Christian-Albrechts-Universität zu Kiel

Bachelor Thesis

Implementing an Algorithm for Orthogonal Graph Layout

Ole Claussen

September 29, 2010

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Miro Spönemann

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

Abstract

A planar and orthogonal layout is a popular approach in automated graph drawing. Minimizing the number of edge crossings and restricting the graph drawing to horizontal and vertical lines usually results in an easy to read and visually appealing layout. In many practical applications, especially UML class diagrams, entity-relationship-diagrams for databases or electric wiring schemes, an orthogonal graph drawing is one of the most useful layouts. This thesis provides a basic layout algorithm that will result in a graph drawing with a low number of edge crossings and an orthogonal layout that is suitable for such applications.

Contents

List of Figures

1 Introduction

There exist various approaches and algorithms on the layout of graphs, each meeting different aesthetic criteria. The best choice for a suitable algorithm usually depends on user preference and the use case. This thesis gives an example for a layout algorithm that provides an orthogonal drawing of a graph. The drawing of a graph is considered orthogonal if all edge sections (i. e. parts of edges, separated by *bend points*) in the drawing are either horizontal or vertical, which implies that all angles in bend points or between edges are multiples of 90°. As a side-effect, and fulfilling additional aesthetic criteria, the presented algorithm used for orthogonalization keeps the number of bend points as well as the number of edge crossings in the graph drawing very low. Such a layout will result in a clearly arranged and highly readable graph. It is therefore a suitable algorithm in many applications. It is especially a preferred choice for many diagram layouts in UML, database applications (such as entity-relationship diagrams) or many technical use-cases such as wiring schemes or VLSI design.
This thesis is based on a Java implementation of the presented algorithm. This implementation was the subject of a bachelor project at the Christian-Albrecht-University in Kiel in 2010. The major part of the project was the implementation of the planarity testing algorithm by John M. Boyer and Wendy J. Myrvold [1] as described in Chapter 4, hence that part occupies a relatively large part of this document. The second part of the project and base of the bachelor thesis consisted in the implementation of algorithms for orthogonalization and compaction of graphs, and the integration of these algorithms in in a graph layouter based on the topology-shape-metrics approach as described in Chapter 3.

1.1 Structure of this document

This first introductory chapter will give a short overview of the thesis and the implementation of the graph layout algorithm, while Chapter 2 will recapture the basic definitions of graph theory relevant to the algorithms presented in this thesis. Chapter 3 will give an overview of the main algorithm used by the layouter and its individual phases, along with the important terminology required in these phases of the algorithm. The following chapters will then deal with each of the phases in the main algorithm: the planarity test in Chapter 4, planarization in Chapter 5, orthogonalization in Chapter 6 and compaction in Chapter 7. Chapter 8 gives a final conclusion with some examples what can still be done in the project.

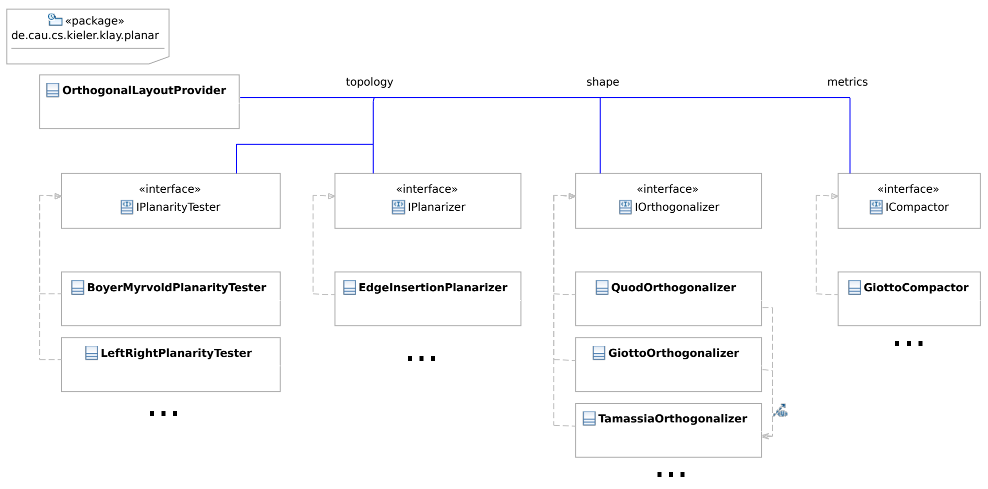


Figure 1.1: An overview of the implementation of the layouter plug-in

1.2 Implementation

The layout algorithm has been implemented in the Java programming language and is part of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1] project. The KIELER project consists of a series of plug-ins to the Rich Client Framework of the Eclipse IDE[2]. It has been developed by the Real-Time and Embedded Systems Group of the Department of Computer Science at the Christian-Albrechts-Universität zu Kiel. Its aim is to enhance the graphical model-based design by integrating various modeling languages and offering automatic layout for their graphical components. The algorithm presented in this thesis is an example of the various layout algorithms provided in the KIELER project for the automatic layout of diagrams. The implementation accompanying this thesis can be found in the KIELER Eclipse plug-in `de.cau.cs.kieler.klay.planar`.
The implementation is based on a simple graph data structure, as seen in Figure 1.2. Since the project consists of a large variety of interchangeable algorithms, it heavily relies on the strategy design pattern to provide common interfaces and abstractions for the individual algorithms. Figure 1.1 shows a rough layout of the project. Each step in the algorithm (i. e. topology, shape and metrics steps) has its own interface for implementing algorithms, which can therefore be easily exchanged or improved separately. As for the implemented algorithms, two state-of-the-art algorithms for planarity testing were successfully implemented, the *Boyer-Myrvold-Algorithm* [1] and the *Left-Right Planarity Test* [13]. Along with a basic algorithm for planarization, which is the subject of an accompanying bachelor thesis by Christian Kutschmar [19], these algorithms provide the topology part of the layouter project. Additionally,

[1] `http://www.informatik.uni-kiel.de/rtsys/kieler`
[2]`http://www.eclipse.org`

some simple approaches to orthogonalization and compaction algorithms were implemented to gain basic layouting functionality. These still provide far from optimal solutions, especially since the implemented orthogonalizer is limited to a maximal vertex degree of 4. Unfortunately, the compaction algorithm has not been finished completely at the time this thesis is published.

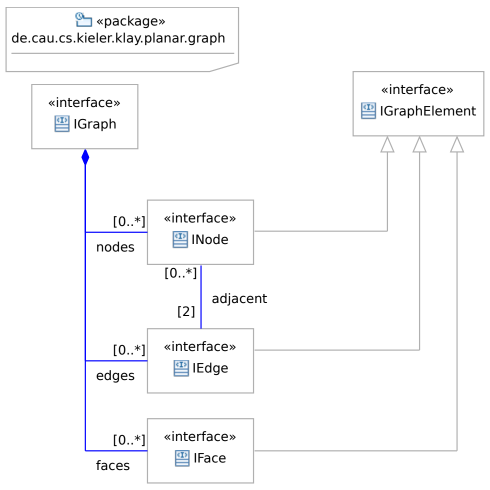


Figure 1.2: On overview over the graph data structure used in the Java implementation

1.3 Related Work

This thesis basically documents the implemented layout algorithm based on the *topology-shape-metrics* approach. Therefore, all works describing this approach or algorithms used in one of its phases are related to this thesis. Some basic information on automated graph drawing and especially orthogonal graph drawing using the used approach can be found in the various works of Tamassia et al. [26, 3, 2]; or the book on graph drawing by Michael Kaufmann and Dorothea Wagner [14]. Single phases of the approach are detailed in various works of Tamassia, Gutwenger et al. or Klau et al. [24, 25, 10, 9, 22, 15, 16, 5, 17]. Special note should be taken of the bachelor thesis by Christian Kutschmar [19], which describes the planarization algorithm used in the topology phase, since it is based on the same graph layouter implementation as this thesis.

2 Basics of Graph Theory

Since the subject of this thesis is a layout algorithm for graphs, a basic understanding of graphs is vital for its comprehension. This chapter will recapture the most important definitions of graph theory that are used in the context of this document.

2.1 Graphs

A *graph* is an abstract object used to model objects and their relations. Graphs are formally defined as an ordered pair $G = (V, E)$, where V is a set of *vertices* and E is a set of *edges*. While vertices are the basic elements in a graph, the edges represent links between these elements. Edges are defined as an unordered pair (v, w), where v and w are elements from V. The existence of an edge (v, w) in a graph G means that the vertices v and w are connected. v is said to be *adjacent* to w. The number of vertices $|V|$ is the *order* of a graph and the number of edges $|E|$ is the *size* of a graph. The number of edges adjacent to a vertex v is called the *degree* $\delta(v)$ of a vertex.

A graph is a *directed* graph if its edges maintain information on its direction, i. e. they are defined as an ordered pair of vertices, where the first vertex is the source and the second vertex the destination of the edge. A *mixed* graph contains both directed and undirected edges. In a *multigraph*, multiple edges between the same vertices are allowed, as well as *self loops*, i. e. directed or undirected edges with the same source and destination vertex.

A *connected component* in a graph is a subset of its vertices and edges, where every vertex is reachable from every other vertex via a path. A *k-connected component* is one such component, where a connected component remains after the removal of any k vertices. In particular, a biconnected component (bicomp) remains a connected component after the removal of one vertex. A vertex connecting separate bicomps in one connected component is called a *cut vertex*, while an edge connecting separate connected components is called a *cut edge* or *bridge*.

2.2 Planarity

The drawing of a graph is considered *planar* if it does not contain any edge crossings. A graph, on the other hand, is considered planar if a planar drawing exists for the graph. A *planar embedding* of a graph defines the order in which edges have to be added to every vertex in order to obtain a planar drawing. For example the graph shown in Figure 2.1a is planar, but the drawing is not. In Figure 2.1b, the same

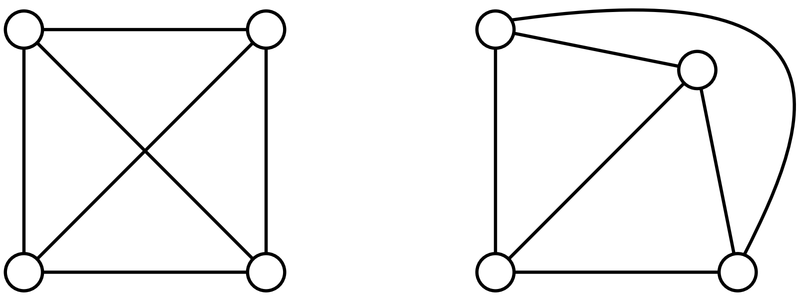

(a) A planar graph (b) A planar drawing of the graph

Figure 2.1: Examples of planar graphs

graph is drawn with a proper planar embedding, and therefore the drawing of the graph is planar. The two graphs in Figure 2.2a and Figure 2.2b are both not planar, since there is no possible way to draw these graphs in a planar way. These two graph are known as *Kuratowski* graphs K_5 and $K_{3,3}$. It is proven that every nonplanar graph contains either a K_5 or a $K_{3,3}$ as a subgraph [18].

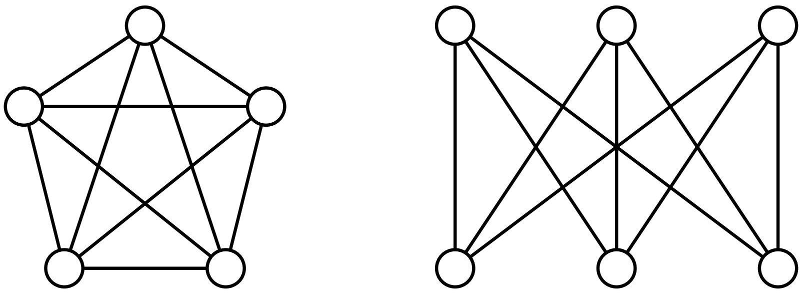

(a) The K_5 graph (b) The $K_{3,3}$ graph

Figure 2.2: Kuratowski graphs, which are both not planar

2.3 Faces and the Dual Graph

A planar embedding defines the set of *faces* in a graph. Every cycle in a planar graph that has no edges leading from the cycle into the region surrounded by it forms a face. A face in a graph is adjacent to another face, if it shares an edge with it. The unbounded area on the outside of a graph drawing is the *external face*, while all other regions are *internal faces*. Dependent on the faces is the *dual graph* $\tilde{G}$ of a planar graph G. The dual graph $\tilde{G}$ contains a vertex for every face in G. Adjacent faces in G are represented by adjacent vertices in $\tilde{G}$. This implies that for every edge in G separating two faces, there is an edge in $\tilde{G}$ that connects the two vertices representing these faces. For every bridge in G, the dual graph contains a self loop.

2.4 Flow Networks

Algorithms often attempt to reduce a problem to another problem with a well known solution. For instance, algorithms in the later phases of the approach discussed in this thesis are usually reduced to problems in *flow networks*. Flow networks are frequently used to model transportation problems such as traffic on roads or electrical current. A flow network is a directed graph in which a *flow* travels on edge paths. The edges in a flow network are called *arcs*. Each arc (u, v) has a capacity value $c(u, v)$, which acts as an upper bound for the allowed flow through theses arcs. They may also have a lower bound value. The vertices in a flow network are usually called *nodes*. Two special nodes in a flow network are the *source* s and the *sink* t. The produce and consume flow respectively. If a problem requires multiple sources or sinks, it can easily be reduced to a single source and sink node: The two nodes s and t are added to the network, for every source v an arc (s, v) is embedded with a capacity equal to the supply value of v and for every sink u an arc (u, t) is embedded with a capacity equal to the demand value of u. Solving a problem in a flow network requires assigning a flow value $f(u, v)$ to every arc.
Every feasible flow in the network must fulfill the following properties:

1. The flow in any arc can never exceed its capacity ($f(u, v) \leq c(u, v)$)
2. The flow from one node u to another node v must be the opposite of the flow from v to u ($f(u, v) = -f(v, u)$)
3. The net flow in every node must be zero, except in the source and the sink ($\sum_{v \in V} f(u, v) = 0$, $u \neq s \wedge v \neq t$)

The most common problem in flow networks is the *maximum flow problem*, which consists of finding the largest possible flow from the source to the sink, i. e. maximizing $\sum_{v \in V} f(s, v)$. Another problem is the *minimum cost flow problem*. In this problem, every arc (u, v) is assigned a *cost* $a(u, v)$ and the cost of sending a flow from u to v is $f(u, v) \cdot a(u, v)$. Solving the problem requires sending a given flow from the source to the sink with a minimal flow cost.

3 Overview of the Topology-Shape-Metrics Approach

A commonly used algorithm for orthogonal layout is the *topology-shape-metrics* approach. It is basically a series of three phases, each with its own individual algorithms. These three phases fix the topology (by finding a planar embedding), the shape (by computing an orthogonal representation) and the metrics of the graph (by minimizing the edge length). This division in phases provides great modularity, so the algorithms can be chosen, developed or improved independently. The topology-shape-metrics approach is detailed by Roberto Tamassia et al. [26], but information on the approach can be found in many sources [2, 14].
Figure 3.1 shows the $K_{3,3}$ Kuratowski graph. This non-planar graph will serve as an example graph for algorithms throughout this thesis. The following sections will give an overview how algorithms in the phases should deal with this example graph.

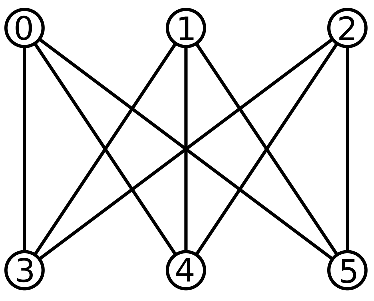
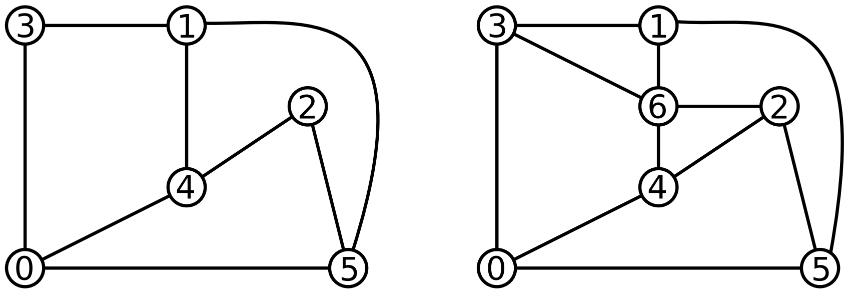


Figure 3.1: The $K3, 3$ Kuratowski graph will be the common example graph

3.1 Topology Phase

The first step is the *topology* phase. The topology of a graph is defined by the clockwise order of edges in the adjacency list of each vertex. Two graphs are equal in their topology if one can be transformed into the other without changing the order of edges adjacent to every vertex. The result of the first phase is a *planar representation* of the graph. Formally, a planar representation P is a function $P(v) = (e_1, \dots, e_{\delta(v)})$, $e_i \in E$, that assigns an ordered list of edges to every vertex v. It defines a fixed topology for the graph. The algorithm used in this project divides the topology phase into two steps:
First, a *planarity testing algorithm* operates on the graph. Although the primary use

of these algorithms is to determine whether a graph is planar or not, they are usually able to find a planar subgraph of the given graph (if it is not initially planar), and compute a planar representation of this subgraph. Figure 3.2a shows the computed planar subgraph of the example graph in Figure 3.1. It contains all the vertices of the example graph, but the edge $(2, 3)$ is missing, since it can't be inserted without losing planarity. An algorithm for planarity testing is explained in Chapter 4 of the thesis.
To acquire a full planar representation of the given input graph, some edges may have to be re-inserted into the planar subgraph resulting from the planarity test. These edges are added by a *planarization algorithm.* To maintain planarity, the inevitable edge crossings are replaced by dummy vertices. The algorithm minimizes the number of inserted dummy vertices and maintains a correct planar embedding of the graph. The example graph with the additional edges and dummy vertices is shown in Figure 3.2b. The edge $(3, 2)$ is now part of the graph, divided by the new dummy vertex 6. The order of the edges on each vertex after planarization defines a valid planar representation of the whole input graph. The algorithm used for planarization is further explained in Chapter 5.



(a) A planar subgraph of the example graph (b) A full planar representation of the graph

Figure 3.2: The topology phase provides a planar representation of the graph

3.2 Shape Phase

The second phase defines the *shape* of the graph. The shape of a graph extends the topology, but also affects edge bends and angles in the graph. Two graph drawings are of equal shape if they differ only in the length of their edges. The shape of the graph is defined by an *orthogonal representation.* An orthogonal representation H is formally defined as a function $H(f) = (r_1, \ldots, r_{\delta(f)})$, $r_i = (e_i, s_i, a_i)$ from the set of faces to a list of 3-tuples (e_r, s_r, a_r), where $e_r \in E$ is an edge adjacent to the face, s_r is a string of bends along this edge (consisting only of 90° or 270° angles) and a_r is the angle the edge forms with the following edge in clockwise order inside the face. Such an orthogonal representation is the result of an *orthogonalization algo-*

rithm. Figure 3.3 shows how the example graph would be drawn with an orthogonal representation computed by the orthogonalization algorithm. Such an algorithm for orthogonalization is detailed in Chapter 6.

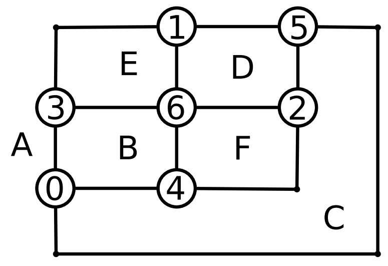


Figure 3.3: The shape phase provides an orthogonal representation of the graph

3.3 Metrics Phase

The final phase is known as the *metrics* phase. Two graphs with equal metrics have an identical drawing. In the metrics phase the last part missing for a final drawing is computed, which is the actual edge length. This is done by a *compaction algorithm*. Usually, such an algorithm minimizes either the total edge length, the total drawing area or the maximal edge length. Examples for compaction algorithms are discussed in Chapter 7 of the thesis. The metrics phase also maps the graph metrics back to the original input graph, thus removing the added dummy vertices and edges. Figure 3.4 shows how the example graph would be drawn after the whole topology-shape-metrics algorithm has performed a layout on the graph.

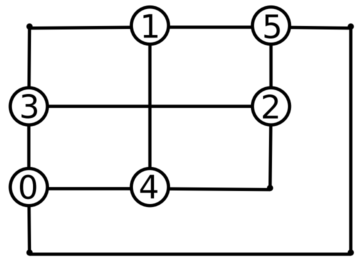


Figure 3.4: The final drawing of the processed graph

4 Planarity Testing

The first step on the way to a planar and orthogonal layout consists of a planarity test. A planarity test basically checks if a graph is planar, i. e. if it can be drawn without any edge crossings. Additionally, and relevant for the layouter, the planarity test can compute a planar subgraph, if the input graph is not initially planar, and also provide a planar representation of the graph or the planar subgraph. A planar representation gives a fixed order of edges on each vertex in clockwise direction and therefore defines a topology for the graph. While the computation of a *maximum* planar subgraph (i.e. the globally largest planar subgraph) is known to be NP-hard [27], the testing algorithm should be able to compute a *maximal* planar subgraph, meaning a subgraph in which any additional edge would break the planarity property.

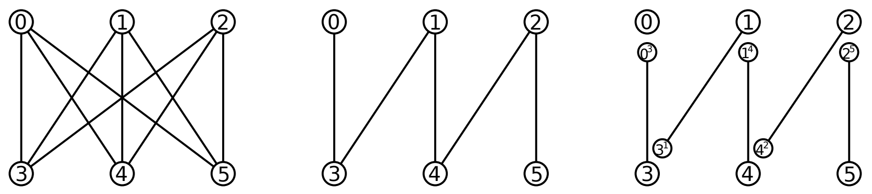
The first linear time algorithm for planarity testing is due to John Hopcroft and Robert E. Tarjan [12]. Over time, many alternative algorithms, extensions or improvements emerged [21, 20, 13, 1]. In the layout implementation accompanying this thesis, two possible testing algorithms were implemented. On the one hand an algorithm based on the *left-right planarity criterion* [13], which will not be discussed further in this thesis. On the other hand the algorithm developed by John M. Boyer and Wendy J. Myrvold in 2004 [1]. This second algorithm will be the main subject of this chapter. It provides a planarity test as well as the computation of a planar subgraph in time linear to the number of vertices in the graph. The basic idea of the algorithm is to copy the graph (i. e. the vertices), and then adding edges one by one until the graph is maximal planar. This is accomplished by first adding the edges of a spanning tree (which is obviously planar), and then adding the back edges one by one, starting at the tree leaves and building the graph upwards to the root of the tree.

4.1 Preprocessing and basic terminology

4.1.1 Depth First Search

The algorithm starts out performing a *Depth First Search (DFS)* on the graph. A DFS on a graph results in a spanning tree, the *DFS-Tree*, and an ordering of the vertices, assigning every vertex a *Depth First Search Index (DFI)*. The DFS also separates the edges in the graph in two groups: The *tree edges* and the *back edges*, Since the algorithm does not distinguish between directed and undirected edges, forward edges and cross edges are also interpreted as back edges.

Figure 4.1a shows the $K3, 3$ example input graph for the algorithm. Figure 4.1b



(a) The example input graph (b) A spanning tree of the graph (c) Insertion of virtual root vertices

Figure 4.1: The preprocessing of an example graph visualized

shows a possible DFS-Tree with all the tree edges. The back edges would be $(0, 4)$, $(0, 5)$, $(1, 5)$ and $(2, 3)$.

4.1.2 Least Ancestor and Low points

During the DFS, the algorithm computes two additional pieces of information for every vertex v: The *least ancestor* is the vertex of lowest DFI that can be reached from v by using only a back edge. The *low point* is the vertex of lowest DFI that can be reached from v by using zero or more tree edges and one back edge. Hence, the low point is also the least ancestor of all the vertices in the subtree below the vertex v. In the example graph, the least ancestor of 5 would be 0, of 2 would be 3, and so on. The low points of all vertices would be 0.

4.1.3 Virtual Root Vertices

The algorithm maintains a bicomp of the graph in form of a *virtual root vertex*. These virtual vertices act as an image of the cut vertex for the corresponding bicomp. Every bicomp contains exactly one virtual vertex at any time, and the virtual vertex has the lowest DFI of vertices in the bicomp. The virtual vertices are inserted into the graph during the preprocessing DFS for every child node in the DFS-Tree, forming a *singleton bicomp* (a bicomp that consist only of the virtual vertex and the child vertex). The virtual vertices are later removed one by one when a bicomp is merged into a large one by adding edges. Figure 4.1c shows the example graph with added virtual vertices.

4.1.4 Pertinence and Activity

Two other important properties of vertices are their *pertinence* and *external activity* state. These are both dependent on the vertex currently processed in the main loop of the algorithm.
A vertex is considered externally active if it has a back edge to a vertex of lower DFI than the processed vertex in the original graph. This means that the vertex will be important in any future step of the algorithm. Externally active vertices have to be kept on the external face of the graph at any time. As soon as an externally active

vertex vanishes from the external face, the back edge can not be embedded without losing the planarity of the graph.
A pertinent bicomp is a bicomp that is considered important for the embedding of a back edge in the current step of the algorithm. A bicomp is pertinent if it contains a pertinent vertex, and a vertex is pertinent either if it has a virtual root vertex that marks a pertinent bicomp, or if it is the endpoint of a back edge to be embedded in that step of the algorithm.

Algorithm 4.1: The main loop of the algorithm

```
procedure planarity($G$: graph)
    initialize $G'$ via depth first search on $G$

    for each vertex $v \in G$ in reversed order of their DFI do

        for each DFS child $c$ of $v$ in $G$
            embed the tree edge $(v^c, c)$ in $G$

        for each back edge $(v, w) \in G$ incident to $v$ do
            if $w$ has a higher DFI than $v$ then
                walkup($G'$, $v$, $w$)

        for each pertinent virtual root $v^c$ of $v \in G$ do
            walkup($G'$, $v^c$)

        for each back edge $(v, w)$ in $G$ incident to $v$ do
            if $w$ has a higher DFI than $v$ then
                if not $(v, w) \in G'$ then
                    $G$ is not planar

    perform postprocessing on $G'$
end
```

4.2 Algorithm Outline

As already mentioned, the algorithm starts performing a DFS on the input graph. During the DFS, every traversed vertex is copied to a new working graph, and the least ancestor and low point values are computed for each vertex. Additionally, the virtual root nodes are added to the working graph, each representing a (singleton) bicomp.
After the preprocessing DFS, the algorithm starts with the main loop. This loop traverses the vertices in the graph in descending order of their DFI, thus rebuilding the graph starting at the leaves of the DFS-Tree and working the way upwards to the root. For every traversed vertex, the loop performs four steps:
First, the tree edge from the parent to the vertex is embedded in the graph. Then the pertinent subgraph is determined by calling the *walkup* method on every vertex

to which a back edge should be embedded. This method marks certain bicomps in the graph as pertinent, so this information can be used in the next step. The third step constitutes the major part of the algorithm, namely embedding the back edges and merging the bicomps on the way. At last, in the fourth step, the algorithm checks if all back edges were properly embedded. If that is not the case, the graph is (obviously) not planar. After the main loop is done, a last postprocessing DFS is performed on the graph to merge all leftover bicomps. Algorithm 4.1 shows the planarity test algorithm in pseudo code. The walkup and walkdown methods are detailed in the next two sections.

Algorithm 4.2: The walkup method

```
procedure walkup($G$: graph, $v$: starting vertex, $w$ vertex)
    raise the back edge flag of $w$

    $(x, dir_x) \leftarrow (w, 1)$
    $(y, dir_y) \leftarrow (w, 0)$

    while $x \neq v$ do

        if $x$ or $y$ are marked as visited then break the loop
        mark $x$ and $y$ as visited

        if $x$ is a virtual root vertex then
            $z^c = x$
        else if $y$ is a virtual root vertex then
            $z^c = y$
        else
            $z^c = nil$

        if $z^c \neq nil$ then
            set $z$ equal to the parent of the virtual root vertex $z^c$
            mark $z^c$ as pertinent
            $(x, dir_x) \leftarrow (z, 1)$
            $(y, dir_y) \leftarrow (z, 0)$

        else
            $(x, dir_x) \leftarrow GetSuccessorOnExternalFace(x, dir_x)$
            $(y, dir_y) \leftarrow GetSuccessorOnExternalFace(y, dir_y)$
end
```

4.3 The Walkup Method

The walkup is a subroutine of the algorithm responsible to mark the pertinent subgraph for the currently processed vertex v. The routine is invoked for every endpoint of a back edge w that has a higher DFI than v. It is therefore invoked for every vertex below v in the DFS-Tree that should be connected to v by a back edge. The

purpose of this method is to determine which vertices and bicomps are pertinent during the embedding of that back edge.
The walkup starts at the vertex w, the endpoint of a back edge, and raises a *back edge flag* for w, thus marking it pertinent. It then traverses the external face of the graph until it encounters a virtual root vertex. The bicomp represented by that virtual root vertex is marked as pertinent and the method jumps to the parent bicomp of that virtual vertex and continues traversal of the external face there. This way, the method works its way upwards (hence the name *walkup*), until the vertex v is reached.
Figure 4.2 shows an example graph section during the main algorithm. In this case, the currently processed vertex is $v = 1$ and the back edge $(1, 5)$ should be embedded (a). The walkup method is called on the vertex 5, where the back edge flag of 5 is raised (b). It then traverses the external face and encounters vertex 3^4, marks it as pertinent and jumps to vertex 3 (c). Similarly, it will encounter vertex 1^2, mark it as pertinent and jump to 1 (d). There it will stop since it has reached the starting vertex.

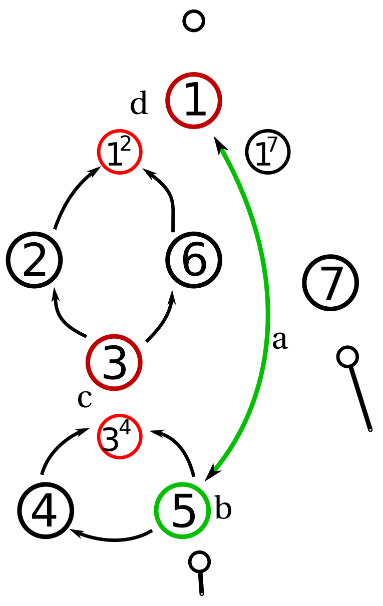


Figure 4.2: The walkup for the back edge $(1, 5)$ (a) during processing of vertex 1: The back edge flag of 5 is raised (b) and the vertices 3 and 3^4 (c) as well as 1 and 1^2 (d) are marked as pertinent.

Algorithm 4.3: The walkdown method

```
procedure walkdown($G$: graph, $v'$: virtual root vertex)
    clear the merge stack $S$

    for $dir_{v'} \in 1, 0$ do
        $(w, dir_w) \leftarrow successoronexternalfaceof(v', dir_{v'})$
        while $w \neq v'$ do

            if $w$ has a raised back edge flag then
                merge vertices on merge stack $S$
                embed the back edge $(w, v')$

            if $w$ has a pertinent virtual root vertex $w'$ then
                choose traversal direction $dir_{w'} \in 1, 0$
                push $(w, dir_w)$ on the merge stack
                push $(w', dir_{w'})$ on the merge stack
                $(w, dir_w) \leftarrow (w', dir_{w'})$

            else if $w$ is externally active then
                break the 'while' loop

            else
                $(w, dir_w) \leftarrow GetSuccessorOnExternalFace(w, dir_w)$

        if not $S$ is empty then
            break the 'for' loop
end
```

4.4 The Walkdown Method

After the walkup has determined the pertinent subgraph, the *walkdown* method can perform the major work of the algorithm. The purpose of the walkdown is to finally embed the back edges in the graph. While the walkup traverses the graph from the endpoint of the back edge upwards, the walkdown starts at the currently processed vertex and makes its way downwards to the endpoint of the back edge. It is invoked for every virtual root vertex of the currently processed vertex v and starts traversing the external face.
If it encounters a vertex that has any virtual root vertices marked as pertinent, it puts the encountered vertex together with the virtual root vertex on a stack, jumps down to the pertinent bicomp and continues traversal there. The direction of the traversal (i.e. clockwise or counterclockwise) does not matter and is up to the implementing programmer.
If it encounters a vertex whose back edge flag is raised, all the bicomps on the stack have to be merged. That means the adjacency list of the virtual root vertex is added to the adjacency list of the parent vertex and the virtual root vertex is removed

from the graph. Any bicomps containing externally active vertices may have to be flipped, so that these vertices remain on the external face. After the bicomps have been merged, the back edge can be embedded into the graph and the method can continue traversing the external face.
Traversal halts as soon as it reaches the virtual root vertex it started on, or if it encounters an externally active vertex that is not pertinent, a so-called *stopping vertex*. Since the embedding of a back edge after such a vertex will break planarity, the algorithm cannot traverse the face any further. But it may still traverse the face in the other direction. If it encountered a stopping vertex in both directions, at least one of the back edges cannot be embedded. The algorithm will then detect these missing edges in the main loop.
Figure 4.3 continues the example from Section 4.3. The currently processed vertex is 1 and the walkdown is invoked for vertex 1^2 (a). In this example the vertex 6 shall be an externally active vertex. The method traverses the face in a random direction. If it travels clockwise, it will encounter the externally active vertex 6 and stop there (b). If it travels counterclockwise, it will encounter the vertex 3, which has a pertinent virtual root 3^4 (c). Both vertices 3 and 4^4 will be put on the merge stack and traversal continues in 3^4. Independent of the now chosen direction, the method will encounter vertex 5, which has a raised back edge flag. Now the method merges all vertices on the merge stack and embed the edge $(1^2, 5)$.

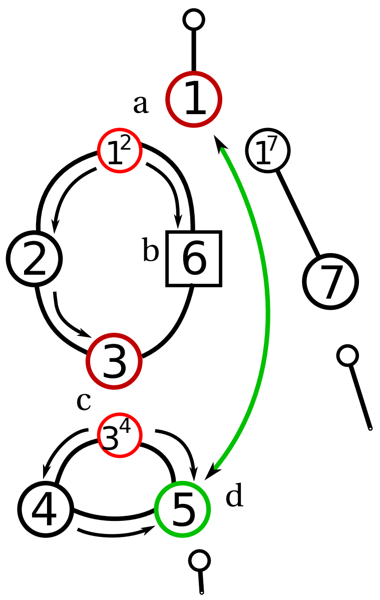
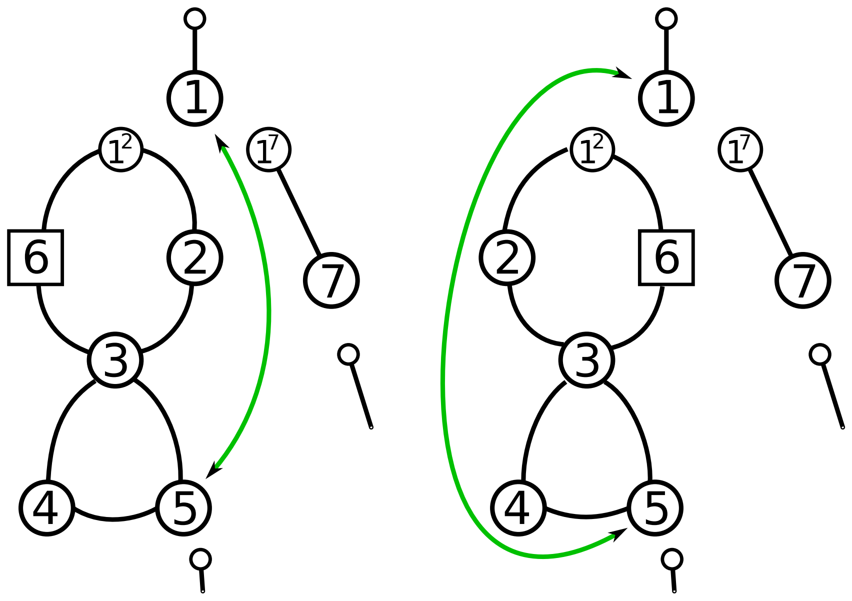


Figure 4.3: The walkdown for root vertex 1^2 (a) during processing of vertex 1: The external face is traversed, while pushing vertices 3 and 3^4 on a merge stack (c), until vertex 5 is reached. Then the vertices on the stack are merged and the edge $(1^2, 5)$ embedded (d).



(a) Embedding an edge after counter-clockwise traversal

(b) Embedding an edge after clockwise traversal

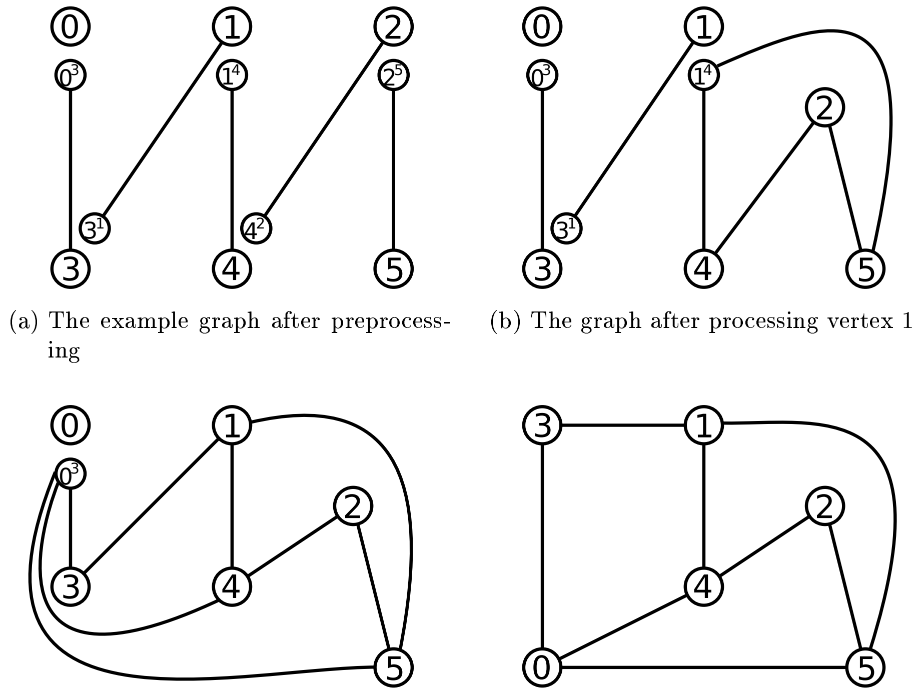
Figure 4.4: The direction in which a bicomp is traversed determines if it has to be flipped before merging.

4.5 Flipping and Merging Biconnected Components

When merging two bicomps, on some occasions the lower bicomp may have to be flipped. That choice is dependent on the direction, in which the algorithm traverses the external face. For example, if the algorithm traverses the face in counterclockwise direction, it will embed the edge as seen in Figure 4.4a, and if it traverses the face in clockwise direction, it will embed the edge on the other side (Figure 4.4b). Since the traversal of a bicomp will not go beyond a stopping vertex, a bicomp will have to be flipped if and only if it is traversed in the same direction as the parent bicomp. Merging the adjacency lists of the two merged vertices is also dependent on the traversal direction. If the face is traversed in clockwise direction, the adjacency list of the virtual root vertex is appended to the adjacency list of its parent vertex, otherwise it is prepended to the list.

4.6 Processing of an example graph

This section should clarify the algorithm by explaining the single steps it performs on the example graph used throughout this thesis, the $K_{3,3}$ graph. The preprocessing step has already been performed on this graph in Section 4.1, so this section starts with the DFS-Tree, including the virtual vertices, as shown in Figure 4.5a. As a



(a) The example graph after preprocessing

(b) The graph after processing vertex 1

(c) The graph after processing vertex 0

(d) The graph after postprocessing

Figure 4.5: Performing the algorithm on the $K3,3$ graph

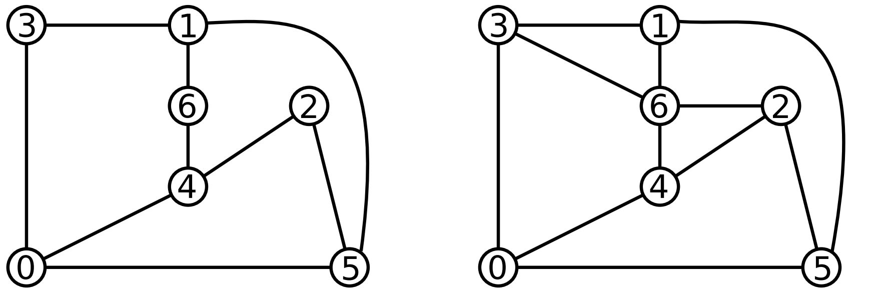
reminder, the missing back edges are $(0,4)$, $(0,5)$, $(1,5)$ and $(2,3)$.

The main loop processes the vertices in reversed order of their DFI, in this case they are processed in the order $5,2,4,1,3,0$. The first significant events occur during processing of vertex 1, when the back edge $(1,5)$ should be embedded. At this point, the walkup method is called on vertex 5. It raises the back edge flag on 5, traverses the external face and encounters virtual vertex 2^5. The vertex is marked as pertinent and traversal resumes at vertex 2. The same happens for vertices 4^2 and 4, as well as 1^4 and 1. Then the currently processed vertex v is reached and the walkup stops. Now the walkdown is called on all pertinent virtual root vertices of 1, namely vertex 1^4. It starts traversal at 1^4 and immediately reaches vertex 4. Since 4^2 is pertinent, both 4 and 4^2 are pushed on the merge stack. The same happens for vertices 2 and 2^5. After that, vertex 5 is encountered, which has a raised back edge flag. Now the vertices 4 and 4^2 as well as 2 and 2^5 are merged, and the edge $(1^4,5)$ is embedded. The result is shown in Figure 4.5b. The method will continue traversal from vertex 5, but will soon encounter the starting vertex 1^4 (since the bicomps are now merged) and stop.
The next vertex in the main loop is vertex 3, which has a back edge to vertex 2. The walkup is called on 2, the back edge flag on this vertex is raised and the vertices 1^4 and 3^1 are marked as pertinent.

The walkup method is now invoked on the virtual root vertex 3^1. It traverses the external face starting at 3^1, finds vertex 1, pushes 1 and 1^4 on the merge stack and continues at vertex 1^4. The clockwise traversal will encounter vertex 5 and the counterclockwise traversal will encounter vertex 4, both of which are externally active (since the input graph contains the edges $(0, 5)$ and $(0, 4)$). The walkdown stops here, and the algorithm detects that the edge $(3^1, 2)$ has not been embedded. The graph is therefore marked as not planar.
At last, the vertex 0 is processed. The walkup is invoked twice this time, once on vertex 4 and once on vertex 5, both endpoints of back edges. The back edge flag for these two vertices are raised and each time the vertices 1^4, 3^1 and 0^3 are marked as pertinent.
The walkdown will again encounter either vertex 4 or vertex 5, after the vertices 3, 3^1, 1 and 1^4 have been pushed on the merge stack. Assuming the algorithm chooses a counterclockwise traversal, the vertex 4 is encountered first. The vertices 3 and 3^1, as well as 1 and 1^4 are merged and the edge $(0^3, 4)$ is embedded. The traversal is resumed and the vertex 5 is encountered. Since the merge stack is empty at this point, the edge $(0^3, 5)$ can be embedded immediately and further traversal will stop at the starting vertex 0^3. The resulting graph is shown in Figure 4.5c.
After the the main loop has terminated, a last postprocessing DFS is performed on the graph. In this example case, the DFS will notice the left-over virtual vertex 0^3, and will merge it with vertex 0. The overall result of the planar testing algorithm on the example graph is shown in Figure 4.5d.

5 Planarization

Since the planarity testing algorithm provides only a planar representation for a subgraph of the original input graph, some edges are missing and have to be re-inserted in the graph. This is done in the second part of the first phase of the topology-shape-metrics algorithm, the planarization. The aim of the planarization is to add all edges that were removed during the computation of the planar subgraph, while keeping the graph planar and the planar representation intact.
Since the graph resulting from the previous step is maximal planar, i. e. any of the missing edges inserted into the graph will break planarity, every edge crossing is replaced with an additional dummy vertex. One possible algorithm that adds edges to a planar graph has been introduced by Gutwenger et al. in 2001 [10]. The simpler algorithm implemented in the graph layouter, and the one discussed in this chapter, is detailed in the bachelor thesis of Christian Kutschmar [19]. His thesis is based upon the same layouter implementation as this thesis and is therefore directly related.



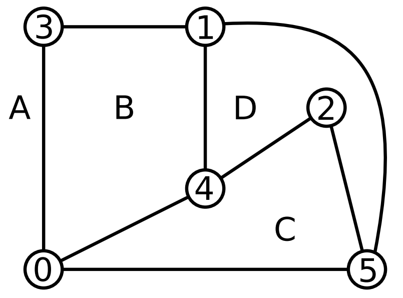
(a) The graph with a dummy vertex (b) The graph with the embedded edges

Figure 5.1: Embedding of an edge requires the addition of dummy vertices on every crossing edge. In this example the vertex 6 is added on the edge $(1, 4)$. All these dummy vertices are then connected by a path of edges $((2, 6), (6, 3))$ representing the original edge $(2, 3)$.

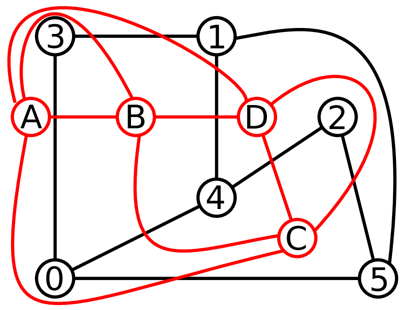
5.1 Inserting an edge

The algorithm will not only embed an additional edge into a planar graph, but will also attempt to do this while creating the minimum number of dummy vertices. It does this by finding an embedding path for the new edge that crosses a minimal number of faces. This path corresponds to the shortest edge path in the dual graph.

Therefore the embedding of edge (v, w) in graph G is performed using the following algorithm: First, the dual graph $\tilde{G}$ of the input graph G is computed based on its faces, as seen in Figure 5.2. For every face f adjacent to v and every face g adjacent to w, the shortest path in the dual graph $\tilde{G}$ between the vertices representing f and g is found. Dijkstra's algorithm [4] can be used to accomplish this. The edge is then added into the graph following this path, i. e. for every edge in the dual path a dummy vertex is added on the corresponding edge in G, and these dummy vertices are connected by edges crossing the relevant faces.
In the example graph, the edge $(2, 3)$ should be embedded. The source vertices of the path would therefore be $v \in \{C, D\}$ and the target vertices would be $w \in \{A, B\}$. All of these vertices are adjacent to each other, so the shortest path can be chosen freely. Choosing the edge (D, B), the dummy vertex 6 is added splitting the edge $(1, 4)$ in the original graph (see Figure 5.1a). Now, embedding the edges $(2, 6)$ and $(6, 3)$ will result in a correct planar graph containing all edges, as seen in Figure 5.1b.



(a) A planar graph with its faces

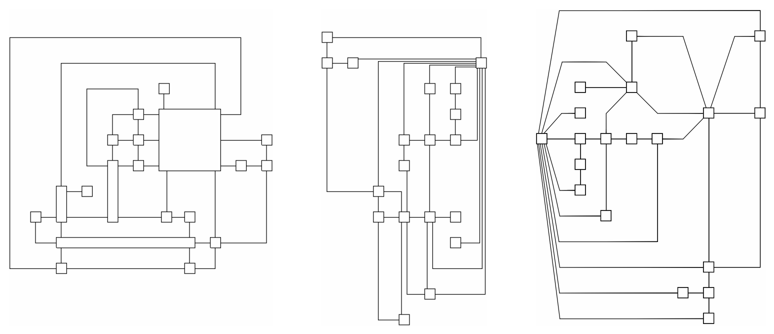


(b) The dual graph

Figure 5.2: The planarization algorithm requires the construction of the dual graph for every inserted edge. The shortest path in the dual graph corresponds to the embedding path with the least edge crossings.

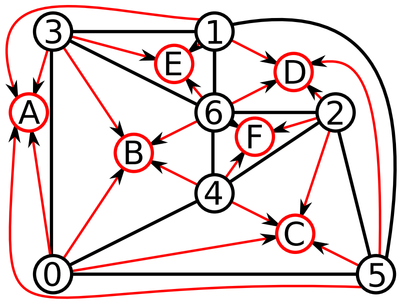
6 Orthogonalization

The first phase in the layouter has defined the *topology* of the input graph, meaning that the graph has a fixed order of edges on every vertex. Now the second phase will define the *shape* of the graph. In other words, it computes an *orthogonal representation*, which defines the number and order of bend points on every edge and the angles (in multiples of 90°) the edges form at these bend points and at the vertices. The basic algorithm to find an orthogonal representation with a minimum number of bend points has been developed in 1987 by Roberto Tamassia [24]. Unfortunately, this algorithm is limited to graphs with a maximal vertex degree of 4, since it places the vertices of the graph on a 2-dimensional grid. Since this is usually not very useful in practice, several alternative approaches have been introduced to extend the basic algorithm and allow the orthogonalization of graphs containing vertices of higher degree. The *Giotto* approach [26], for instance, allows vertices to occupy multiple points in the grid. This requires them to be larger, making this approach simple to implement but problematic if used with graphs that contain size constraints on vertices. Another approach is the *Kandinsky* orthogonalization [7], which allows edges to leave the grid while remaining vertical or horizontal. This way, the edges can be put closer to each other on the surface of a vertex. Another approach is the *Quod* orthogonalization [15]. It stands for quasi-orthogonal, as it allows the edges to leave the orthogonal grid and become diagonal. Figure 6.1, taken from the paper by Gunnar Klau and Petra Mutzel [15], shows a comparison of the different approaches on high-degree vertices.

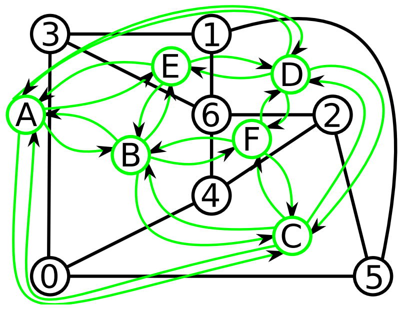


(a) Giotto (b) Kandinsky (c) Quod

Figure 6.1: While the *Giotto* approach resizes high-degree vertices, the *Kandinsky* approach packs edge segments together and the *Quod* approach allows edge segments to leave the grid.



(a) The network arcs in A_V



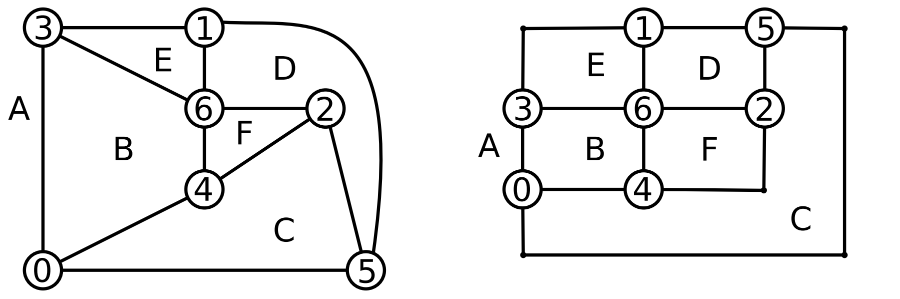
(b) The network arcs in A_F

Figure 6.2: The bend-minimizing algorithm reduces the problem to a minimum cost flow problem in a flow network. The network contains vertices for every face and vertex in the graph, and arcs for vertices (red) and faces (green) adjacent to each face.

6.1 Bend minimization for 4-planar graphs

This section will first detail on the original bend-minimizing algorithm by Tamassia [24], which is limited to 4-planar graphs, i. e. planar graphs with a maximal vertex degree of 4. It reduces the bend minimization problem to a flow network. This flow network is constructed according to the following rules:
The set of nodes in the network consists of two node sets $U = U_F \cup U_V$. The set U_V contains a source with a flow supply value of 4 for every vertex in the input graph. The set U_F on the other hand contains a sink node for every face in the input graph, with a flow demand value of $4 - 2 * \delta$, where δ is the number of vertices on the face. In the special case of the external face, the demand value is $4 - 2 * \delta - 8$ instead.
The set of arcs in the flow network also consists of two arc sets $A = A_F \cup A_V$. The set A_V contains an arc from every node in $v \in U_V$ to a node in $w \in U_F$, where the vertex corresponding to v is adjacent to the face corresponding to w. The arcs in this set have a lower bound of 1, a capacity of 4 and a cost of 0 and their construction is visualized in Figure 6.2a. The second set A_F contains an arc from every node $v \in U_F$ to another node $w \in U_F$, where the faces corresponding to v and w are adjacent. These faces can be equal, if they contain a bridge. The arcs in this set have a lower bound of 0, unlimited capacity and a cost of 1 and Figure 6.2b shows the network with only this second arc set.
After the network is constructed according to these rules, every feasible flow in it encodes a valid orthogonal representation of the input graph. If the minimum cost flow problem is solved for the network, the resulting flow encodes the orthogonal representation with a minimum number of bend points. A possible algorithm to solve the minimum cost flow problem is described by Tamassia in [8].
Since every arc in A_V leads from a node representing a vertex v to a node representing a face f in the input graph, the flow in these arcs represent the sum of angles the

face f forms at vertex v in multiples of 90°. A face can form multiple angles in a vertex if it contains a bridge. Every arc in A_F connects two nodes representing faces f and g in the input graph, and every unit of flow in these arcs represents a bend of 90° in f, or a bend of 270° in g. These bends are part of the edge dividing f and g. Computing this information allows the construction of an orthogonal representation of the input graph. A drawing based on the orthogonal representation that would result from this algorithm can be seen in Figure 6.3b.



(a) The example input graph with its faces (b) A possible orthogonal drawing of the graph

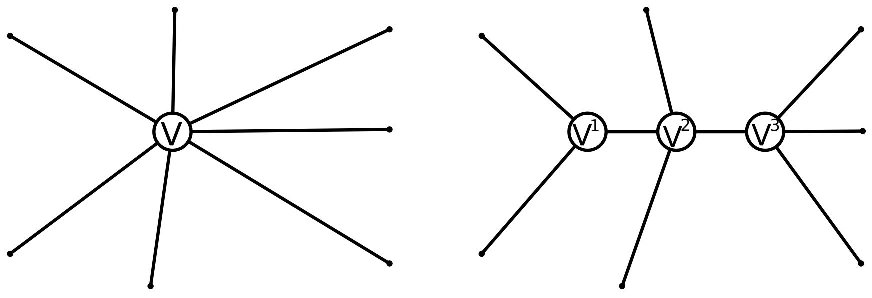
Figure 6.3: The orthogonalization algorithm computes the number of bend points along each edge and the angles formed by edges in vertices and bend points.

6.2 Giotto Orthogonalization

The first and simplest approach to handle vertices with a degree higher than 4 by the orthogonalization algorithm is known as the *Giotto* approach. It was presented together with the bend minimizing algorithm from Section 6.1 [24]. It simply replaces every vertex of high degree in the graph with multiple vertices, each connected to some of the edges of the original vertex. An example transformation is shown in Figure 6.4. The resulting graph can be orthogonalized with the original bend-minimizing algorithm, with the additional restriction that all vertices replacing the same original vertex have to form a rectangle in the orthogonal representation. After replacing these vertices with the original, the vertex will occupy multiple point on the grid, therefore resizing it. Hence, the approach is usually avoided if the vertices of the input graph have a restricted or fixed size (e. g. in UML class diagrams).

6.3 Quod Orthogonalization

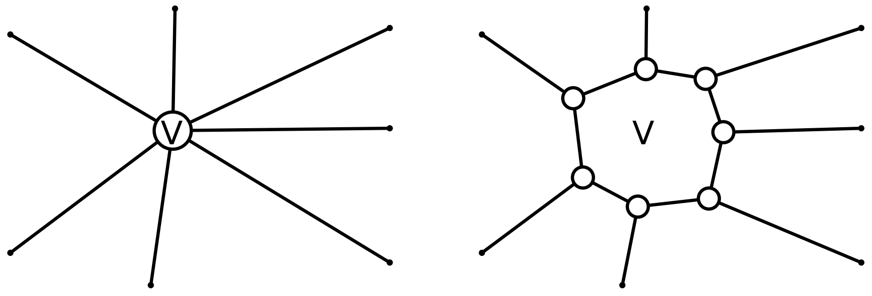
The *Quod* orthogonalization approach provides an algorithm to handle vertices with a degree higher than 4 in a quasi-orthogonal layout. The approach has been introduced by Gunnar W. Klau and Petra Mutzel in 1998 [15]. Its basic idea is to replace every



(a) A high-degree vertex (b) The split vertex

Figure 6.4: The *Giotto* approach splits Vertices of high degree.

vertex in the graph with a higher degree with a *cage*. A cage consists of a ring of vertices forming a face, which represents the original node. The ring contains a vertex for every edge adjacent to the original vertex. Every vertex is connected to a vertex in the graph with this edge and to the two neighboring vertices in the ring. Since the vertex with a high degree is removed from the graph and every additional vertex has a degree of exactly 3, the resulting graph has a vertex degree of at most 4. Figure 6.5 shows how a cage would replace a high degree vertex. After the graph is reduced to have a vertex degree of at most 4, the bend minimization algorithm described in Section 6.1 can be used to build an orthogonal representation. The only additional restriction on the algorithm is that every cage has to form a rectangle on the grid. After the orthogonalization algorithm is done, the cages have to be replaced by the original vertices. Unlike the Giotto approach, the vertices do not have to be resized. Instead, the edges are allowed to leave the grid in the segment between their endpoint at the cage vertex and the endpoint at the actual vertex. This results in diagonal edge segments close to high-degree vertices.

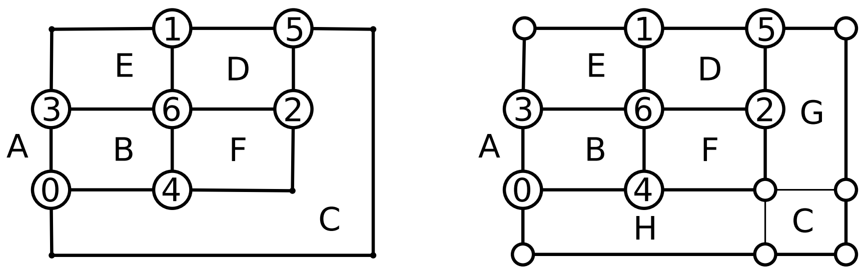


(a) A high-degree vertex (b) A cage replacing the vertex

Figure 6.5: The *Quod* approach replaces vertices of high degree with cages.

7 Compaction

The last phase in the topology-shape-metrics approach is the *metrics* phase. This last step will finally result in a proper drawing of the input graph, by calculating actual coordinates to vertices and bend points in the graph. Since the basic graph structure, including all angles and bend points, have already been computed in previous phases, all that is left to do now is computing the length of the edge segments, or the distance between the vertices and bend points. The algorithms used in this *compaction* step should also minimize the graph size in some way. While some algorithms focus on minimizing the required drawing area, other algorithms minimize the average or the maximum edge (segment) length. There exist many different approaches on compaction algorithms [24] [16] [23], but the one detailed in this chapter will be the approach by Roberto Tamassia that was introduced together with his approach on orthogonalization in 1987 [24].



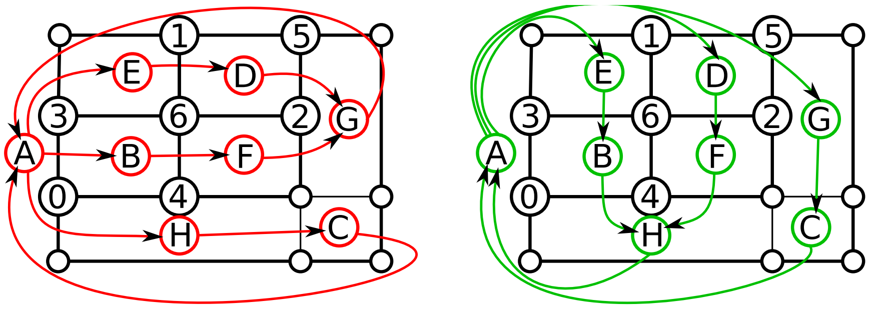
(a) Drawing of a graph with a regular orthogonal representation (b) A derived simple orthogonal representation of the graph

Figure 7.1: Every orthogonal representation can be reduced to a simple orthogonal representation by adding virtual vertices and edges.

7.1 Simple Orthogonal Representations

Many approaches on compaction require in their first step to reduce the orthogonal representation H of the graph to a *simple orthogonal representation* H'. An orthogonal representation H' is called *simple* or *refined* if all the faces in the graph are defined as rectangles by H'. To acquire a simple orthogonal representation H' from H, additional virtual vertices replace every bend point in H. Then the rectangular parts in the faces have to be separated by virtual edges. To do this for a face f,

the circular bend string representing the face is built from H. As a reminder, the orthogonal representation assigns a string of angles, as well as the angle formed with the next edge, to every edge surrounding the face f in the graph. To get the bend string for the total face, the strings for the edges are appended in clockwise order. In between these strings, an additional angle is added for the vertex between the edges. This additional bend is a right bend for 90° angles, a left bend for 270° angles or two left bends for 360° angles. 180° angles are ignored, as they don't contribute to the shape of the face.
After the bend string representing the face is built, the face is reduced to rectangles by replacing every occurrence of *left-right-right* with a *right*. The resulting bend string contains either only four right bends, or no consecutive right bends at all if it encodes the external face. A simple orthogonal representation derived from the orthogonal representation of an example graph is shown in Figure 7.1. It visualizes where virtual vertices and edges have been added to reduce all faces to a rectangular shape.



(a) The flow network for horizontal minimization (b) The flow network for vertical minimization

Figure 7.2: To minimize the edge length in a graph, two separate flow networks are constructed. One network minimizes the length of vertical edge segments (red) and the other minimizes the length of horizontal edge segments (green).

7.2 Giotto Compaction

A simple and easy to implement compaction algorithm is introduced by Roberto Tamassia, along with the orthogonalization algorithm presented in Section 6.1 [24]. It reduces the edge minimization problem to a one-dimensional problem, i. e. the length of the horizontal and the length of the vertical edges are minimized independently. This is in general not an optimal approach, since if edge length in one dimension are minimized first, they can block further minimization in the other dimension and vice versa. For example in Figure 7.3, no further minimization of either the horizontal or vertical edge segments is possible, but the edge lengths are not minimized.

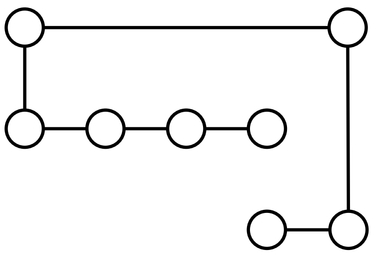


Figure 7.3: The compaction algorithm cannot minimize the vertical or horizontal edge segments any further, but the edge lengths are not yet optimal.

The algorithm first requires reduction of the the orthogonal representation to a simple orthogonal representation as detailed in Section 7.1. It then creates two flow networks to solve the problem: One for the horizontal edge segments and one for the vertical edge segments. These flow networks contain a node for each face in the orthogonal representation. The first network contains an arc for each horizontal edge segment dividing two faces in the graph, the second network contains an arc for each vertical edge segment. All arcs have a lower bound and cost of 1. Figure 7.2 shows the two networks for the example graph. The network in Figure 7.2a minimizes the vertical edge segment lengths, and the network in Figure 7.2b minimizes the horizontal ones. Solving the minimum cost flow problem in these networks will result in a graph drawing with minimal edge length. A final orthogonal drawing of the example graph is shown in Figure 7.4.

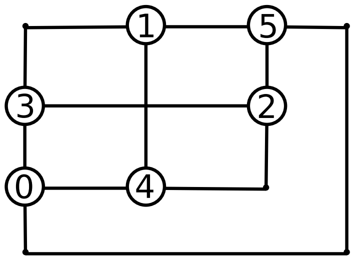


Figure 7.4: A final orthogonal drawing of the example graph

8 Conclusion and Future Work

The layout algorithm based on the topology-shape-metrics approach presented in this thesis provides a fully orthogonal layout with minimized number of edge crossing and bend points. It also provides a modular frame for implementing algorithms of its three phases. The algorithms implemented in the context of the bachelor project build up to a working, yet very simple graph layouter. There is still a lot of work left for a fast and useful graph layout. While the algorithms used for planarity testing are state-of-the-art, there is still place for improvements in all the other phases.
The planarization phase, for instance, could be enhanced to find better planar representations through the use of *SPQR-Trees* [9]. The orthogonalization phase is missing algorithms that handle high-degree vertices correctly. Here an assortment of approaches (e. g. the *Giotto* [26], *Quod* [15] and *Kandinski* [7] approach) would provide the user some choices over the resulting layout. There are also more advanced algorithms for compaction, some of which take predefined vertex sizes into account and some of which doesn't [24, 16, 23, 5, 17].
It would also be a nice addition to add an option for the layout of k-gonal graphs (as opposed to orthogonal graphs). While an orthogonal graph drawing is embedded in a grid with at most 4 edges per grid point and angles in multiples of 90°, a k-gonal graph drawing has k edges on a grid point, and all the angles are multiples of $360°/k$. A 3-gonal graph, for instance, would result in triangular structures, and 6-gonal graph drawings would contain comb-shaped elements. Approaches for such k-gonal graph drawings are already present in the basic literature [24, 14].
Another possible enhancement is to allow compound vertices (i. e. vertices that contain other graphs), and especially edges that connect vertices in different graphs (e. g. a vertex in a graph connected to a vertex in a compound vertex). Also, most algorithms in the project need to be modified or even replaced if embedding constraints should be allowed [11, 6]. Such embedding constraints could include for example a predefined order of edges on specific vertices.

Bibliography

[1] John Boyer and Wendy Myrvold. On the cutting edge: Simplified $\mathcal{O}(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.

[2] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, June 1994.

[3] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.

[4] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[5] Markus Eiglsperger and Michael Kaufmann. Fast compaction for orthogonal drawings with vertices of prescribed size. In *GD '01: Revised Papers from the 9th International Symposium on Graph Drawing*, volume 2265 of *LNCS*, pages 124–138. Springer-Verlag, 2002. ISBN 3-540-43309-0.

[6] Markus Eiglsperger, Ulrich Fößmeier, and Michael Kaufmann. Orthogonal graph drawing with constraints. In *SODA '00: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 3–11. SIAM, 2000. ISBN 0-89871-453-2.

[7] Ulrich Fößmeier and Michael Kaufmann. Drawing high degree graphs with low bend numbers. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 254–266, London, UK, 1996. Springer-Verlag. ISBN 3-540-60723-4.

[8] Ashim Garg and Roberto Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In *GD '96: Proceedings of the Symposium on Graph Drawing*, pages 201–216, London, UK, 1997. Springer-Verlag. ISBN 3-540-62495-3.

[9] Carsten Gutwenger and Petra Mutzel. A linear time implementation of SPQR-trees. In *GD '00: Proceedings of the 8th International Symposium on Graph Drawing*, volume 1984 of *LNCS*, pages 77–90. Springer-Verlag, 2001. ISBN 3-540-41554-8.

[10] Carsten Gutwenger, Petra Mutzel, and René Weiskircher. Inserting an edge into a planar graph. In *SODA '01: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 246–255. SIAM, 2001. ISBN 0-89871-490-7.

[11] Carsten Gutwenger, Karsten Klein, and Petra Mutzel. Planarity testing and optimal edge insertion with embedding constraints. In Michael Kaufmann and Dorothea Wagner, editors, *GD 2006: Proceedings of the 14th International Symposium on Graph Drawing*, volume 4372 of *LNCS*, pages 126–137. Springer-Verlag, 2007.

[12] John Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974. ISSN 0004-5411.

[13] Daniel Kaiser. Das Links-Rechts-Planaritätskriterium. http://www.inf.uni-konstanz.de/algo/lehre/ws08/projekt/ausarbeitungen/kaiser.pdf.

[14] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs: Methods and Models*. Number 2025 in LNCS. Springer-Verlag, Berlin, Germany, 2001. ISBN 3-540-42062-2.

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

[16] Gunnar W. Klau and Petra Mutzel. Optimal compaction of orthogonal grid drawings. In *Proceedings of the 7th International IPCO Conference on Integer Programming and Combinatorial Optimization*, volume 1610 of *LNCS*, pages 304–319. Springer-Verlag, 1999. ISBN 3-540-66019-4.

[17] Gunnar W. Klau, Karsten Klein, and Petra Mutzel. An experimental comparison of orthogonal compaction algorithms. In *GD '00: Proceedings of the 8th International Symposium on Graph Drawing*, pages 37–51, London, UK, 2001. Springer-Verlag. ISBN 3-540-41554-8.

[18] Casimir Kuratowski. Sur le probleme des courbes gauches en topologie. *Fund. Math.*, 15:271–283, 1930.

[19] Christian Kutschmar. Planarisierung von hypergraphen, 2010. to appear.

[20] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Theory of Graphs: International Symposium*, pages 215–232, New York, NY, USA, 1967. Gordon and Breach.

[21] Kurt Mehlhorn and Petra Mutzel. On the embedding phase of Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.

[22] Petra Mutzel. The SPQR-tree data structure in graph drawing. In *Automata, Languages and Programming, 30th International Colloquium*, volume 2719 of *LNCS*, pages 34–46. Springer-Verlag, 2003.

[23] Maurizio Patrignani. On the complexity of orthogonal compaction. In *WADS '99: Proceedings of the 6th International Workshop on Algorithms and Data Structures*, volume 1663 of *LNCS*, pages 56–61. Springer-Verlag, 1999. ISBN 3-540-66279-0.

[24] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal of Computing*, 16(3):421–444, 1987. ISSN 0097-5397.

[25] Roberto Tamassia. Constraints in graph drawing algorithms. *Constraints*, 3(1):87–120, 1998. ISSN 1383-7133. doi: http://dx.doi.org/10.1023/A:1009760732249.

[26] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):61–79, 1988. ISSN 0018-9472.

[27] Mihalis Yannakakis. Node-and edge-deletion np-complete problems. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 253–264, New York, NY, USA, 1978. ACM.