *before* any left bends on any edge that contains bends of both types (giving up this property would lead to much more undesirable complications). This means that a right bend on p_i would happen *before* the demerge of the bundle and thus invalidate the drawing (see below). Nevertheless, wrong bends on e_i or on the edge separating f_i from f_{i+1} cannot produce errors anymore.

Our induction starts with the coltri, and we explore the possibilities of producing errors within that face:

- If the coltri is not collapsed, it – as well as all following subfaces – is free of errors.
- If the coltri is collapsed, the following errors may occur:
  1. At least one left bend on e_0
  2. At least one right bend on p_0
  3. At least one bend on the edge the coltri shares with f_1

In the third case we distinguish between *upflows* (a flow from f_0 to f_1) and *downflows* (f_1 to f_0).
Furthermore, we have to consider that the coltri demands two units, following its network definition; it obviously gets them from its two incident dummy nodes. Since the coltri is starting with a 0° angle, the incident original node delivers no units. Thus the coltri is balanced by default. This implies that any inflow has to have a corresponding outflow (and vice versa).
Thus only the cases shown in Fig. 5.16 exist. The rows show all possibilities where flows that could potentially produce errors are involved. The first row shows left bends on e_0, the second shows right bends on p_0; the rows three and four display up- and downflows. The cases where errors actually occur are encircled. Since some situations recur, one of each error type is boldly encircled to make it more recognizable.
Note that all situations that include a downflow generate an invalid shape; all cases incorporating an upflow – if there is no downflow – demerge the face.
A situation in which a unit is sent in both directions over a single edge is called *FlowReFlow* (or *FRF* for short). If this FRF includes a downflow, we call it *updown–FRF* (Fig. 5.16, second of fourth row); if the FRF is on the partner edge of a hyperface, we call it *left–FRF* (Fig. 5.16, last of second row). These cases will be analyzed in greater detail below.
We summarize this observation as a lemma:

Lemma 5.4. *There are only two different circumstances that can invalidate the shape of a collapsed coltri: a downflow into the coltri and a left–FRF on it.*

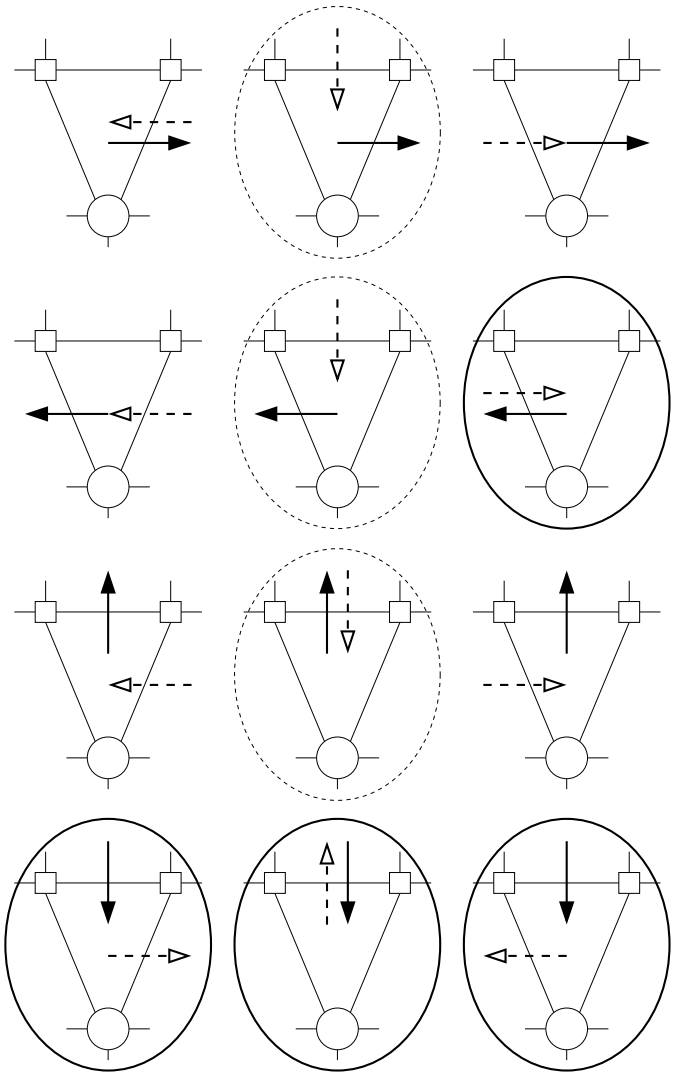
Proof. Obvious, based on the above observations. □

As our induction step, we look for the occurrence of an error up to a certain stage i ($i > 0$). An invalid shape can either be caused by a stage below (so it will have been recognized in stage $i-1$) or by an invalid flow in this very stage.
The erroneous cases on stage i are analogous to the cases in the coltri; the following situations may induce errors:

1. At least one left bend on e_i
2. At least one right bend on p_i
3. At least one bend on the edge the colquod f_i shares with f_{i+1}

- We do not consider errors produced by bends on the edge f_i shares with f_{i-1} since they were considered in stage $i-1$.

Since a colquod has a demand of four units which it retrieves from its four incident dummy nodes, every inflow into the face has to have a corresponding outflow; that is the same property as before when analyzing the coltris.
Essentially, we can reuse Fig. 5.16: for every row we only have to add the cases where the in-/outflow runs over the edge underneath (between f_i and f_{i-1}). These are quite simple: if the flow goes from f_i to f_{i-1}, it basically was a downflow in the stage $i-1$ and has already generated an error. But if the flow goes from f_{i-1} to f_i it was an upflow in the stage below, thus f_i is not collapsed at all.
We use this argument for another lemma:



(a) The erroneous flows are encircled; since identical errors recur, one of each type is boldly marked.

row	left	center	right
1	V right bend on e	E bend on q, left bend on e	V left bend on p
2	V right bend on e	E bend on q, right bend on p	E right bend on p before left bend
3	V right bend on e	E bends on q	V left bend on p
4	E bend on q, left bend on e	E bends on q	E bend on q, right bend on p

(b) The reasons; V = valid, E = error; e = subedge of hyperedge, p = partner(e), q = top edge of coltri

Figure 5.16: Sources of errors in a coltri

Lemma 5.5. *There are only three different circumstances that can invalidate the shape of a collapsed colquod: an error in preceding faces of its collapsepath, a downflow into the colquod and a left–FRF on it.*

Proof. Obvious, considering the above observations. □

Based on the two lemmata above, we can formulate a theorem on the possible errors:

Theorem 5.1. *There are only two different circumstances that can invalidate the shape of a hyperface f^H: downflows into, and left–FRFs on collapsed subfaces of f^H.*

Proof. (Inductive) We use Lemma 5.4 as the induction start. Lemma 5.5 is the induction step. The final step of the inductive proof considers the end face f_n of the hyperface. If the faces $f_0 ... f_{n-1}$ below are all still collapsed, there has not been any right bend on the hyperedge yet. Since the definition of the ILP demands at least one such bend on the hyperedge, it has to happen on e_n. Thus f_n is never a collapsed face and can never produce any errors. □

By this induction, we now know the possible structures of all errors that may occur. It is easy to see that a downflow is involved in most of the errors; the only error without it is a FlowReFlow on the bundle partner.
Alternatively we can distinguish the errors if they are caused by an FRF-situation or not. It will be pointed out that the invalid FRFs are especially complicated to get rid of.

5.6.3 Restricting Downflows

Since all downflow situations seem to cause trouble, one may be tempted to simply forbid downflows completely. But this would be too much: actually, downflows are necessary and turning them off would cause the algorithm to cut off good and valid solutions, as this section will point out. Downflows are only bad if they are directed into a collapsed face. As soon as a hyperface demerges, downflows have to be allowed. Since such an implication would result in a quite complex constraints, we have to analyze the downflows more thoroughly.
As we have seen in the previous section, any downflow into a collapsed face introduces an error. Hence it only arises in a valid solution if the lower of the two involved faces is demerging or demerged.
Let us assume that we have such a hyperface including a valid non–FRF downflow. We can distinguish between several classes of situations (Fig. 5.17):

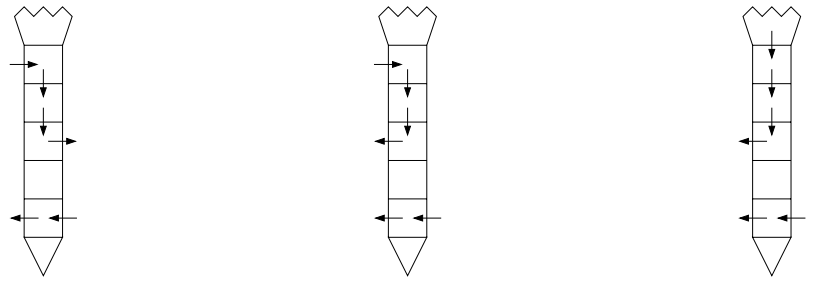
Class 1 Downflows: Downflow that moves from one side to the other (from left to right/from right to left)

Class 2 Downflows: Downflow attached only to the left/right

Class 3 Downflows: Downflow coming from the end face and leaving to the left/right

We will need the following remark for the task of analyzing these classes in more detail:

Remark 5.8. *We know that hyperfaces can neither start nor end with a colquod. If a colquod is shared by two hyperfaces (orthogonal to each other), the adjacent faces above and below are always part of one of the hyperfaces, as well as the faces to its left and right are automatically part of the other hyperface.*



(a) Class 1 (b) Class 2 (c) Class 3

Figure 5.17: Classes of valid non–FRF downflows

Class 1 Downflows

Lemma 5.6. *For any valid and optimal solution with class 1 downflows, there exists a solution with equal objective value but without any class 1 downflow. Furthermore, such a related solution does not introduce new class 2 downflows.*

Proof. All situations that include class 1 downflows are not really necessary as there always exists an equally rated solution without them (see Fig. 5.18). These alternative solutions can be generated by moving the downflow to the side of the higher entry point into the hyperface. Moving it to the other side may not result in a valid solution, as shown in Figure 5.19.
There exist situations where none of the two sides of a bundle contains the lowest part of the considered flow path (Fig. 5.20, left). They happen if a class 1 downflow is combined with an updown–FRF: Fig. 5.20(right) shows that the downflow can be shifted as described above and a pure FRF remains.
The described shifts do not introduce class 2 downflows, but they may generate class 3 downflows (Fig. 5.21). □

Class 2 Downflows

Lemma 5.7. *For any valid and optimal solution with class 2 downflows, there exists a solution with equal objective value but without any class 2 downflow. Furthermore, such a related solution does not introduce new class 1 downflows.*

Proof. We have to prove the lemma for both subclasses: the one attached to the right and the one attached to the left of the hyperface.
The former is particularly simple: the hyperface has to be demerged when a downflow happens. Thus such a detour for the path that contains the downflow is only reasonable if the flow is needed to generate a right bend on an opposing hyperface (as shown in Fig. 5.22). We can simply shift the downflow to the right and create an FRF for the opposing hyperface. This FRF can always be placed where the flow left the opposing hyperface to enter into the hyperface that contained the downflow.
If the downflow is attached to the left, the approach is quite similar (Fig. 5.23). The detour for the path that contains the downflow is only reasonable, if there is a neighboring hyperface to its left. Hence we can move the downflow there and generate an FRF.
This imaginary construction produces a new downflow situation in the hyperface to the left. This downflow can only be either a class 2 downflow attached to the left or a class

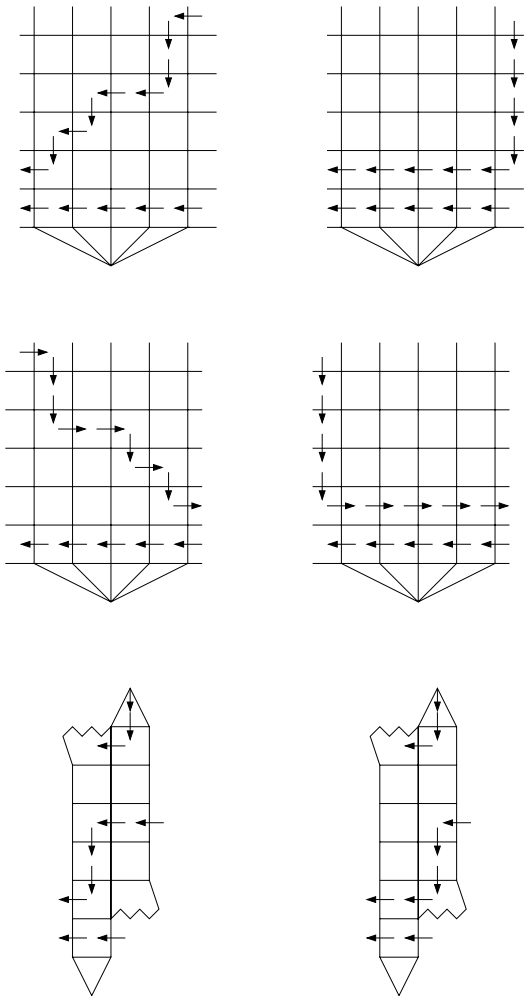


Figure 5.18: Class 1 downflows are not necessary; (left: with, right: without downflows)

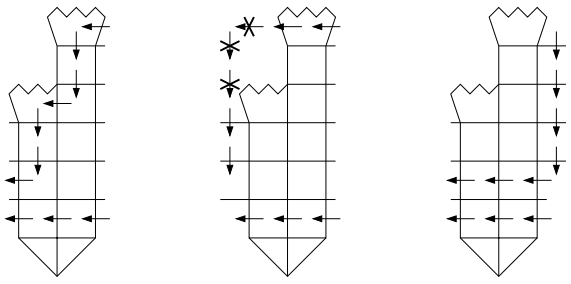


Figure 5.19: There always exists a related downflow–free solution that has the former downflows on the correct side of the bundle

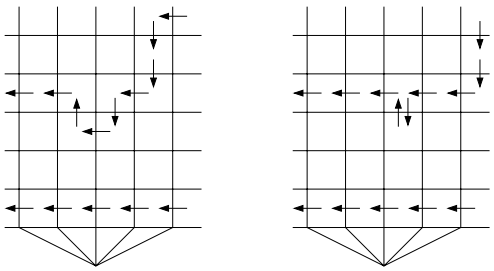


Figure 5.20: Solvable without non–FRF–downflows

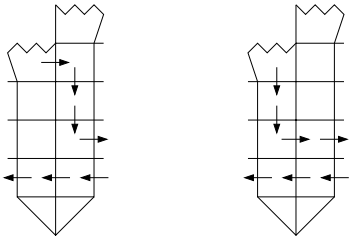


Figure 5.21: A class 1 downflow can resolve into a class 3 downflow

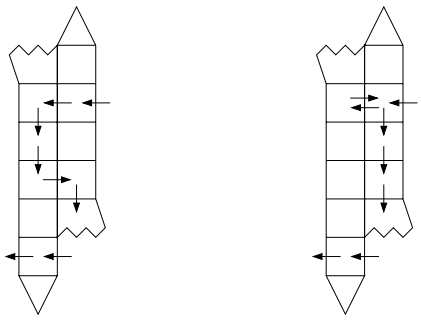


Figure 5.22: Class 2 downflow attached to the right and the related downflow–free solution

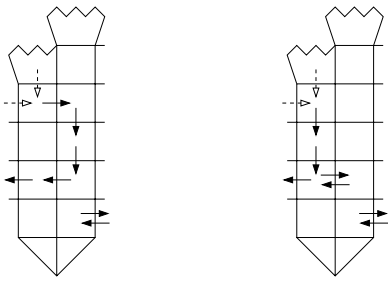


Figure 5.23: Class 2 downflow attached to the left and the related downflow–free solution

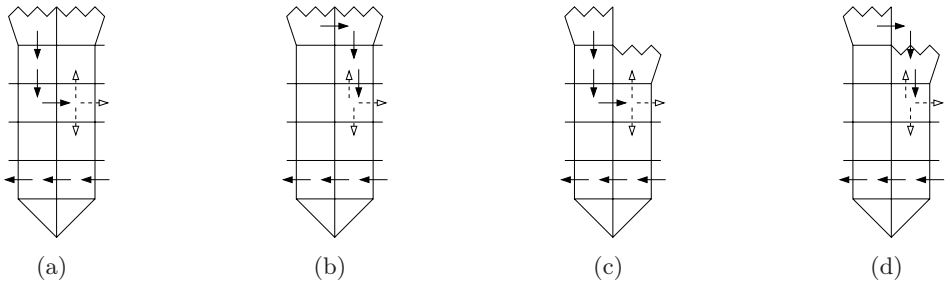


Figure 5.24: Repetitive modifications (a)→(b) and (c)→(d), until there are no downflows left

3 downflow leaving to the left (Note that it cannot be a class 1 downflow from the right to the left). If it is the former one, the same construction will be reapplied to this new downflow on the left hyperface (and perhaps again to the hyperface left to the left one, and so on); but this repetition will always stop with a class 3 downflow leaving to the left since otherwise all class 2 downflows before would have been useless (which would be a contradiction to the optimality of the solution). □

Class 3 Downflows

Lemma 5.8. *For any valid and optimal solution with class 3 downflows, there exists a solution with equal objective value but without any class 3 downflow. Furthermore, such a related solution does neither introduce new class 1 nor new class 2 downflows.*

Proof. As before, we have to prove this property for the two subclasses differing in the side where the flow leaves the hyperface:

1. *Flow leaves to the right; there is a neighboring hyperface to the right:* We shift the downflow to the right. This modification alone would generate a new class 1 downflow which would resolve back into the original class 3 downflow by Lemma 5.6. Thus this shift cannot be the only modification: we will apply it repetitively until there is no neighboring hyperface to the right that satisfies one of the properties below. Hence after this process, we can apply the modification given in case *(2)* ("Flow leaves to the right; there is no neighboring hyperface to the right").

    - *The neighboring hyperface is at least as high as the the colquod where the downflow starts:* Figure 5.24(a)→(b): All downflows remain, but are on the neighboring hyperface.
    - *Otherwise, if the neighboring hyperface is at least as high as the the colquod where the downflow ends:* Figure 5.24(c)→(d): We reduce the downflows. The remaining downflows are on the neighboring hyperface.

2. *Flow leaves to the right; there is no neighboring hyperface to the right:* (Fig. 5.25) The downflow can be shifted to the right. Note that the flow that leaves to the right may be used as the necessary right bend of an opposing hyperface. Thanks to Lemma 5.3, we know that the new position of this right bend is still on that opposing hyperface. Furthermore it now happens "earlier", hence no new errors are introduced by this transformation.

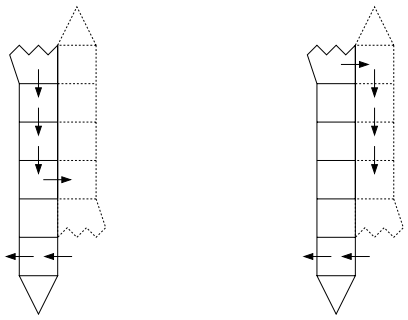


Figure 5.25: Class 3 downflow leaving to the right and the related downflow–free solution

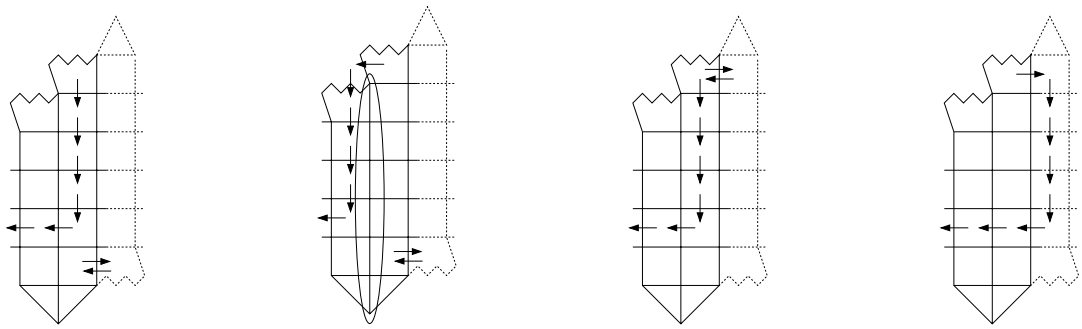


Figure 5.26: (a) Class 3 downflow leaving to the left, (b) an invalid transformation, (c) a hypothetical change, and (d) the related downflow–free solution

3. *Flow leaves to the left:* As Figure 5.26(a) shows, the flow may also be used by a neighboring hyperface to the left. A simple shift to the left may not result in a valid solution, as demonstrated in Figure 5.26(b). The situation in which such a transformation is not possible needs the right hyperface to be demerged by an FRF. Otherwise the flow demerging the right hyperface would also demerge the left one. That would render the right bend produced by the downflow *not* crucial for the shape of the left hyperface (and hence making the mentioned left shift possible).
   Note that this FRF may also be crucial for a hyperface that is opposing the right one, but because of Lemma 5.3 the following thoughts are not influenced by this fact: If you move the FRF up to the end face of its hyperface, the situation looks more like an easy class 1 downflow (Fig. 5.26(c)) that we can transform as described in Lemma 5.6. This results in a valid solution for these complicated cases (Fig. 5.26(d)).
   If there is a neighboring hyperface to the right instead of an opposing one, we have to apply the same method not once but twice (and so on), using the FRF–argument above.

□

Main Downflow Lemma

As a result of the above observations we can now deduce the following main lemma of this section:

Lemma 5.9. *Only downflows that occur as a part of FRFs have to be allowed.*

Proof. Lemmata 5.6, 5.7 and 5.8 point out that there always exists a related solution S' for any optimal solution S with equal objective value. S' contains no downflows except of updown–FRFs.
Since any optimal solution is good enough for our algorithm, we have proven that if we allow downflows only as a part of an FRF, at least one of the optimal solutions remains in the solution space of the ILP. □

Using the knowledge of this main lemma, we can now simply add an inequality to the ILP. Constraint (5.10) assures that downflows only occur as part of FRFs. It does so by demanding that the upflow is at least as strong as the downflow. This inequality has to be added for all inner edges of all hyperfaces.

$$\forall n \in H(V), \forall e \in \{e_1 \in \mathrm{IE}(n) \mid |e_1^H| > 1\}, \forall e' \in \{e_2 \in e_1^H \mid \mathrm{source}(e_2) \notin H(V)\}: \quad (5.10)$$

$$b_{\mathrm{facepred}(e')} \geq b_{\mathrm{rev}(\mathrm{facepred}(e'))}$$

Corollary 5.1. *Constraint* (5.10) *prohibits all errors induced by downflows except for invalid updown–FRFs.*

Proof. This is a direct result of the above description. □

Remark 5.9. *This extension does not prohibit all errors induced by downflows but reduces the possible erroneous downflows to ones accompanied by an FRF situations. Section 5.7 will describe why and how such errors can always be repaired after the optimization process without adding additional bends.*

Remark 5.10. *We can describe* (5.10) *as a constuction in the underlying min–cost–flow network: we just have to remove the possibility for arcs other than the upflow to send units into the downflow arc (see Fig. 5.27). Hence – unlike* (5.9) *– this extension does not introduce any non–integer solutions (see Section 6.1 for details).*

5.6.4 Restricting Left FRFs

The second type of errors is the one without downflows. Invalid shapes are caused by an FRF on the partner edge of a hyperedge. Such an FRF is called *left–FRF*.
Because the ILPs objective function minimizes the bend count, such an FRF can only be caused by a second hyperface that lies directly to the left of the first one. Thus the halfedge on which the FRF lies is not only part of a partner edge, but also an element of an additional hyperedge. While such an FRF demerges the second hyperface, it may invalidate the first one (to its right), if that hyperedge is still collapsed at this stage.
So what we have to do is to assure that such a left–FRF only happens if the hyperedge is already demerged at this stage (Fig. 5.28). Since this would be quite troublesome to include into the ILP, we analyze this situation a bit further.

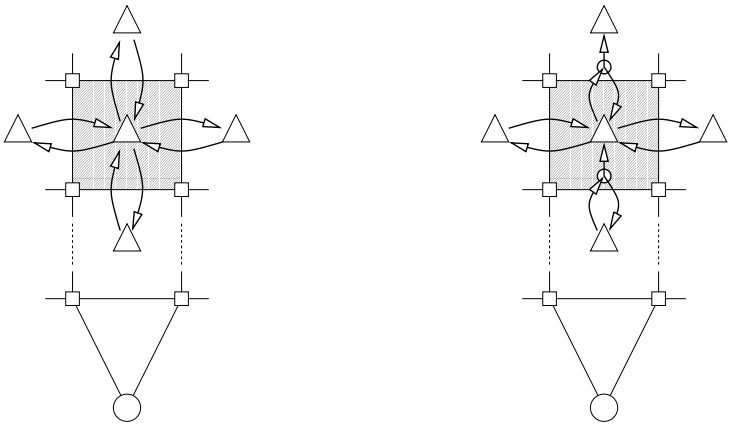


Figure 5.27: Forbidding downflows except for FRF–situations can be realized as a normal min–cost–flow network; left: original network of a colquod, right: modified (triangles represent face–nodes)

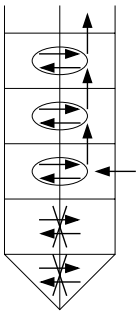


Figure 5.28: The FRF is only possible on the encircled positions (and above)

Lemma 5.10. *For every valid and optimal solution with a left–FRF on a hyperface with an opening angle of 0°, there exists a solution with equal objective value, where the FRF lies on the same height as the first right bend on the hyperedge.*

Proof. We can move the left FRF downwards to the lowest possible position: the stage where the (right) hyperface demerges. Such a move does not effect orthogonal hyperfaces (if there are any): the FRF only looks like an updown–FRF to them, which we already considered irrelevant for now (see Section 5.6.3 and 5.7).
The demerge of the hyperface can either happen by a right bend on its hyperface or by a left bend on its bundle partner. The latter is a left bend on the hyperedge of the left hyperface. Thus in the first case the lowest possible place is the one specified by the lemma.
If the right hyperface demerges by a left bend on the bundle partner, the left hyperface demerges at exactly the same stage or even below since it has no non–FRF downflows. Hence the left–FRF is not needed to assure a demerge but only satisfies the demand for at least one right bend on every hyperedge.
Thus we can simply move this FRF to the position we want: the stage where the first right bend occurs on the hyperedge of the right hyperface. □

Since hyperfaces with an opening angle of 90° are not influenced by any left FRFs, we can add an additional constraint based on the above lemma and these thoughts:
It is certainly enough to allow left–FRFs to happen either:

1. On the same face as a right bend of the hyperedge (if the opening angle is 0°)
2. Directly on the coltri (if the opening angle is 90°)
3. On a face that has an effective upflow (= upflow that is not used in an FRF) coming from the face below (if the opening angle is 90°)

This restriction can be further weakened by allowing case 3 also to happen if the opening angle is 0°, because we know that effective upflows only happen if the face below is demerged.
Fortunately, such a constraint is quite easy to implement:

$$\forall n \in H(V), \forall e \in \{e_1 \in \mathrm{IE}(n) \mid |e_1^H| > 1\}, \forall e' \in e^H \text{ with } \mathrm{target}(e') \in D(V): \tag{5.11}$$

$$b_{\mathrm{partner}(e')} \leq b_{e'} + \begin{cases} a_{e'} & \text{if } \mathrm{source}(e') \in H(V) \\ b_{\mathrm{facepred}(e')} - b_{\mathrm{rev}(\mathrm{facepred}(e'))} & \text{otherwise} \end{cases}$$

The "bad" part of the left–FRF is the right bend on the partner edge; thus we only have to restrict it by demanding that the flow for this bend comes either from a right bend on the hyperedge or from "below". In case of a coltri, this "below" means the high degree node that may send a unit to generate an opening angle of 90°. For the other subfaces of the hyperface, "below" means the face below. This implies that – in the case of the coltri – a right bend on the bundle partner can only occur if there is a right bend on the respective hyperedge that early, too, or if the opening angle of the hyperface is 90°.
Hence we assured that no errors are introduced by any left FRF, without cutting off all optimal solutions – at least one remains. We also did neither introduce additional class 1, class 2 nor class 3 downflow errors.

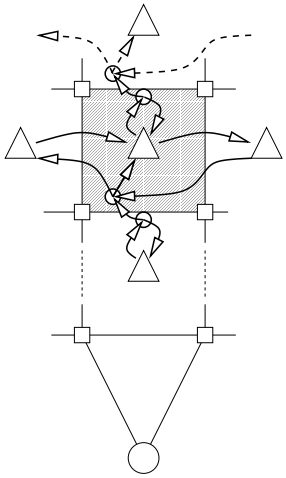


Figure 5.29: Restricting left–FRFs can be included in a normal min–cost–flow network; the network of a colquod that restricts the right bends on the bundle partner implicitly includes the downflow restriction.

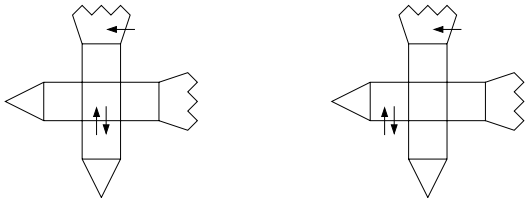


Figure 5.30: Invalid updown–FRF and its valid counterpart

Corollary 5.2. *Constraint* (5.11) *prohibits all errors induced by invalid left–FRFs.*

Proof. This is a direct result of the above description. □

Remark 5.11. *We can describe* (5.11) *as a construction in the underlying min–cost–flow network: we just have to remove the possibility for arcs other than the allowed ones to send flow into the arc of the partner edge (see Fig. 5.29). Hence – unlike* (5.9) *– this extension does not introduce any non–integer solutions (see Section 6.1 for details). This construction also includes the downflow restriction, discussed in Section 5.6.3.*

5.7 Repairing the Solution

As mentioned earlier, there are cases in which a solution generated by the ILP may not be applicable due to FRFs consisting of an up- and downflow (*updown–FRFs*, see Fig. 5.30). Fortunately, the flow creating such a situation can be rerouted: we modify the flow in such a way that the objective function never increases and the invalid shapes are completely replaced by valid ones.
The main structure of the function is based on a scan through all collapsed hyperfaces, searching for invalid updown–FRFs.Since they may only occur for as long as the hyperface is collapsed, the scanning of a hyperface may stop once a demerge is detected. If such an invalid updown–FRF is detected, it will be removed as described in the next subsection.

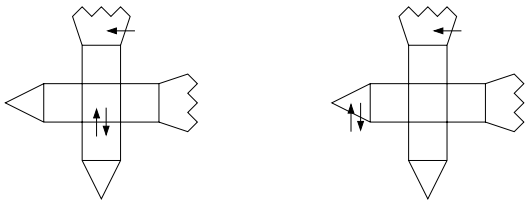


Figure 5.31: Moving the invalid updown–FRF to its coltri so it cannot cause any trouble

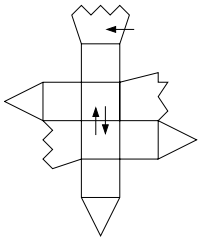


Figure 5.32: Moving the invalid updown–FRF to its coltri may not always be possible

The algorithm executing these repairs is given in Section 5.7.2.

5.7.1 Theory of Repairing Invalid UpDown–FRFs

We move each invalid updown–FRF somewhere where it does not invalidate any hyperface, but still correctly demerges the hyperface that needs it. The best place for such an FRF would be the first edge of the hyperedge they lie on (cf. Fig. 5.31).

Lemma 5.11. *Moving an FRF that causes an invalid downflow to the hyperface's subedge that is incident to the coltri (if this coltri is unambiguous) has the effect of removing any invalid downflows caused by it. (But it may generate an invalid left–FRF.)*

Proof. By moving it there, the corresponding hyperface demerges earlier. Hence such a movement does not introduce errors on the hyperface the FRF is needed for. Since the target subedge can never be an inner edge of any hyperface (coltris cannot be end faces of a hyperface; Remark 5.2), such a shift cannot result in an updown–FRF. □

So we only have to pay attention not to introduce an invalid left FRF by such a shift. We also have to watch out for situations where such a shift is not possible at all: if the FRF causing the downflow–error is needed by two opposing hyperfaces (hence the coltri to move the FRF onto is ambiguous), it may not be possible to shift it to any coltri without introducing errors on the opposing hyperface (see Fig. 5.32).
Thus we have to take a detailed look on the structure of such updown–FRF–situations:

Stacked FRFs

Invalid updown–FRFs have the property that they can occur *stacked*: there may exist updown–FRFs between a continuous sequence of subfaces.

Definition 5.10. FRF Stack, stacked FRFs: *An* FRF stack *is a continuous sequence of dependent updown–FRFs of a hyperface. Continuity is defined by the property that*

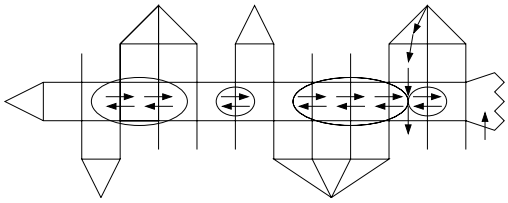


Figure 5.33: Example of a hyperface containing several (encircled) FRF stacks

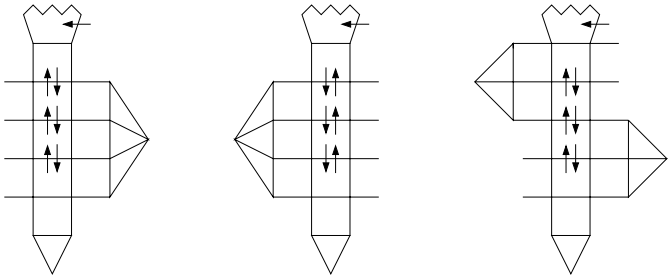


Figure 5.34: The three types of FRF stacks

for all but the last FRFs of the sequence, there exists another FRF directly above it. Dependency is defined as the need for an FRF to move, if the FRF below moves (if such FRF exists), due to the left–FRF–restriction (formulated in Eq. 5.11).

Figure 5.33 shows an example of several distinguishable FRF stacks on a single hyperface. Since such stacks are defined to be dependent, it is obvious that it is necessary to repair each such stack as a whole. Thus the repair–function will always find an invalid FRF stack, repair it, and then continue looking for the next invalid stack, until there is none left.
There are three different types of FRF stacks; the third one is the conjunction of the other two (Fig. 5.34):

1. A bundle of neighboring hyperfaces with their respective coltris to the right of the hyperface containing the erroneous updown–FRFs

2. A bundle of neighboring hyperfaces with their respective coltris to the left of the hyperface containing the erroneous updown–FRFs

3. A block of hyperfaces that starts with a bundle of the first type and continues with a bundle of the second. The last of the first, and the first of the second bundle are opposing hyperfaces; both of them need the FRF.

Note that bundles can consist of a single hyperface.

Remark 5.12. *If not both, but only one of the opposing hyperfaces mentioned in the third case needs the FRF, the stack would actually split up into two stacks, since the required dependency would not be existent.*

Remark 5.13. *A block consisting of a type 2 stack first, and followed by a type 1 stack, can never have a continuous sequence of FRFs since the pseudo–opposing hyperfaces only*

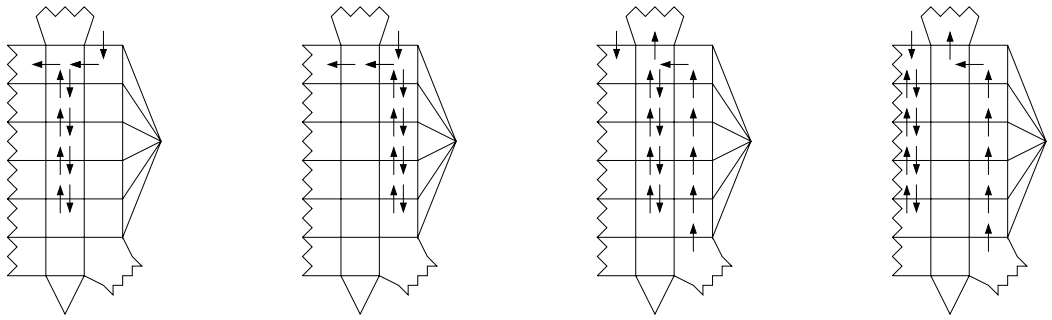


Figure 5.35: Two repairs for type 1 stacks: (a)→(b), (c)→(d)

share their bundle partners; SPED will never generate an FRF there since it could be simply removed without side effects which would improve the objective function by two.

Repairing Type 1 Stacks

Lemma 5.12. *Errors caused by type 1 FRF stacks can be removed in polynomial time without introducing new errors.*

Proof. Let h be the hyperface that contains the stack. As pointed out earlier (Lemma 5.11), shifting FRFs to their coltris is in general a good idea, since they cannot produce any downflow errors there. Thus we will try to shift the stack onto their respective coltris. We know from Lemma 5.2 that neighboring hyperfaces have a common source node; hence the stack will continue to be a stack after such a movement.
We analyze the two different situations that may occur:

1. *There is a neighboring hyperface h^* (with an opening angle of 0°) to the right of the bundle that contains the stack:* We know that the h^* has at least one right bend on its hyperface. This bend will not be directly above the stack, since this would either be a downflow in h or part of an FRF that itself is part of the stack. Hence there are two possibilities:

   - *The right bend is below the left–FRF on h^*:* (Fig. 5.35)(a)→(b)
     In this case, we can shift the stack towards its coltris until we reach this right bend. This shift cannot introduce downflow errors: (Indirect) Let h' be the hyperface that contains the shifted stack. Thus the right bend on h^* is either an invalid downflow or an upflow in h'. The latter renders h' demerged, hence the stack cannot introduce downflow errors.
   - *The right bend is above the left–FRF on h^*:* (Fig. 5.35)(c)→(d)
     We know that there is a flow that enables the left–FRF on h^* (the top FRF of the stack). This flow has to be an upflow in h^*, which is a right bend on the hyperedge of h. Furthermore, every outflow of a colquod has an associated inflow, and the hyperedges of the bundle contain no right bends except for the FRFs. Hence we know that there is a flow path p on the right side of the stack downwards. Hence the hyperfaces that need the elements of the stack are already demerged and we can shift the FRFs towards the end faces of these hyperfaces without introducing errors. We stop this movement if we

find either a position where this stack does not introduce invalid downflows, or when we reach the first end face of the bundle. The latter position may still introduce invalid downflows if there is a hyperface $\bar{h}$ in which the stack resides. But $\bar{h}$ is smaller that the stack and has a right bend on its hyperedge. Hence the number of errors decreases after such a shift. We can reapply the same movement for the smaller stack until no errors remain.

2. *Otherwise:* We can simply shift the stack onto its coltris. Based on Lemma 5.11 we know that such a shift could only introduce invalid left–FRFs. The only place where this could happen would be below the stack. The generation of an invalid left–FRF would require a left–FRF that was valid before the shift. But such an FRF will remain valid, since the right bends now happen earlier on their hyperedges. (Note that an FRF directly below the stack is not possible, since it would be part of the stack due to its dependency.)

Note that in both cases, we cannot introduce an invalid left–FRF on the left–most hyperface of the bundle: this FRF would either be part of the stack (because of its dependency) or it has a different – and therefore unmodified – flow that enables its existence (cf. Constraint (5.11)). □

Repairing Type 2 Stacks

Lemma 5.13. *Errors caused by type 2 FRF stacks can be removed in polynomial time by shifting them to their respective coltris, without introducing new errors.*

Proof. Based on Lemma 5.11, we only have to prove that such a shift does not introduce any invalid left–FRFs. Such errors could only arise below or above the stack.

1. *No invalid left–FRFs above the stack:* The generation of an invalid left–FRF above the stack would require a left–FRF that was valid before the shift. But such an FRF will remain valid, since the right bends now happen earlier on their hyperedges. (Note that an FRF directly above the stack is not possible, since it would be part of the stack due to its dependency.)

2. *No invalid left–FRFs below the stack:* (Indirect) Let h^* be the hyperface for which the lowest FRF of the stack is a left–FRF. If h^* does not exist, there obviously can be no left–FRF–error. If it exists, there is a flow that enables its existence (Constraint 5.11). This flow can either be a right bend on the hyperedge of h^* or an upflow in h^*. The former is an upflow in h, the latter is a left bend on the bundle partner of the hyperedge of h. In both cases, these flows render h demerged. Hence the stack will not generate any invalid downflow, which is a contradiction to the need for a repair.

□

Repairing Type 3 Stacks

Moving such stacks to coltris is in general not possible: the stack would be split up and the hyperedges of the opposing hyperfaces would need two instead of only one FRF to generate a valid solution. Thus we handle such a situation differently:

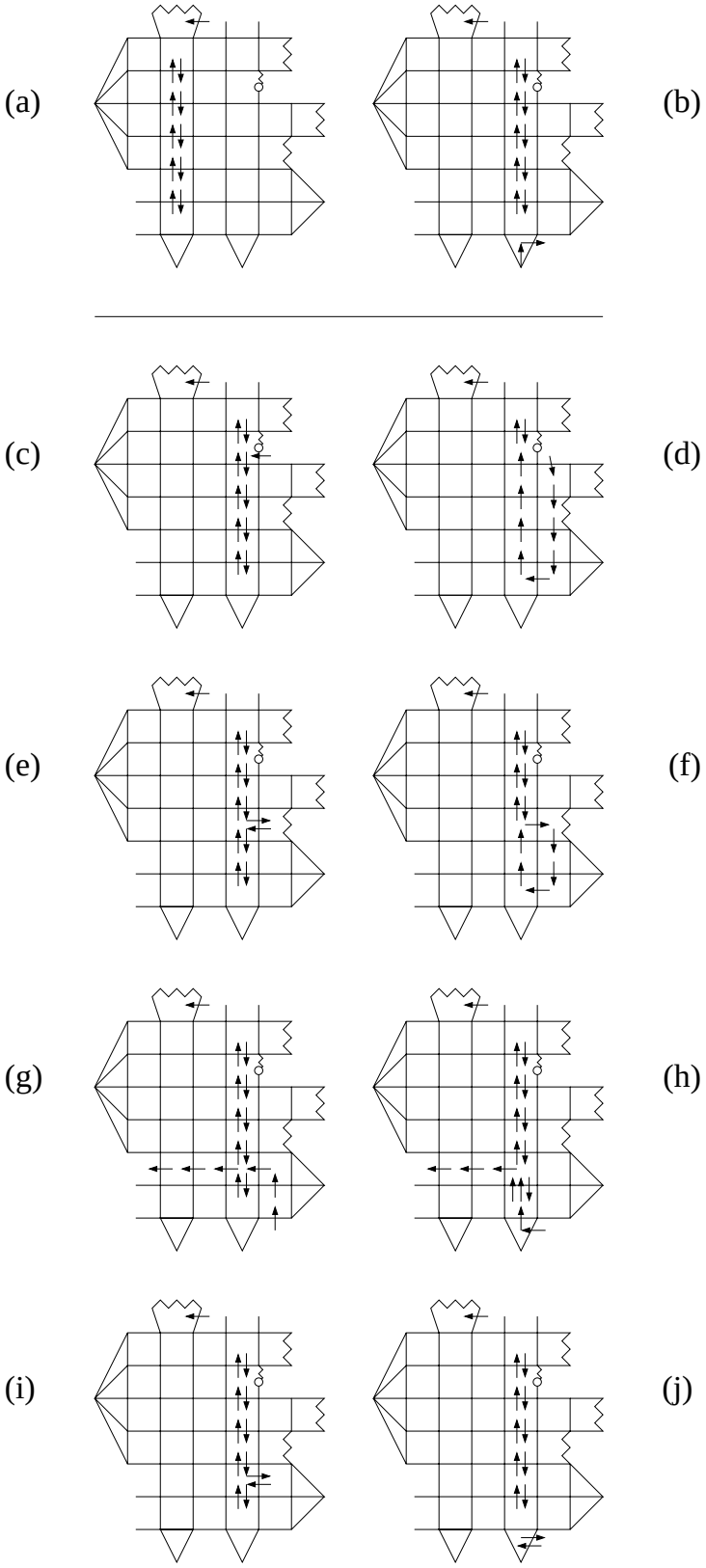


Figure 5.36: Moving Type 3 Stacks: (a) Invalid solution; (b) Shift suffices; (c)(e)(g)(i) Shift does not suffice; (d)(f)(h)(j) Additional modification

We shift the stack to the right as far as possible. If it reaches the coltris on the right, a valid solution has been found; if not, and the solution is still invalid, there exist small modifications to repair it (Fig. 5.36).
At first, we have to prove that such a shift to the right does not invalidate the upper bundle of the block the stack belongs to although we move the FRFs farther away from their coltris.

Lemma 5.14. *In case of a type 3 FRF stack, a shift of the FRFs to the right – and therefore farther away from the coltris of the second bundle of the stack – will not introduce any errors other than updown–FRFs that are revealed.*

Proof. To generate an error, there would be either a left bend on a hyperedge, a right bend on a bundle partner, or an updown–FRF that now – after the shift – happens earlier than the FRF that demerges the bundle (note that only the hyperfaces of the second bundle of the block have to be considered):

- *left bend on hyperedge:* Since such a left bend is an outflow from the hyperface, there would be a related inflow.
  Obviously, downflows are not allowed and the hyperface starts with an opening angle of 0° – otherwise there would be no need for the FRF. Furthermore, the inflow can never come anywhere over the hyperedge since that would mean two FRF–like situations on a single hyperedge; this is impossible since we minimized the objective function.
  So the inflow has to come from the side of the bundle partner – perhaps from some colquods below the left bend we analyze. But since such an inflow is a left bend on the bundle partner, it would demerge the hyperface. Thus the left bend on the hyperedge will not cause any trouble.

- *right bend on bundle partner:* If this right bend would be part of an FRF, it is either part of the stack itself (thus moving together with it) or there exists another flow that enables that FRF. Such a flow is of course not influenced by the movement of the stack and remains to validate the FRF.
  If the right bend is not part of an FRF, is also has an enabling flow that will remain.

- *revealed updown–FRF:* If there are any invalid FRFs revealed by this shift, they will be removed when scanning through the corresponding hyperface that includes these newly generated errors. Lemma 5.16 will prove that there cannot be a situation where this leads to an infinite loop.

□

Lemma 5.15. *The errors caused by a type 3 FRF stack can be removed. We may reveal some additional updown–FRFs on the orthogonal hyperfaces thereby.*

Proof. We shift a type 3 stack as far right as possible (Lemma 5.14, this shift may reveal invalid updown–FRFs). A hyperedge of the second bundle of the block stops this movement, if that hyperedge ends before the right coltris are reached. In this case, the stack will always go through an end face after such a move.
If the stack lies on the right coltris after such a movement, this part of the solution is valid, due to Lemmata 5.11, 5.12 and 5.14.

If the stack does not lie on the right coltris but does not induce any downflow error, the corresponding part of the solution is valid, too: there cannot occur any left–FRF–errors (Lemmata 5.12 and 5.14; Fig. 5.36(b)).
Thus there only remains the situation where there is still a downflow error generated by the stack: We know that at least one face the stack uses is an end face, and that the stack lies completely in a hyperface h (otherwise there could not be any downflow errors). Hence we know that h will have to stop at the lowest end face – the lowest non-colquod – the stack goes through. (Note that it is irrelevant if h is parallel to the hyperface the stack lay in originally or if it points in the opposite direction.)
Since h contains a downflow error, it obviously does not demerge below the stack. Hence its necessary right bend either happens:

1. On the end face of h (Fig. 5.36(c))
2. Below the end face, on the second bundle of the stack (Fig. 5.36(e))
3. On the first bundle of the stack (Fig. 5.36(g,i))

Ad 2) In the second case, the right bend has to occur as an FRF, because otherwise it would be a forbidden downflow on the orthogonal hyperface.
Ad 3) If the right bend occurs on the first bundle of the stack, it is an upflow in the corresponding orthogonal hyperface $\bar{h}$. This right bend can either be part of an FRF, or the flow that generates the bend enters $\bar{h}$ over the bundle partner of its hyperedge (Fig. 5.36(g,h)).
Note that it could neither enter over the hyperedge of $\bar{h}$, nor origin from an opening angle of 90°: this would render the type 3 stack nonexistent.

- *Repair in the first two cases* (Fig. 5.36(d,f)): Since only the stack's elements below that discussed right bend introduce the errors, we have to reroute only these FRFs, taking advantage of the right bend on h. We apply the following rerouting scheme from the highest error–causing FRF down to the lowest FRF of the stack: we shift the downflow–part of the FRFs to the right and the right bend down.
- *Repair in the third case, right bend is not part of an FRF* (Fig. 5.36(h)): As shown in the figure, we can simply let the right bend happen earlier by elementary rerouting.
- *Repair in the third case, right bend is part of an FRF:* (Fig. 5.36(j)): Although it may not seem so at first glance, this is a simple case, too. Because of the second block of the stack and the property that hyperfaces cannot end with a coltri, such an FRF may only be part of a type 2 FRF stack s (if any). Thus s can simply be shifted to its coltris (as described in Lemma 5.13). Such a move clearly renders the type 3 stack completely harmless.

Obviously, this leads to a removal of the errors caused by the FRF stack. □

Lemma 5.16. *The revealed FRFs will not cause an infinite loop.*

Proof. Since the FRFs are not newly generated but only revealed, there exists only a polynomial number of them. The following property holds for every removal step for all stack types: we remove FRF–errors either by shifting them to coltris – where they will never generate an error again – or by splitting them up. Thus the number of invalid FRFs is decreased by each step, not counting additionally revealed ones. Hence the process will terminate after a polynomial amount of steps. □

Corollary 5.3. *Errors caused by type 3 FRF stacks can be removed in polynomial time by applying the above scheme.*

Proof. This corollary is a direct result of Lemmata 5.15 and 5.16. □

5.7.2 Algorithm

Algorithm 5.1 Repair–Function

Require: feasible and optimal solution of the ILP (using the arrays a and b)
Ensure: valid solution of the orthogonalization problem

```
 1: Q.clear()
 2: for all v ∈ H(V) do
 3:   for all e ∈ IE(v) do
 4:     if face(e) is a coltri then
 5:       Q.append(e)
 6:     end if
 7:   end for
 8: end for
 9: while Q is not empty do
10:   e ← Q.pop()
11:   if a[e] = 0 then
12:     h ← e
13:     while h ∈ e^H, except the last one do
14:       p ← partner(h)
15:       t ← facesucc(h)
16:       if b[h] > 0 ∨ b[p] > 0 then
17:         exit inner loop
18:       else if b[t] > 0 ∧ b[rev(t)] > 0 then
19:         S ← scan_stack(t)
20:         if S is of type 1 then
21:           apply repair scheme type 1 (Lemma 5.12)
22:         else if S is of type 2 then
23:           apply repair scheme type 2 (Lemma 5.13)
24:         else
25:           A ← apply repair scheme type 3 (Lemma 5.15)
26:           Q.append(A)
27:         end if
28:         advance h by length(S)
29:       else
30:         advance h by 1
31:       end if
32:     end while
33:   end if
34: end while
```

The repair–function (Algorithm 5.1) consists of two parts. The first part (line 1–8) stores the first subedges of all hyperedges into a queue Q; the second part (line 9–34) uses the content of this queue to perform the actual repairs: it scans the queue for hyperfaces

with an opening angle of 0° (line 11) and analyzes each of them from bottom upwards (line 12, 13). As soon as there is a demerge of the hyperface, it is guaranteed to have a valid shape, thus its examination can be aborted (line 16, 17).
If an (invalid) updown–FRF is detected (line 18), the situation is examined in detail by a helper–function called *scan_ stack* that returns the complete stack that belongs to the invalid FRF. Lines 20–27 select and execute the appropriate method to eliminate the errors based on the type of the stack. These schemes have been discussed in the previous section. Since the scheme for type 3 stacks may reveal invalid FRFs, it returns a list of the hyperfaces that (may) have been invalidated by the repair. This list is added to the queue (line 26).
After repairing the errors caused by the FRF stack, we can skip as many subedges of the hyperedge as the stack was high, since these errors have been removed as well (line 28). If the algorithm detected no updown–FRF, it simply makes one step up the hyperedge (line 30).
The repair–function repeats this procedure for each hyperedge, as long as there exist more colquods to examine (line 13, 32) or until a demerge is detected (line 16, 17).

Runtime Analysis of Algorithm 5.1: *The loop in lines 2–8 processes each hyperface exactly once, and appending to a queue requires only constant time. We can reduce its first three lines by explicitly listing the hyperfaces during the generation of the ILP:*

$$t_{2-8} = O\left(\sum_{v \in H(V)} \deg(v)\right) = O(E') \Rightarrow t_{2-8} = O(h)$$

The loop between line 9 and 34 scans every hyperface at least once. But the list may grow during the execution, due to repairs of type 3 stacks:
Let h denote the number of hyperfaces and l_{max} the maximal length of a hyperface. Since no invalid FRF can reoccur after it was repaired (Lemma 5.16), we have to repair at most $O(h \cdot l_{max})$ invalid FRFs. Each of these repairs may reintroduce a hyperface into the queue. (Note that the scan of a hyperface only needs $O(l_{max})$ time, since the scan_stack*–function needs only linear execution time if the halfedges are labeled appropriately.)*
Hence the loop in line 9 – in conjunction with the loop in line 13 – has a time complexity of at most

$$t_{9-34} = O\left((h \cdot l_{max})^2\right) \quad .$$

Additionally, we have to consider the complexity of the individual repair schemes. Each FRF that we repair, is shifted alongside a hyperface. To find the final shift–position, we need $O(l_{max})$ steps. If we need an additional modification – in case of complicated type 3 repairs – we can perform this in $O(l_{max})$, too. Hence the $O(h \cdot l_{max})$ repairs can be done in

$$t_{\text{repairs}} = O(h \cdot l_{max} \cdot (2l_{max})) = O\left(h \cdot l_{max}^2\right) \quad .$$

We sum up these results to get the complexity of the whole algorithm:

$$\begin{aligned} t_{\text{algorithm}} &= t_{2-8} + t_{9-34} + t_{\text{repairs}} = \\ &= O\left(h + h^2 \cdot l_{max}^2 + h \cdot l_{max}^2\right) = \\ &= O\left(h^2 \cdot l_{max}^2\right) \quad . \end{aligned}$$

Alternatively, we know that $O(h \cdot l_{max}) \leq O(E)$, hence the performance can be estimated more generously as

$$t_{\text{algorithm}} = O\left(E^2\right) \quad .$$

Lemma 5.17. *There exists a polynomial algorithm transforming any optimal solution of the described ILP into an optimal valid solution without changing the value of the objective function.*

Proof. The proof is given by presenting such an algorithm (Alg. 5.1). The correctness of the algorithm is a direct result of the preceding Lemmata 5.12, 5.13 and Corollary 5.3. □

Remark 5.14. *Experiments show that the repair–function is not needed very often (cf. Appendix B). Especially, the more complex the repair scheme is, the more uncommon is its appearance. In most cases there are no invalid updown–FRFs at all.*

5.8 Summary of SPED

Since the lengthy sections above may have obscured the resulting ILP and the overall process of calculating the shape, this section recapitulates the results of the above observations and conclusions:
We solve the ILP below and run the repair–function on its output. The flow that results from that function represents a valid and optimal shape for SPED.
The ILP consists of the following parts:

Eq.	Purpose
(5.1)	Minimize the bend count
(5.2)	Balance in nodes
(5.3)	Balance in faces
(5.9)	At least one right bend on the hyperface
(5.10)	Downflows only as a part of an FRF
(5.11)	Restriction for Left–FRFs
–	Bounds and domains

$$\min \sum_{e \in E_h} b_e \tag{5.1}$$

subject to

$$\forall n \in V : \sum_{e \in \mathrm{IE}(n)} a_e = 4 \tag{5.2}$$

$$\forall f \in F : \sum_{e \in \mathrm{IE}(f)} \left(a_e + b_e - b_{\mathrm{rev}(e)} \right) = 2\deg(f) + \begin{cases} +4 & \text{if } f = f_0 \\ -4 & \text{otherwise} \end{cases} \tag{5.3}$$

$$\forall n \in H(V), \forall e \in \mathrm{IE}(n) : a_e + \sum_{t \in e^H} b_t \geq 1 \tag{5.9}$$

$$\forall n \in H(V), \forall e \in \{e_1 \in \mathrm{IE}(n) \mid |e_1^H| > 1\}, \forall e' \in \{e_2 \in e^H \mid \mathrm{source}(e_2) \notin H(V)\} : b_{\mathrm{facepred}(e')} \geq b_{\mathrm{rev}(\mathrm{facepred}(e'))} \tag{5.10}$$

$$\forall n \in H(V), \forall e \in \{e_1 \in \mathrm{IE}(n) \mid |e_1^H| > 1\}, \forall e' \in e^H \text{ with } \mathrm{target}(e') \in D(V) : b_{\mathrm{partner}(e')} \leq b_{e'} + \begin{cases} a_{e'} & \text{if } \mathrm{source}(e') \in H(V) \\ b_{\mathrm{facepred}(e')} - b_{\mathrm{rev}(\mathrm{facepred}(e'))} & \text{otherwise} \end{cases} \tag{5.11}$$

and the following bounds/domains[6]:

$$\begin{aligned} a_e &\in \begin{cases} [1, 5 - \deg(\sigma(e))] & \text{if } \mathrm{source}(e) \in L(V) \\ [0,1] & \text{if } \mathrm{source}(e) \in H(V) \end{cases} \\ b_e &\in [0, \infty] \\ a_e &\in \mathbb{R} \\ b_e &\in \begin{cases} \mathbb{Z} & \text{if } \exists e_1 \in E_h : e \in e_1^H \\ \mathbb{R} & \text{otherwise} \end{cases} \end{aligned}$$

Theorem 5.2. *For any planarized, simple, and self–loop–free graph G with given planar topology T and the dummy nodes $D(V)$ representing edge crossings, we can calculate a valid shape with the minimum number of bends obeying the rules of SPED with the above ILP, in conjunction with the described repair–function.*
The rules of SPED are the rules of Simple–Podevsnef with the difference that a right bend on bundles is demanded for the original edges of the non–planar graph, instead of the planarized graph G.

Proof. This theorem is a direct result of Theorem 5.1, the lemmata considering downflows (5.9), left–FRFs (5.10), and the repair–function (5.17), the Corollaries 5.1 and 5.2 as well as of the complete chapter above. □

[6] see details in Section 6.1

5.9 Preparing for Compaction

The precompaction algorithm has to be expanded because of the dummy–merging; prior to the classic precompaction, a merge–function has to be applied. This function identifies the collapsed faces and merges the dummy nodes, as well as the arising *double edges*[7]. Thus the overall compaction step looks like this (cf. Fig. 5.37):

1. Dummy merging
2. Precompaction[8]
3. Classic grid compaction
4. Undo precompaction
5. Undo dummy merging

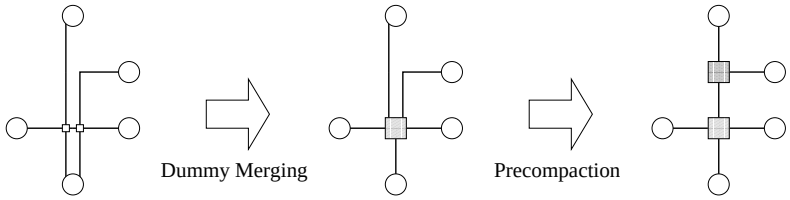


Figure 5.37: Preparing a SPED solution for compaction

Algorithm 5.2 Dummy Merging

```
1: for all v ∈ H(V) do
2:   for all e ∈ IE(v) do
3:     if face(e) is a coltri ∧ a[e] = 0 then
4:       for all h ∈ e^H, except the last one do
5:         p ← partner(h)
6:         if b[h] > 0 ∨ b[p] > 0 then
7:           exit inner loop
8:         else
9:           merge target(h) with target(p)
10:        end if
11:      end for
12:    end if
13:  end for
14: end for
```

The main body of the presented algorithm (Alg. 5.2) has similarities to the structure of the repair–function, but its overall functionality is much less complicated. It runs through all collapsed hyperfaces and simply merges the dummy nodes as long as it is collapsed.
By storing the merges in a stack, it is easy to undo them in step 5 in the inverse (and thus correct) order.

[7] two edges with identical source and target nodes are called *double edge*.
[8] as described in Section 4.2.5

Runtime Analysis of Algorithm 5.2: *We need line 1, 2, and (the first part of) 3 to process each hyperface exactly once. These lines can be further reduced, if we explicitly list the hyperfaces during the generation of the ILP (h is the number of hyperfaces):*

$$t_{1-14} = O\left(\sum_{v \in H(V)} \deg(v)\right) = O(E') \Rightarrow t_{1-14} = O(h)$$

Let l_{max} denote the maximum hyperface length in the planarized graph. The loop in line 4 has a running time of

$$t_{4-11} = O(l_{max}) \ .$$

Since Lines 5–10 have constant running times, this results in an overall performance of

$$t_{\text{algorithm}} = t_{1-14} \cdot t_{4-11} =$$

$$= O(h \cdot l_{max}) \ .$$

Since it could be possible that nearly all edges are part of a hyperedge, we can estimate this more generously as

$$t_{\text{algorithm}} = O\left(E\right) \ .$$

Remark 5.15. *We can enhance this function by the introduction of* bendable bundles*: This concept features the drawing of hyperfaces that do not demerge when their first bend occurs (if possible). E.g., consider a hyperedge e^H that has a right bend on its first and on its second subedge, and the bundle partner of e^H has a right bend on its first, and no bend on its second subedge: the hyperface can remain collapsed for the first parallel right bend, and demerge at the second right bend on e^H.*
This function would not modify the shape of the graph (the bend count does not change), but the area needed for the resulting drawing would be reduced.

Chapter 6

Related Topics

Wenn die anderen glauben, man ist am Ende, so muss man erst richtig anfangen
Konrad Adenauer

This chapter is a collection of topics related to SPED. Section 6.1 discusses the problem of determining the exact time complexity of SPED. It shows examples of fractional solutions with lower objective values than the best ILP solution, and offers some thoughts on them. Section 6.2 discusses the use of classic Simple–Podevsnef as a heuristics for SPED and proves its quality guarantee. Section 6.3 shows various directions for developing further heuristics for SPED.
Finally, Section 6.4 describes how SPED can be further extended to handle non–planar clustergraphs – a special type of graphs where nodes are hierarchically collocated.

6.1 Complexity of SPED

From a mathematical point of view, we can transform every algorithmic problem into a corresponding language, and we can simulate a computer by a one–band Turing machine. Amongst others, we distinguish between languages that can be recognized by a deterministic polynomially time bounded one–band Turing machine, and languages that require a nondeterministic one.
Problems that only require such a deterministic machine are solvable in polynomial time, and are elements of P. Languages that can be recognized by a nondeterministic machine are *in* NP [29].
The identity of P and NP could neither be proven nor disproven for decades, but seems unlikely [13]. Hence NP–hard problems are only solvable using exponential running times with current knowledge. Many – if not most – interesting problems belong to this class, e.g., satisfiability and 3SAT, traveling–salesman, vertex cover and clique, graph 3–colorability, ILP, etc.
Such problems are also said to be *intractable*.

As described earlier, a classic min–cost–flow network is solvable in polynomial time. If it is transformed into an ILP, the LP–relaxation – the ILP without the integer constraints – will always result in an optimal integer solution.
Sadly, most extensions of such networks – namely path–constrained networks and networks with homologous arcs, both of which will be described in the next few paragraphs – do not have this property [13][29]: there exist optimal solutions better than any solution that consists of integer values only.
As pointed out in Chapter 5, the ILP described for the SPED problem consists of an extended min–cost–flow network. All constraints except for the right–bend–on–hyperedge constraint have an analogon in such a network. Unfortunately, this is the very constraint that has to be included in SPED to extend Simple–Podevsnef for non–planar graphs.

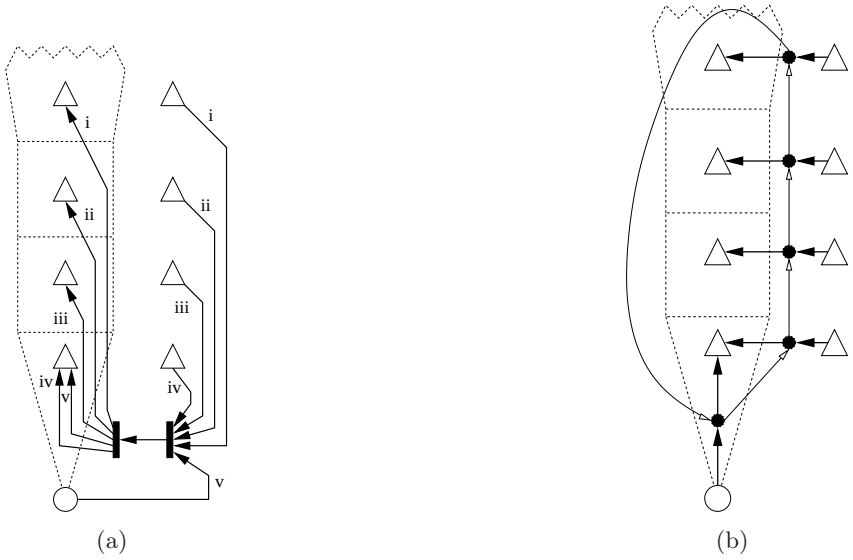


Figure 6.1: Right–bend constraint (5.9) as network: (a) with homologous arcs, (b) as path–contraint network

Figure 6.1 shows the two most obvious attempts to simulate this constraint in flow–networks:
Figure 6.1(a) makes use of *homologous arcs*. Such a network is a min–cost–flow network with the ability to force the flow to be identical for pairs of arcs. The homologous pairs are identifiable by their roman numbers. The single solid arc with five inputs and five outputs has the lower bound of one unit to resemble (5.9).
Figure 6.1(b) uses an approach similar to the negative cycles used in Fößmeier's Podevsnef [11][12]. As in Podevsnef, we would have to make use of a path–constrained network to forbid the solver to "fill" the cycles without sending any flow into them. The arcs of the cycle have a lower bound of one unit, and cost $-C$ (C being a constant "big enough", e.g. $C > 4|V|$). The costs on the ingoing edges of the cycle are the same as the costs on the arcs they represent: 0 units for the arc from the high degree node, 1 unit for the others. The cost on the outgoing edges of the cycle is $l \cdot C$, where l is the length of the cycle .
But since computing flow in both network types is NP–complete in general, such constructions seem not help us to achieve polynomial time bounds.[9]

As shown in Figure 6.2, there exist certain situations where the LP–relaxation of SPED will generate non–integer values. These situations can only occur on quite dense graphs, since they require certain overlaps of hyperfaces. Experiments show that such cases only appear for graphs that are both quite dense and large. Furthermore the standard cuts for mixed–integer–programs included in CPLEX [16] resolve these situations easily (cf. Appendix B.1).
The examples of Figure 6.2 consist of rather small blocks of hyperfaces, and the exact solution of one such block does not affect any part outside of the block.
The question remains, if we can use such building blocks to generate bigger interconnected blocks. This would be a first step for a proof of NP–hardness of this problem.

[9] Fößmeier's argumentation for planar Podevsnef cannot be applied since the outgoing arcs of the cycles have different target faces.

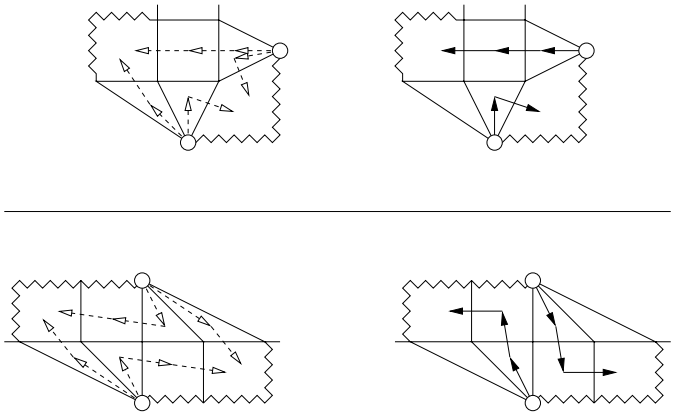


Figure 6.2: Examples of optimal fractional blocks: (left) LP–relaxed, (right) ILP. The arrows on the left symbolize a flow of 0.5 units, each arrow on the right stands for a flow of 1 unit. The first example has a difference of 0.5, the second of 1.

Furthermore, we do not know if there exists an infinite number of such blocks. If there are only a few, it could be possible to precompute their exact integer–optimal solution, and plug them into the solution of the LP–relaxation. This would result in a polynomial algorithm.
Thus the question whether SPED is mathematically intractable or not remains open for now.

6.2 Simple–Podevsnef as a Heuristics

Simple–Podevsnef has been used since the year 2000 and its network seems to be a good compromise between simplicity, execution time, and pleasing results. Therefore, it seems quite natural to use it even for non–planar graphs. Although this introduces unnecessarily demerging bundles, and it wastes much room for dummy nodes, it still has the advantage of being quick and useful. It even does not need any special algorithm for merging dummy nodes before the compaction stage, neither does it have to demerge them afterwards. Thus it is quite interesting to analyze, how much worse Simple–Podevsnef – used as a heuristics – handles non–planar graphs than SPED does.
We first need some lemmata to help us proof its quality guarantee.

Lemma 6.1. *Applying both Simple–Podevsnef and SPED on planar graphs results in two identical solutions.*

Proof. Since all extensions made to SPED only consider hyperfaces – and hyperfaces do not exist in planar graphs due to the absence of dummy nodes – SPED is reduced to a normal Simple–Podevsnef min–cost–flow network (as can be seen in Section 5.8). Hence the SPED–network is identical to Simple–Podevsnef and both have to produce the same results. In particular, the bend count has to be identical. □

Lemma 6.2. *Simple-Podevsnef can never calculate better solutions than SPED does.*

Proof. To proof the above lemma for all graphs, we only have to recognize that the set of solutions of Simple–Podevsnef is a subset of all valid SPED solutions:
As pointed out when constructing the SPED network, the only real extension modifies the rule of an edge to have a right bend, if its node angle with the edge to its left equals zero. Instead of demanding that this bend occurs on the edge itself, it has to happen anywhere on the corresponding hyperedge. All other additional modifications described in the sections 5.6.3 and 5.6.4 did not cut off all optimal solutions; they only guaranteed drawable results.
A right bend on the first subedge of a hyperedge – as Simple–Podevsnef generates it – also satisfies the modified right–bend constraint; the other restrictions do not increase the bend count. Thus any (optimal) Simple–Podevsnef solution is always easily transformable into a valid (not necessarily optimal) SPED solution, without changing the objective value. Hence it is impossible for Simple–Podevsnef to produce less bends than SPED does. □

Lemma 6.3. *For any given planarized graph $G = (V, E)$ and its planar topology T with h hyperfaces, Simple–Podevsnef needs at most h more bends than SPED does.*

Proof. (Refer to Fig. 6.3 as an illustrating example.)
Let SPED have calculated a valid shape, where each hyperface h_i has k_{h_i} bends. The following observation holds true for every hyperface h_i:
If the opening angle of the hyperface h_i is 90° or if there is a right bend on the subedge of h_i that lies next to the coltri, this shape obviously could also have been produced by Simple–Podevsnef, hence the latter algorithm may also need only k_{h_i} bends for h_i.
Let us assume that the coltri has an opening angle of 0°, and there is no right bend on the first subedge of h_i. Hence this shape is no valid Simple–Podevsnef shape (Fig. 6.3(left)). We can achieve a Simple–Podevsnef shape that is similar to the one SPED produced: we increase the node angle from 0° to 90° and add one right bend on the bundle partner of

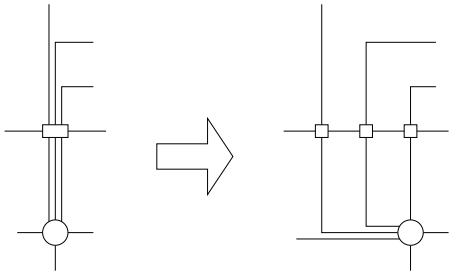


Figure 6.3: Hyperfaces in SPED (left), one more bend per hyperface in Simple–Podevsnef (right)

h_i (Fig. 6.3(right)). This transformation is similar to the handling of parallel edges in the Italian model of Simple–Podevsnef (Sec. 4.2, Fig. 4.3).
Note that this transformation is not influenced by collapsed coltris to the left of h_i, since these have to be extended by an analogous bend itself to become Simple–Podevsnef compliant. Neither does the transformation influence its surrounding area, since its overall shape remains identical.
Thus we need at most one more bend for each hyperface to transform the shape generated by SPED into a valid Simple–Podevsnef drawing. The optimal solution of Simple–Podevsnef is at least as good as this drawing. □

Theorem 6.1. *For any given planarized graph $G = (V, E)$ and its planar topology T with h hyperfaces, Simple–Podevsnef is a valid heuristics and needs at most h more bends than SPED.*

Proof. This theorem is directly implied by the above Lemmata 6.2 and 6.3. □

Corollary 6.1. *The bound given in Theorem 6.1 is sharp.*

Proof. Fig. 6.4 shows the graph used as a building block. If you put an arbitrary number of these blocks next to each other (joined by a simple edge, represented by a dotted line), every building block has exactly one hyperface. For every block, Simple–Podevsnef needs two bends, but SPED needs only one. Hence Simple–Podevsnef needs exactly h more bends than SPED.
Note that each block is optimally planarized, since there exists no other planarization generating less than two dummy nodes per block. □

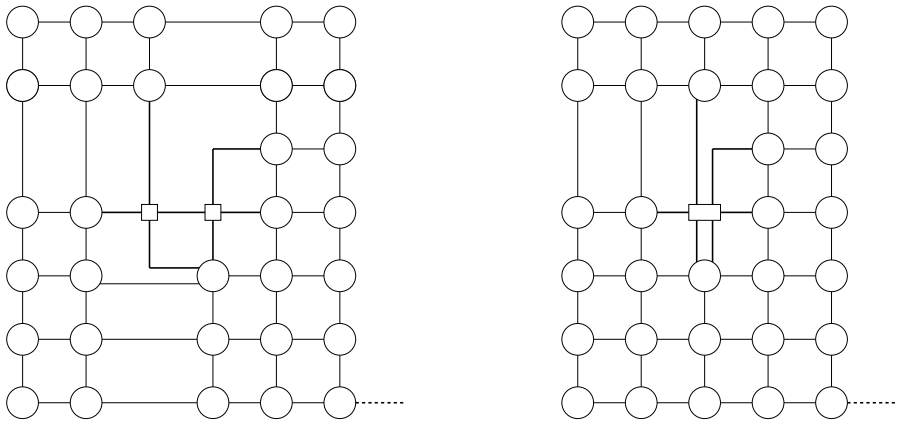


Figure 6.4: Example where the heuristic solution is as bad as it gets

6.3 Other Heuristic Approaches

Heuristic solutions for the SPED problem are an interesting topic. Thus – after the analysis of the approximation quality of Simple–Podevsnef – one may be tempted to look for other – perhaps better – algorithms.
The following is an incomplete description of concepts and experiments regarding other networks: it shows some easy approaches and eventually points out why these simple ideas do not work. After all, the classic Simple–Podevsnef seems to be quite a good heuristics for non-planar graphs.
The problem with SPED is the property of forcing a right bend "somewhere" on the hyperedge. Simple–Podevsnef circumvents this complexity by putting the right bend – if one is needed due to a 0° opening angle – on the first subedge of the hyperedge. But this also removes the possibilities of dummy merging.

6.3.1 Bend–on–End

A quite obvious idea would be to force this right bend on the last such subedge.
Such a network – named *Bend–on–End* – can easily be implemented by setting the capacities of all arcs that generate bends on the hyperedge and its bundle partner to zero, except for the arcs on the end face (Fig. 6.5).

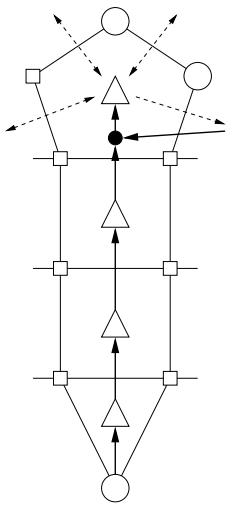


Figure 6.5: The network of a hyperface for the Bend–on–End heuristics

Thus no downflow can exist anymore; upflow can only be emitted from the source node of the hyperface, and has to be transmitted completely through the entire hyperface. Hence it is possible to use a construction similar to the one described for the alternative Simple–Podevsnef network (Sec. 4.3). This would lead to drawings where a hyperedge has to be either totally collapsed until its end face, or has an opening angle of 90°.
This is not just a bad heuristics, but even an invalid one: for graphs with a high density of hyperfaces – e.g., nearly complete graphs – this restriction is much too strong and no feasible solution can be found at all. The problem becomes obvious when thinking about orthogonal hyperfaces. As Figure 6.6 points out, there may exist vertices surrounded by "too many" coltris. Due to the symmetrical distribution property of Simple–Podevsnef–like drawing standards, some hyperfaces h_i are forced to have an opening angle of 90°.

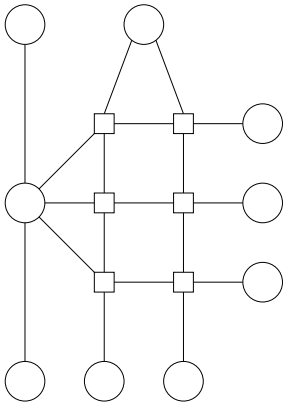


Figure 6.6: Bend–on–End may not be able to generate valid solutions at all

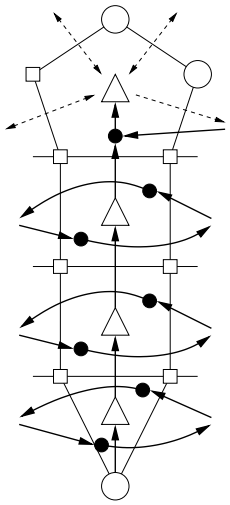


Figure 6.7: The network of a hyperface for the Lazy Bend–on–End heuristics

Thus there is an upflow in every such h_i. If too many of these h_i have orthogonal hyperfaces that prohibit the upflows, no valid solution can be found.

6.3.2 Lazy Bend–on–End

To circumvent this problem and allow some more flexibility, we enhance the above heuristics to *Lazy Bend–on–End* by allowing *bendable bundles*: In addition to the properties described for the above heuristics, a hyperface can bend itself completely, without demerging (cf. Remark 5.15).
This can be achieved by connecting the bend producing arcs of the subedges (except the last one) of each hyperedge directly to their counterparts on the bundle partner (Fig. 6.7). This allows flow to travel orthogonally through a hyperface, but it still prohibits all downflows and any upflow except the one from the source node.
This heuristics is now capable of drawing all possible graphs, but, as Figure 6.8[10] demon-

[10]The hyperfaces obviously could be drawn without a demerge on their first bundle bend, using an appropriately modified precompaction algorithm

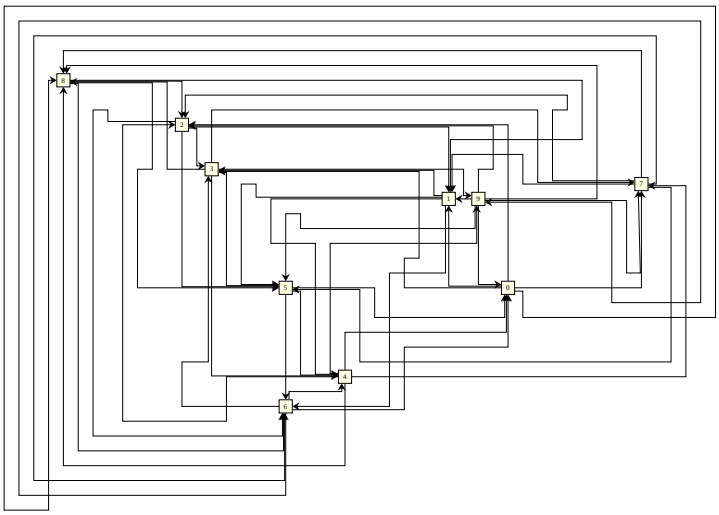


Figure 6.8: Example of a graph drawn by Lazy Bend–on–End. (124 bends)

strates, the results are quite poor: For the given non–planar graph with 10 nodes and 45 edges – the planarization step introduced 62 dummy nodes – the optimal SPED result requires 69 bends. While classic Simple–Podevsnef generates an approximation using 77 bends, Lazy Bend–on–End introduces 124.
Also note the unaesthetic bumps that occur on the end faces of the hyperfaces, due to the forced right bend on the last subedge of their hyperedges.

6.3.3 Righteous Bend–on–End

We can refine the above ideas based on the following thoughts: Simple–Podevsnef has the property that for every two edges e_1 and e_2 that start parallel (e_2 being the right one), the first bend on e_2 is a right bend. All other bends – whether on e_1 or e_2 – happen afterwards.
Coupled with the idea of bends to occur on the last hyperface and the necessity of bendable bundles, this leads to a heuristic with the following property: All hyperfaces have their bends collocated on the end face, except for right bends on the hyperedge and bends of the complete bundle.
This description can still be transformed into a min–cost–flow network, as Figure 6.9 demonstrates.
This heuristics leads to greatly improved results compared to Lazy Bend–on–End – Fig. 6.10 shows the same graph as before – but is still inferior to Simple–Podevsnef. It is easy to see that the unaesthetic bumps mentioned above also remain only slightly mitigated in this heuristics.

6.3.4 Simple–Podevsnef with Simple Postprocessing

Since Simple–Podevsnef seems to be a quite efficient heuristics, it appears to be a good idea to generate even better solutions based on this scheme.

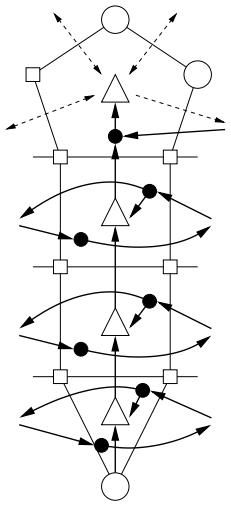


Figure 6.9: The network of a hyperface for the Righteous Bend–on–End heuristics

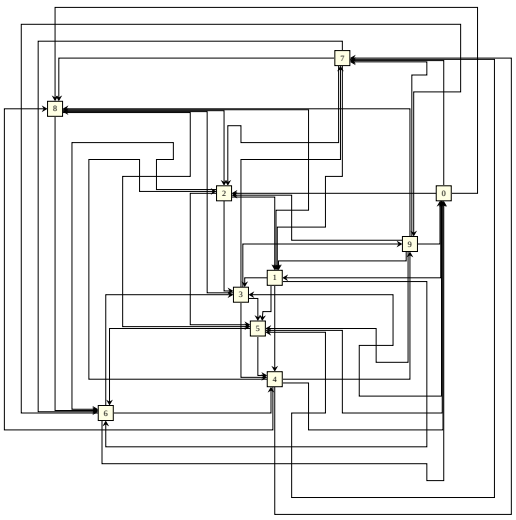


Figure 6.10: Example of a graph drawn by Righteous Bend–on–End. (102 bends)

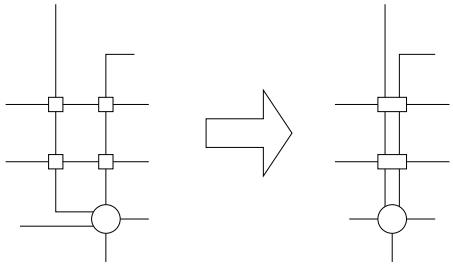


Figure 6.11: Postprocessing after Simple–Podevsnef

We can use Simple–Podevsnef as a first step in generating a valid drawing. We know that such a solution will not introduce dummy merges and bundle crossings. In a second step – the postprocessing – we search all hyperfaces for "unnecessary" bends, only introduced by Simple–Podevsnef's short–sighted demand for right bends on the first subedge (cf. Fig. 6.11).
We seek for a situation with the following properties:

1. The hyperface f^H (with the hyperedge e^H) has an opening angle of 90°.
2. The subedge e_1 – the first subedge of the bundle partner of e^H – has at least one right bend.
3. The angle between e_1 and its counterclockwise successor in the incidence list of source(e_1) is 0°.
4. The shape of f^H remains valid (in terms of the definition of SPED), if it would start collapsed and e_1 would have one right bend less.

Ad 4) This property can be easily checked by running through e^H and its bundle partner: e^H has to have at least one right bend, and f^H has to demerge before any otherwise invalid bend may occur.

If we find such a situation, we can safely reduce the opening angle of the hyperface to 0°, remove one right bend on e_1, and increase the angle between e_1 and its successor to 90°. Obviously the solution of this heuristics can never be inferior to Simple–Podevsnef. It even would optimally solve the example given in Corollary 6.1.

6.3.5 Simple–Podevsnef with Extended Postprocessing

We can enhance the above idea by the introduction of flow modifying postprocessing algorithms similar to the repair–function. These may perform operations related to downflows and left–FRFs based on concepts described in the Sections 5.6.3 and 5.6.4.
Such algorithms can broaden the field of hyperfaces that satisfy the fourth of the above properties (concerning the valid shapes of hyperfaces).

6.4 SPED for Clustergraphs

6.4.1 Definitions and General Approach

SPED can be easily extended to calculate the shape with a minimal number of bends for an especially interesting class of graphs: the so–called clustergraphs. Details on them and the concepts of calculating bend–minimal drawings of 4–planar clustergraphs can be found in Lütke–Hüttmann's diploma thesis [25].

Definition 6.1. General Clustergraph: *A tuple $C = (G, T)$ is called* clustergraph*, iff $G = (V, E)$ is a graph and T is a rooted tree, whose leaves are the nodes of G.*

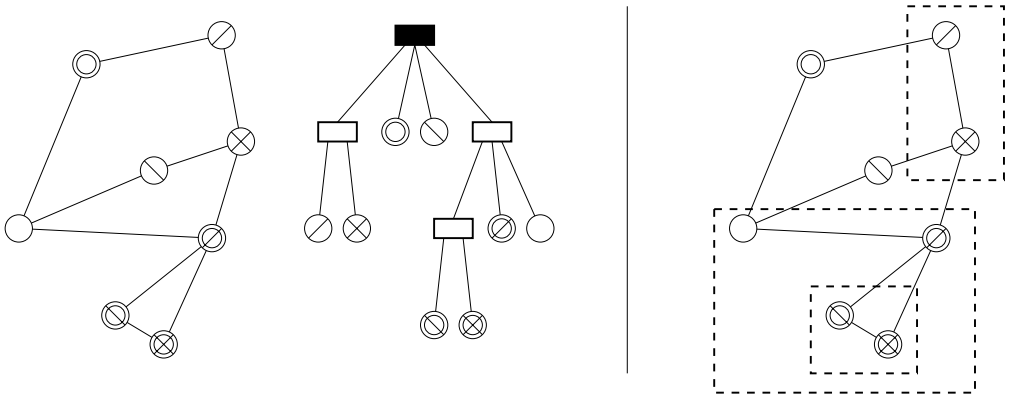


Figure 6.12: A clustergraph (right) consists of an underlying graph and a rooted tree

The inner nodes of T are called *clusters*, its root ($root(T)$) is the *root cluster*. A cluster c_1 is a *subcluster* of a cluster c_2, if and only if c_2 is an ancestor of c_1 in T. $T(c)$ denotes a subtree of T with root c. Its leaves $V(c)$ naturally induce the graph $G(c)$ (subgraph of G).
We define a subtype of such a clustergraph to simplify the proofs below:

Definition 6.2. Regular Clustergraph: *A clustergraph is regular, if all inner nodes of T have at least two children. This property is not necessary for the root cluster, if the clustergraph contains only one node.*

Thus, degenerated clusters containing only one node or only one direct subcluster are forbidden.

Definition 6.3. Cluster–Connected Clustergraph: *A clustergraph is cluster–connected, iff the induced graph $G(c)$ is connected for every cluster c.*

Unless stated otherwise, we will always discuss cluster–connected clustergraphs.
To discuss the process of generating drawable shapes, we need to define a planar embedding of clustergraphs based on the one given for regular graphs (Def. 2.7, page 4):

Definition 6.4. Planar Embedding of Clustergraphs: *A planar embedding Γ of a clustergraph $C = (G, T)$ satisfies the properties of a planar embedding for the underlying graph G. The clusters (except of the root cluster) are described as regions, represented by their borders. The region $R(c)$ of any cluster c has to satisfy the following properties:*

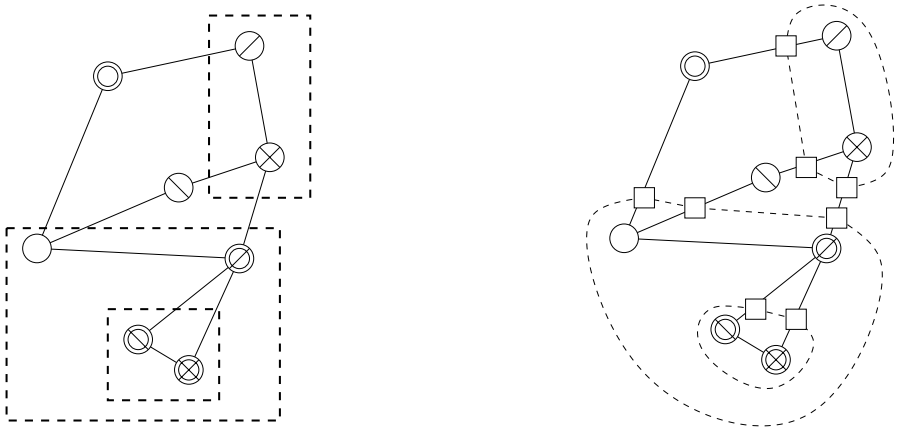


Figure 6.13: A clustergraph (left) can be transformed into a corresponding graph (right) by introducing dummy edges

1. *The shape of $R(c)$ is topologically equivalent to a circle.*
2. *The nodes of c are inside of $R(c)$.*
3. *The regions of the subclusters of c are inside of $R(c)$.*
4. *The regions of all other clusters are completely outside of $R(c)$.*
5. *Any edge connecting two nodes $u, v \in V(c)$ lies completely in $R(c)$.*
6. *Any edge connecting two nodes $u \in V(c)$, $v \notin V(c)$ crosses the border of $R(c)$ exactly once.*
7. *No edge connecting two nodes $u, v \notin V(c)$ crosses $R(c)$.*

The shape of a cluster region has to be rectilinear for orthogonal drawings. Traditionally, we want it to be rectangular, due to aesthetic reasons.
The basic idea behind drawing clustergraphs is quite simple and mostly a matter of the planarization step (Sec. 3.2.1): the clusters are "materialized" as cycles of edges representing its border (Fig. 6.13). The nodes (edges) introduced thereby are called *border nodes (border edges)*.
Thus we transform a clustergraph into a classic (planar) graph which we can draw with – more or less unmodified – classic algorithms for orthogonalization and compaction. After these steps, the borders of the clusters just have to be visually "marked".
We have to demand two things to assure appealing results:

1. The border nodes (they have a degree of 4 and are a special kind of dummy nodes) only have node angles of 90°.
2. If clusters have to be rectangular: For every circle of border edges we have to demand that only convex bends occur. Since clusters are never empty, and the edges are only allowed to be orthogonal, this will always lead to four 90° angles.

These demands can be easily included for approaches based on Tamassia's network:

Ad 1) If no 0° angles are allowed at all, this property is automatically satisfied. Otherwise, the capacity of the edge that controls the opening angle just has to be set accordingly.
Ad 2) There exist two arcs in the min–cost–flow network for every border edge incident to the faces f_1 and f_2 (the latter being outside of the respective cluster). The one from f_1 to f_2 obviously generates convex bends, whereby its reversal – sending flow inside the cluster – would generate concave bends. Thus we simply have to delete all such edges (f_2, f_1) (or give them a capacity of 0).
Just as for non–planar graphs, we can planarize cluster–connected clustergraphs for which no planar embedding exists [7], and use the resulting topologies for the calculation of the shapes and the dimensions.

6.4.2 Using SPED

By introducing border nodes $B(V)$ there is a third class of nodes in addition to the original nodes and the dummy nodes $D(V)$ (introduced by edge crossings). It is quite obvious that border nodes are a special kind of dummy nodes. Hence we want to define $B(V) \subseteq D(V)$. Thereby we can allow whole bundles to cross cluster borders, without having them to split up before.
As pointed out in Section 5.5.1, the complete set of lemmata considering the downflows, the left–FRFs and the repairability strongly rely on Lemma 5.1. This lemma assures that a node cannot be completely surrounded by coltris. Otherwise, all the described shifts of flows may result in infinite loops.
Lemma 5.1 assumes that metaedges are basically paths of subedges connecting two original nodes; but by the introduction of border edges and by treating border nodes as dummy nodes, we get such cyclic metaedges.

Lemma 6.4. *A high degree node of a planarized regular cluster–connected clustergraph cannot be completely surrounded by coltris.*

Proof. The situation to discuss consists of a node n surrounded by coltris and the responsible metaedge that is a cycle of border edges. Let c be the cluster represented by that border. Due to the definition of planar embeddings and since the clustergraph is cluster–connected we can deduce that n is the only element of c. Since there are no clusters with only one single element in regular clustergraphs, the above situation cannot exist. □

Thus there is no problem as long as we draw only regular (cluster–connected) clustergraphs.
Due to the property of Simple–Podevsnef to allow 0° angles only for high order nodes, the demanded property for the node bends on border nodes is automatically satisfied.
The convex–bend property for border edges can be implemented as described above.[11]
Thus we can formulate the following theorem:

Theorem 6.2. *For any planarized, regular, cluster–connected, simple, and self–loop–free clustergraph, SPED calculates a shape with the minimum bend number with respect to the SPED drawing paradigm. Rectangular cluster borders can be guaranteed if needed.*

Proof. This is a direct result of Theorem 5.2, Lemma 6.4 and the above description. □

[11] A scan through the previously described shifts concerning downflows, left–FRFs and repairability shows that there neither is a need for any concave bend, nor is any introduced.

Clusters with only one child – as they appear in general clustergraphs – are not only futile in nearly all cases, but also mostly irrelevant for the shape of the drawing. Thus, although drawing such questionable graphs is a quite dispensible functionality, it can also be done by SPED.

Lemma 6.5. *Every general clustergraph can be transformed reversibly into a corresponding regular clustergraph.*

Proof. We can remove every cluster c (except for the root cluster) that has only one child a and add a to the father of c. This father cluster simply stores information about this merge for later retracting. □

We only have to discuss what to do with the clusters removed by this transformation: Every cluster border introduces at least four bends. This bound may be exceeded by allowing non–rectangular borders.
Let c be a cluster that got removed, c' its father, and a its only child. Hence, after the transformation into a regular clustergraph, a became the child of c'. Since the calculated shape of a (cluster)graph includes no information about the dimensions of the drawing, we can simply reintroduce the deleted cluster borders using the shape induced by a:
If a is a node, the shape of c is a simple square around it. This only introduced the minimum number of bends required for any additional cluster. If a is a cluster, the shape of c should be identical to the shape of a. If only rectangular borders are allowed, c will introduce exactly four additional bends; thus the drawing remains optimal.
If the cluster borders are not restricted, the reintroduction of c could generate many bends more and render the solution suboptimal. To circumvent this, we can simply store the *thickness* of every cluster during the transformation process: all clusters have a thickness of 1 in the general clustergraph. If c is deleted, and its child a is a cluster, the thickness of a is simply increased by 1. The thickness of a cluster is then used for the costs on the arcs determining the bends on the border edges of the cluster.[12]

Theorem 6.3. *For any planarized, cluster–connected, simple, and self–loop–free clustergraph, SPED calculates a shape with the minimum bend number (with respect to the SPED drawing paradigm) by the usage of an additional polynomial pre- and postprocessing function. Rectangular cluster borders can be guaranteed if needed.*

Proof. This is a direct result of Theorem 6.2, Lemma 6.5 and the above description. □

Remark 6.1. *Due to the structure of cluster borders containing only one single element, and the fact that a cluster border can never be part of a hyperedge or its bundle partner, the fact that there may exist a high degree node surrounded by coltris assumably will not influence the shifts described in Chapter 5. Thus the pre- and postprocessing to remove and reintroduce such degenerated clusters may be completely unnecessary.*

[12] The shifts described in chapter 5 are transparent for this modification.

Chapter 7

Summary and Outlook

So eine Arbeit wird eigentlich nie fertig;
man muss sie für fertig erklären, wenn man nach Zeit und Umständen das Mögliche getan hat
J. W. v. Goethe

This thesis showed how to extend Simple–Podevsnef in such a way that non–planar graphs with given topology can be drawn with the minimum number of bends. This extension differentiates between original nodes and nodes only inserted to define edge crossings. It uses the concept of bundle crossings, necessary to achieve true bend minimality.
All constraints necessary to guarantee a valid shape are reduced to normal min–cost–flow constructions except for the main constraint that forces a right bend to appear on collapsed hyperfaces. We could not simplify this constraint due to the definition of non–planar Simple–Podevsnef (SPED). Any simplification would lead to a possible increase of the bend count.
Using Simple–Podevsnef even for non–planar graphs does not lead to optimal solutions in general, but the quality of the approximation is not that bad compared to other simple approaches using a min–cost–flow network (see also Appendix B). Its quality can be further increased by clever postprocessing.
Furthermore, SPED can be easily extended for non–planar cluster–connected cluster-graphs.

The thesis opens several problems and fields:

- (Dis)proving the intractability of SPED.
- Further heuristics, e.g., based on the LP-relaxation of SPED.[13]
- The addition of a secondary objective to encourage hyperfaces to demerge as late as possible, or the implementation of this demand by the means of an additional postprocessing–function.
- Modification to count bends on bundles (consisting of b edges) only once, instead of b times. How does the result change, using an adequate compaction step (especially when using a bus–type drawing scheme)?
- Extending Podevsnef for non–planar graphs.

[13] Note that you cannot simply round the values up due to the necessary balance in nodes/faces

Appendix A

Remarks on Fößmeier's Non–Planar Podevsnef

Ein Argument gleicht dem Schuss einer Armbrust -
es ist gleichermaßen wirksam, ob ein Riese oder ein Zwerg geschossen hat
Francis Bacon

Fößmeier describes some ideas on how to use his Podevsnef network for non–planar graphs in Chapter 5.6 of his doctoral thesis [11]. But there are some flaws in his proofs of time complexity as well as in the model itself.
The next two sections give an overview about his chapter dealing with non–planar graphs, and point out two mistakes. The first section points out why the proof of polynomial time complexity is invalid; the second section shows that his drawing standard is only able to draw a restricted class of planarized graphs.
Note that this appendix only discusses his chapter about non–planar graphs, not the Podevsnef–network itself as it is used for planar graphs.
All the following citations are from Fößmeier's doctoral thesis [11][14], unless otherwise stated. The thesis was written in German and there seems to be no English translation of it, except for published papers like [12]. It seems that only his planar Podevsnef (he calls it *Kandinsky* in his thesis) was ever published by him but not the non–planar one. The citations of his thesis are given in original German and summarized in English directly afterwards. It is not an aim to produce an exact translation.

A.1 Integer Properties and NP

Fößmeier makes strong use of negative cycles for his network constructions; particularly, he uses them for his non–planar Kandinsky network, too {Ch. 5.6, p. 91 et sqq.}.
In Section 5.6.1 he constructs a complex network that should guarantee a polynomial algorithm {page 93, last paragraph}. (In Section 5.6.2 he describes an alternative (exponential) approach using an ILP that is said to be quicker in most cases; this ILP would not be a direct 1–to–1 transformation of the network, and would have nothing to do with the LP mentioned below.)
A network with negative cycles is not solvable with normal min–cost–flow algorithms the way we need, because we want the flow inside a cycle to be equal to the flow that gets into it. We have to solve such networks using a *path–based min–cost–flow* algorithm {page 62, bottom; Sec.5.7.4, p. 109}.
Although the problem of path–based min–cost–flow is NP-complete in general [13], Fößmeier proves the polynomial running time for the special case of the Kandinsky base network (thus for planar graphs only) in Section 5.7.4 {page 109-111}. This simplification

[14] References to chapters and pages are marked by curly brackets.

is based on the property that all outgoing edges of a negative cycle lead into a single target node – which does not hold for the non–planar Kandinsky.
He mentions on page 111 that this augmenting algorithm is not polynomial for the non–planar Kandinsky network. Thus he proposes the following algorithm {in the proof of Theorem 5.3, page 102}:
The complex network should be transformed into a corresponding LP without changing anything. The path based property regarding negative cycles can be assured by simple constraints, forbidding the flow on such a cycle to be greater than the sum of the flows on its ingoing edges.
The mistake seems to reside in the following second paragraph of his proof {page 103}. It reads:

> *Für das klassische Min–Cost–Flow–Problem gilt, dass immer eine ganzzahlige optimale Lösung existiert, falls alle Kapazitäten ganzzahlig sind (siehe etwa [54]* [15]*). Daraus kann man folgern, dass das Polyeder der zulässigen Lösungen des LPs nur ganzzahlige Ecken hat. Daher kann eine ganzzahlige optimale Lösung in polynomialer Zeit gefunden werden. Da der Beweis in [54] ein Augmentierungsargument entlang von s-t-Pfaden benutzt, gilt dieselbe Aussage auch für pfadbasierte Min-Cost-Flow-Probleme, was die polynomiale Laufzeit des Knickminimierungsalgorithmus beweist.*

Fößmeier points out that since we know that normal min–cost–flow problems with integer capacities always have an optimal flow with integer values (as proven by Papadimitriou and Steiglitz [28]), the polyhedron of the feasible solutions has only vertices with integer coordinates. Therefore a polynomial algorithm for finding an optimal solution exists.
In his last sentence, he claims that since the cited proof uses an augmentation argument on s–t–paths, this conclusion considering the polynomial algorithm would also hold for path–based min–cost–flow problems. This conclusion is wrong:

- He gives the proof using an augmentation argument. But such an augmentation has an exponential running time in the case of non–planar Kandinsky, as he mentions on page 111.
- There are simple networks with negative cycles and the demand that these cycles may only transport as many units as are sent into them, that have fractional solutions better than the best integer solution (see Sec. 6.1 for examples). Hence the polyhedrons of such problems have non–integer vertices (cf. next list entry).
- As Garey and Johnson [13] point out, the problem of *path constrained network flow* – as they call it – is in general NP–complete (proven by Prömel, 1978). This classification also holds true for the related problem of integral flow with homologous arcs (Sahni, 1974). (Note that if you remove the integer property of these problems, you get a straight forward LP; thus their relaxations become solvable in polynomial time.)

Hence we know of no polynomial augmentation that can by applied a polynomial number of times to solve the path–based min–cost–flow network for non–planar graphs.

[15] Papadimitriou, C., K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice–Hall, 1982 (→ this thesis: [28])

Fößmeier proposes a different formulation as an ILP, instead of his complex network and its LP counterpart; he points out that although this ILP has exponential execution time, it runs much quicker in reality than the first approach. But nevertheless, the specification of his ILP {Section 5.6.2, page 103 et sqq.} is similar to the network: he basically replaced the negative cycles by a more direct formulation as inequalities. Such a transformation cannot change an NP–hard problem into one with polynomial execution time, and vice versa (if $P \neq NP$).

A.2 Undrawable Graphs

Fößmeier's non–planar Kandinsky network is quite restricted to keep it simple. In Theorem 5.3 {page 102} he claims:

> *Für eine gegebene Zeichnung* Γ *eines beliebigen Graphen* $G = (V, E)$ *mit* n *Knoten und* K *Kreuzungen kann eine topologieerhaltende Kandinsky–Zeichnung mit minimaler Knickanzahl in polynomialer Zeit berechnet werden unter der Nebenbedingung, dass für jeden Knoten* $v \in V$ *und je zwei zu* v *inzidente Kanten* e_1 *und* e_2 *alle Kanten, die ein kritisches Dreieck bezüglich* (v, e_1, e_2) *oder ein kritisches Viereck bezüglich* (v, e_1, e_2) *begrenzen, zusammen höchstens einen Knick aufweisen.*

His network should be able to draw any non–planar graph with given topology (in polynomial time), with the following constraint for every hyperface: the sum of all bends on all subedges of its hyperedge and its bundle partner that lie next to the coltri or to a colquod, is at most one.
He proves this theorem by referring to his Lemmata 5.6 and 5.7, and claims that the correctness of the theorem is directly implied by them. These two lemmata basically say that we can generate a valid drawing out of every optimal solution of the network and that every valid drawing can be transformed into a corresponding flow solution.
But these lemmata fail to prove that the described network is actually solvable; Fößmeier did not prove that there exists any valid drawing with the given bend restrictions *for any arbitrary non–planar graph with given topology*, as he claims in Theorem 5.3. There do exist graphs/topologies that cannot be drawn with the given restriction. Fig. A.1 shows a graph with corresponding topology that cannot be drawn, if both sides of a hyperface (excluding the end face) together always have only one single bend.

Remark A.1. *Of course the planarization of the given graph is quite unfortunate but valid. Furthermore, Theorem 5.3 claims to be able to calculate the shape for* any *topology. We can consider this example embedded in a bigger and more complex graph. Under these circumstances such a planarization is not so unlikely.*

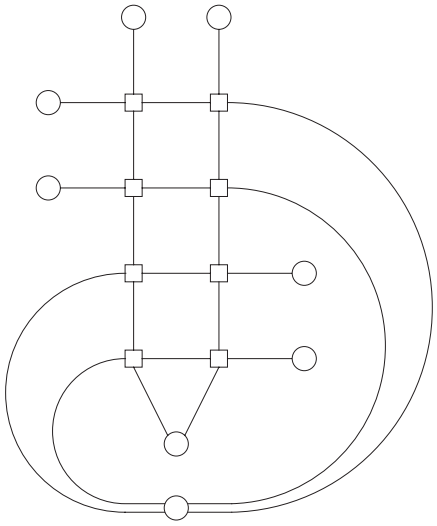


Figure A.1: Undrawable by non–planar Kandinsky; (circles are original nodes, squares are dummy nodes)

Appendix B

Computational Results and Examples

Lang ist der Weg durch Lehren, kurz und wirksam durch Beispiele.
Seneca

This section focuses on the real–world performance of SPED. It therefore uses the *Rome Graphs* test suite [8] containing over 11000 graphs with between 10 and 100 nodes each (Sec. B.1.1). But since those graphs are quite sparse – many of them are even planar – there is little difference between Simple–Podevsnef and SPED. Hence we also ran tests with *squared* Rome graphs (Sec. B.1.2). To see the full power of SPED, we ran tests with complete graphs, too (Sec. B.1.3).
The chapter closes with some examples of graphs drawn by SPED (Sec. B.2).

B.1 Computational Results

The following statistics are based on planarizations created by the standard *Subgraph-Planarizer* of the AGD library [1].

B.1.1 Rome Graphs

The *Rome graphs* [8] are a collection of about 11600 graphs based on 112 graphs taken from real–world applications. They have between 10 and 100 nodes each, and most of them are planar. Even the non–planar graphs are only very sparse, i.e., they contain very few edges and therefore very few dummy nodes after the planarization step. Thus we know from Lemma 6.1 and Theorem 6.1 that we cannot expect big differences between Simple–Podevsnef and SPED.
As Figure B.1 demonstrates, nearly 7000 graphs do not have any hyperface at all and 1500 others contain only one.
SPED solves all these graphs without any need for the repair–function, nor does the LP–relaxation ever produce non–integer solutions. The following diagrams only consider graphs with one or more hyperfaces, so there is potential for any improvement:
Figure B.2 underlines that the Rome graphs are inappropriate to effectively test SPED. Figure B.3 shows that the performance of SPED is below 0.1 second for every instance and there are no noteworthy differences to Simple–Podevsnef.

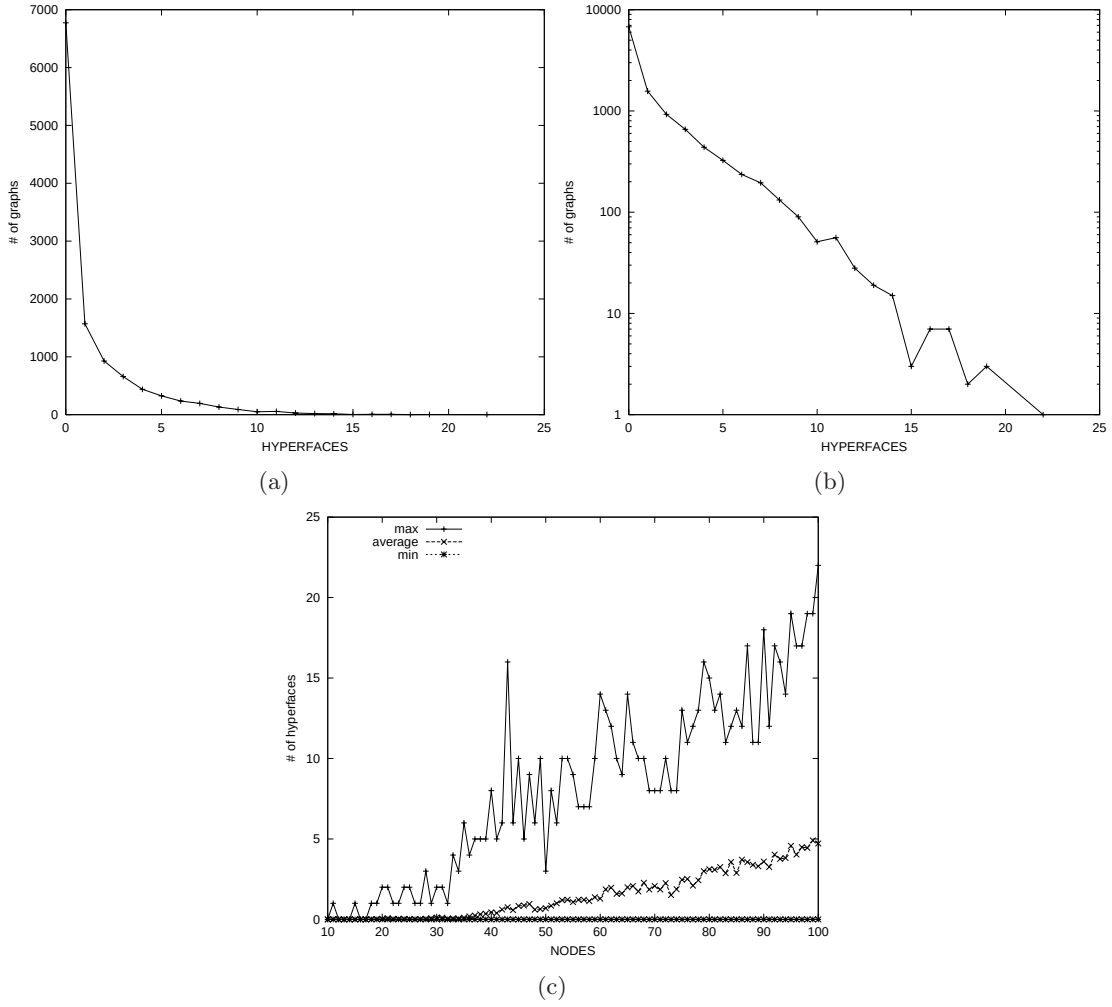
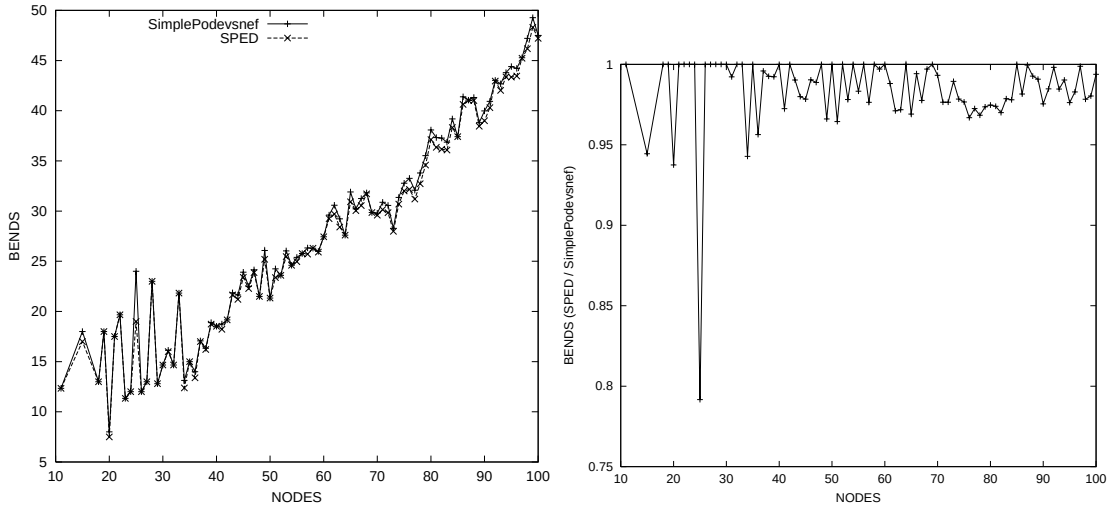


Figure B.1: [Rome Graphs] (a)(b) How many graphs exist with certain hyperface count? (linearly and logarithmically scaled); (c) minimal, average, maximum hyperface count per graph size.



(a) (b)

Figure B.2: [Rome Graphs] Bend counts are very similar between SPED and Simple–Podevsnef. Improvements are normally under 5%.

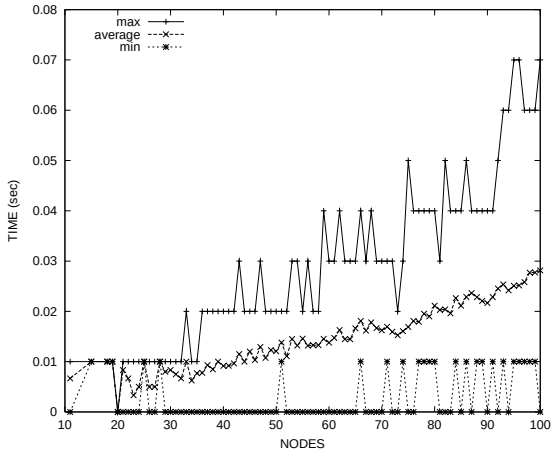


Figure B.3: [Rome Graphs] Time needed by SPED (minimal, average and maximum per graph size). The discretization is due to the resolution of the timer.

B.1.2 Squared Rome Graphs

Since the Rome graphs do not satisfy our demands for testing SPED appropriately, we *square* each Rome graph before drawing it:

Definition B.1. Squared Graph [9]: *Given a graph $G = (V, E)$, we denote by G^2 the graph on V in which two vertices are adjacent iff they have a distance of at most 2 in G.*

By this reproduceable operation we get denser graphs. These graphs are still not extremely dense, but have some more hyperfaces (Fig. B.4): only 1200 have still none, and there exist graphs with up to 280 hyperfaces.
As before, the following statistics are based only on graphs with at least one hyperface. When we look at the bend counts, we see that the bigger the graph gets, the bigger is the difference between SPED and Simple–Podevsnef. This is not only true for absolute values, but even the percentage of improvements itself increases (Fig. B.5): while for small graphs the improvement is similar to the one observed for classic Rome graphs,

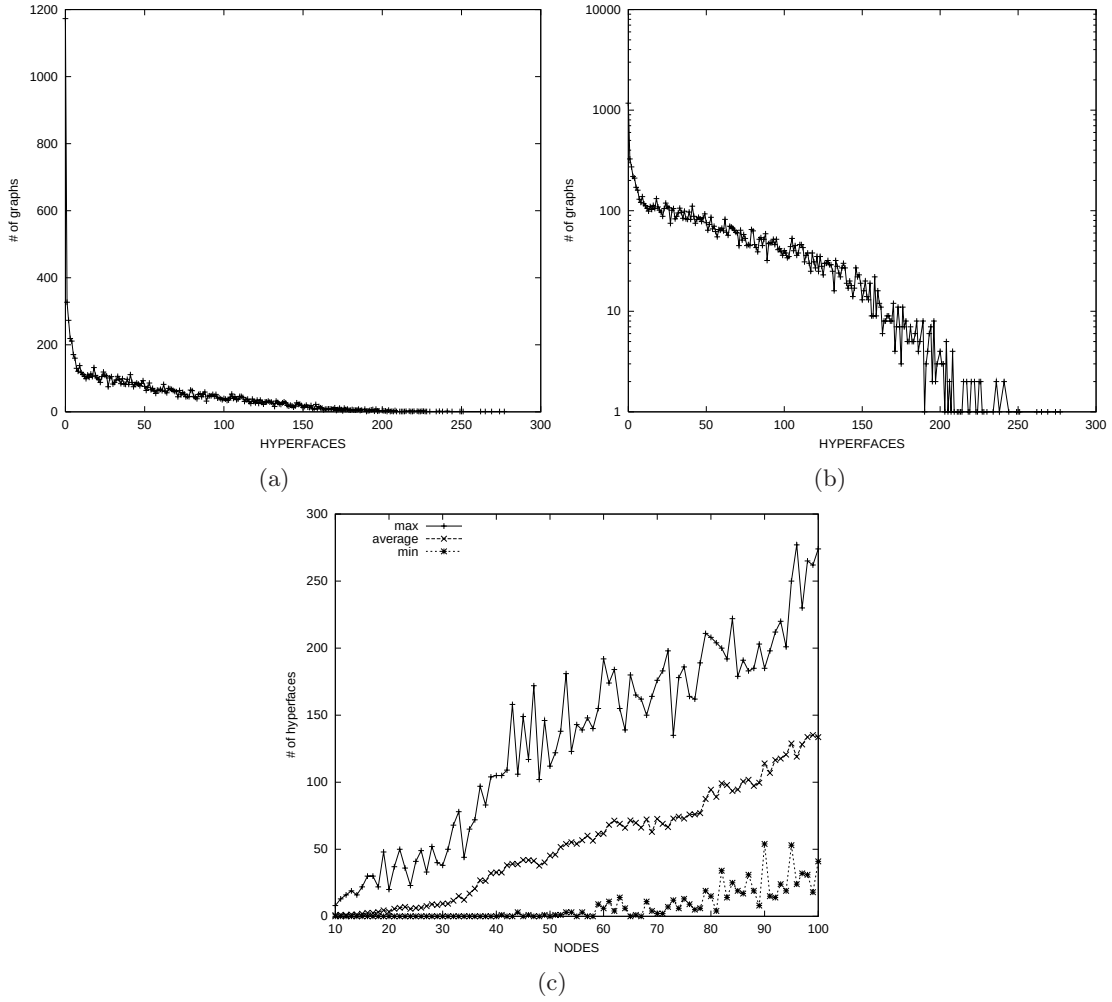


Figure B.4: [Squared Rome Graphs] (a)(b) How many graphs exist with certain hyperface count? (linearly and logarithmically scaled); (c) minimal, average, maximum hyperface count per graph size.

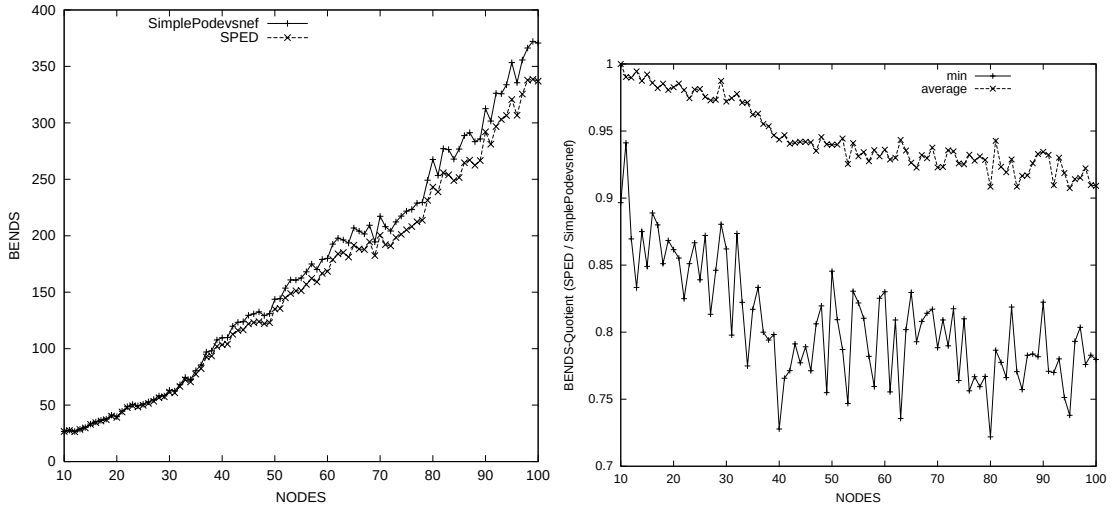
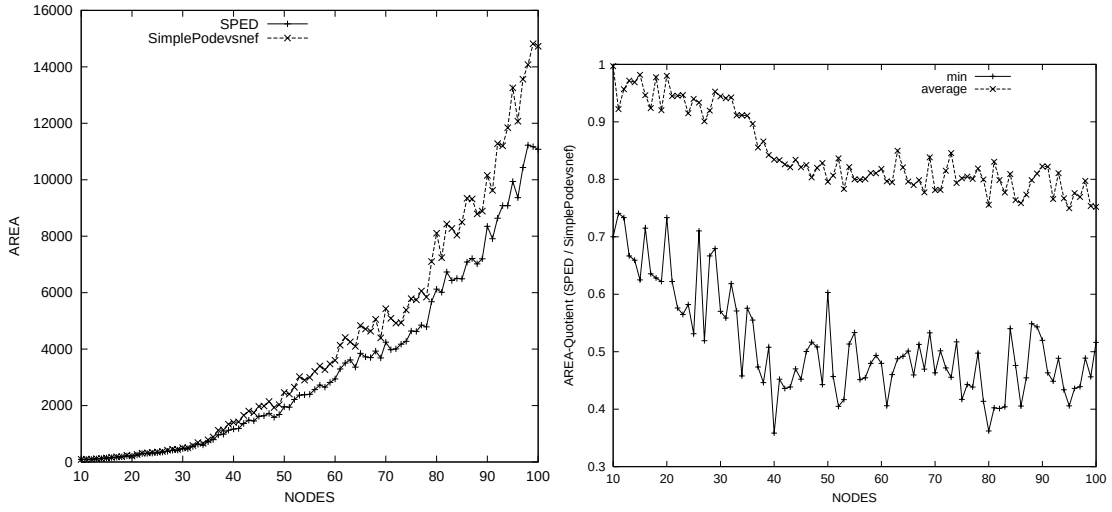


Figure B.5: [Squared Rome Graphs] Bend count: the bigger the graphs get, the better is the improvement rate.

we get an average of nearly 10% percent less bends for the big graphs. There even exist graphs where there is an improvement of over 25%.
The observations are similar when we look at the area needed for the final drawing (Fig. B.6). Even though the compaction step does not use *bendable bundles* (cf. Remark 5.15) – thus bundles demerge at the their first bend – there are big savings compared to Simple–Podevsnef: even middle–sized graphs profit of space reductions of about 20% on average. There exists reductions of over 60%.
Figure B.7 shows the time performance of SPED. While its average running times are below 1 second even for the big graphs, there are certain mavericks when SPED has to do branches after solving the LP relaxation. The performance of Simple–Podevsnef (not shown in the diagram for clarity reasons) is just a little better than the average performance of SPED (cf. Sec. B.1.3).
262 graphs needed the repair–function to become drawable; most of them had only one invalid FRF, 7 graphs had 2, and only one had 3 invalid FRFs. Figure B.8 shows the necessity of the repair–function for each graph size: only about 5 percent of the bigger graphs need it. The y–axis gives the ratio of graphs, hence a value of 0.1 means that we need to apply the repair–function on 1 out of 10 graphs.
Based on Corollary 6.1, we know that Simple–Podevsnef may need an additional bend for every hyperface compared to SPED. Figure B.9 shows that – in case of the squared Rome graphs – Simple–Podevsnef leads to far better results than its quality guarantee assures: only every fourth or fifth hyperfaces generates an additional bend.



(a) (b)

Figure B.6: [Squared Rome Graphs] The area requirement of the SPED is much smaller than Simple–Podevsnef's.

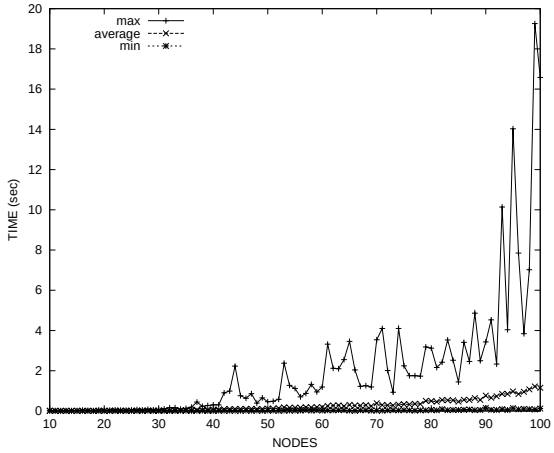


Figure B.7: [Squared Rome Graphs] Time needed by SPED (minimal, average and maximum per graph size)

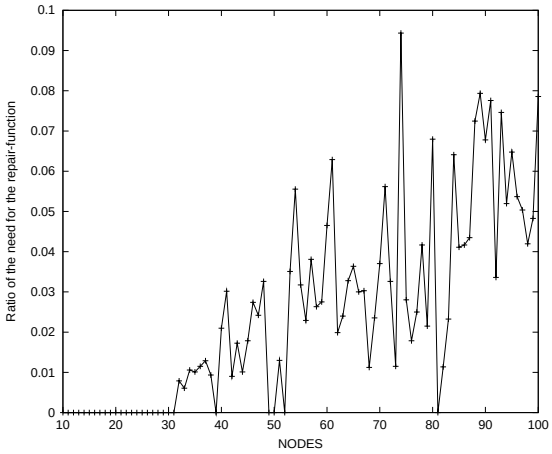
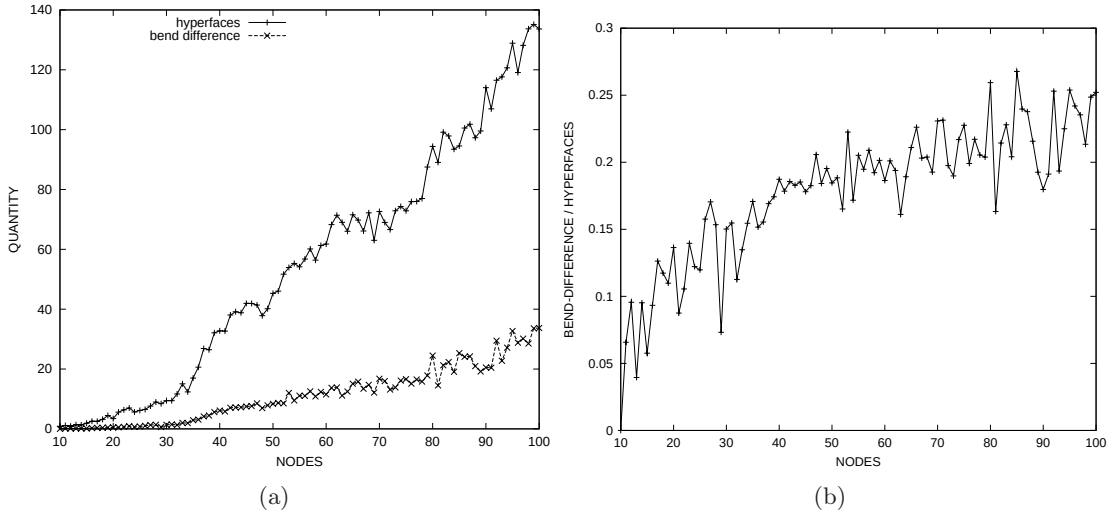


Figure B.8: [Squared Rome Graphs] Necessity of the repair–function

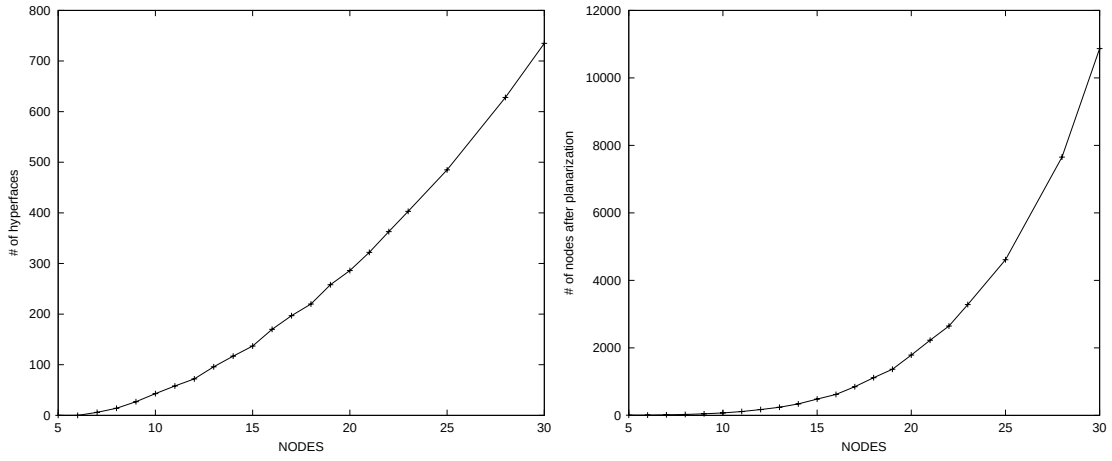


(a) (b)

Figure B.9: [Squared Rome Graphs] The relation between hyperface count and the difference of bend count between SPED and Simple–Podevsnef

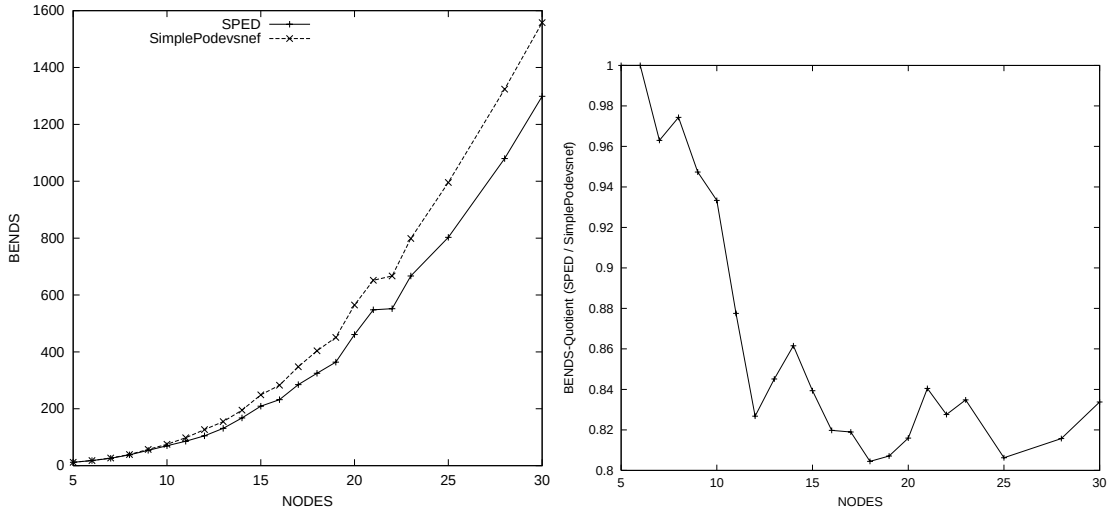
B.1.3 Complete Graphs

For better insight on the differences between SPED and Simple–Podevsnef we also used complete graphs (Fig. B.10). These graphs had between 5 and 30 nodes. Note that the planarization of the graph with originally 30 nodes (and therefore 435 edges) results in a graph with nearly 11000 nodes (and 22000 edges).
The repair–function is used only once for a single invalid FRF in K16. Figures B.11 and B.12 show the further increased benefit of SPED compared to Simple–Podevsnef. The performance of SPED is shown in Figure B.13; Figure B.14 shows the quality of Simple–Podevsnef in relation to the hyperface count.



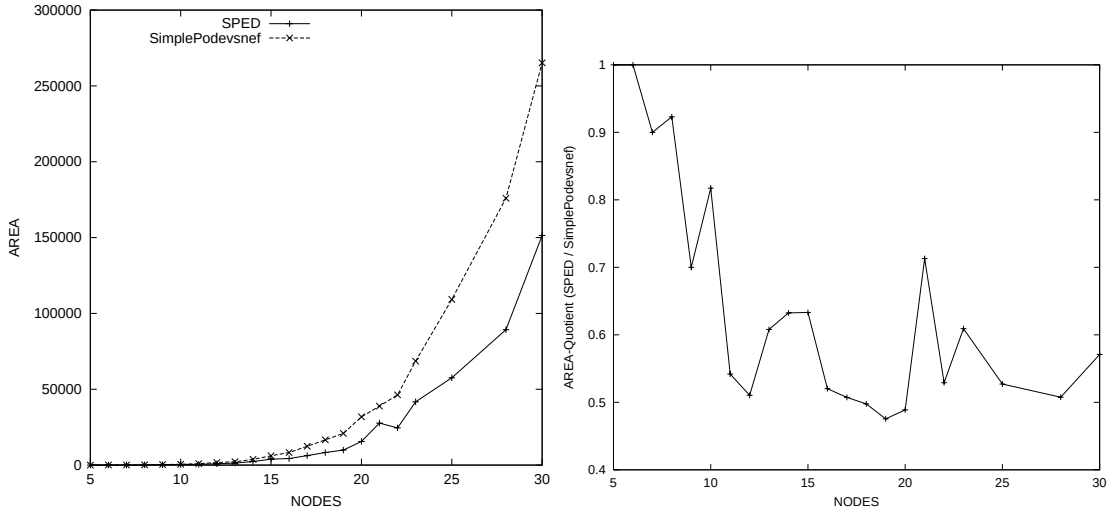
(a) Hyperface count per graph (b) Nodes after planarization per graph

Figure B.10: [Complete Graphs] Hyperface– and node count



(a) (b)

Figure B.11: [Complete Graphs] Bend count



(a) (b)

Figure B.12: [Complete Graphs] Area

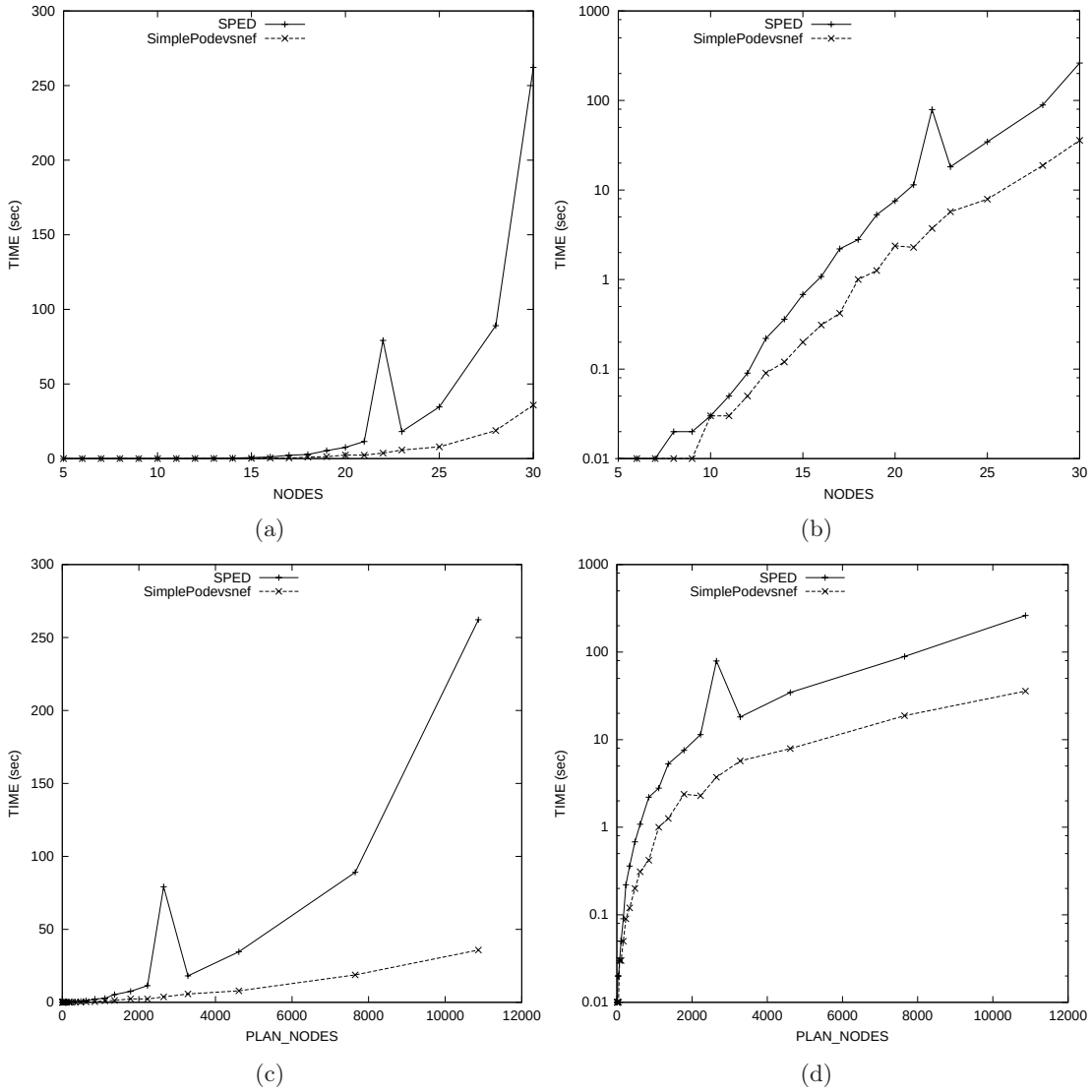
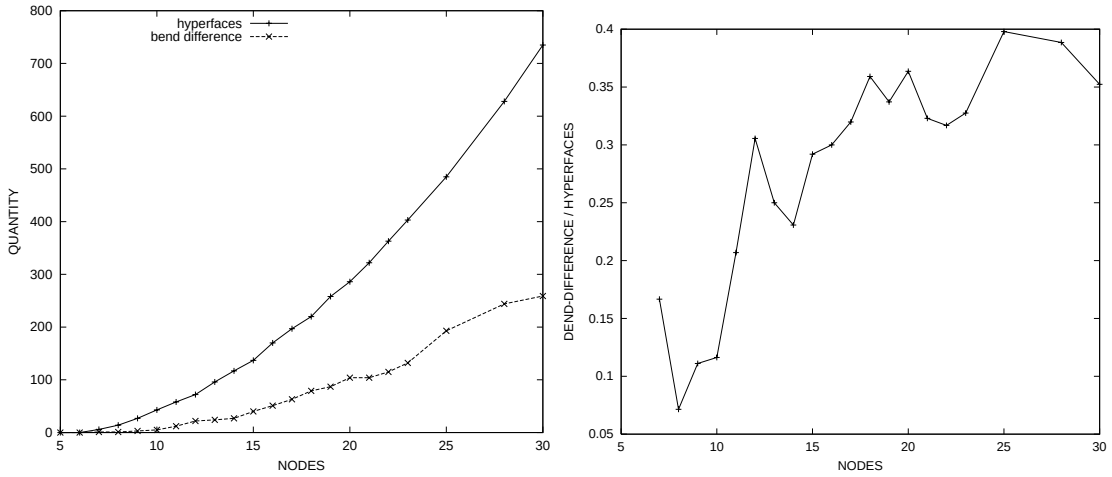


Figure B.13: [Complete Graphs] Time needed by SPED, compared to Simple–Podevsnef. (a)(b) use the node count of the original graphs on the x–axis (linearly and logarithmically scaled), (c)(d) use the node count after the planarization step (linearly and logarithmically scaled)

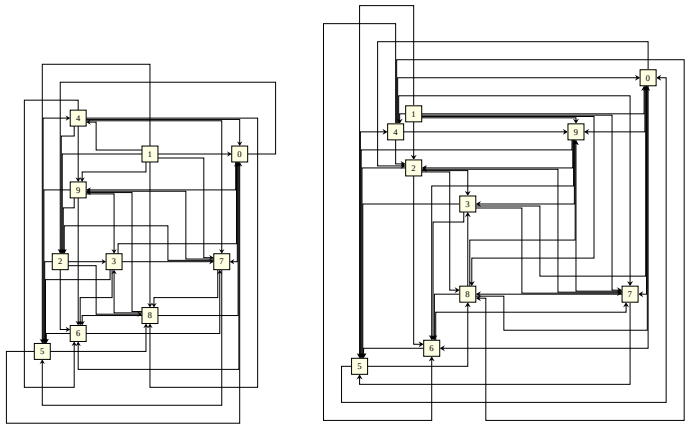


(a) (b)

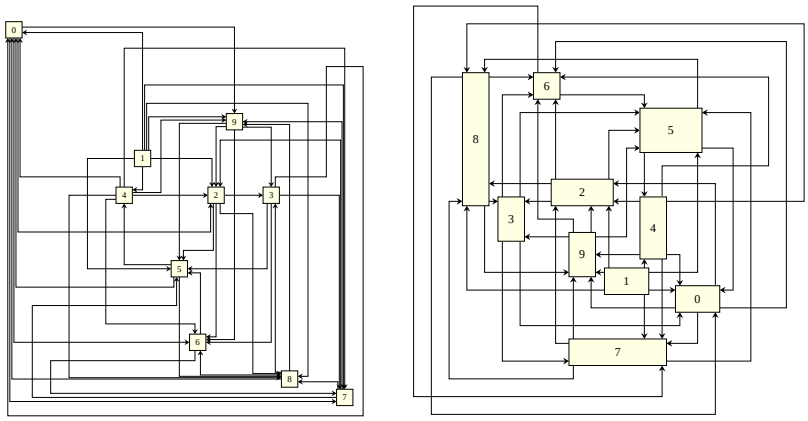
Figure B.14: [Complete Graphs] The relation between hyperface count and the difference of bend count between SPED and Simple–Podevsnef

B.2 Examples

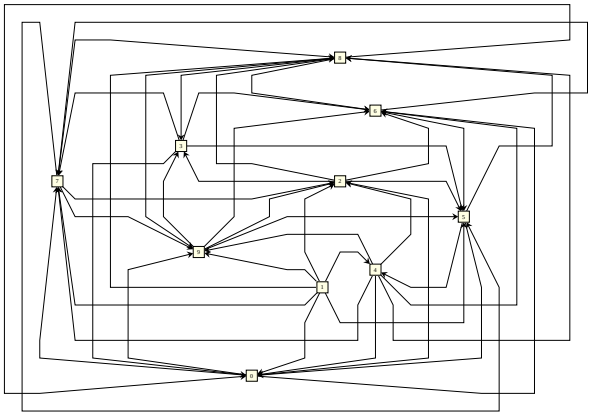
This final section shows some graphs drawn by SPED, compared to other drawing standards.
Figure B.15 shows a graph with 10 nodes and 42 edges, drawn by SPED, Simple–Podevsnef, Podevsnef, Giotto, and Quasi–Orthogonal. The latter four are not able to produce bundle crossings; the latter two do not have any concept of bundles at all). SPED needed only the LP–relaxation and no repair–function; its running time was indistinguishable from Simple–Podevsnef.
Figure B.16 shows the complete graphs K7–K15 drawn by SPED. Note that K7 is also a valid Simple–Podevsnef drawing. Figure B.17 shows K11 with the classic drawing style both for SPED and for Simple–Podevsnef.
Figure B.18/B.19/B.20 is based on a graph with 15/20/20 nodes and 75/90/150 edges. It contains 185/218/946 dummy nodes and 76/95/203 hyperfaces. SPED needs 120/143/324 bends and an area of 1152/1340/8400 grid squares. Simple–Podevsnef would need 136/161/401 bends and 1596/2548/16166 grid squares.



(a) SPED (b) Simple–Podevsnef

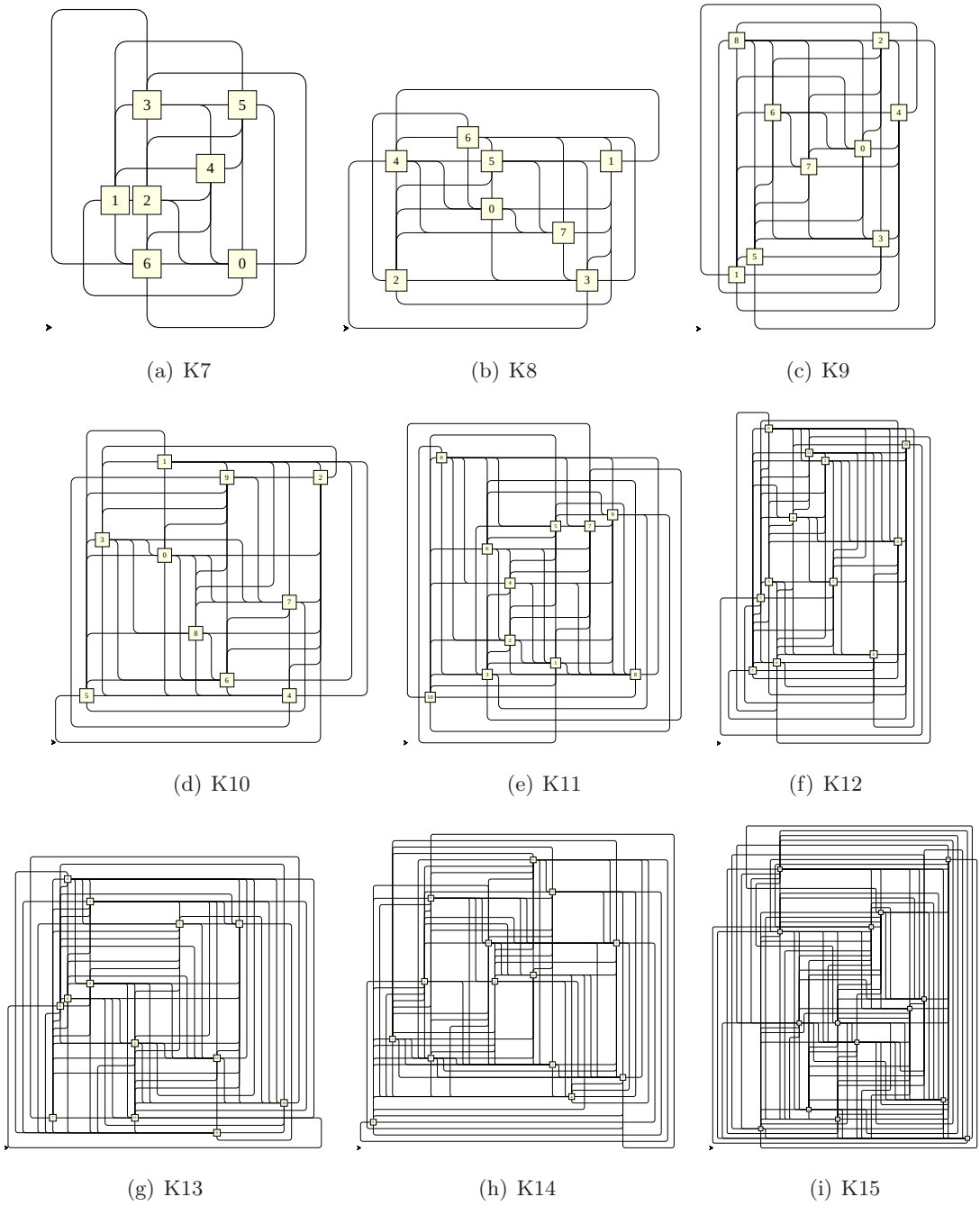


(c) Podevsnef (d) Giotto



(e) Quasi–Orthogonal

Figure B.15: 10 nodes, 42 edges, 5 different drawing standards (equally scaled)

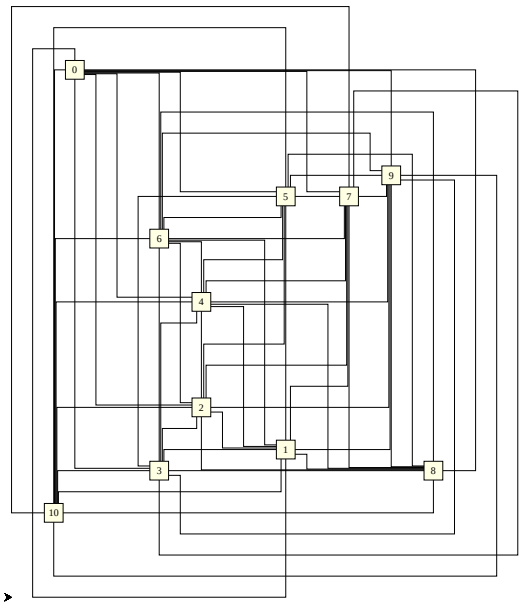


(a) K7 (b) K8 (c) K9

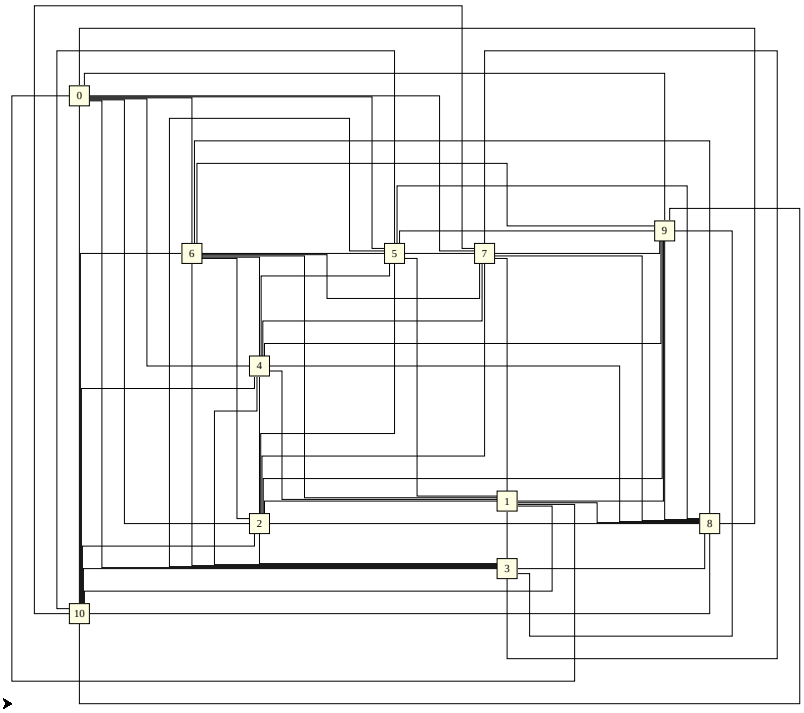
(d) K10 (e) K11 (f) K12

(g) K13 (h) K14 (i) K15

Figure B.16: Some complete Graphs, bus drawing style



(a) SPED



(b) Simple–Podevsnef

Figure B.17: K11, classic drawing style (equally scaled)

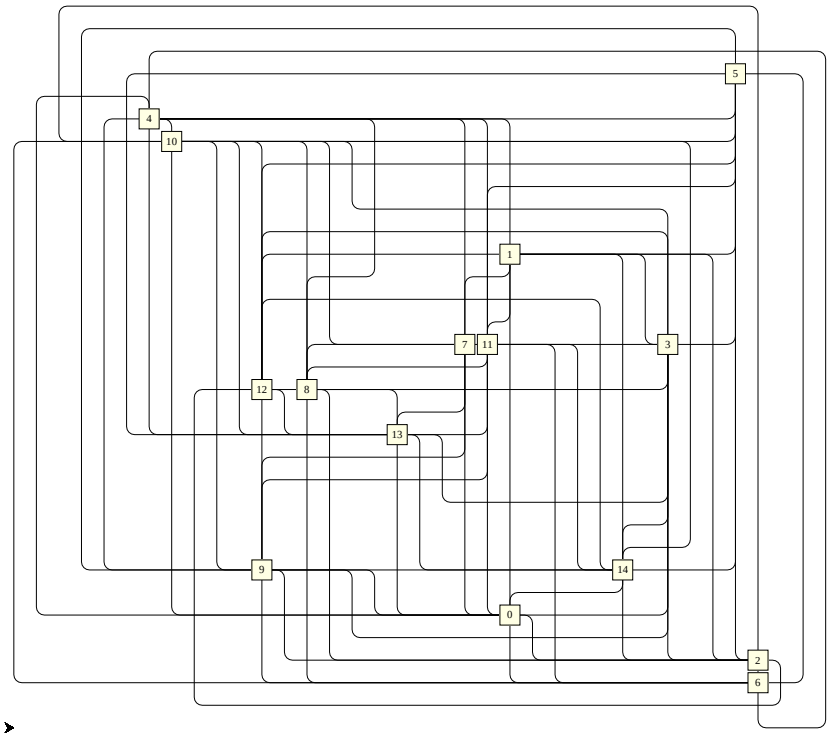


Figure B.18: 15 nodes, 75 edges

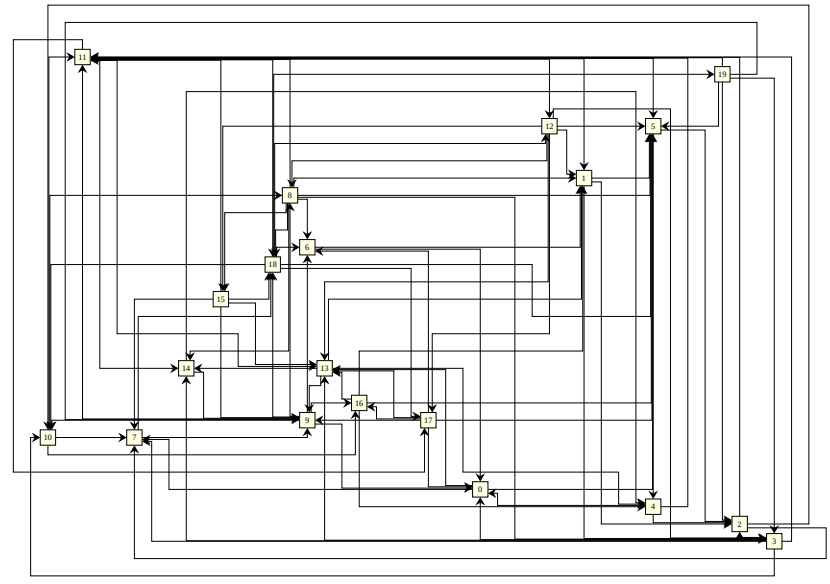
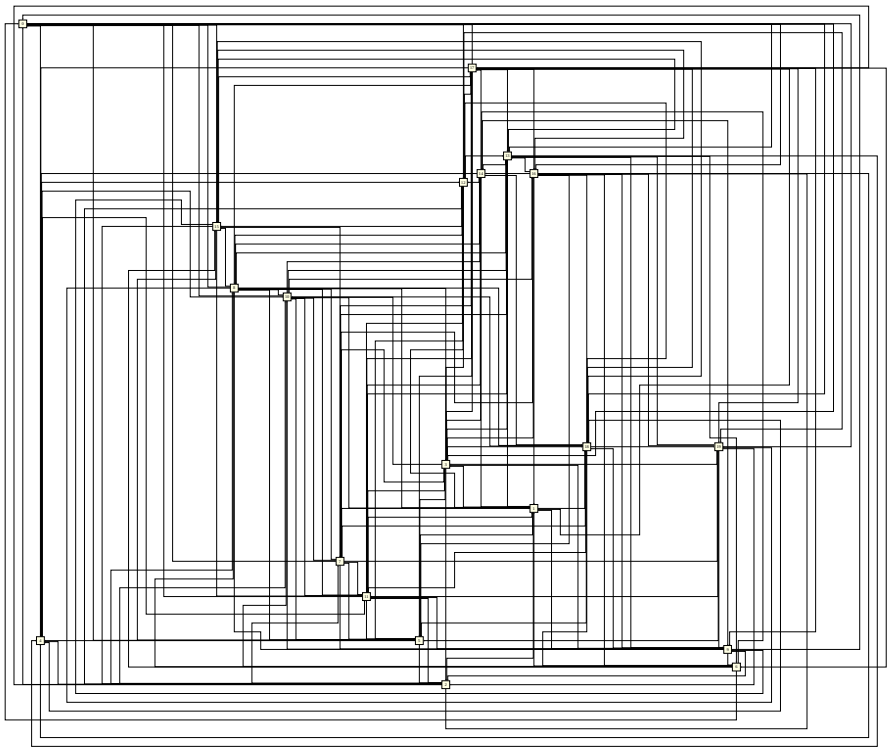
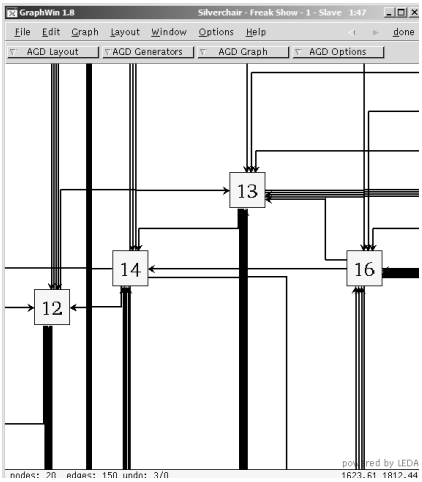


Figure B.19: 20 nodes, 90 edges



(a)



(b) Zoom to the four nodes on the top of the drawing

Figure B.20: 20 nodes, 150 edges; note the two merged bundles crossing each other between node 13 and 16

List of Figures

Die Mutter des Wortes
Hugo Ball, über den Begriff "Bild"

Bibliography

Seien Sie vorsichtig mit Gesundheitsbüchern - Sie können an einem Druckfehler sterben.
Mark Twain

[1] AGD – Library of Algorithms for Graph Drawing. *Online-Manual*. `www.ads.tuwien.ac.at/AGD`, v1.3, 2004.

[2] Ahuja, R.K., Magnanti, T.L., Orlin, J.B. *Network Flows*. Prentice Hall, 1993.

[3] Batini, C., Nardelli, E., and Tamassia, R. *A layout algorithm for data-flow diagrams*. IEEE Transactions on Software Engineering, SE-12(4), 1986.

[4] Bertolazzi, P., Di Battista, G., and Didimo, W. *Computing Orthogonal Drawings with Minimum Number of Bends*. IEEE Transactions on Computers, VOL 49, No.8, 2000.

[5] COIN–OR – COmputational INfrastructure for Operations Research. *Online-Manual*. `www.coin-or.org`, 2004.

[6] Di Battista, G. , Eades, P., Tamassia, R., and Tollis, I.G. *Graph Drawing*. Prentice Hall, 1999.

[7] Di Battista, G., Didimo, W., and Marcandalli, A. *Planarization of Clustered Graphs*. 9th International Symposium on Graph Drawing, 2001.

[8] Di Battista, G., Garg, A., Liotta, G., Tamassia, R., Tassinari, E., and Vargiu, F. *An Experimental Comparison of Three Graph Drawing Algorithms*. Proc. 11th Annu. ACM Sympos. Comput. Geom., 1995.

[9] Diestel, R. *Graph Theory*. Springer Verlag, 2nd Edition, 1997, 2000.

[10] Edmonds, J. and Karp, R.M. *Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems*. Journal of the ACM, Vol. 19, No. 2, 1972.

[11] Fößmeier, U. *Orthogonale Visualisierungstechniken für Graphen*. Dissertation, Eberhard-Karls-Universität zu Tübingen, 1997.

[12] Fößmeier, U. and Kaufmann, M. *Drawing High Degree Graphs with Low Bend Numbers*. Graph Drawing (Proc. GD '95), 1996.

[13] Garey, M.R. and Johnson, D.S. *Computers and Intractabilities*. Bell Labs, W.H. Freeman, 1979, 1999.

[14] Grötschel, M., Lovasz, L., and Schrijver, A. *The ellipsoid method and its consequences in combinatorial optimization*. Combinatorica, 1:169–197, 1981.

[15] Gutwenger, C., Jünger, M., Klau, G.W., Leipert, S., and Mutzel, P. *Graph Drawing Algorithm Engineering with AGD*. Lecture Notes in Computer Science Vol. 2269 / 2002, Springer-Verlag Heidelberg, 2001.

[16] ILOG CPLEX. *Online-Manual*. `www.cplex.com`, v8.1, 2004.

[17] Jayakumar, R., Thulasiraman, K., and Swamy, M.N.S. *On maximal planarization of non-planar graphs*. IEEE Transactions on Circuits Systems, 33(8):843–844, 1986.

[18] Klau, G. W. and Mutzel, P. *Quasi-orthogonal drawing of planar graphs*. Technical Report MPI-I-98-1-013, Max-Planck-Institut für Informatik, Saarbrücken, 1998.

[19] Klau, G. W., Klein, K., and Mutzel, P. *An experimental comparison of orthogonal compaction algorithms*. Proceedings of the 8th International Symposium on Graph Drawing, 2000.

[20] Klau, G.W. *Quasi–orthogonales Zeichnen planarer Graphen mit wenigen Knicken*. Diplomarbeit, Universität des Saarlandes, 1997.

[21] Klau, G.W. *A Combinatorial Approach to Orthogonal Placement Problems*. Dissertation, Universität des Saarlandes, 2001.

[22] LEDA – Library of Efficient Data Types and Algorithms. *Online-Manual*. `www.algorithmic-solutions.com`, v4.4.1, 2004.

[23] Leitsch, A. *Algorithmen-, Rekursions- und Komplexitätstheorie*. Skriptum zur Vorlesung, TU Wien, 2002.

[24] Lemhauser, G.L. and Wolsey, L.A. *Integer & Combinatorial Optimization*. Wiley Interscience, 1988.

[25] Lütke-Hüttmann, D. *Knickminimales Zeichnen 4–planarer Clustergraphen*. Diplomarbeit, Universität des Saarlandes, 2000.

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

[27] Mehlmann, A. and Hartl, R. *Methoden der Optimierung*. Skriptum zur Vorlesung, TU Wien, 2001.

[28] Papadimitriou, C. H. and Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[29] Sahni, S. *Computationally related problems*. SIAM J. Comput. 3, 262–279, 1974.

[30] Stroustrup, B. *The C++ Programming Language, Special Edition*. Addison-Wesley & AT&T Labs, 3rd Edition, 1997.

[31] Tamassia, R. *On embedding a graph in the grid with the minimum number of bends*. SIAM Journal on Computing 16(3), 1987.

[32] Tamassia, R., Di Battista, G. and Batini, C. *Automatic graph drawing and readability of diagrams*. IEEE Trans. Syst. Man. Cybern., SMC-18(1), 1988.

[33] Weiskircher, R. *New Applications of SPQR-Trees in Graph Drawing*. Dissertation, Universität des Saarlandes, 2002.