UNIVERSITÄT PASSAU

Fakultät für Mathematik und Informatik

Dissertation

Crossings in Clustered Level Graphs

Michael Forster

Supervisor
Prof. Dr. Franz J. Brandenburg

30th November 2004

Dissertation for the aquisition of the degree of a doctor in natural sciences at the Faculty of Mathematics and Computer Science of the University of Passau.

1st referee: Prof. Dr. Franz J. Brandenburg, University of Passau
2nd referee: Prof. Dr. Ulrik Brandes, University of Konstanz

Abstract

Clustered graphs are an enhanced graph model with a recursive clustering of the vertices according to a given nesting relation. This prime technique for expressing coherence of certain parts of the graph is used in many applications, such as biochemical pathways and UML class diagrams. For directed clustered graphs usually level drawings are used, leading to clustered level graphs. In this thesis we analyze the interrelation of clusters and levels and their influence on edge crossings and cluster/edge crossings.

We present a new method for the application of two-level crossing reduction algorithms to clustered level graphs. Our approach is optimal in the sense that it does not introduce unnecessary crossings, and therefore produces fewer crossings as previous results. In contrast to other approaches, our extension scheme retains the optimality of a one-sided two-level crossing reduction algorithm when extended to clustered level graphs.

We also give a new algorithm for constrained one-sided two-level crossing reduction in level graphs, which appears as a subproblem in clustered crossing reduction. Here, the relative position of some vertex pairs on the second level is fixed. Based on the barycenter heuristic, we present a new algorithm that runs in quadratic time and generates fewer crossings than existing simple extensions. It is significantly faster than previous advanced algorithms, while it compares well in terms of crossing number and is easy to implement.

Minimizing crossings also leads to the concept of planarity. Planar drawings are easy to understand and thus preferable to non-planar drawings. Variations of planarity have been studied intensively for level graphs and clustered graphs. We combine these concepts and analyze a new problem: clustered level planarity. We give an efficient algorithm that decides clustered level planarity of elementary clustered level graphs and computes a clustered level embedding, if one exists.

Contents

Every time we remember to say "thank you", we experience nothing less than heaven on earth.

Sarah Ban Breathnach

So long and thanks for all the fish.

Douglas Adams

1 Acknowledgments

Writing this thesis would not have been possible without the great support of various people. First of all, I would like to express my thankfulness to my supervisor Prof. Dr. Franz J. Brandenburg. He supported me very much throughout all of my computer science studies, by letting me be a part of his research group – starting in my first term nearly nine years ago as a student research assistant, up to now as a research assistant and PhD student. It was also him, who introduced me to the topic of graph drawing, and who made it possible for me to attend the graph drawing conference multiple times.

I am also very grateful to my second referee, Prof. Dr. Ulrik Brandes, and to my colleagues Dr. Christian Bachmaier, Andreas Pick, Marcus Raitner, and Dr. Falk Schreiber. They always had an open ear for discussions about various scientific problems and thus really helped me to advance my work. Especially Christian contributed a major share to finding many of the results in this thesis and in various co-authored papers. More thanks go to Silvia Breu and also to Franz, Christian, and Andreas for proof-reading long parts of my thesis.

Last but not least, I would like to thank my family. Without the support of my parents Elfriede and Andreas Forster it would not have been possible for me to study computer science. And without Stefanie Awdejew, the girl I love, there would have been various occasions to despair. I am very sorry for the little time I had to spend with her while writing this thesis.

It's been said that a picture's worth 1,000 words. Many have said that the Chinese philosopher, Confucius said it in 500 BC. But what kind of picture was Confucius talking about? Probably a simple line drawing. And as it turns out, in most word processing programs, 1,000 words is about 11K—equivalent to a simple line drawing. But when Hollywood digitizes a frame for a special effect, a single frame is about a 40 Meg file. So, a picture today is worth about 25 million words!

Paul Martin Lester [152]

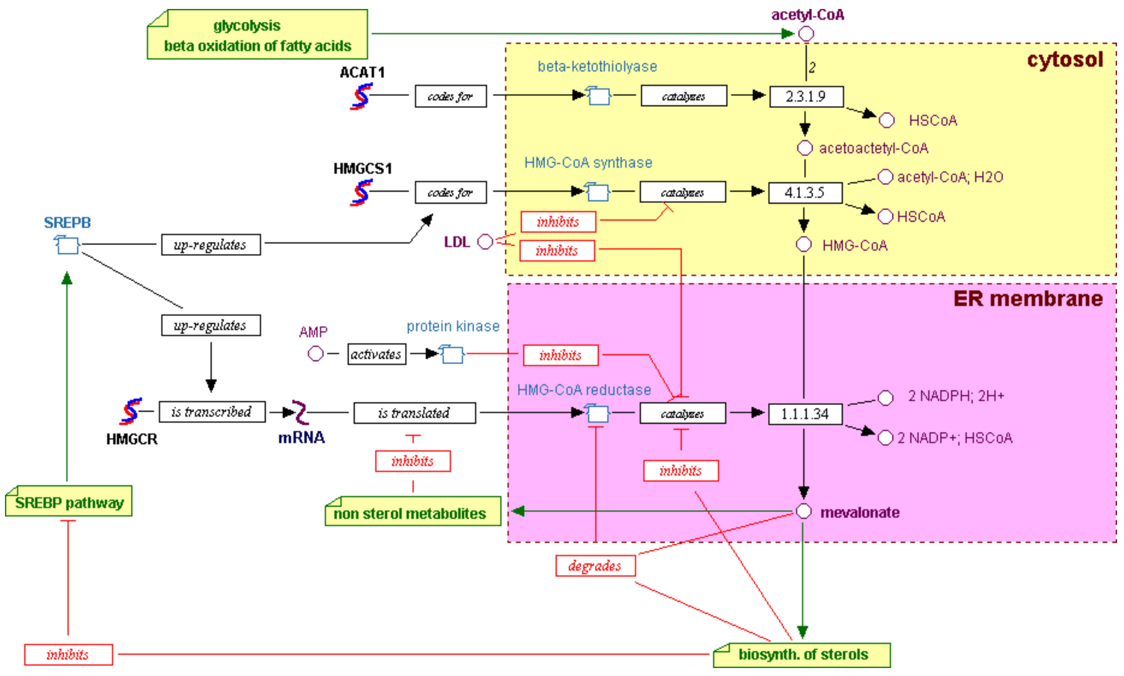
2 Introduction

"A picture is worth a thousand words." This famous quote ascribed to the Chinese philosopher Confucius may be even more true today, in the context of huge and complex data sets in, e. g., physics, biology, and information technology. Of course, analyzing the information content of data is not only a matter of its size, but also of its structure. Complex sets are more difficult to understand than simple ones, even if they are smaller. Though, if they are presented in visual form, humans can recognize and understand them much more easily than in textual or mathematical form. For example, the shortest path between two street locations can be found fastest by using a street map, instead of a list of street coordinates. The superiority of visualizations, however, is apparently only given for good pictures. Bad visualizations can be confusing or even misleading. For street maps, criteria for distinguishing "good" from "bad" visualizations are easy to define. But in general the key problem is: What exactly is a good picture? Obviously, a good picture must be clear and uncluttered to be easily understandable. But what does this mean in mathematical terms? And, even more important: How can such a picture be generated from given data?

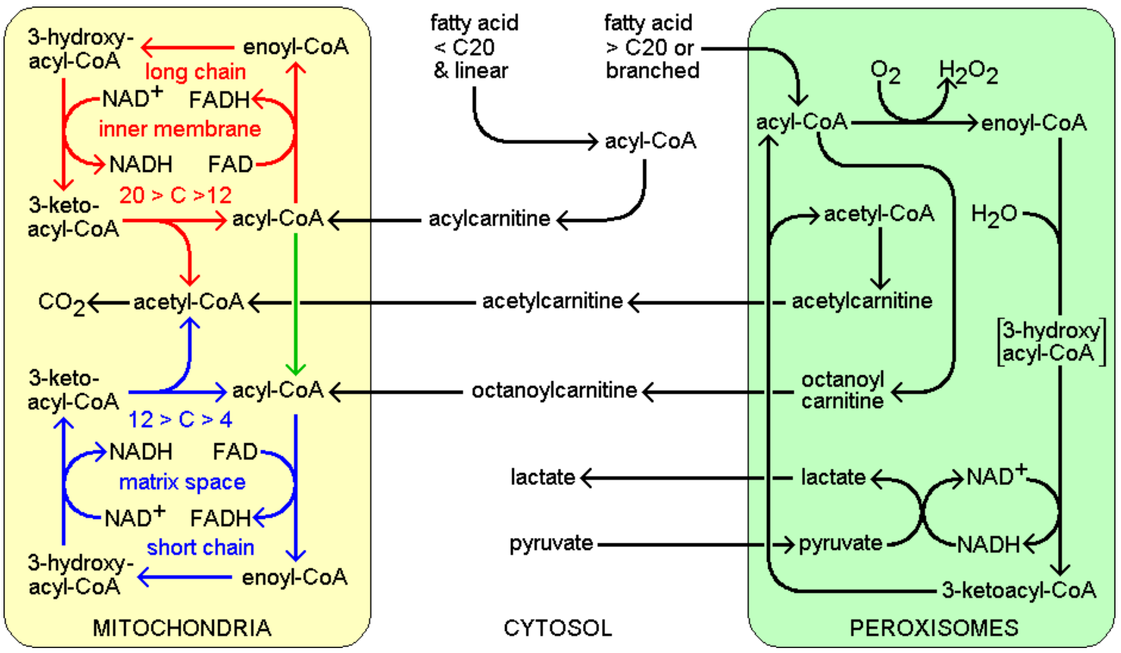
In real applications, the answers to these questions depend on the intention of the picture and on the form of the given data. Graph drawing considers data that can be modeled as graphs with associated attributes. Examples are easy to find, because nearly every finite data set that represents some kind of relations between some objects can be modeled as a graph. Graph-like structures are ubiquitous in information visualization: data flow diagrams, class hierarchies, entity-relationship diagrams, Petri nets, state transition diagrams, PERT charts, electronic circuits, file system hierarchies, and many more.

While graphs are well suited for displaying most categories of relational data, very large and complex structures in real life applications cannot be represented appropriately by standard graphs. In these cases extended graph models are used, such as *clustered graphs* or similar hierarchically structured graphs. There are several advantages of such graph models. As the size and complexity of data sets grow, it becomes more and more difficult to represent and visualize them in their entirety. Really large graphs, such as the web graph, circuit diagrams, or biochemical pathways cannot be visualized as a whole at once. Partial solutions for this problem are to show only a clipping of the whole drawing, or to emphasize certain parts with fisheye views [90, 99, 139, 143, 198]. These techniques, however, cannot satisfactorily give a global overview of the graph, but only of local relations. Furthermore they do not take into account additional structural information. A superior solution is to exploit a hierarchical clustering structure of the graph, which is available in most applications. The graph is partitioned recursively into a hierarchy of subgraphs. Then different views on the graph at different levels of detail can be generated. This is particularly useful for interactive exploration of the graph. Clusters of vertices can be expanded and contracted by the user, see [29, 30, 184–186, 200]. There are various tools that support such navigation techniques [1, 20–22, 64, 94, 96, 121, 143, 181, 214, 215].

As a second application, clustered graph models can also be used for the visualization of graphs that are not necessarily large or complex, but where a hierarchical structure represents important additional information. There are many application areas for this type of clustered graphs, for example in statistics [106] or linguistics [7]. Another example for directed clustered graphs are biochemical pathways, which directly motivated the research in this thesis. Leading to many new and interesting challenges for graph drawing, they have attracted much attention recently [8, 20–24, 61, 94, 104, 136, 165, 200–203, 209]. Biochemical pathways are reaction networks modeling parts of, e. g., the human metabolism. They are usually represented by directed (hyper) graphs consisting of vertices for the substances and directed (hyper) edges for the reactions, see Figure 2.1. Parts of a reaction network take place in different regions of the cell, e. g., in the nucleus or in the cytosol. These *cell compartments* define different components (clusters) of the graph. This information can be visualized by drawing boxes around related parts of the graph. In Figure 2.1(a), the boxes for the cytosol and the ER membrane make it evident which reactions occur in these compartments. Due to nesting of cell compartments it may also be necessary to nest the boxes. This leads to clustered graphs.



(a) A part of the cholesterol biosynthesis [226]



(b) Metabolic specialization and cooperation between compartments [124]

Figure 2.1. Visualizations of clustered biochemical pathways

As another application for directed clustered graphs consider class diagrams in the unified modeling language UML [192], which share similar requirements. Again, there is an underlying directed graph, consisting of vertices for the classes and edges modeling inheritance and association. The classes are grouped in boxes that declare the UML package they belong to. The boxes are recursively nested in arbitrary depth. In Figure 2.2, which shows a simple UML class diagram, it is immediately evident, which classes belong together. For example, the position of the `Cloneable` class has been chosen so that it can be drawn within the cluster for the `java.lang` package.

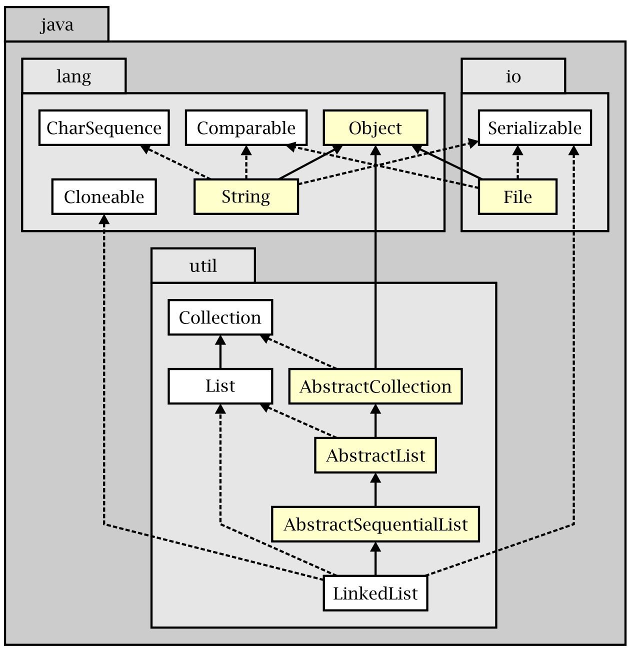
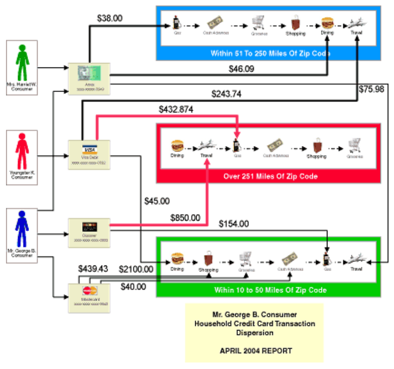


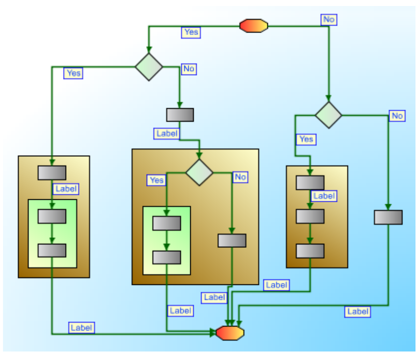
Figure 2.2. A UML diagram showing a part of the Java API [217]

Finally, clustered graphs also appear in declarative approaches to graph drawing, such as in layout graph grammars and similar work [16, 17, 119, 153, 162, 163, 205, 206]. Layout graph grammars are a rule-based method for the construction of graphs and graph drawings. They consist of an underlying context free graph grammar and layout specifications attached to the productions. Starting with a single vertex, successive application of productions to nonterminal vertices finally leads to a clustered graph. Here a cluster corresponds to a nonterminal vertex and contains the subgraph that has been derived from it. The clustered graphs that are constructed from a graph grammar are special, because any production replaces a vertex by a constant number of new vertices. Therefore the cluster tree in the resulting clustered graph is of bounded degree. Because of this, several otherwise intractable problems can be solved efficiently on such clustered graphs [119].

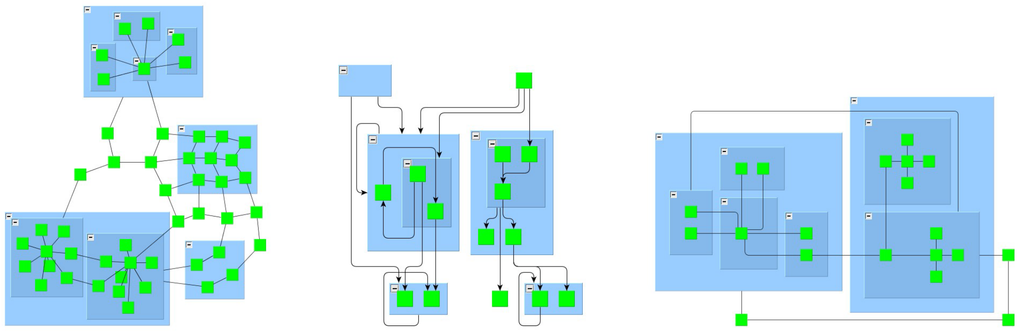
The practical relevance of clustered graphs is stressed by the fact that all well-known commercial graph drawing libraries include support for clustered graphs or similar graph models, see Figure 2.3.


$38.00
Within 51 To 250 Miles Of Zip Code
$46.09
$243.74
$75.98
$432.874
Over 251 Miles Of Zip Code
$45.00
$850.00
$154.00
$439.43
$2100.00
$40.00
Within 10 to 50 Miles Of Zip Code
Mr. George B. Consumer
Household Credit Card Transaction
Dispersion
APRIL 2004 REPORT

(a) Tom Sawyer Software [221]



(b) ILOG [125]



(c) yWorks [236]

Figure 2.3. Nested drawings in commercial graph drawing

This thesis is structured as follows: The next chapter introduces some concepts and definitions that are fundamental for all of the presented work. In Chapter 4 we present clustered level graphs. These are special clustered graphs where all vertices and edge bends are drawn on horizontal levels. Such drawings are the de-facto standard for drawings of directed graphs. We will analyze in detail the characteristics of edge crossings and cluster/edge crossings in clustered level graphs. The minimization of edge crossings is one of the most important aesthetic criteria for level drawings [182, 183].

The subsequent three chapters investigate specific algorithmic problems related to crossings in clustered level graphs: Chapter 5 addresses the crossing reduction problem in clustered level graphs. For the solution of this problem, an algorithm for constrained crossing reduction is needed, which is presented in chapter 6. Chapter 7 investigates the problem of clustered level planarity, i. e., whether drawings without any crossings are possible. Finally, we close with a summary in Chapter 8.

If you have built castles in the air, your work need not be lost; that is where they should be. Now put the foundations under them.

Henry David Thoreau

He who has not first laid his foundations may be able with great ability to lay them afterwards, but they will be laid with trouble to the architect and danger to the building.

Niccolo Machiavelli

3 Foundations

This chapter gives a brief introduction to graph drawing and to graphs in general. It introduces the basic graph theoretical terms and concepts, and presents the notation used in this thesis. A further overview of graph theory resp. graph drawing algorithms can be found in [41, 43] resp. [55, 138, 213]. The proceedings of the graph drawing symposiums [18, 52, 54, 142, 144, 156, 158, 172, 176, 179, 220, 233] may serve as a guideline for the ongoing research in the field.

3.1 Graphs

Graphs are a general purpose data structure for the representation of binary relational data. Depending on the application, there are various notations for graphs. Since we only consider finite simple directed graphs, the following standard definition for directed graphs without self loops or multiple parallel edges is used:

Definition 3.1 (Graph). *A* (directed) graph $G = (V, E)$ *consists of a finite set of vertices* V *and a finite set of directed edges* $E \subseteq \{\, (u, v) \in V \times V \mid u \neq v \,\}$*.*

There are many basic graph theoretical terms associated with graphs. The following definition summarizes the most important concepts needed for the descriptions in the following chapters. For a more detailed introduction to graphs please refer to a book about graph theory, such as [41].

Definition 3.2 (Basic Graph Terms). *Let $G = (V, E)$ be a graph.*

1. *For an edge $e = (u, v) \in E$ the vertices u and v are called* source (vertex) *and* target (vertex) *of e, respectively, while e is called an* outgoing *edge of u and an* incoming *edge of v. The vertices u and v are called* adjacent *to each other and* incident *to e.*

2. *The* direct predecessors *and* direct successors *of a vertex $v \in V$ are defined as* $\mathrm{pred}_G(v) = \{u \in V \mid (u, v) \in E\}$ *and* $\mathrm{succ}_G(v) = \{u \in V \mid (v, u) \in E\}$, *respectively. The* successors $\mathrm{succ}^*_G(v)$ *and* predecessors $\mathrm{pred}^*_G(v)$ *of v are the respective reflexive transitive closures. A* source *of the graph is a vertex without predecessors, a* sink *is a vertex without successors.*

3. *A* path $p = (v_1, v_2, \ldots, v_k) \in V^k$ *is a sequence of vertices connected by a sequence of edges: $\forall i \in \{1, \ldots, k-1\}\colon (v_i, v_{i+1}) \in E$. p is a* cycle *if $(v_k, v_1) \in E$. G is* acyclic, *i. e., a* directed acyclic graph (DAG), *if it contains no cycles.*

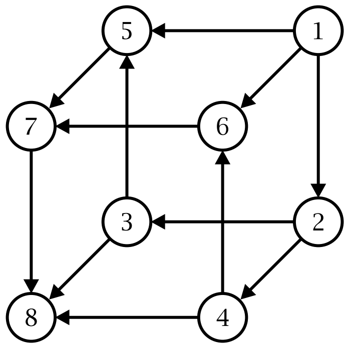
4. *An acyclic graph is a* tree *if it has a single source (its* root*) and each other vertex has exactly one direct predecessor (its* parent*). In a tree, sinks are also called* leaves *and the terms* children, descendants, *and* ancestors *are used for direct successors, successors and predecessors, respectively.*

5. *A graph is strongly connected, if for every pair of vertices $u, v \in V$ there is a path from u to v. It is (weakly) connected, if its symmetric closure is strongly connected.*

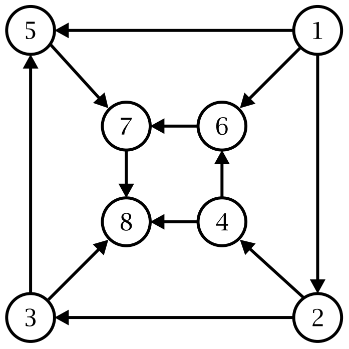
6. *A graph $G' = (V', E')$ is a* subgraph *of G if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. It is an* induced subgraph *of G if $E' = E \cap (V' \times V')$.*

3.2 How to Draw a Graph

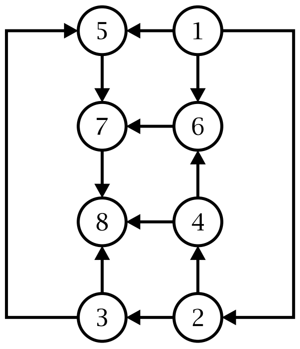
Before we investigate how a good drawing of a graph can be computed, it is necessary to define what “good” means. Unfortunately, this is very hard to define. The quality of a drawing is very dependent on the semantics of the given graph and on the intended purpose of the drawing. For instance, Figure 3.1 shows the same graph drawn in various ways. None of the drawings is inherently better than the others since each of them emphasizes different properties of the graph. In consequence, there are various graph drawing algorithms with different results.



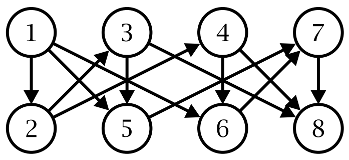
(a) 3D symmetries, uniform edge length, no edge bends



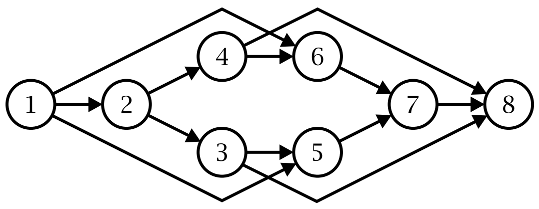
(b) 2D symmetries, planar, no edge bends



(c) Planar, orthogonal



(d) Bipartite, minimum area, no edge bends



(e) Acyclic, uniform edge direction

Figure 3.1. The same graph drawn with different priorities

The classification of graph drawing algorithms is subject to several characteristics, which can be put into three categories: drawing conventions, aesthetic criteria and constraints. Every algorithm produces drawings according to its *drawing conventions*, a basic set of rules that is observed by every generated drawing, irrespective of the input graph. There are various drawing conventions used by different graph drawing algorithms. The most simple kind of drawings are *straight-line drawings* as in Figures 3.1(a), 3.1(b), and 3.1(d).

Definition 3.3 (Straight-Line Drawing). *Let $G = (V, E)$ be a graph. A* (two-dimensional) straight-line drawing *of G is a function $\delta\colon V \to \mathbb{R}^2$ that assigns coordinates to each vertex. The vertices are drawn as points in the plane (or as some small geometric object) and are connected by the edges drawn as straight lines.*

Slightly more complex are *polyline drawings*, as those in Figures 3.1(c) and 3.1(e). They differ from straight-line drawings in the routing of the edges, which are allowed to have a finite number of bends between multiple straight-line segments:

Definition 3.4 (Polyline Drawing). *Let $G = (V, E)$ be a graph. A* (two-dimensional) polyline drawing *$\delta = (\delta_V, \delta_E)$ of G consists of two functions $\delta_V\colon V \to \mathbb{R}^2$ and $\delta_E\colon V \to (\mathbb{R}^2)^*$ that assign coordinates to each vertex resp. each edge bend.*

Within the bounds of the drawing conventions an algorithms tries to optimize some *aesthetic criteria* of the drawing: small area, uniform edge length, few edge bends and crossings, good spatial and angular resolution, and others. These criteria typically are in conflict with each other, and thus different algorithms prioritize differently. The most important aesthetic criterion for the purpose of this work is the minimization of edge crossings. Empiric studies by Purchase [182, 183] show that few edge crossings are very important for the ease of human understanding.

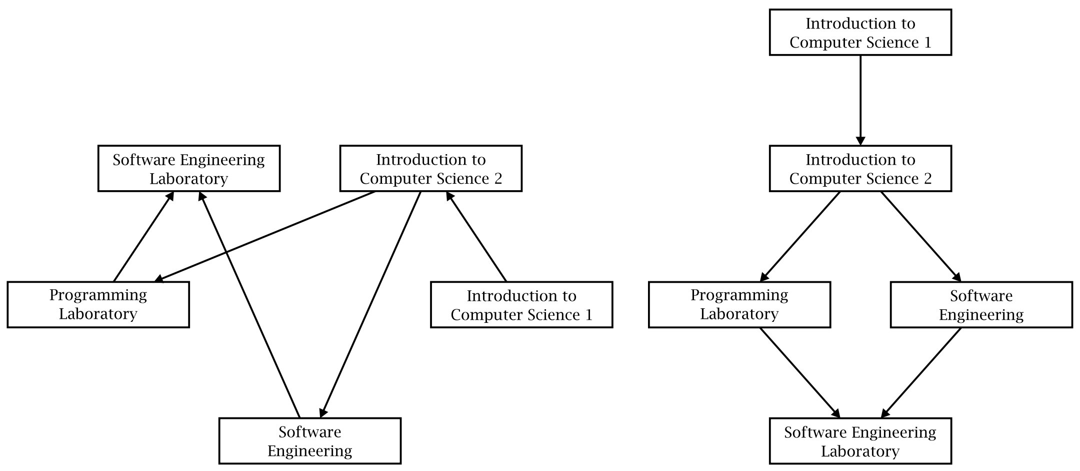
Some algorithms also have the ability to observe drawing constraints. A constraint is a given restriction for drawing a specific subgraph of the actual input graph. For instance, an edge can be constrained to have a certain direction, or two vertices can be restricted to have the same x-coordinate. The set of given constraints must be consistent with the drawing conventions, but they may conflict with the aesthetic criteria. The objective is to compute a drawing within the bounds of the drawing conventions that satisfies all constraints and optimizes the aesthetic criteria.

3.3 Drawing Directed Acyclic Graphs

The basis of our considerations are drawings of directed acyclic graphs (DAGs), one of the most important classes of graphs for applications. When drawing directed graphs, it is often desirable to emphasize the orientation of the edges. This applies particularly for acyclic graphs, because then it is possible to draw all edges in the same general direction. This can be used to better visualize the semantics of the orientation, for instance dependency in logic applications, a main direction of flow, or the time line in scheduling applications.

Example 3.1 (University Course Dependencies)**.** *Consider a dependency graph of courses given at some university. Each course is represented by a vertex, and course prerequisites are connected by directed edges. The graph is acyclic, because cyclic dependencies would make it impossible to attend any of the involved courses. The semantics of the dependencies imply that all edges should be drawn in the same direction.*

Figure 3.2 shows a small part of the dependency graph of computer science courses at the University of Passau. Comparing two drawings of the same graph, the drawing with uniform edge directions in Figure 3.2(b) clearly gives a better overview. Even in this very small example, the overall relationship between advanced courses at the bottom and basic courses at the top is much more visible than in Figure 3.2(a).


Software Engineering Laboratory
Introduction to Computer Science 2
Programming Laboratory
Introduction to Computer Science 1
Software Engineering
Introduction to Computer Science 1
Introduction to Computer Science 2
Programming Laboratory
Software Engineering
Software Engineering Laboratory

(a) Varying edge directions (b) Uniform edge directions

Figure 3.2. Drawing directed acyclic graphs

The most common approach for drawing DAGs are *level drawings*, also called *layered drawings* or *hierarchical*[1] *drawings*. These are polyline drawings with all vertices and edge bends arranged in horizontal levels and all edges oriented downwards.[2] Although there was some initial work on level drawings by Warfield [230] and Carpano [31] before, this approach is commonly attributed to Sugiyama et al. [216]. The Sugiyama algorithm is divided into four phases, see Figure 3.3. Each of the phases is computationally hard and has been intensively studied. We will present a short description of each phase and an overview of the many heuristics and algorithmic variations that exist. For a more detailed investigation see [6] and [55, chapter 9].

3.3.1 Cycle Removal

Although level drawings are primarily suited for drawing DAGs, they can also be used for graphs that contain cycles. Of course, then it is not possible to draw all edges downwards. In every drawing at least one edge of each cycle points upwards. Even so, there are many applications, where graphs with cycles should be drawn with a maximum number of edges being directed into the same direction. Therefore the first step for a level drawing is to eliminate

[1]The term *hierarchical* is overloaded in graph drawing. A *hierarchical drawing* is different from a *hierarchical graph* and also from a level graph that is a *hierarchy*. To avoid confusion we will avoid using these terms where possible.

[2]By convention we always draw DAGs top down with the origin in the upper left corner and coordinates growing down and rightwards. Of course this is equivalent to drawings in other directions by a simple coordinate transformation.

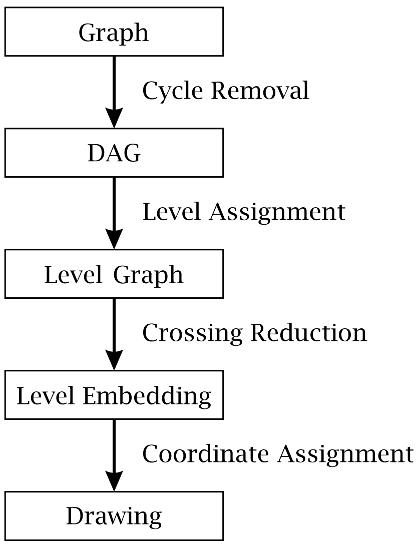


Figure 3.3. The phases of the Sugiyama algorithm

all cycles. This can be done by removing or alternatively by reversing some edges. The latter is preferred, because the quality of the drawing decreases, if some edges are entirely ignored throughout the algorithm. The adjacency information represented by the removed edges is lost for the following phases of the algorithm. Furthermore, removed edges are difficult to reinsert.

It is desirable to remove or reverse as few edges as possible. Finding a minimum set of edges whose removal makes the graph acyclic is known as the *feedback arc set* problem, a well-known NP-hard problem [102, 137]. Although edge removal and reversion are different problems, finding a minimum set of edges whose reversal makes the graph acyclic is NP-hard as well [55]. Removing all edges makes any graph acyclic, but reversion of all edges only inverts the direction of each cycle. However, the minimum solutions of the removal problem are exactly the minimum solutions of the reversal problem and hence the NP-hardness can be transferred.

Thus, efficient heuristics are used for the cycle removal problem. A simple solution is to traverse the graph in depth first search (DFS) order and to reverse all back edges and cross edges. This approach can lead to $|E| - |V| + 1$ reversed edges. A better and even simpler method is to arbitrarily order all vertices from top to bottom and to reverse all edges that point upwards. If this reverses more than half of the edges, the opposite direction is used for all edges, leading to at most $\frac{|E|}{2}$ reversed edges. A better quality, but also a considerably higher running time is achieved by computing the biconnected components of the graph and iteratively removing edges in biconnected components until the graph is acyclic. For more sophisticated algorithms see [6, 9, 55, 73, 74, 114, 134].

3.3.2 Level Assignment

If the given graph $G = (V, E)$ is a DAG, or after all cycles have been removed, the vertex set V is partitioned into levels $V_1, \dots, V_k$. All vertices of the same level V_i are later drawn on the horizontal line $l_i = \{ (x, i) \mid x \in \mathbb{R} \}$. The level number is identical to the y-coordinate of the vertex. The result of the level assignment step is a level graph:

Definition 3.5 (Level Graph). *A* k-level graph $G = (V, E, \phi)$ *is a graph* (V, E) *with a* leveling $\phi\colon V \to \{1, \dots, k\}$ *that partitions the vertex set into* k *disjoint* levels $V_1, \dots, V_k$, $V_i = \phi^{-1}(i)$*, such that each edge* $(u, v) \in E$ *has a positive* span $\phi(v) - \phi(u) > 0$*, i. e., all edges point downwards. Edges are called* proper *if their span is* 1 *and* long span edges *otherwise.* G *is proper if all its edges are proper.*

The quality of a leveling depends on various parameters. First it is desirable that drawings are compact, i. e., that they have small height and width. Because the level numbers are equivalent to the vertical coordinates of the vertices, the height of a drawing is an immediate result of the number of levels. An obvious lower bound for the number of levels is the length of the longest path in G. The lower bound can be reached by the simple *longest path leveling* algorithm, which is also called *critical path method*. All vertices without incoming edges are placed on level 1, and are then removed from the input graph together with their incident edges. All remaining vertices that now have no incoming edges left are placed on level 2, then removed, and so on. This leads to a drawing of minimum height, but ignores the width of the drawing.

Although the exact width is not defined until the last phase of the Sugiyama algorithm, it mainly depends on the leveling, because levels with many vertices need more horizontal space than sparse levels. With a uniform level distribution of the vertices, a smaller width can be expected. However, minimizing the width of a minimum height leveling is NP-hard, as can be shown by a simple reduction from the multiprocessor scheduling problem [102]. Therefore, many heuristics have been developed, for instance the well-known Coffman Graham heuristic, see [42].

For drawing proper level graphs, no edge bends are needed. For non-proper level graphs a corresponding proper level graph is constructed by splitting long span edges into a sequence of proper edges (edge segments):

$$(u, v) \longrightarrow (u = w_0, w_1), (w_1, w_2), \dots, (w_{s-1}, w_s = v).$$

New dummy vertices $w_1, \dots, w_{s-1}$ with $\phi(w_i) = \phi(u) + i$ are introduced and later replaced by edge bends, if necessary. It is desirable to introduce as few dummy vertices as possible for several reasons: In the worst case a quadratic number of dummy vertices may have to be created, slowing down considerably the later phases of the algorithm. Even more important, edge bends and long span edges make the drawing more difficult to understand, because short edges are much easier to follow visually. The bend minimization problem can be solved in polynomial time by exact integer linear programming techniques [100]. However, it becomes NP-hard again when simultaneously minimizing the height of the drawing [154]. Recently, Eiglsperger, Siebenhaller and Kaufmann [81] presented a new technique, how the Sugiyama algorithm can be implemented without the explicit generation of dummy vertices.

In some applications the level assignment is already given by the semantics of the data, as presented in the following example:

Example 3.2 (Given Leveling). *In the university course dependency graph in Example 3.1 each course is intended to be attended in a specific semester. It is desirable to draw the courses of a semester on a single line and to partition the vertices into levels according to the semester number, see Figure 3.4. Since a course can only depend on courses in lower semesters, all edges point from higher to lower levels and thus the graph is a k-level graph, where k is the expected number of needed semesters.*

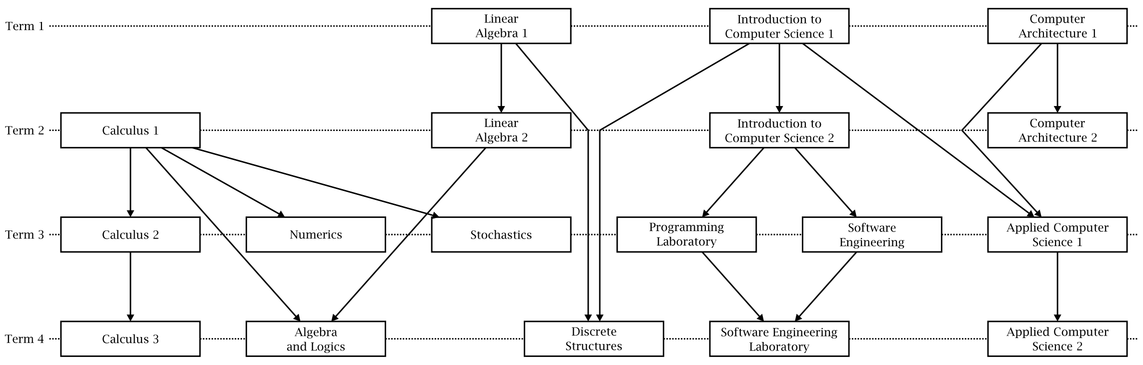


Figure 3.4. Dependencies of computer science courses at the University of Passau

3.3.3 Crossing Reduction

In straight-line drawings of proper level graphs, edge crossings do not depend on the exact coordinates of the source and target vertices, but only on their relative positions. Because the vertical positions are fixed by the leveling, only the ordering of the vertices on each level is significant. Therefore, given a level graph, the next step is to compute these orderings, a so-called level embedding:

Definition 3.6 (Level Embedding). *Let $G = (V, E, \phi)$ be a proper level graph. A* level embedding[3] $\pi\colon V \to \mathbb{N}$ *of G is a linear ordering of the vertices in each level $\pi(V_i) = \{1, \ldots, |V_i|\}$.*

Two proper edges (u, v) and (u', v') starting on the same level $\phi(u) = \phi(u')$ cross each other if and only if $(\pi(u') - \pi(u)) \cdot (\pi(v') - \pi(v)) < 0$. To compute drawings with few crossings, we are interested in level embeddings with few crossings. In the optimal case there are no crossings at all, and the drawing is planar:

Definition 3.7 (Level Planar Embedding). *A level embedding is* level planar *if it induces no edge crossings. A proper level graph is* level planar *if there exists a level planar embedding of it.*[4]

Crossings in level embeddings have been studied extensively for about 35 years, with first results from Harary, Schwenk, and Watkins [109, 110, 231]. Although it can be tested in linear time, whether a level planar embedding exists [129, 130], the problem of finding a level embedding with a minimum number of edge crossings is NP-hard [103]. The common heuristic approach, first proposed by Warfield [230], is to reduce the problem to successive applications of the supposedly simpler *one-sided two-level crossing minimization* problem. Starting with an arbitrary ordering of the first level, an ordering of the second level is computed, minimizing the number of crossings between these two levels. This step is repeated for each level in multiple top-down and bottom-up level-by-level sweeps over the graph.

Although one-sided two-level crossing minimization is also NP-hard [78, 80], it can be solved efficiently in many cases [126, 132, 133, 225]. For larger instances, heuristics are applied, such as the barycenter [101, 216] and median [79, 80, 100, 168] heuristics. The vertices are sorted by the barycenter or median of their predecessors' positions, respectively. The barycenter is the average position of all direct predecessors

$$b(v) = \sum_{u \in \mathrm{pred}_G(v)} \frac{\pi(u)}{|\mathrm{pred}_G(v)|} ,$$

while the median considers only the central direct predecessor(s). The median heuristic misses the optimum by provably at most a factor of three [55]. The barycenter heuristic, however, gives better experimental results. A better theoretical bound is obtained with a heuristic by Yamaguchi and Sugimoto [235].

[3]The term *embedding* is often used for *planar embeddings* only. Note that here a level embedding is not necessarily planar.

[4]Level planar graphs have also been called *h-planar* graphs, e. g., in [63, 68, 69].

The practical importance of the crossing reduction problem is stressed by the large number of alternative heuristics that optimize running time and crossing number in experimental results, such as greedy strategies [71, 167], stochastic insertion [62], assignment heuristics [34, 35], and greedy randomized adaptive search [145, 159, 160]. Other approaches initialize the algorithm by first embedding only a part of the graph, like a depth first search tree [196], a spanning tree [164] or a maximum planar subgraph [171, 218]. Furthermore, several generic algorithmic concepts have been applied, like genetic algorithms [169], tabu search [146] and sifting [105, 161]. There are also some variations of the level sweep, such as sorting the vertices on a level according to both adjacent levels [190], or by using *two-sided two-level crossing minimization* [133, 204, 210], i. e., simultaneously optimizing two consecutive levels. With Tutte's Algorithm [77] the crossings on all levels can be minimized simultaneously, if a permutation of the first and last level are given.

An overview and experimental comparison of many heuristics can be found in [132, 133, 160]. As a general rule, the results of the heuristics do not vary very much for dense graphs, and thus the barycenter heuristic is often a good and efficient choice. For sparse graphs the decision depends on the envisioned running time. The barycenter heuristic then also gives good results, but there are some heuristics with higher running time that outperform it significantly, such as the greedy randomized adaptive search approach by Martí and Laguna [145, 159, 160], and the exact branch-and-cut approach by Jünger and Mutzel [133]. The time needed for counting the crossings also becomes significant when using a fast heuristic. While a trivial implementation needs $\mathcal{O}(|E|^2)$, there are also more efficient algorithms [5, 228] with an optimal running time of $\mathcal{O}(|E| + c)$, where c is the number of crossings.

A related problem, which is especially important for our investigations, is the minimization of crossings with respect to given drawing constraints, i. e., if for some vertex pairs $(u, v) \in V^2$ the vertex u must be positioned to the left of the vertex v. The constraints can be given by the user or by an algorithm that uses constrained crossing reduction as a subroutine, e. g., the minimization of crossings for clustered level graphs. Heuristic solutions are given in [50, 51, 89, 196, 200, 227] and in Chapter 6.

3.3.4 Coordinate Assignment

The last step in drawing the graph is to replace all dummy vertices by edge bends, and to assign coordinates to all vertices and edge bends. Finally, a level drawing is generated:

Definition 3.8 (Level Drawing). *A* level drawing *of a DAG is a two-dimensional polyline drawing where all vertices and edge bends lie on horizontal levels and all edge segments point downwards.*

Because the vertical positions are already fixed by the leveling, only the horizontal coordinates remain to be computed. This is done by retaining the ordering computed before, and optimizing several aesthetic criteria: The number of edge bends and the horizontal stretch of the edges are minimized while the number of vertical edge segments is maximized. For a harmonic picture the balancing of vertices subject to the incident edges is also important. The width of the drawing is often given a lower priority than the number of edge bends [25]. For level planar embeddings Eades et al. [68, 69] give an algorithm that does not generate bends at all. However, the drawings may need exponential area.

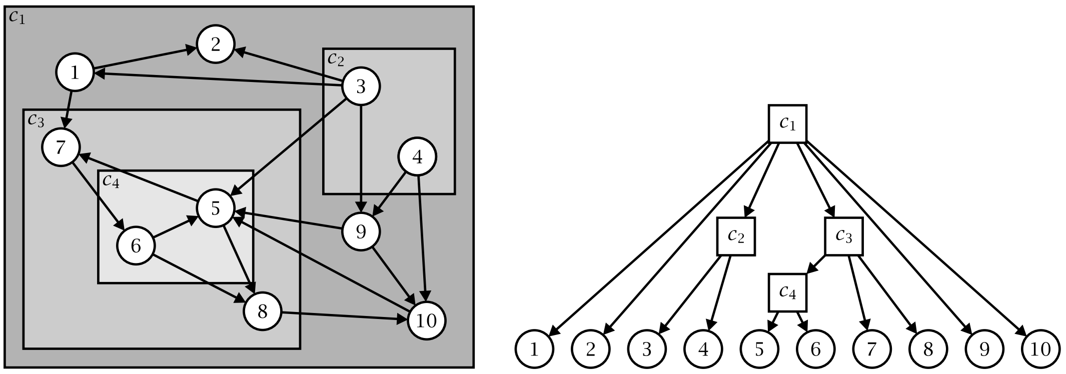
There are several algorithms for horizontal coordinate assignment [28, 75, 77, 95, 100, 101, 193, 194, 196, 197, 214, 216] using different approaches for the optimization of various objective functions or iterative improvement techniques. Most interesting is the algorithm of Brandes and Köpf [25], which generates at most two bends per edge if no two inner segments cross each other. It also gives good results for the other aesthetic criteria. The algorithm has a linear running time and is also very fast in practice.

3.4 Drawing Clustered Graphs

3.4.1 Clustering in Graphs

Clustering is a widely used technique for the reduction of complexity or for the representation of hierarchical structures. In graphs, there are two main approaches to clustering: vertex clustering and edge clustering. Edge clustering is used, for example by Schreiber [200], who describes a navigation technique in biochemical pathways that represents subgraphs (“pathways”) by edges. Other techniques like edge concentration [155, 174, 180], factoring [36], or graph compression [85] replace complete subgraphs or complete bipartite subgraphs by single vertices that represent the edges. Recently this approach has also been used to reduce the number of crossings in so-called confluent drawings [60, 82, 122].

Motivated by the applications mentioned in Chapter 2, however, the main focus of this thesis is on vertex clustering. There are various extended graph models for representing the grouping of vertices such as clustered graphs [67–69, 86–88], statecharts [32, 33, 111], higraphs [112], cigraphs [147], hierarchical graphs [150, 151], compound graphs [214], (layout) graph grammars [16, 17, 119], and clan-based graph decompositions [162, 163, 205, 206]. See also [26] for an overview. All of these approaches have in common that subsets of vertices are grouped together to form a hierarchical clustering of the graph. They differ in particular features, such as whether clusters are allowed to overlap, whether edges can be drawn between clusters, and in which way vertices are grouped together. Such features improve the expressiveness of the drawings, but they also increase their complexity. The more features the graph model has, the more difficult it is to handle. Additional features have to be considered in all definitions, theorems and algorithms. Therefore, we choose a comparatively simple, yet powerful graph model: directed clustered graphs [67–69, 86–88], see Figure 3.5.

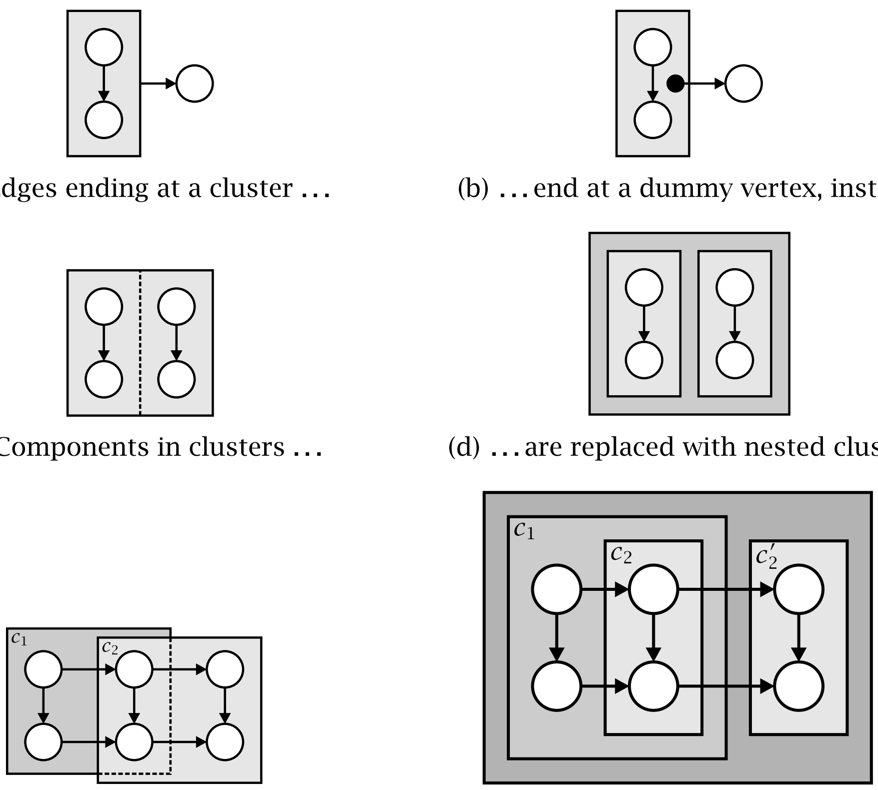


(a) A drawing of G (b) The cluster tree $\Gamma = (V \cup C, I)$

Figure 3.5. A clustered graph $G = (V, E, C, I)$

Clustered graphs represent a recursive vertex clustering of arbitrary finite depth. The vertices are the leaves of an inclusion relation that is required to form a tree, i. e., clusters do not overlap. Edges are only allowed to connect vertices but not clusters. To a certain extent, more complex features of other clustered graph models can be emulated in clustered graphs, see Figure 3.6. Using these replacements and techniques like drawing constraints, graph drawing algorithms can generate drawings that are similar to those of more complex graph models. Figure 3.7 shows a rather complex statechart and how it can be represented as a clustered graph. Formally, clustered graphs are defined as follows:

Definition 3.9 (Clustered Graph). *A* clustered graph $G = (V, E, C, I)$ *consists of an* underlying graph (V, E), *clusters* C, *and a recursive* inclusion relation I. I *builds a rooted tree* $\Gamma = (V \cup C, I)$ *with the clusters* C *as inner nodes and the vertices* V *as leaves, such that each cluster has at least two children.* Γ *is called the* cluster tree *of* G.



(a) Edges ending at a cluster ...

(b) ... end at a dummy vertex, instead

(c) Components in clusters ...

(d) ... are replaced with nested clusters

(e) Overlapping clusters ...

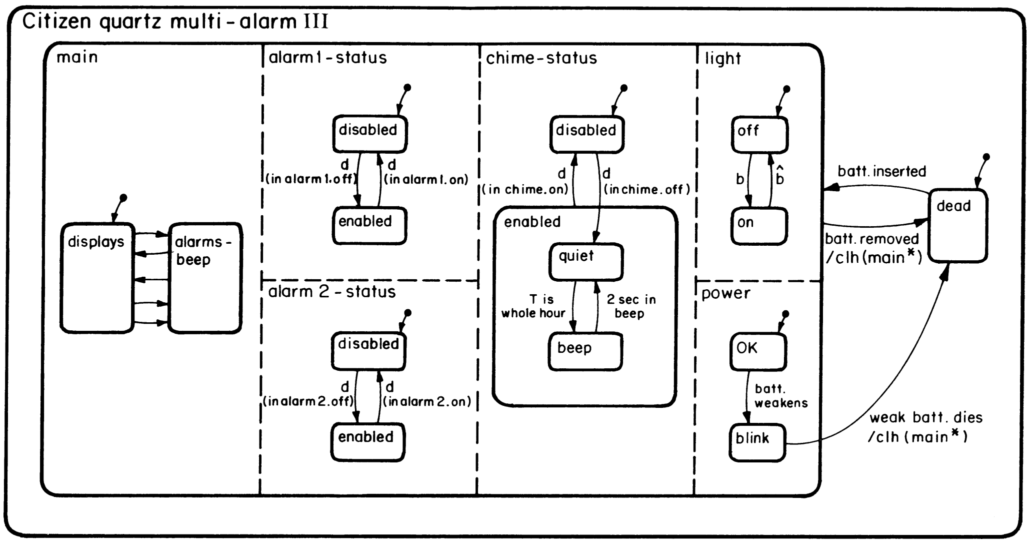
(f) ... are divided into multiple clusters

Figure 3.6. Modeling advanced clustering techniques with clustered graphs

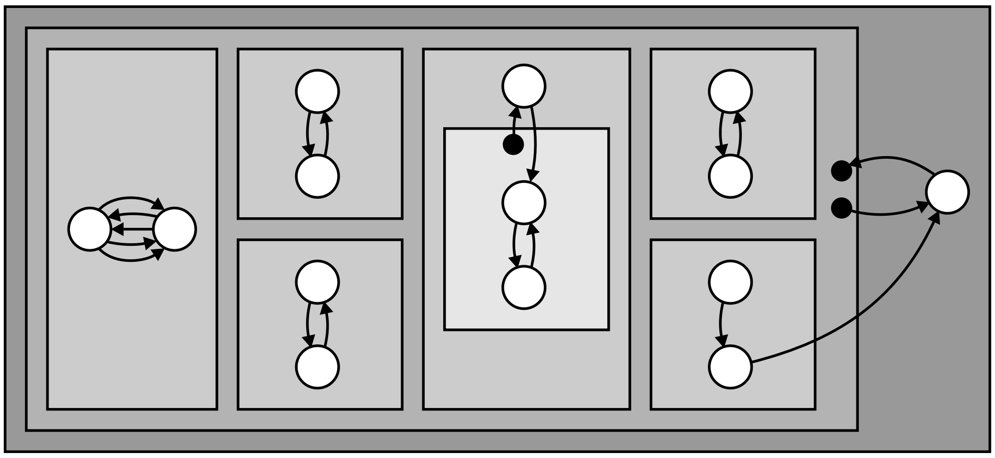
If in an application clusters with only a single child occur, a preprocessing step is used to combine sequences of these clusters into a single cluster. Then the number of clusters $|C|$ is linear in the number of vertices $|V|$. Consider Figure 3.5 for a drawing of a clustered graph and an illustration of its cluster tree. Each cluster $c \in C$ induces a subgraph $G_c = (V_c, E_c)$ of G. The vertices $V_c \subseteq V$ of this subgraph are the leaves of the cluster tree $\Gamma = (V \cup C, I)$ that are reachable from c.

Definition 3.10 (Contained Vertices). *Let $c \in C$ be a cluster of a clustered graph $G = (V, E, C, I)$. A vertex $v \in V$ is* contained *in c if it is a descendant of c in the cluster tree. The set of vertices contained in c is denoted by $V_c = \mathrm{succ}^*_\Gamma(c) \cap V$. These are the leaves of the subtree rooted at c.*

Example 3.3. *In Figure 3.5 the root cluster c_1 contains all vertices of the graph ($V_{c_1} = V$) while c_2 contains only its children: $V_{c_2} = \{3, 4\}$. The cluster c_4 is nested within c_3, and thus its set of contained vertices $V_{c_4} = \{5, 6\}$ is a subset of $V_{c_3} = \{5, 6, 7, 8\}$.*



(a) A complex statechart [111] . . .



(b) . . . and its abstraction as a clustered graph

Figure 3.7. Modeling statecharts with clustered graphs

3.4.2 Drawing Conventions

Clustered graphs can be drawn in various ways. Firstly, a clustered graph consists internally of two graphs, the underlying graph and the cluster tree. Thus different drawing styles have two choices for their main focus. It would be possible to generate a drawing of the cluster tree with a tree drawing algorithm [27, 123, 188, 232], and to connect the leaves with the edges of the underlying graph. This way, the inclusion relation is very evident, but the drawing of the underlying graph is a linear arrangement of the vertices, and thus not well readable.

Eades and Feng [67] present a three-dimensional drawing style for clustered level graphs that uses two dimensions for drawing the underlying graph, and the third dimension for drawing the cluster tree, see Figure 3.8. In these so-called multilevel drawings of clustered graphs both the cluster tree and the underlying graph are reasonably visible.

Most drawing algorithms for clustered graphs, however, primarily emphasize the underlying graph and draw the cluster tree as nested regions, similar to the nested box inclusion diagrams of Eades, Lin and Lin [72], see Figure 3.5(a). The edges of the cluster tree are not drawn at all, but they are only represented by the nesting of the clusters. The clusters are drawn as simple closed curves that define closed regions of the plane. The region of a cluster contains exactly the clustered drawing of the subgraph induced by its vertices. Regions are nested recursively according to the cluster tree.

Nearly all drawing algorithms for clustered graphs use convex shapes for the cluster regions. The cluster regions should be as simple as possible to improve the readability. If the shape of a cluster region is to complex it is difficult so see which vertices are contained in the cluster and which are not. Thus convex regions like circles or rectangles are preferred. Because we will later combine clustered drawings and level drawings, we use rectangles, which visually match well with the horizontal level lines. This is no strong restriction on the drawings. For example, Eades, Feng and Lin [69] have shown that every clustered planar graph admits a planar straight-line drawing with rectangular cluster regions. See also Section 7.1.4.

Definition 3.11 (Clustered Drawing). *Let $G = (V, E, C, I)$ be a clustered graph. A (nested) clustered drawing $\delta = (\delta_V, \delta_E, \delta_C)$ of G consists of a two-dimensional polyline drawing (δ_V, δ_E) of the graph (V, E) and a function $\delta_C\colon C \to \mathcal{P}(\mathbb{R})$ such that:*

- *The drawing $\delta_C(c)$ of a cluster $c \in C$ is an axially parallel rectangle with non-zero area.*

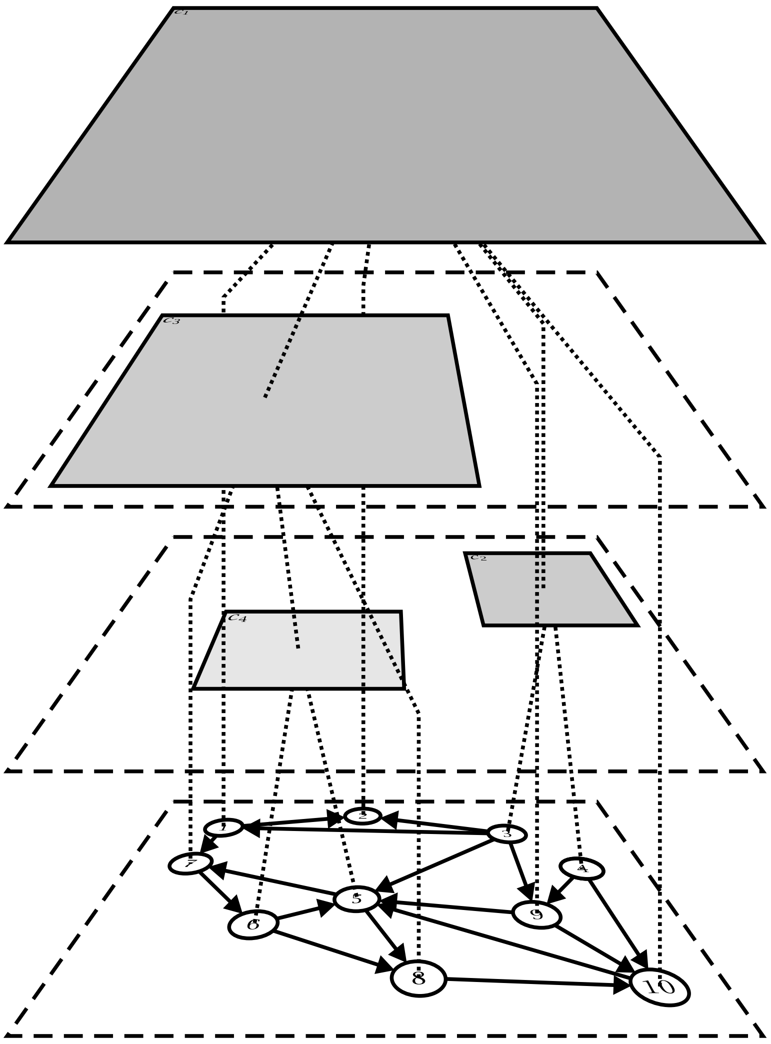


Figure 3.8. A multilevel drawing of the graph in Figure 3.5

- *The drawing of each vertex $v \in V$ is contained within the drawing of its parent cluster $c \in C$: $\delta_V(v) \in \delta_C(c)$.*
- *The drawing of each non-root cluster $c \in C$ is completely contained within the drawing of its parent cluster c': $\delta_C(c) \subseteq \delta_C(')$*
- *If two clusters c and c' are not are not related, i. e., none of them is a descendant of the other one, then their drawings do not overlap: $\delta_C(c) \cap \delta_C(c') = \emptyset$.*

All major techniques for drawing graphs have been extended to nested clustered graphs. There are algorithms for computing planar drawings [65, 66, 68, 69, 87, 173], level drawings [32, 33, 177, 195–197, 214], and also force directed methods for clustered graphs [3, 61, 70]. The choice of the right algorithm for drawing a clustered graph depends on the same reasons as for graphs. The applications that motivated this thesis use directed graphs, and the direction is also very important to be visible in these drawings. In biochemical pathways, there is the time line of reactions, and UML diagrams contain the class hierarchy, which should be drawn top down. In many other applications of clustered graphs the underlying graph is also directed. In these cases, level drawings are the best choice for a visualization, since these are the only drawings that guarantee the same direction for all edges, if possible. We will see in the next chapter, how level graphs and clustered graphs are combined to clustered level graphs.

There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

C. A. R. Hoare

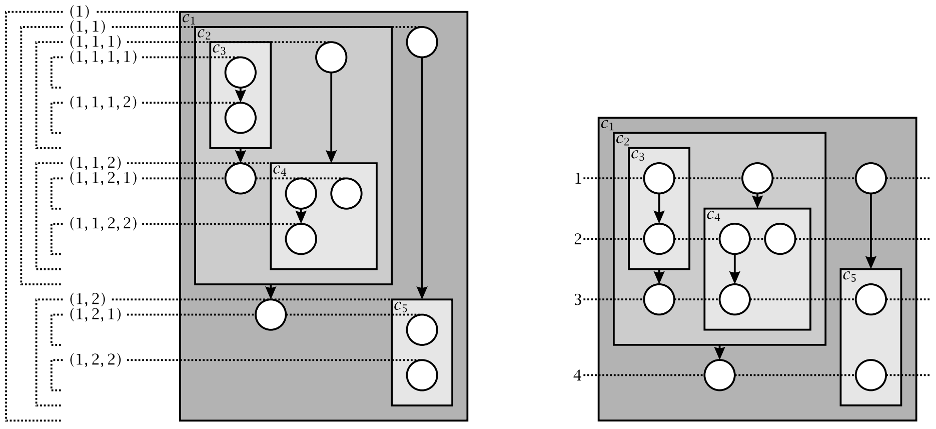
4 Clustered Level Graphs

The main focus of this thesis is on level drawings of directed clustered graphs. Level drawings are well suited for the visualization of directed graphs, because they explicitly emphasize the direction of the edges by letting them point into a uniform direction. The main aesthetic criterion for level graphs is the minimization of edge crossings. Few edge crossings are very important for understandable drawings [182, 183]. Thus the principal topic of the following chapters is the minimization or total prevention of crossings in level drawings of clustered graphs. We will present algorithms for minimizing edge crossings, and for testing whether a drawing without any crossings is possible. In this chapter we start with a detailed analysis, how the concepts of leveling and clustering are combined, and how this is related to crossings.

4.1 Previous Results

Previously, there have been two main concepts for combining leveling and clustering. Sugiyama and Misue [214] present an algorithm for drawing *compound graphs* on horizontal levels. Compound graphs are a generalization of clustered graphs that also allow edges between two clusters or between a cluster and a vertex. A similar algorithm is proposed by Sander [195–197]. Both algorithms extend the classical level drawing algorithm of Sugiyama, Tagawa, and Toda [216]. The vertices are drawn on horizontal levels, and the clusters are drawn as nested rectangles. After partitioning the vertices into levels, permutations of each level are computed for minimizing edge crossings. Finally, horizontal coordinates are assigned to the vertices and clusters.

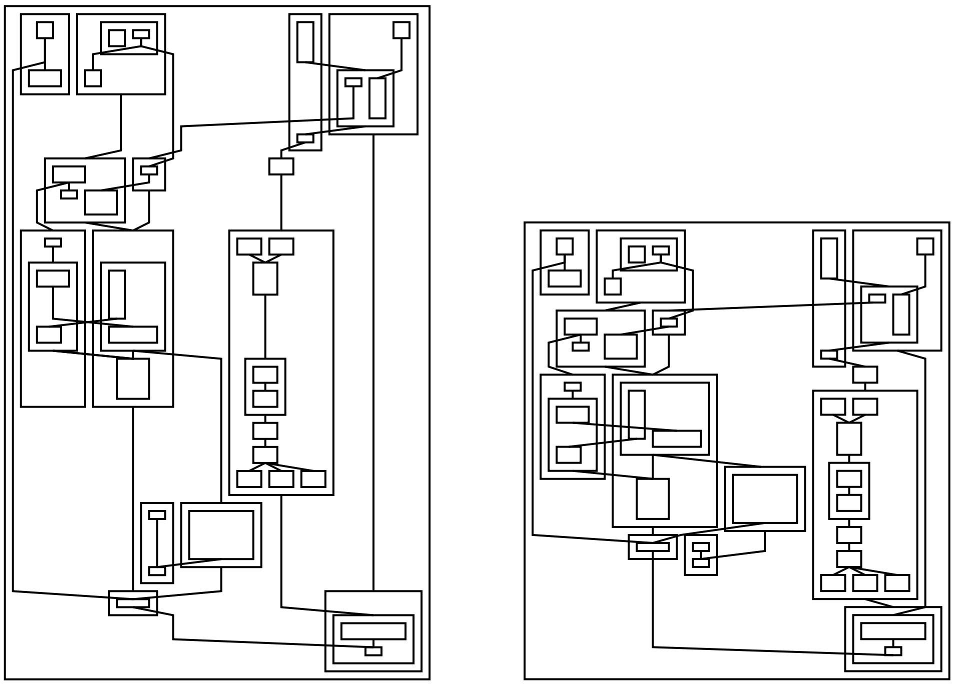
The main difference between the two algorithms is the leveling. In the algorithm of Sugiyama and Misue each vertex and each cluster spans exactly one level. Levels are nested according to the nesting of the clusters. The label of a level does not consist of a single number, but of a sequence of numbers that specifies the nesting of the level. For example in Figure 4.1(a) the levels $(1,2,1)$ and $(1,2,2)$ are nested within level $(1,2)$. If a cluster is drawn on level $(a_1,\dots,a_q)$, $q \in \mathbb{N}$, then all its children are drawn on the levels $(a_1,\dots,a_q,1)$ to $(a_1,\dots,a_q,p)$, $p \in \mathbb{N}$. A cluster c is nested within a cluster c' if and only if the level label of c' is a prefix of the level label of c. The subgraph contained within a cluster is drawn on the levels that are nested within the level of the cluster. In this drawing convention, the leveling is called a *local leveling*. It is a main property of a local leveling that clusters on different levels are not allowed to overlap vertically, even if their horizontal range is disjoint. For example in Figure 4.1(a), the clusters c_2 and c_5 are on the different levels $(1,1)$ and $(1,2)$, respectively. These levels have a disjoint vertical range, because they are not nested. Therefore c_2 and c_5 cannot share common y-coordinates. The intention of this partitioning of vertical space is the prevention of cluster overlap. In the crossing reduction step and in the coordinate assignment step no care has to be taken against overlaps of clusters on different levels.



(a) Local leveling (b) Global leveling

Figure 4.1. Different leveling concepts

Sander's algorithm uses a different drawing convention: In the computed drawings there is one plain set of levels for all vertices and clusters. The levels are not nested, and they are numbered $1,\dots,k$. Each vertex is assigned exactly one level, but clusters are allowed to span multiple levels. Accordingly, this is called a *global leveling*. See Figure 4.1(b) for an example. In a global leveling it is possible that two clusters have a different but overlapping set of levels. For example the cluster c_2 is on the levels $\{1, 2, 3\}$, and cluster c_5 is on the levels $\{3, 4\}$. Both clusters span level 3, i. e., in contrast to local levelings, vertical overlap is possible. This typically results in more compact drawings than with a local leveling. Figure 4.2(a) shows a drawing generated by the algorithm of Sugiyama and Misue. The used leveling clearly can be vertically compacted. Figure 4.2(b) shows the same graph drawn with a global leveling. The large clusters are now allowed to vertically overlap, which leads to a much smaller area. Also the vertices are more uniformly distributed. Because of this, we consider global levelings as superior to local levelings. However, global levelings are often computationally more expensive than local levelings, because more crossings between edges and level lines have to be represented by dummy vertices.



(a) Local leveling (b) Global leveling

Figure 4.2. Comparing the compactness of different leveling concepts

There is also an algorithm for level drawings of clustered graphs by Eades, Feng and Lin [69], which draws edges as straight lines, but generates drawings with up to exponential area. The leveling used by this algorithm gives each vertex its own level, such that the vertices of a cluster are on consecutive levels, see Figure 4.3. Clusters do not overlap vertically, so this very similar to a local leveling, while formally, it is a global leveling.

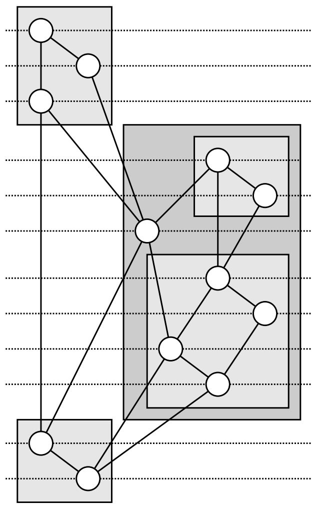


Figure 4.3. Straight-line drawings by Eades, Feng, and Lin [69]

4.2 Definitions and Basic Properties

Because of the more compact drawings and the better vertex distribution, from now on we focus on global levelings. A clustered graph with a global leveling is called a clustered level graph and is defined as follows:

Definition 4.1 (Clustered Level Graph). *A* clustered k-level graph $G = (V, E, C, I, \phi)$ *is a k-level graph (V, E, ϕ) with a cluster tree $\Gamma = (V \dot\cup C, I)$, $V \cap C = \emptyset$.*

In this definition only the vertices are assigned to levels. The levels of the clusters result directly from the levels of the contained vertices and clusters. A cluster starts on the minimum level of a contained vertex and ends at the maximum level.

Definition 4.2 (Minimum/Maximum Level). *In a clustered k-level graph $G = (V, E, C, I, \phi)$, the* minimum level $\phi_{\min}(c)$ *and the* maximum level $\phi_{\max}(c)$ *of a cluster $c \in C$ are defined as the minimum and maximum level of its contained vertices, respectively:*

$$\phi_{\min}(c) = \min_{v \in V_c} \phi(v) \qquad\qquad \phi_{\max}(c) = \max_{v \in V_c} \phi(v).$$

The cluster c is said to span the levels between $\phi_{\min}(c)$ and $\phi_{\max}(c)$. The set of spanned levels is denoted by

$$\Phi(c) = \{\, i \in \mathbb{N} \mid \phi_{\min}(c) \le i \le \phi_{\max}(c) \,\}$$

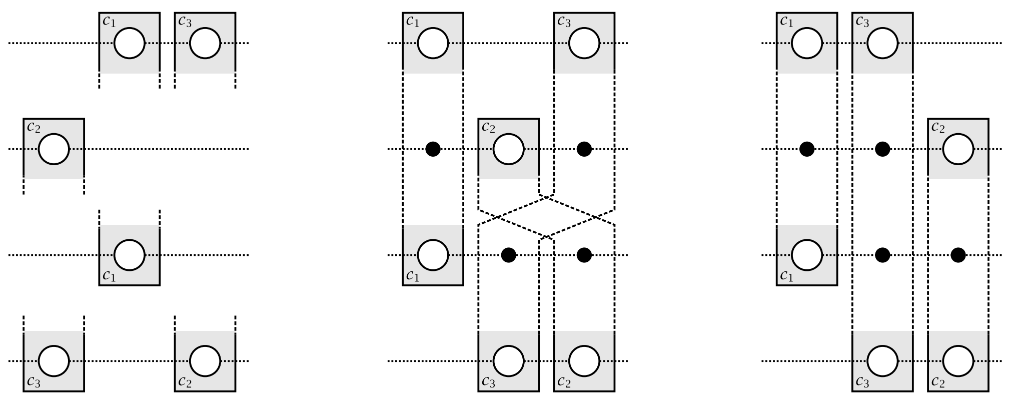
For a vertex $v \in V$ the minimum and maximum level are defined as $\phi_{\min}(v) = \phi_{\max}(v) = \phi(v)$, $\Phi(v) = \{\phi(v)\}$.

4.2.1 Proper Clustered Level Graphs

After computing a leveling for a clustered graph, some edges may span more than one level. This is analogous to drawings of level graphs, see Section 3.3.2. Before in the next step edge crossings can be minimized, long span edges must be split into segments that span only a single level. This is necessary, because otherwise the exact routing of the edges cannot be defined. At each crossing between a level line and an edge it may be necessary to introduce an edge bend. These bends are represented by dummy vertices.

In a clustered level graph not only edges can span multiple levels, but also clusters. Because of this, dummy vertices are not only needed for the edges, but also for the clusters. As long as a cluster contains a vertex on each spanned level, the “routing” of its region is well defined. But it is also possible that a cluster in a clustered level graph spans a level on which it does not contain any vertex. This can lead to problems, because on such a level the exact region of the cluster is not defined, and it may be unclear if a cluster is crossed by edges or by other clusters, see Figure 4.4(a). Figures 4.4(b) and 4.4(c) show two drawings of the same graph. Although the relative position of the original vertices on each level has not been changed, two clusters cross in one of the drawings but not in the other. These problems are very similar to long span edges. While long span edges are split into proper edge segments, we require clusters to contain a vertex on each spanned level. In analogy to proper level graphs, we call this a proper clustered level graph:

Definition 4.3 (Proper Clustered Level Graph). *A clustered k-level graph $G = (V, E, C, I, \phi)$ is* proper *if all edges are proper and each cluster $c \in C$ contains a vertex on any spanned level:* $\forall i \in \Phi(c) : V_c \cap V_i \neq \emptyset$.



(a) Unknown cluster crossings (b) With a cluster crossing (c) Without a cluster crossing

Figure 4.4. In non-proper graphs the region of clusters is not fully defined

From now on, we will only consider proper clustered level graphs. Any clustered level graph can be made proper by introducing dummy vertices for long span edges and for clusters with empty levels. In contrast to level graphs there may be several alternatives for making an edge of a clustered level graph proper. In particular, there is typically more than one choice for the parent of the dummy vertices in the cluster tree, see Figure 4.5. For a readable drawing, long span edges should enter or leave a cluster at most once, and they should not touch unrelated clusters. Therefore the parents of the dummy vertices should follow a simple path in the cluster tree. Different strategies lead to a trade-off between symmetry and the number of dummy vertices. Routing the edges outside of a cluster may lead to an overall higher number of dummy vertices, as shown in Figure 4.5(c), but the routing is more symmetric. Such a routing is generated by inserting the dummy vertices for a long span edge (s, t) as children of the lowest common ancestor of s and t in the cluster tree. For our purposes any strategy for inserting the dummy vertices can be chosen, as long as no edge enters or leaves a cluster twice.

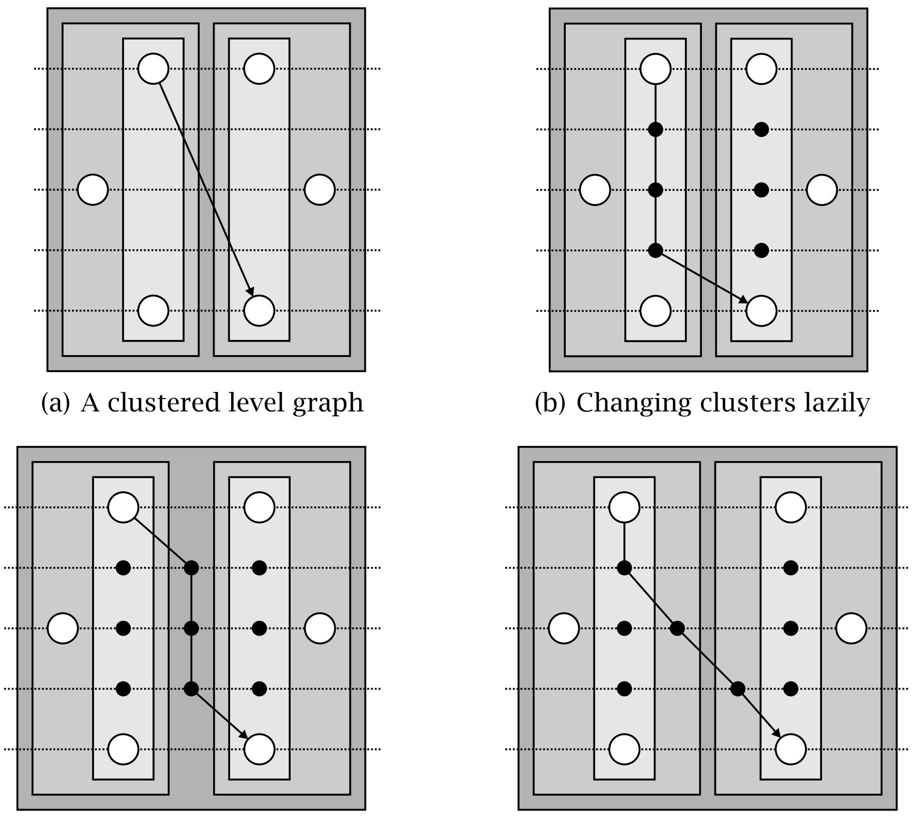

(a) A clustered level graph
(b) Changing clusters lazily
(c) Lowest common ancestor
(d) Mixed strategy

Figure 4.5. Making a clustered level graph proper

For the dummy vertices that are inserted for clusters there is no such choice. For each empty spanned level a dummy vertex is introduced as a direct child of the cluster. A dummy vertex is not necessary, however, if on the same level another dummy vertex has already been inserted into the cluster for a nested long span edge or cluster. Because of this, fewer dummy vertices are generated by first considering long span edges, and then traversing the cluster tree bottom up for inserting dummy vertices for the clusters.

It is clear that making a clustered level graph proper requires up to a quadratic number of dummy vertices, because for each crossing between a level line and a cluster or a long span edge, a dummy vertex may have to be inserted. This is analogous to level graphs and increases the running time of the remaining parts of the algorithm. Therefore the number of dummy vertices should be taken into account when computing the leveling. Here, an extended version of the dummy vertex minimization algorithm by Gansner et al. [100] can be used.

4.2.2 Level Cluster Trees

The leveling and clustering of a graph are in a sense orthogonal concepts. Since the layout of a graph is constrained by both, we want to analyse where the different restrictions influence each other. The ordering of the vertices on a single level is constrained by the clustering, but not the whole cluster tree is relevant for all levels. Therefore, we introduce the level cluster tree, which is that part of the cluster tree that is relevant for a given level. We will see later that thoughtfully ordering the children of clusters in the level cluster tree leads to drawings with few crossings.

Definition 4.4 (Level Cluster Tree). *Let* $\Gamma = (V \cup C, I)$ *be the cluster tree of a clustered* k*-level graph* $G = (V, E, C, I, \phi)$*. The* i-th level cluster tree Γ_i *is the subgraph of* Γ *induced by all vertices and clusters* $x \in V \cup C$ *spanning the* i*-th level,* $i \in \Phi(x)$*.*

For example, the clustered level graph in Figure 4.6(a) has the cluster tree shown in Figure 4.6(b). Since there are two levels, there are also two level cluster trees, one for each level, see Figures 4.6(c) and 4.6(e).

As this example shows, the level cluster trees may contain clusters with a single child, such as cluster B in Γ_1. This implies that the size of a level cluster tree may be greater than linear in the size of the level. Actually, the accumulated size of the level cluster trees can be quadratic in the number of vertices. Each intersection of a cluster with a level line leads to a new node in the corresponding level cluster tree. Figure 4.7 shows how clustered level graphs can be constructed with $\Omega(|V|)$ clusters that span $\Omega(|V|)$ levels, each.

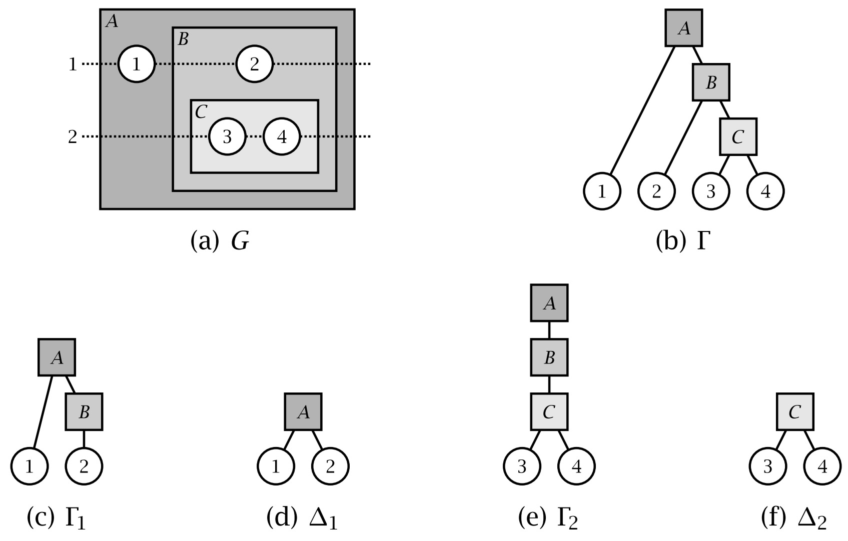


Figure 4.6. Defining level cluster trees

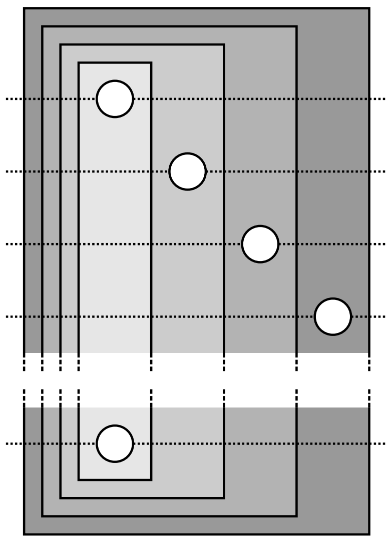


Figure 4.7. A clustered level graph where all clusters span all levels

Also, single-childed clusters do not carry much information, because there is only one permutation of the single child. So we further reduce the size of the level cluster tree by eliminating single-childed clusters:

Definition 4.5 (Contracted Level Cluster Tree). *Let Γ_i be the i-th level cluster tree of a clustered k-level graph. The i-*th contracted level cluster tree *Δ_i is constructed from Γ_i by removing each single-childed cluster and connecting the child directly to the grandparent. If no grandparent exists (at the root), no connection is made.*

The contracted level cluster trees are significantly smaller than the full level cluster trees. While the overall size of the level cluster trees can be quadratic, the size of the contracted level cluster trees is linear, as the following lemma shows:

Lemma 4.1. *Let G be a clustered k-level graph. Then the accumulated size of the contracted level cluster trees $(\Delta_i)_{1\le i\le k}$ is linear in the size of G.*

Proof. In a contracted level cluster tree every inner node has at least two children. Thus, its size is smaller than twice the number of leaves, $|\Delta_i| < 2 \cdot |V_i|$, and the correctness follows immediately:

$$\sum_{1\le i\le k} |\Delta_i| < \sum_{1\le i\le k} 2 \cdot |V_i| = 2 \cdot |V|.$$

□

The sequence of contracted level cluster trees can be constructed easily by pruning the level cluster trees. An alternative is to traverse the cluster tree bottom up and to simultaneously insert each vertex or cluster in the corresponding contracted level tree(s). Both approaches lead to quadratic time algorithms. Although both the cluster tree and the sequence of contracted level cluster trees are linear in the number of vertices, no linear time algorithm is known, yet.

4.3 Drawing Conventions

Drawings of clustered level graphs are the straight-forward combination of level drawings and clustered drawings. As in level graphs, vertices and edge bends are drawn on horizontal levels, and the clusters are nested as in clustered graphs:

Definition 4.6 (Clustered Level Drawing). *Let $G = (V, E, C, I, \phi)$ be a clustered level graph. A clustered drawing $\delta = (\delta_V, \delta_E, \delta_C)$ of (V, E, C, I) is a* clustered level drawing *of G if (δ_V, δ_E) is a level drawing of (V, E, ϕ).*

It is not necessarily clear, whether this definition is well-formed. Drawings of clustered level graphs have been defined as drawings with rectangular cluster regions. So this definition requires rectangular cluster regions as well. These are desirable for readable drawings, but they also restrict the number of possible drawings. As we will see, however, for any leveling of a clustered level graph there is a clustered level drawing. Also, as already mentioned in Section 3.4.2, every clustered planar graph admits a planar straight-line drawing with rectangular cluster regions. This is also true for clustered level graphs [69], so the restrictions imposed by rectangular clusters are manageable. They may, however, increase the number of crossings or edge bends.

Before drawing a clustered level graph, an embedding must be computed. An embedding of a clustered level graph is very similar to a level embedding of the underlying level graph, but there are some additional restrictions for the clusters. For some vertex permutations, it is not possible to draw the graph without overlapping clusters. In the following we will analyze these restrictions in detail. We will also analyze how the embedding affects edge crossings and cluster/edge crossings. This leads to four different restrictions for the underlying level embedding, which we will define and analyze in the rest of this chapter, see Table 4.1 on page 47. We will show that a valid clustered level embedding can be characterized by the cluster/level and cluster/cluster restrictions. If additionally the edge/edge and cluster/edge restrictions are satisfied, then there are no crossings.

As the embedding defines the relative position of the vertices on a level, it also determines the shape of the clusters and whether they overlap or not. In the same way as the leveling determines the vertical range of a cluster, the embedding defines its horizontal ranges. In analogy to the definition of the minimum and maximum level of a cluster subject to the levels of the contained vertices, an embedding defines the minimum and maximum level position of a cluster subject to the level positions of the contained vertices:

Definition 4.7 (Minimum/Maximum Level Position). *Let $G = (V, E, C, I, \phi)$ be a clustered level graph and π a level embedding of (V, E, ϕ). The minimum resp. maximum position of a cluster $c \in C$ on level i is defined by*

$$\pi_{\min}(c,i) = \min_{v \in V_c \cap V_i} \pi(v) \ , \qquad \pi_{\max}(c,i) = \max_{v \in V_c \cap V_i} \pi(v) \ .$$

The horizontal range *of a cluster is denoted by*

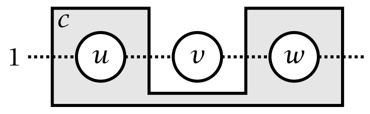
$$\Pi(c,i) = \{\, j \in \mathbb{N} \mid \pi_{\min}(c,i) \le j \text{ and } j \le \pi_{\max}(c,i) \,\} \ .$$

If $V_c \cap V_i = \emptyset$, then $\pi_{\min}(c,i) = \infty$, $\pi_{\max}(c,i) = -\infty$, and $\Pi(c,i) = \emptyset$.

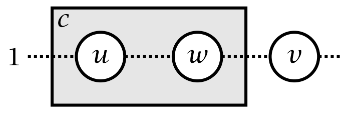
Per definition the relative position of every vertex is within the range of an enclosing cluster and the range of a cluster is nested within the range of any ancestor. For a valid clustered drawing with rectangular cluster regions the reverse direction must also be true:

Definition 4.8 (Cluster/Level Restriction). *Let $G = (V, E, C, I, \phi)$ be a clustered level graph. A level embedding π of (V, E, ϕ) satisfies the* cluster/level restriction *if the horizontal range of any cluster $c \in C$ includes only contained vertices: $\forall v \notin V_c \colon \pi(v) \notin \Pi(c, \phi(v))$.*

Example 4.1. *Figure 4.8 shows two level embeddings of the same clustered level graph. In the first level embedding the cluster/level restriction is violated, because the level position of vertex v is within the horizontal range of the cluster. In the second embedding the cluster/level restriction is satisfied, and a cluster drawing with rectangular clusters is possible.*



(a) The restriction is violated: $\pi(v) = 2 \in \{1, 2, 3\} = \Pi(c, 1)$.



(b) The restriction is satisfied: $\pi(v) = 3 \notin \{1, 2\} = \Pi(c, 1)$.

Figure 4.8. The cluster/level restriction

While the cluster/level restriction assures that clusters can be drawn as rectangles and avoids overlap of clusters on a single level, two clusters with more than one common level may still overlap. This leads to a second condition that is necessary for the existence of a clustered level drawing, which deals with the interference of clusters across multiple levels:

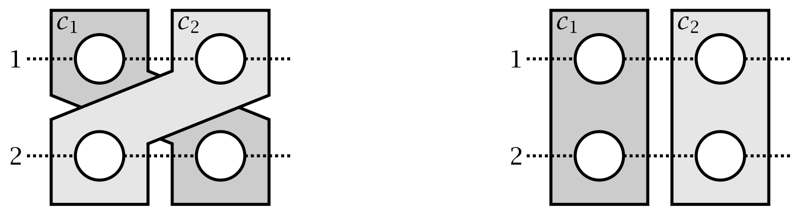
Definition 4.9 (Cluster/Cluster Restriction). *Let $G = (V, E, C, I, \phi)$ be a proper clustered level graph. A level embedding π of (V, E, ϕ) satisfies the* cluster/cluster restriction *if for any two clusters $c_1, c_2 \in C$ that are not nested, c_1 lies to the left of c_2,*

$$\forall i \in \Phi(c_1) \cap \Phi(c_2) \colon \pi_{\max}(c_1, i) < \pi_{\min}(c_2, i) \ ,$$

or c_2 lies to the left of c_1,

$$\forall i \in \Phi(c_1) \cap \Phi(c_2) \colon \pi_{\max}(c_2, i) < \pi_{\min}(c_1, i) \ .$$

Example 4.2. *Figure 4.9 shows two level embeddings of the same clustered level graph. In the first level embedding the cluster/cluster restriction is violated, because the order of the clusters on level 1 is different to the order on level 2. In the second embedding the cluster/cluster restriction is satisfied and a cluster drawing with disjoint cluster regions is possible.*



(a) The restriction is violated. (b) The restriction is satisfied.

Figure 4.9. The cluster/cluster restriction

While the necessity of the cluster/level and cluster/cluster restrictions is obvious, the following lemma shows that both restrictions together are also a sufficient condition and therefore a characterization for the existence of a rectangular clustered drawing. Such a drawing can be generated by using the algorithm of Sander [195–197]. For proving the following lemma, however, we use a much more simple algorithm, which ignores most aesthetic criteria. It needs quadratic area, which is optimal. Clustered level graphs like that in Figure 4.7 cannot be drawn with rectangular clusters on less than quadratic area.

Lemma 4.2. *For any level embedding of a clustered level graph that satisfies the cluster/level and cluster/cluster restrictions there is a clustered level drawing with rectangular cluster regions.*

Proof. Consider a clustered level graph $G = (V, E, C, I, \phi)$ with a level embedding π that satisfies the cluster/level and cluster/cluster restrictions. We describe a simple algorithm that generates a clustered drawing with rectangular cluster regions: The vertical coordinates of the drawing are given by the leveling. Horizontal coordinates are computed as follows: For the children of the root r of the cluster tree Γ, the "is left of" graph $G' = (V', E')$ is defined as:

$$\begin{aligned} V' &= \mathrm{succ}_\Gamma(r) \ , \\ E' &= \{ (u, v) \in V' \times V' \mid \exists i \colon \pi_{\max}(u, i) < \pi_{\min}(v, i) \} \ . \end{aligned}$$

Because of the cluster/cluster restriction, G' is acyclic, and a linear arrangement of V' can be found by topological sorting. The topsort numbers are used as horizontal coordinates for these vertices and clusters. Clusters are assigned a width of $\frac{1}{2}$. Then any two clusters have disjoint horizontal ranges, and therefore do not overlap. Because of the cluster/level restriction, the contents of a cluster can be drawn independently and then scaled down to fit into the region of the enclosing cluster. The cluster tree is traversed top down and the children of each cluster c are drawn into the region of c as described above. □

Now that we have seen, that any embedding that satisfies the cluster/level and cluster/cluster restrictions can be drawn with rectangular clusters, it remains to show, that such embeddings exist.

Lemma 4.3. *For any clustered level graph there is an embedding that simultaneously satisfies the cluster/level restriction and the cluster/cluster restriction.*

Proof. Let $G = (V, E, C, I, \phi)$ be a clustered level graph. A suitable embedding for G is constructed using a preorder[1] traversal of the cluster tree. Every cluster and vertex is assigned a preorder number $p(v)$. Then the vertices on each level are sorted by $p(v)$. This defines an embedding π such that $\pi(u) < \pi(v)$ if and only if $p(u) < p(v)$. In the preorder traversal every vertex that is not contained in a cluster c is traversed either before or after all vertices in c. Thus, π satisfies the cluster/level restriction. It satisfies the cluster/cluster restriction, because the preorder traversal is independent of the levels. Therefore, if two clusters c_1 and c_2 are not nested, all vertices contained in c_1 have smaller preorder numbers than all vertices in c_2 or vice versa. □

In summary, rectangular clustered drawings are possible if and only if the above restrictions are satisfied, and such embeddings always exist. This justifies the following definition, which requires a clustered level embedding to satisfy the restrictions:

Definition 4.10 (Clustered Level Embedding). *Let $G = (V, E, C, I, \phi)$ be a proper clustered level graph. A level embedding π of (V, E, ϕ) is a* clustered level embedding *of G if it satisfies the cluster/level and cluster/cluster restrictions.*

4.4 Characterizing Crossings

Up to now, we have only considered vertices and clusters, and have ignored the edges. Because crossings decrease the readability of a drawing, we obviously cannot ignore the edges when computing an embedding. In clustered level graphs, there are two kinds of crossings: edge crossings and cluster/edge crossings.

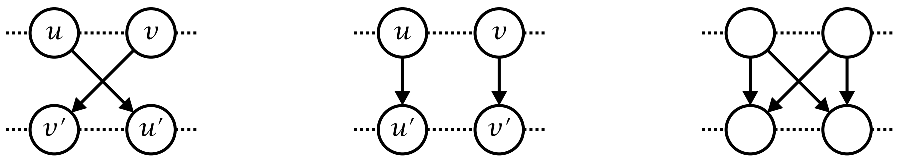
[1]Because the algorithm is irrespective of the numbers assigned to clusters, a postorder traversal can be used as well.

4.4.1 Edge Crossings

Edge crossings in clustered level graphs are very similar to edge crossings in level graphs, see Section. 3.3.3. Embeddings that do not induce edge crossings are characterized by the following restriction:

Definition 4.11 (Edge/Edge Restriction). *A clustered level embedding π for a proper clustered level graph $G = (V, E, C, I, \phi)$ satisfies the* edge/edge restriction *if there are no edge crossings, i. e., if there is no pair of edges $(u, v) \in E$ and $(u', v') \in E$ with $\phi(u) = \phi(u')$ and $(\pi(u') - \pi(u)) \cdot (\pi(v') - \pi(v)) < 0$.*

Example 4.3. *Figure 4.10(a) shows a (clustered) level embedding that violates the edge/edge restriction, because the vertices u' and v' have a different relative order than the vertices u and v. Figure 4.10(b) shows an embedding of the same graph with the restriction satisfied.*



(a) The restriction is violated.

(b) The restriction is satisfied.

(c) A (clustered) level graph without a satisfying embedding.

Figure 4.10. The edge/edge restriction

Minimizing crossings is a computationally hard problem. First, we observe that not every (clustered) level graph has an embedding without crossings, see Figure 4.10(c). Further, we know from level graphs that minimizing the number of crossings in a level embedding is NP-hard [102]. This is also true for clustered level graphs, because per definition every level graph is also a clustered level graph.[2]

4.4.2 Cluster/Edge Crossings

In addition to edges crossing each other, we also want to avoid edges crossing a cluster, i. e., edges that cross the boundary of a cluster region more than once. If cluster regions are rectangular, this can be enforced by a simple restriction:

[2]Formally, every level graph $G = (V, E, \phi)$ has a corresponding clustered level graph $G' = (V, E, C, I, \phi)$ with a single cluster $C = \{c\}$ that contains all vertices $I = \{(c, v) \mid v \in V\}$. This is only a notational difference, however.

Definition 4.12 (Cluster/Edge Restriction). *Let $G = (V, E, C, I, \phi)$ be a proper clustered level graph. In a clustered level embedding π of G, an edge $e = (u, v) \in E$ crosses a cluster $c \in C$ if $\phi(u) \geq \phi_{\min}(c)$ and $\phi(v) \leq \phi_{\max}(c)$ and*

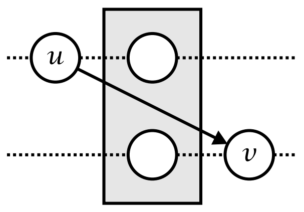
$$\pi(u) > \pi_{\max}(c, \phi(u)) \wedge \pi(v) < \pi_{\min}(c, \phi(v))$$

or

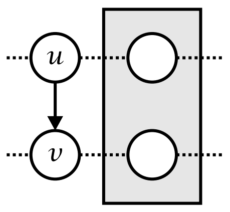
$$\pi(u) < \pi_{\min}(c, \phi(u)) \wedge \pi(v) > \pi_{\max}(c, \phi(v)) \ .$$

π satisfies the cluster/edge restriction *if there are no cluster/edge crossings.*

Example 4.4. *Figure 4.11(a) shows a clustered level embedding that violates the cluster/edge restriction, because vertex u is on the other side of the cluster than v. Figure 4.11(b) shows an embedding of the same graph with the restriction satisfied.*



(a) The restriction is violated.



(b) The restriction is satisfied.

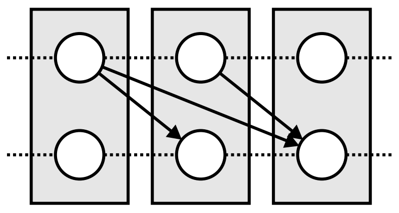

(c) A clustered level graph without a satisfying embedding.

Figure 4.11. The cluster/edge restriction

Similar to the edge/edge restriction, the cluster/edge restriction cannot be satisfied for all clustered level graphs. The graph in Figure 4.11(c) has no such embedding. Because there is an edge between any pair of clusters, the cluster in the middle is always crossed by an edge that connects the other two clusters. See Chapter 7 for a further analysis, when drawings without crossings are possible. Minimizing the number of cluster/edge crossings is NP-hard:

Theorem 4.1. *Cluster/edge crossing reduction is NP-complete:*

Instance: *A clustered level graph $G = (V, E, C, I, \phi)$ and a positive integer K.*

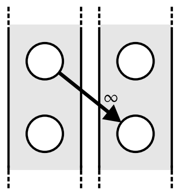
Question: *Is there a clustered level embedding of G with at most K cluster/edge crossings?*

Proof. (Sketch) It is obvious that the problem is in NP. The challenging part of the proof is showing the NP-hardness. We use a reduction from one-sided two-level crossing reduction, which is known to be NP-hard [78, 80]. The input is a two-level graph $G = (V, E, \phi)$, a permutation $u_1, \dots, u_{|V_1|}$ of the first level V_1, and an integer K. The permutation of V_1 is fixed, while the second level V_2 may be reordered. The question is, whether G has a level embedding π with at most K edge crossings and the first level ordered according to the given permutation: $\forall i \in \{1, \dots, |V_1|\}: \pi(u_i) = i$. For the reduction we construct a clustered level graph $G' = (V', E', C', I', \phi')$ that has a clustered level embedding with at most K cluster/edge-crossings if and only if there is a solution for the one-sided two-level crossing reduction problem.

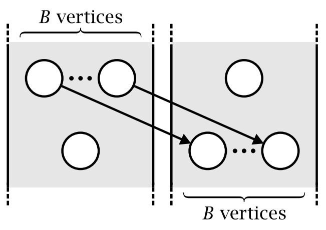
The proof consists of three parts. We first describe the reduction, which is essentially the construction of G'. The construction is illustrated in Figure 4.14. Figure 4.15 shows the construction for an example graph. In the second part, we prove the correctness of the reduction.

Before describing the construction of G' in detail, we discuss some of the used construction techniques. For controlling possible embeddings of the constructed clustered level graph, we use so-called *∞-edges*. An ∞-edge consists of a large number of parallel edges and multiple source vertices and target vertices, see Figure 4.12. In illustrations, these edges are bold and marked with ∞. The actual number B of edges that are represented by an ∞-edge depends on the input level graph and is large enough to prevent the edges from crossing any cluster at all. For any level graph, $B = |V_1| \cdot |V_2| + 1$ is sufficiently large, because any level embedding of G has fewer edge crossings. Thus, if $K \geq |V_1| \cdot |V_2| + 1$, there is always an solution for the one-sided two-level crossing reduction problem, and therefore embeddings with so many crossings never need to be considered.

Another technique is connecting clusters to *blocks*. As illustrated in Figure 4.13 multiple clusters that span at least two adjacent levels are connected by a sequence of ∞-edges. Because of the edges, no other cluster can be positioned between two of the connected clusters. In particular the connected clusters will be consecutive, and they will be positioned in the given order or in the reverse order.



(a) An ∞-edge…



(b) …and its meaning

Figure 4.12. ∞-edges

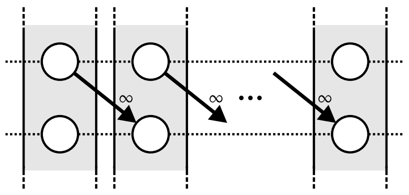
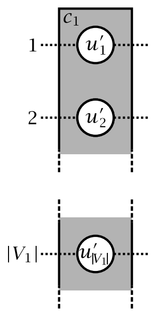
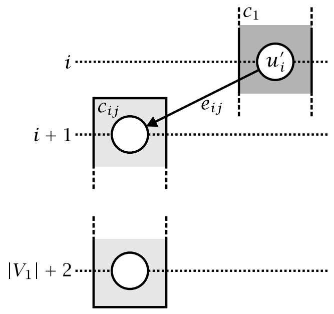


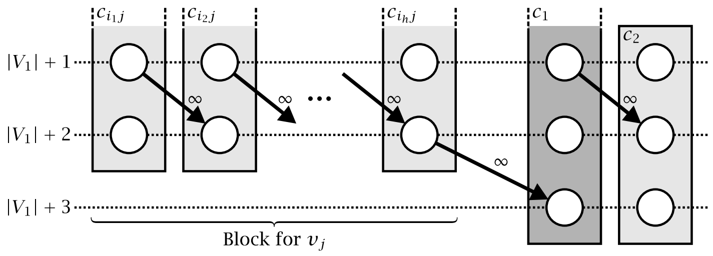
Figure 4.13. Using ∞-edges to build blocks



(a) Replacement for the first level vertices.



(b) Replacement for an edge.



(c) Replacement for a vertex on the second level.

Figure 4.14. Reducing edge crossing minimization to cluster/edge-crossing minimization

In detail, G' is constructed on $|V_1| + 3$ levels. The replacement for the vertices $u_1, \dots, u_{|V_1|}$ of the first level in G is shown in Figure 4.14(a). Each vertex u_i is represented by a vertex u'_i on level $\phi'(u'_i) = i$. The leveling of these vertices directly corresponds to the given permutation of the first level in G. The vertices $u'_1, \dots, u'_n$ are contained in a common cluster c_1, which also contains some additional vertices, whose purpose shall be explained later.

Each edge (u_i, v_j) of G is represented in G' by a proper edge e_{ij} and a cluster c_{ij} as shown in Figure 4.14(b). The edge e_{ij} connects the vertex u'_i with the topmost vertex contained in c_{ij} on level $i + 1$. The cluster c_{ij} spans the levels $i + 1, \dots, |V_1| + 2$ and is filled with one vertex per level, accordingly.

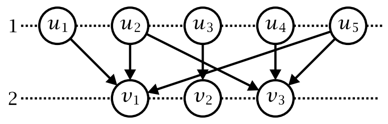
The vertices of the second level in G are not represented directly by vertices in G'. It is only ensured that the clusters that represent endpoints of the incoming edges of a vertex are consecutive. If vertex v_j has h incoming edges $e_{i_1 j}, \dots, e_{i_h j}$, the corresponding clusters $c_{i_1 j}, \dots, c_{i_h j}$ are connected to a block as shown in Figure 4.14(c). For the correctness of the proof, it is necessary, that the clusters are connected in ascending order $i_1 \leq i_2 \leq \cdots \leq i_h$, i. e., according to descending span of the clusters.

The construction is completed by some glue, namely another cluster c_2 and some more ∞-edges. The cluster c_2 spans the levels $|V_1| + 1, \dots, |V_1| + 3$, and is connected to c_1 by an ∞-edge as shown in Figure 4.14(c). Also, the blocks representing a second level vertex are connected by an ∞-edge to an vertex in c_1 on level $|V_1| + 3$. This ensures that c_1 is positioned between c_2 and all other clusters.

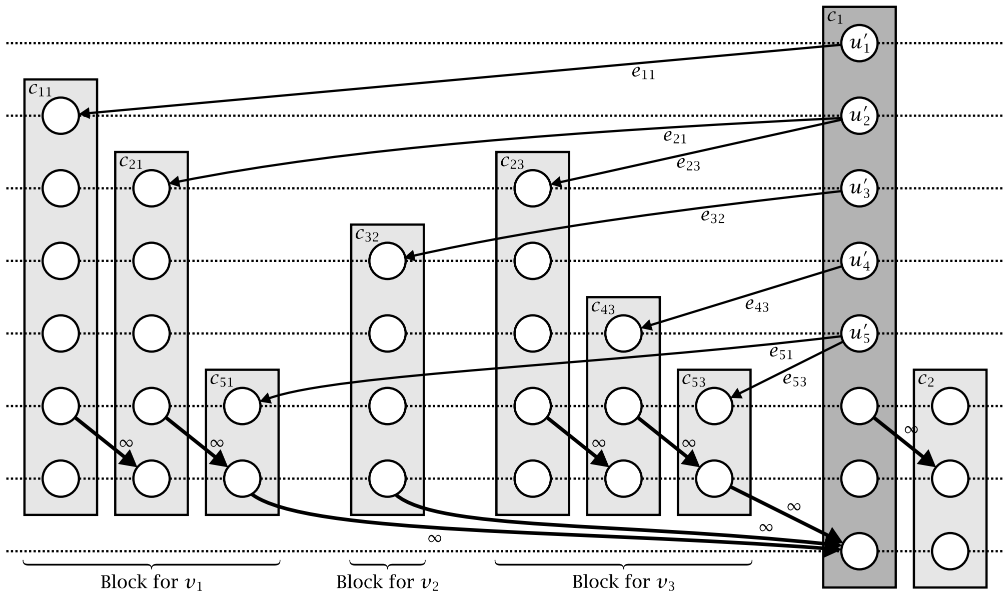
For the correctness of the reduction, we will show two properties: First, there is a one-to-one correspondence between the following sets of embeddings: (i) the embeddings of G with the given permutation of the first level and (ii) the clustered level embeddings of G' with no crossings between an ∞-edge and a cluster. Second, the level embedding of G has the same number of edge crossings as the number of cluster/edge crossings in the corresponding embedding of G'.

We have already seen that c_1 is always positioned between c_2 and all other clusters. We assume w. l. o. g. that c_2 is to the right of c_1 and all other clusters are to the left. The other case is symmetric. We have also seen that in an embedding of G' the clusters representing incoming edges of a second level vertex are consecutive, while any permutation of the blocks is possible. This is equivalent to permuting the vertices on the second level of G. The permutation of $u_1, \dots, u_{|V_1|}$ is fixed by the leveling. This is equivalent to the fixed permutation of the first level of G.

The clusters in G' representing incoming edges of a second level vertex have only been constrained to be consecutive, but it has not been assured, that they



(a) An example graph G.

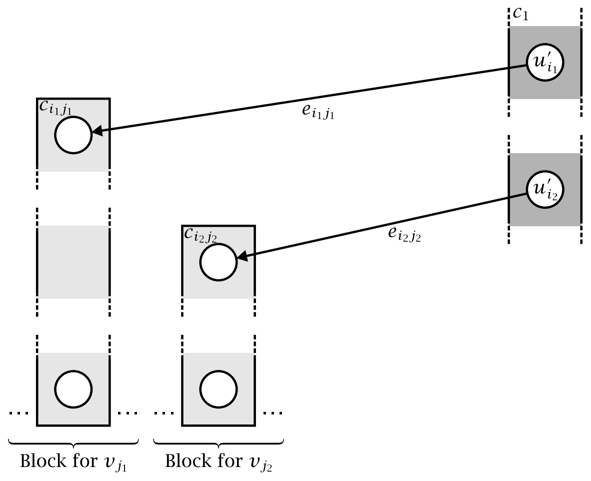


(b) The transformation G'.

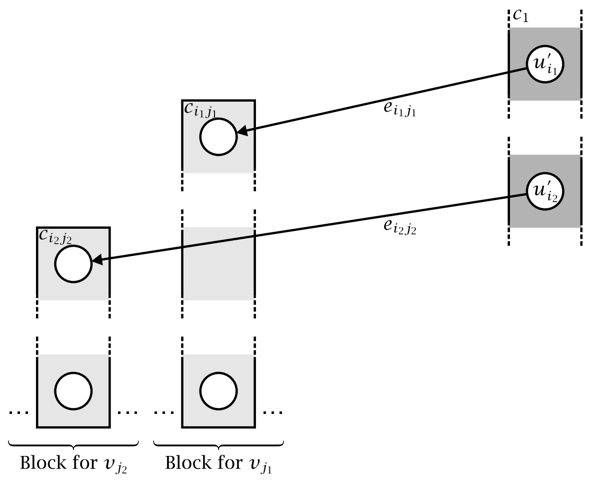
Figure 4.15. An example for the reduction

are ordered from left to right according to descending span. It would also be possible for a whole block to be flipped and ordered in descending order. However, it is easy to see that such an embedding never has fewer crossings than the same embedding with ascending order.

Two edges with different source and target vertices cross in G if and only if the corresponding clusters and edges in G' cross. If two edges cross in the embedding of G, their source and target vertices have different order on both levels. This is equivalent to a different order of blocks in the embedding of G'. Figure 4.16 illustrates the crossing of two edges.



(a) Non-crossing edges.



(b) Crossing edges.

Figure 4.16. Crossing vs. non-crossing edges

Two edges with a common end vertex never cross in G, and their replacement in G' does not induce a crossing either. If they have a common source vertex, both clusters have the same height and therefore cannot cross the other's edge. If the target vertex is common, there is no crossing either, because the clusters have ben ordered according descending height.

Because two edges in an level embedding of G cross if and only if in the corresponding cluster level embedding of G' their replacements induce a cluster/edge crossing, embeddings with a minimum number of crossings correspond to each other. □

4.4.3 Clustered Level Planarity

Reducing crossings also leads to the question, whether a drawing without any crossings is possible. Analogously to defining a level planar embedding as a level embedding without edge crossings, we define clustered level planar embeddings in a straight-forward way:

Definition 4.13 (Clustered Level Planar Embedding). *Let $G = (V, E, C, I, \phi)$ be a proper clustered level graph. A clustered level embedding π of (V, E, ϕ) is a* clustered level planar embedding *of G if it fulfills the edge/edge and cluster/edge restrictions. A clustered level graph is* clustered level planar *if there exists a clustered level planar embedding of it.*

This completes our analysis of needed and desirable properties of embeddings. Table 4.1 gives an overview of the considered embedding restrictions and summarizes the presented results. In the following chapters the problem of avoid crossings is further investigated: Chapter 5 describes heuristics for clustered crossing minimization, and Chapter 7 further analyses clustered level planarity.

	restriction	always satisfiable	crossing minimization
clustered level embedding; clustered level planar embedding	cluster/level	yes, see Lemma 4.3	not applicable
	cluster/cluster	yes, see Lemma 4.3	not applicable
	edge/edge	no, Figure 4.10(c)	NP-complete, see [102]
	cluster/edge	no, Figure 4.11(c)	NP-complete, see Theorem 4.1

Table 4.1. Overview of embedding restrictions

It is best to do things systematically, since we are only human, and disorder is our worst enemy.

Hesiod

5 Clustered Crossing Reduction

This chapter investigates the crossing reduction problem for clustered level graphs. Given a clustered level graph, we want to find a clustered level embedding with few crossings. That is, we compute a level embedding that satisfies the cluster/level and cluster/cluster restrictions while it has as few violations of the edge/edge and cluster/edge restrictions as possible. We have already seen that this problem is NP-hard. Thus heuristics are used for the efficient computation of an embedding.

We do not consider the computation of a drawing for a clustered level graph. We have already seen in Lemma 4.2 that for any clustered level embedding of a clustered level graph there is a drawing with rectangular cluster regions. Refer to [195–197] for a drawing algorithm.

5.1 Previous Results

Before presenting our new results, we first review two previous heuristics for crossing reduction in clustered graphs. Both algorithms extend crossing reduction heuristics that were originally developed for level graphs. They can be implemented on top of any traditional crossing reduction heuristic, which is an important property. As seen in chapter 3.3.3, there are many such heuristics, which are adapted for different classes of level graphs. There is no clear winner, and the choice depends on various parameters. For example, in dense graphs most heuristics give similar results, and therefore a simple and fast heuristic like the barycenter heuristic is the best choice. In sparse graphs, however, advanced heuristics give clearly better results in exchange for a higher running time. For small graphs it is even feasible to use exact algorithms [126, 132, 133, 225]. It is therefore desirable to transfer the diversity of crossing reduction algorithms to clustered level graphs. This leads to several concrete crossing reduction algorithms suited for different classes of clustered level graphs.

Our goal is to give a generic scheme how crossing reduction algorithms for level graphs can be extended to clustered level graphs. Obviously, the extension of different heuristics results in algorithms with different quality. However, this depends both on the original algorithm and on the extension scheme. Crossings in clustered level embeddings can have three different reasons. Some of them are unavoidable, because the clustered level graph is not clustered level planar. The rest come either from the original heuristic or from the extension scheme. For measuring the quality of an extension scheme, we only consider the latter. A weak extension scheme decreases the quality of a heuristic, not only because the clusters reduce the number of admissible permutations, but also because new unnecessary crossings are created by the extension scheme.

5.1.1 Considering Clusters Independently

In clustered graphs, the most obvious idea for reducing crossings and for solving problems in general is to recursively solve the problem for every cluster. This approach has been used many times, for example in graph editing tools [181], drawing algorithms [10, 166], and layout graph grammars [16, 17, 119].

The contents and the outside of clusters are considered independently of each other. First, the contents of the innermost clusters are drawn, i. e., of those clusters that contain only vertices and no other clusters. Then the internal structure is hidden, and the clusters are treated as single large vertices. Edges to and from a vertex contained in a collapsed cluster are connected to the cluster vertex instead. This is repeated for every cluster, traversing the cluster tree bottom up. At the end, the contents of each cluster are reinserted, leading to an embedding for the whole graph. Instead of processing the cluster tree bottom up, it is also possible to start at the root of the cluster tree, first embedding the outside of the clusters and then inserting the contents recursively.

Note that clusters across multiple levels are collapsed to vertices that span multiple levels. This is not supported by all crossing reduction heuristics. There are some algorithms that support vertices with arbitrary size by assigning large vertices to multiple levels [97, 178, 200, 201]. These algorithms also incorporate crossing reduction heuristics that handle multi-level vertices. Those could be used to ensure the cluster/cluster restriction.

Even if we ignore the cluster/cluster restriction, Example 5.1 shows that considering clusters independently leads to unnecessary crossings. This is because parts of the layout are computed without considering the global connectivity of the graph, no matter whether the cluster tree is traversed top down or bottom up. Even if an optimal level embedding is computed for the contents of each cluster, this is no guarantee for an overall optimal clustered level embedding.

Example 5.1. *Figure 5.1(a) shows a given clustered graph after an embedding has been computed outside of the cluster. Up to now the contents of the cluster have been ignored, and so the shown embedding has a minimum number of crossings and is unique except for reflection. From now on the embedding of the outer vertices is fixed. When later in Figure 5.1(b) the contents of the cluster are considered, any permutation of the inner vertices results in four crossings. However, if the algorithm had chosen a suboptimal outer embedding as shown in Figure 5.1(c), a total number of one crossing would have been possible.*

Swapping the contents and the outside of the cluster shows that traversing the cluster tree bottom up also generates unnecessary crossings. Please note that in general it is also not sufficient to additionally consider the edges entering a cluster, because the shown effect does not have to occur at the border of the cluster. Just imagine the entering edges replaced by arbitrarily long chains within the cluster.

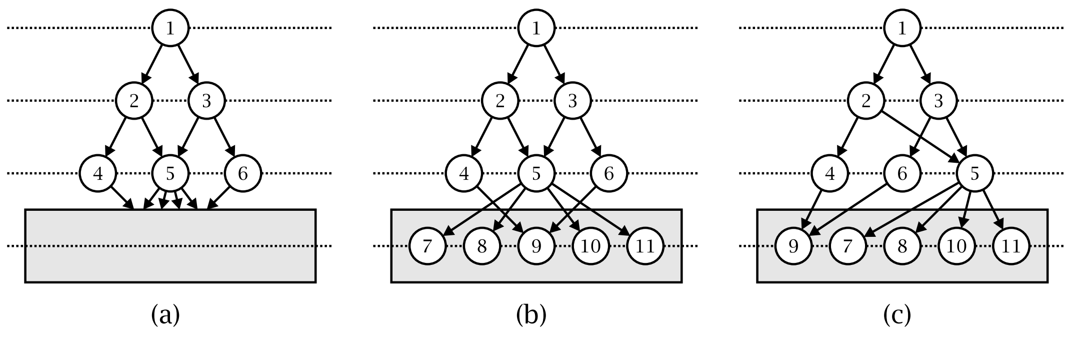


Figure 5.1. Unnecessary edge crossings when considering cluster contents independently

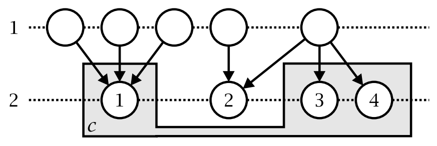
5.1.2 Sander's Crossing Reduction

While the main focus in the approach shown above is on the clustering, the algorithm of Sander [195–197] primarily considers the leveling. His crossing reduction method for clustered level graphs is based on conventional crossing reduction in level graphs. The clusters are considered only secondarily to meet the drawing conventions. They are ignored first, which leads to violations of the cluster/level and cluster/cluster restrictions. These violations are resolved afterwards.

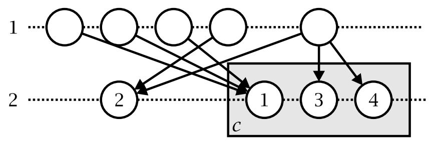
The correction of an embedding π to satisfy the restrictions is done in two stages. Each restriction is satisfied independently. To satisfy the cluster/level restriction, for each cluster $c \in C$ the average position of the contained vertices

$$b(c,i) = \sum_{v \in V_c \cap V_i} \frac{\pi(v)}{|V_c \cap V_i|}$$

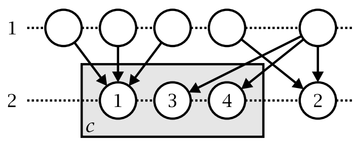
is used to sort each level i again. Because clusters are positioned as a whole, the contained vertices are then consecutive. The main problem of this approach is that only the intermediate vertex order is considered, and the edges are ignored. Because of this, $b(c,i)$ does not directly correspond to the average position of adjacent outer vertices, and therefore is kind of a "wrong" sorting criterion. It is easy to construct simple examples, where this strategy gives many unnecessary crossings, see Figure 5.2.



(a) An illegal permutation before reordering, $b(c,2) = 2\frac{2}{3}$



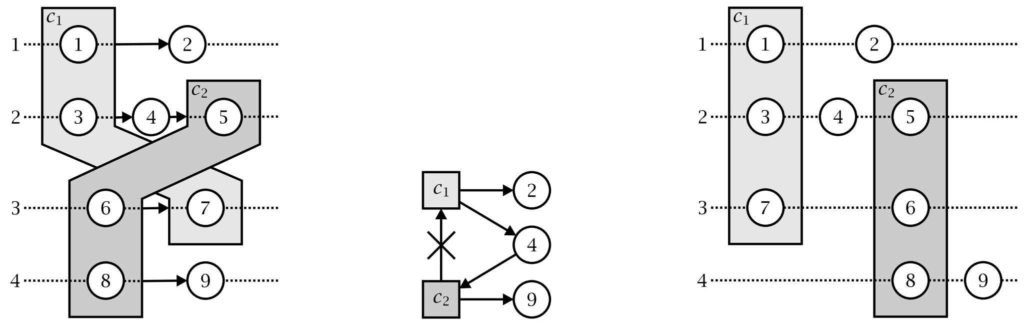
(b) The corrected permutation after reordering



(c) Optimal result

Figure 5.2. Unnecessary crossings with Sander's method

The cluster/cluster restriction is then ensured by breaking cycles in the "is left of" graph as illustrated in Figure 5.3. The levels are reordered by topologically sorting the de-cycled graph. To introduce as few additional crossings as possible, the algorithm does not remove a minimum number of edges, but instead uses a heuristic to minimize the number of generated crossings. However, the heuristic does not guarantee optimality, and therefore again new unnecessary crossings may be introduced.



(a) Before reordering (b) The "is left of" graph (c) After reordering

Figure 5.3. Sander's method for respecting the cluster/cluster restriction

5.2 Advanced Clustered Crossing Reduction

Both of the presented algorithms generate unnecessary crossings. Since crossing reduction is NP-hard, we cannot expect an efficient optimal crossing reduction algorithm for clustered level graphs. But in the above heuristics, some crossings do not originate from the used crossing reduction method itself, they arise from the application of the crossing reduction method to clustered level graphs. Thus even with an optimal crossing reduction strategy for level graphs these crossings are unavoidable. Furthermore, none of the algorithms accounts for cluster/edge crossings. Only edge crossings are minimized.

We present a new algorithm for clustered crossing reduction that improves the known results. The main idea is a scheme, how a one-sided two-level crossing reduction method for level graphs can be applied to proper clustered level graphs. The level sweep in the crossing reduction step stays the same as for level graphs. The underlying graph is traversed top down and bottom up in the same way, considering the clusters only during the two-level crossing reduction. In this regard our algorithm is very similar to the algorithm of Sander. The main difference is the strict enforcement of the cluster/level and cluster/cluster restrictions. While Sander's algorithm first ignores the restrictions, our strategy is to consider them right away. It therefore does not introduce unnecessary crossings. If it is used to extend an optimal one-sided two-level crossing reduction algorithm, the result is an optimal one-sided crossing reduction algorithm for clustered two-level graphs. Global optimality is not reached, because still the level sweep may introduce unnecessary crossings. This is not different to level graphs, however.

Our method works by modifying the one-sided two-level crossing reduction step used as a subroutine in the global crossing reduction to satisfy the cluster/level and cluster/cluster restrictions. We assume a clustered level graph $G = (V, E, C, I, \phi)$, with two levels $V = V_1 \cup V_2$. The order of V_1 is fixed, V_2 must be reordered. Therefore, only the level cluster tree Γ_2 of the second level is considered.

5.2.1 Respecting the Cluster/Level Restriction

As a first step to the solution of the problem, we focus on the cluster/level restriction and ignore the cluster/cluster restriction for now. Since clusters with only a single child on the second level cannot lead to violations of the cluster/level restriction, they are ignored for efficiency reasons, and the contracted level cluster tree Δ_2 is used instead of Γ_2. We observe the following lemma:

Lemma 5.1. *Let $G = (V, E, C, I, \phi)$ be a clustered level graph. A level embedding π of G satisfies the cluster/level restriction if and only if there exists a child order of each contracted level cluster tree Δ_i such that a preorder (or postorder) traversal of Δ_i traverses the vertices in embedding order.*

Proof. For the only if direction let π be a satisfying embedding. For each level i we sort the children $z_1, \dots, z_h$ of each cluster $c \in C$ by $\pi_{\min}(z_1, i) \leq \cdots \leq \pi_{\min}(z_h, i)$. Because of the cluster/level restriction, the horizontal ranges of any two children are disjoint $\Pi(z_j, i) \cap \Pi(z_k, i) = \emptyset$, and $1 \leq j < k \leq h$ implies $\pi_{\max}(z_j, i) < \pi_{\min}(z_k, i)$. Thus in a preorder traversal of Δ_i with this child order, all vertices $V_{z_j} \cap V_i$ contained in z_j are traversed before those contained in z_k. Induction over the structure of Δ_i delivers the desired result.

For the if direction consider an arbitrary child order in Δ_i and a level embedding π induced by a preorder traversal of Δ_i. For a cluster c on level i and a vertex $v \in (V_i - V_c)$ we must show that $\pi(v, i) \notin \Pi(c, i)$. This is easy to see, because the vertices contained in c are traversed consecutively, and $\Pi(c, i)$ contains exactly this horizontal range. Since in Δ_i the vertex v is no successor of c, it is traversed either before or after all vertices contained in c. □

Because of this observation we will concentrate on finding child orders for the level cluster tree and thereof obtain the embedding. From now on, the terms child order and embedding will be used interchangeably. Next we will analyse which child orders induce few edge crossings. Cluster/edge crossings are considered later. The following lemma characterizes the relationship between edge crossings and the child order of some cluster:

Lemma 5.2. *Let $e = (u, v)$, $e' = (u', v') \in E$ be two edges in a clustered two-level graph $G = (V_1 \cup V_2, E, C, I, \phi)$, and let x be the lowest common ancestor of v and v' in the contracted level cluster tree Δ_2 with two children y and y' that are ancestors of v and v', respectively. Then the edges e and e' cross if and only if y and y' have a different relative order than u and u'.*

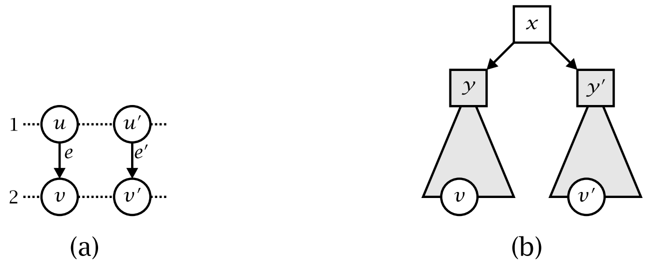


Figure 5.4. Illustration for Lemma 5.2

Proof. Note that v and y resp. v' and y' do not need to be different. The following argumentation is correct in both cases.

For the only if direction let e and e' cross each other. This implies $u \neq u'$. We assume w. l. o. g. that $\pi(u) < \pi(u')$. Therefore $\pi(v') < \pi(v)$. Since π is induced by a preorder traversal of the cluster tree, y' must be before y in the child order of x.

For the if direction assume that y and y' have a different relative order than u and u', w. l. o. g. assume that $\pi(u) < \pi(u')$ and y' comes before y in the child order of x. Then the preorder traversal implies $\pi(v') < \pi(v)$ and therefore e and e' cross. □

Thus each possible edge crossing can be associated to a unique cluster in the level cluster tree. Whether two edges cross depends only on the child order of the lowest common ancestor of their target vertices. It is independent of the child order of all other clusters. Therefore, we say that the child order of some cluster x *induces* a crossing of two edges $e = (u, v)$ and $e' = (u', v') \in E$ if x is the lowest common ancestor of v and v'. Since the total number of edge crossings induced by some child order of the cluster tree is the sum of the edge crossings induced by the child order of each cluster, we directly get the following lemma:

Lemma 5.3. *Let $G = (V_1 \cup V_2, E, C, I, \phi)$ be a clustered two-level graph with a fixed order of the first level. An embedding of G has a minimum number of edge crossings subject to this order if and only if the child order of each cluster in the contracted level cluster tree Δ_2 induces a minimal number of edge crossings.*

This is an important result, which will lead to an optimal one-sided crossing reduction for clustered two-level graphs. The decomposition of the clustered crossing reduction problem used in Section 5.1.1 was suboptimal, because the resulting instances of the one-sided two-level crossing reduction problem were not independent of each other. Because of Lemma 5.3, we can use a new decomposition, which yields the desired independency. We can compute the child order of all clusters independently without losing quality.

To minimize the overall number of crossings, we independently minimize for each cluster $x \in C$ the number of crossings induced by its child order. We construct a set of new weighted two-level graphs $\{\, G'_x \mid x \in C \,\}$, called the *crossing reduction graphs* of G. In each G'_x the upper level V_1 is the same as in G, the lower level V'_2 consists of the children of x in Δ_2. V'_2 contains vertices and clusters of the original graph. The relevant edges of G are then transferred to G'_x in the following manner, independent of the depth of the clustering:

- Edges (u, v) ending in a vertex v that is not a successor of x in Δ_2 are ignored.
- For each remaining edge (u, v), an edge (u, y) with weight $w(u, y) = 1$ is created, where y is the unique child of x which is an ancestor of v. If the edge already exists, its weight is increased by 1.

We get the following weight function

$$w\colon \begin{cases} V_1 \times V'_2 & \to \mathbb{N}_0, \\ (u, y) & \mapsto \left|\{\, v \in V_2 \cap \operatorname{succ}^*_{\Delta_2}(x) \mid (u, v) \in E \,\}\right| \end{cases}$$

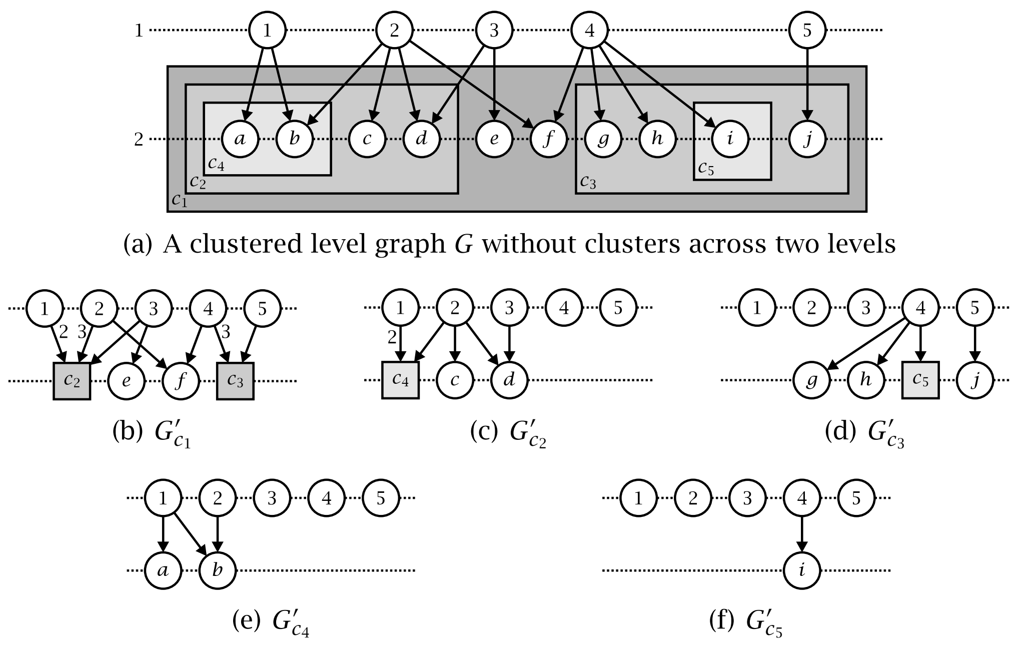
and a corresponding weighted graph G'_x:

$$\begin{aligned} G'_x &= (V'_x, E'_x, w) \\ V'_x &= V_1 \cup V'_2, \quad V'_2 = \operatorname{succ}_{\Delta_2}(x) \\ E'_x &= \{\, (u, y) \in V_1 \times V'_2 \mid w(u, y) > 0 \,\} \,. \end{aligned}$$

See Figure 5.5 for an illustration. As a direct consequence of this definition and because of Lemma 5.3, we observe the following lemma:

Lemma 5.4. *Let $G = (V_1 \cup V_2, E, C, I, \phi)$ be a clustered two-level graph with a fixed permutation of V_1. An embedding π of V_2 has a minimal number of crossings if and only if each crossing reduction graph has a minimal number of crossings.*

For a crossing reduction over more than two levels using the level sweep technique, two crossings reduction graphs have to be computed for each clus-



(a) A clustered level graph G without clusters across two levels

(b) G'_{c_1} (c) G'_{c_2} (d) G'_{c_3}

(e) G'_{c_4} (f) G'_{c_5}

Figure 5.5. Creating the crossing reduction graphs

ter and level: One for each direction. Please note that all crossing reduction graphs can be pre-computed before starting the level sweep, since they do not change during the whole crossing reduction. Algorithm 5.1 shows how the crossing reduction graphs can be computed efficiently by recursively combining the crossing reduction graphs of the children in Δ_2.

Theorem 5.1. *Algorithm 5.1 runs in $\mathcal{O}(|V_2| \cdot |E|)$ time. In a balanced level cluster tree the running time is $\mathcal{O}(|V_2| + |E| \log |V_2|)$.*

Proof. The algorithm traverses the contracted level cluster tree Δ_2 in postorder (lines 13–14). When started at the root of the cluster tree, it is executed once for each cluster, i. e., $\mathcal{O}(|V_2|)$ times. Lines 2–4 can be implemented in constant time and therefore lead to a running time of $\mathcal{O}(|V_2|)$ over all invocations. Lines 7–10 consider every vertex of V_2 and every edge of E totally once and therefore need a total running time of $\mathcal{O}(|V_2| + |E|)$. In lines 15–19 every edge of a crossing reduction graph is inherited to the crossing reduction graph of the parent cluster. For each edge this happens at most $\mathcal{O}(|V_2|)$ times, or $\mathcal{O}(\log |V_2|)$ times in a balanced cluster level tree. This sums up to $\mathcal{O}(|V_2| \cdot |E|)$ or $\mathcal{O}(|E| \log |V_2|)$, respectively. The overall running time of the algorithm is therefore $\mathcal{O}(|V_2| \cdot |E|)$ or $\mathcal{O}(|V_2| + |E| \log |V_2|)$ in balanced cluster trees, respectively. □

A conventional algorithm for weighted one-sided two-level crossing reduction is then applied to the crossing reduction graph and the given order of the first level. If the one-sided two-level crossing reduction algorithm does not

Algorithm 5.1. CROSSING-REDUCTION-GRAPH

Input: A clustered two-level graph $G = (V_1 \cup V_2, E, C, I, \phi)$,
A cluster $x \in C$, $2 \in \Phi(C)$

Output: The crossing reduction graph G'_x of x

```
begin
    $V'_2 \leftarrow \mathrm{succ}_{\Delta_2}(x)$                          // vertices
    $V' \leftarrow V_1 \cup V'_2$                              // only implicitly
    $E' = \emptyset$                                    // edges
    // for vertices in the crossing reduction graph:
    // transfer incident edges
    foreach $v \in V'_2 \cap V$ do
        foreach incoming edge $e$ of $v$ do
            $E' \leftarrow E' \cup \{e\}$                      // keep children edges
            $w(e) \leftarrow 1$
    // for clusters in the crossing reduction graph:
    // transfer edges of the crossing reduction graphs of the children
    foreach $y \in V'_2 \cap C$ do
        $G'_y = (V'_y, E'_y, \phi'_y, w'_y) \leftarrow$ CROSSING-REDUCTION-GRAPH$(G, y)$
        foreach $e = (u, v) \in E'_y$ do
            if $(u, y) \notin E'$ then
                $E' = E' \cup \{(u, y)\}$                // inherit children edges
                $w'(u, y) \leftarrow 0$
            $w'(u, y) \leftarrow w'(u, y) + w'_y(u, v)$     // sum up edge weights
    return $G'_x = (V', E', \phi', w')$
end
```

support weighted edges, it is also possible to use $w(e)$ multiple parallel edges. The resulting order of the second level is used as the order for the children of x. In the same way a child order for all other clusters is computed, and we get an order for the vertices by traversing the tree.

5.2.2 Respecting the Cluster/Edge Restriction

Up to now only edge crossings have been considered, but we also want to avoid cluster/edge crossings. Analyzing how cluster/edge crossings are related to the child order of the level cluster tree, we observe the following lemma, which transfers Lemma 5.2 to cluster/edge crossings:

Lemma 5.5. *Let $G = (V, E, C, I, \phi)$ be a clustered two-level graph with an edge $e = (u, v) \in E$, and a cluster $c \in C$ that spans both levels. Let x be the lowest common ancestor in Δ_2 of v and c, and let y and y' be the two children of x that are ancestors of v and c, respectively. Then e crosses c if and only if $u \notin V_c$ and y and y' have a different relative order than u and c on level 1.*

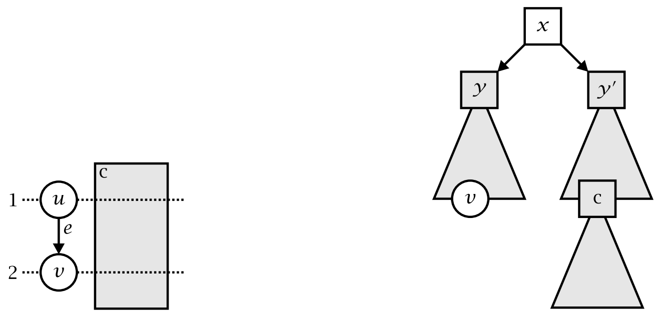
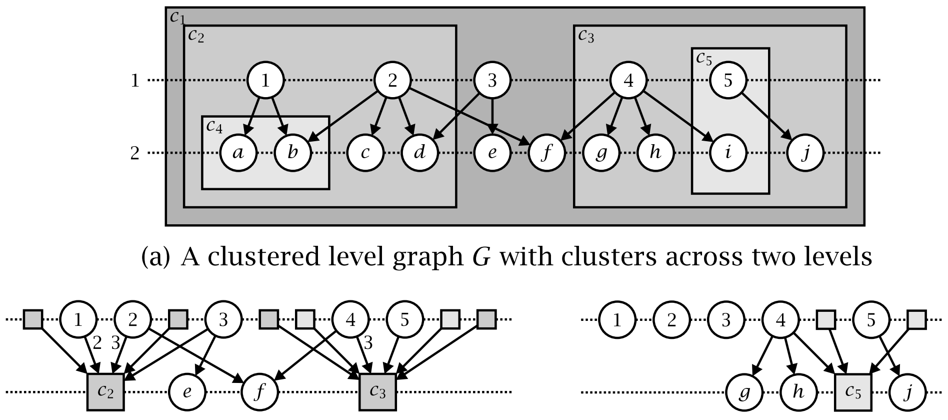


Figure 5.6. Illustration for Lemma 5.5

Proof. If $u \in V_c$ the edge and the cluster do not cross by definition. The rest of the proof is analogous to that of Lemma 5.2. □

This allows us to handle cluster/edge crossings in a similar way to edge crossings. In a crossing reduction graph every relevant cluster is represented by two *border-edges*, one for each vertical border of the cluster region. See Figure 5.7 for an illustration.

Edges (u, v) that do not start within the cluster, $u \notin V_c$, always cross both border-edges or none of them. A single cluster/edge crossing is equivalent to a crossing of the edge with both cluster borders. Therefore, a weight of $\frac{1}{2}$ is a



(a) A clustered level graph G with clusters across two levels

(b) G'_{c_1}

(c) G'_{c_3}

Figure 5.7. Avoiding cluster/edge crossings

good choice for the border-edges. Then a cluster/edge crossing is equivalent to two crossings with weight $\frac{1}{2}$ in the crossing reduction graph.

Depending on the application, cluster/edge crossings are more or less important than edge crossings. Minimizing crossings is a multi-valued optimization problem. Since edges and clusters are represented by distinct edges in the crossing reduction graphs, edge/edge and cluster/edge crossings can be balanced by the weight of the additional edges for the clusters. If the importance of edge/edge and cluster/edge crossings is not the same, their weight is multiplied by a balancing factor.

5.2.3 Respecting the Cluster/Cluster Restriction

For the final solution of the problem, only the cluster/cluster restriction remains to be resolved. It is ignored by the algorithm as it has been described up to now. To an extent the border-edges decrease the probability of cluster/cluster crossings, but they do not prevent them entirely. We present three alternative strategies for respecting the the cluster/cluster restriction.

The *re-sort method* is to use the same heuristic as described by Sander. We assume that the border-edges already prevent cluster/cluster crossings in most of the cases. Otherwise, the resulting embedding is re-sorted to eliminate cycles in the “is left of” graph. It is clear that this way we lose the optimality of our algorithm, because for solving the feedback arc set problem, we have to use a heuristic. However, the results should still be better than in Sander’s algorithm, because we have typically fewer violations of the cluster/edge restriction.

An alternative is the *heavy edge method*. Instead of using a small weight of $\frac{1}{2}$ for the border-edges, a large weight ist used, similar to the ∞-edges in Section 4.4. If the weight of these edges is large enough, an optimum permutation in the crossing reduction graphs never induces a cluster/cluster crossing. This is also true for some non-optimal crossing reduction algorithms. It is easy to see that, e. g., the barycenter and median heuristics then prevent cluster/cluster crossings. Unfortunately, this is not guaranteed for an arbitrary non-optimal crossing reduction heuristic. Consequently, it may be necessary to reorder the vertices in the same way as in the re-sort method, although less likely. Further, the greater edge weight additionally penalizes crossings between edges and clusters. Depending on the application, this may be desirable or not.

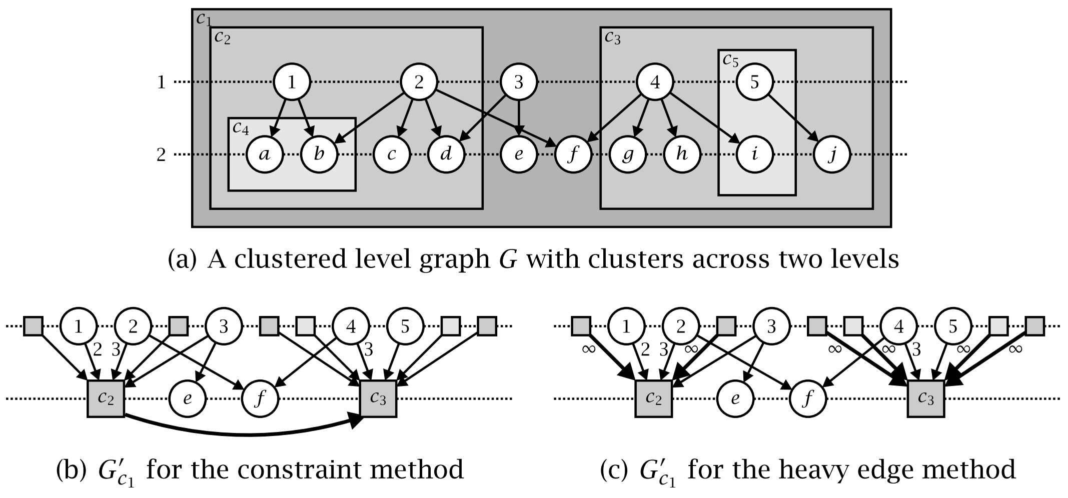
The *constraint method* is the most promising alternative. It guarantees compliance with the cluster/cluster restriction, but it depends on a two-level crossing reduction algorithm that supports constraints, i. e., predefined relative orders of some vertex pairs. A constraint (u, v) means that the vertex u must be positioned left of the vertex v. Some of the conventional crossing reduction methods have been extended to support constraints with varying success. There are also some crossing reduction methods designed specifically for the support of constraints. In Chapter 6, we will further analyse constrained crossing reduction in level graphs.

To satisfy the cluster/cluster restriction, the constraint method prevents the relative position of two clusters spanning adjacent levels from being different. This is done by inserting a constraint into the crossing reduction graph of one specific cluster. Because of Lemma 5.5 it is sufficient to add constraints between clusters having the same parent in the level cluster tree. All clusters $c_1, \dots, c_q$ that are children of some parent cluster p are connected by a chain of constraints $(c_1, c_2), (c_2, c_3), \dots, (c_{q-1}, c_q)$.

Example 5.2. *Figure 5.8(a) shows a graph G with five clusters and the crossing reduction graphs for each cluster. The bold horizontal arrow in Figure 5.8(b) shows the constraint (c_2, c_3) which has been inserted, because c_2 and c_3 are children of the same parent cluster c_1, and both span both levels. Because of the constraint, c_5, which is nested within c_3, is also automatically positioned to the right of c_2.*

The constraint method effectively prevents cluster/cluster crossings. Assuming an optimal constrained one-sided two-level crossing reduction, this leads to an optimal crossing reduction for clustered two-level graphs, see Algorithm 5.2.

Theorem 5.2. *Algorithm 5.2 gives a minimum number of crossings for the clustered one-sided two-level crossing reduction problem when used with an optimal constrained one-sided two-level crossing algorithm.*



(a) A clustered level graph G with clusters across two levels

(b) G'_{c_1} for the constraint method

(c) G'_{c_1} for the heavy edge method

Figure 5.8. Respecting the the cluster/cluster restriction

Algorithm 5.2. CLUSTERED-CROSSING-REDUCTION

```
Input: A clustered two-level graph $G = (V_1 \cup V_2, E, C, I, \phi)$,
       The contracted level cluster tree $\Delta_2$
Output: A clustered level embedding of $G$

begin
    $r \leftarrow$ root cluster of $G$
    CROSSING-REDUCTION-GRAPH($r$)
    insert border-edges
    insert constraints for multi-level clusters
    foreach $c \in C$ do
        minimize crossings in $G'_c$
    obtain an embedding $\pi$ of $V_2$ by a DFS traversal of $\Delta_2$
    return $\pi$
end
```

Corollary 5.1. *Let $G = (V_1 \cup V_2, E, C, I, \phi)$ be a clustered two-level graph such that the level cluster tree Δ_2 has bounded degree. Then the one-sided crossing reduction problem can be solved in $\mathcal{O}(|V_2| \cdot |E|)$ time.*

Proof. Because of the bounded degree, every crossing reduction graph has a bounded number on vertices on the second level. Thus there is only a constant number of permutations of the second level, and an exhaustive search runs in constant time. □

5.3 Experimental Analysis

To analyse the performance of our heuristic, we have implemented both the constraint method and Sander's algorithm in Java. We have used three different heuristics for constrained one-sided two-level crossing reduction: The penalty graph heuristic, a straight-forward extension of the sifting heuristic [105, 161, 191], and a new extension of the barycenter heuristic, see Chapter 6.

We have tested the implementations using a total number of 37,500 random clustered two-level graphs: 150 graphs for every combination of the following parameters:

$$\begin{aligned} |V_2| &\in \{50, 100, 150, 200, 250\}, \\ |E|/|V_2| &\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}, \\ |C|/|V_2| &\in \{0, 0.25, 0.5, 0.75, 1.0\}. \end{aligned}$$

Figure 5.9 displays a direct comparison of the four algorithms. The three diagrams show, how the results vary, when one of the three parameters is changed. Because the number of crossings grows very fast in the number of edges, we do not compare absolute crossing numbers, but the number of crossings divided by the number of crossings before the crossing reduction. As expected, the constraint method gives significantly better results than Sander's heuristic. For a more detailed comparison, we have also analyzed the quotient of the crossing numbers in Figure 5.10. These graphs show that results of the constraint algorithm have up to 15% less crossings than the results of Sander's algorithm. It can also be seen, that the results for different constrained crossing reduction algorithms is very small.

The running time of the algorithms is compared in Figure 5.11. As expected, Sander's algorithm is fastest, but it can also be seen, that the higher running time of the constraint method very much depends on the used constrained crossing reduction algorithm. Together with our extended barycenter heuristic, the running time is still comparable to Sander's algorithm, while its quality is

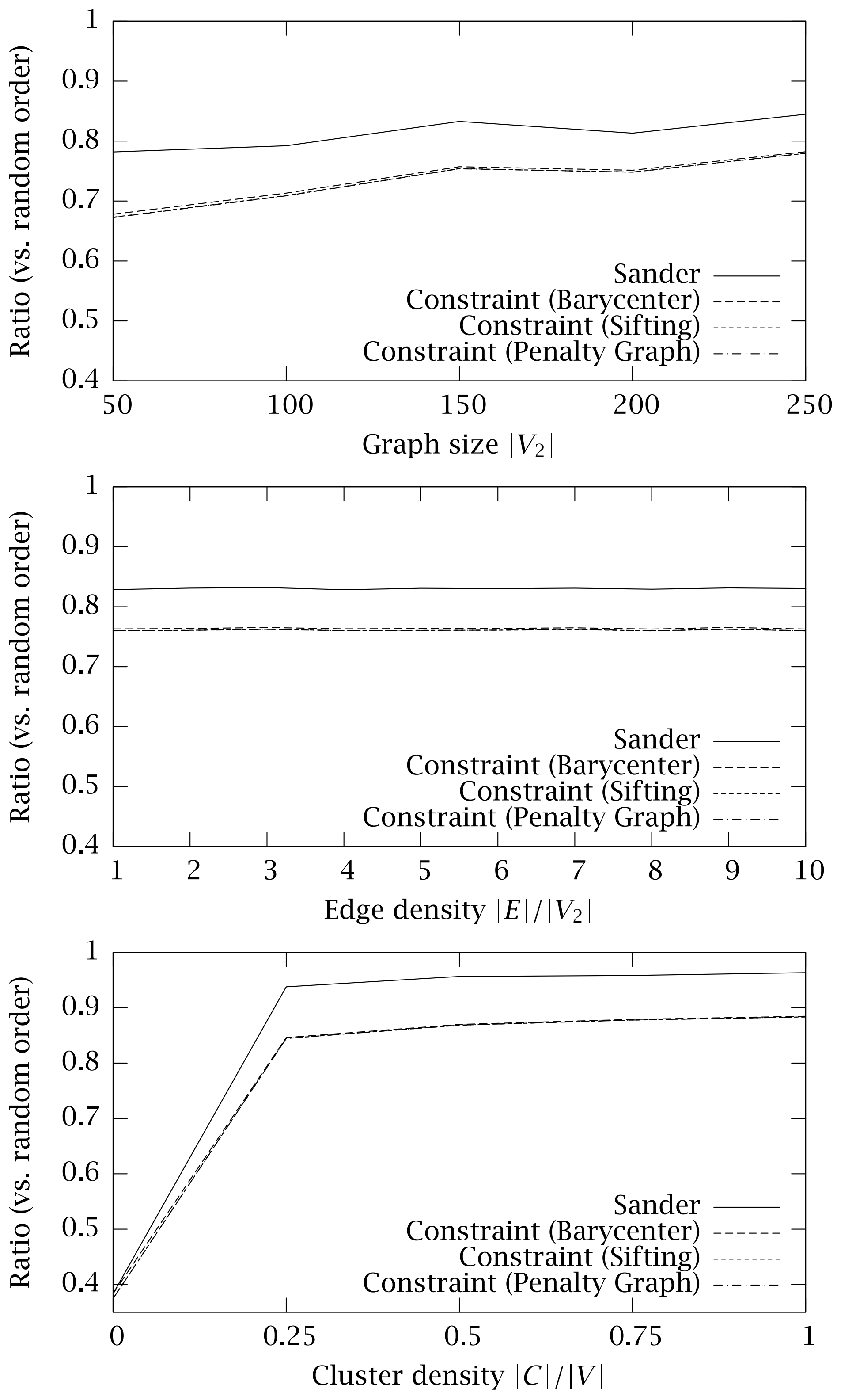


Figure 5.9. The number of crossings compared to random order. Smaller values are better.

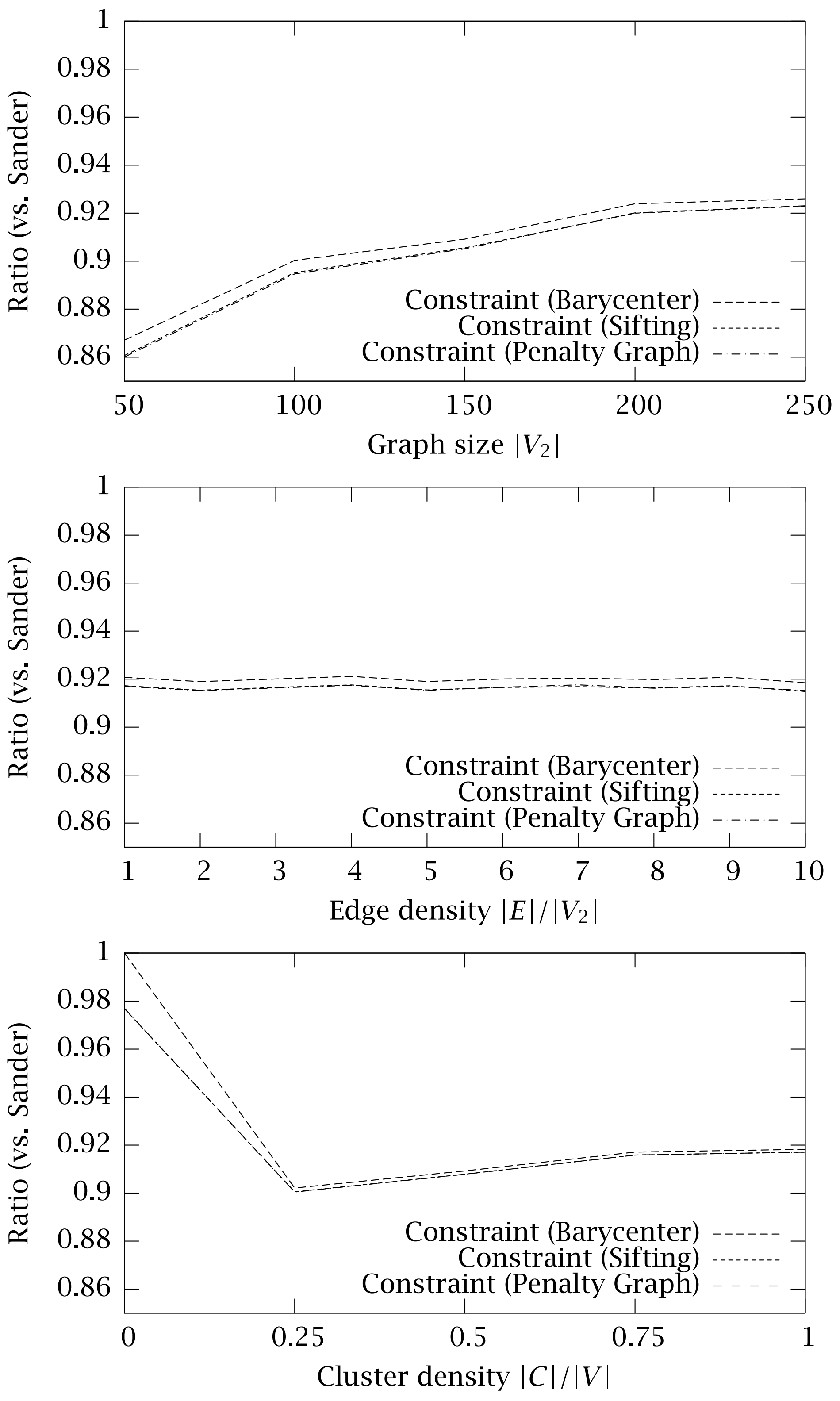
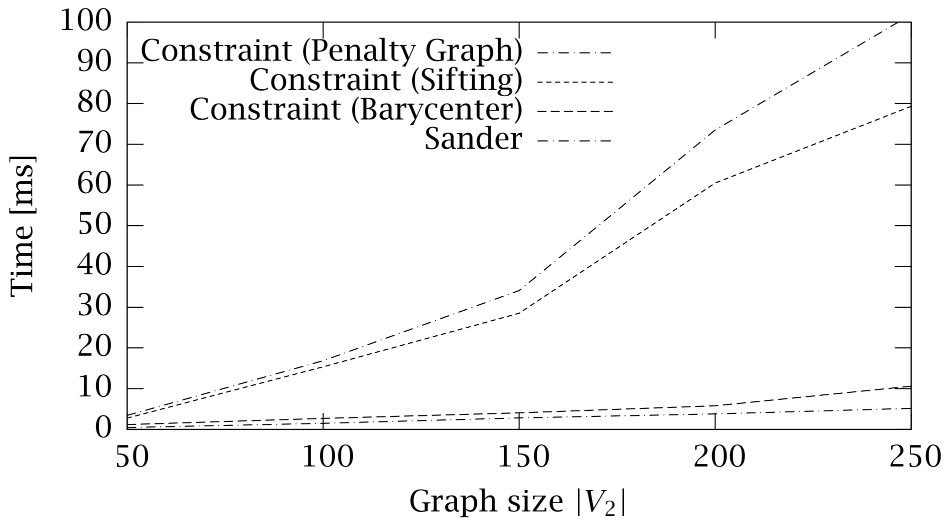


Figure 5.10. The number of crossings compared to Sander's algorithm. Smaller values are better.



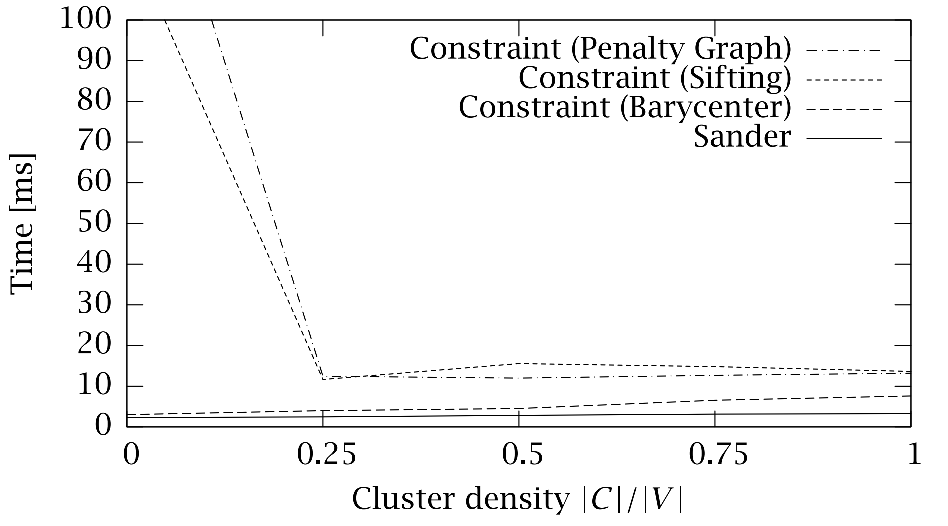


Figure 5.11. The running time in milliseconds

significantly better. The second diagram shows the running time in relation to the cluster density. This reveals an interesting result: Computational intensive constrained crossing reduction algorithms, i. e., the penalty graph method and sifting, run faster with clusters than without. This is, because with clusters, the instances for which the constrained crossing reduction is used are smaller. Also, exact methods are more feasible with many clusters, because then every crossing reduction graphs consist of only few vertices, see also Corollary 5.1.

My freedom will be so much the greater and more meaningful the more narrowly I limit my field of action and the more I surround myself with obstacles. Whatever diminishes constraint diminishes strength. The more constraints one imposes, the more one frees one's self of the chains that shackle the spirit.

Igor Stravinsky

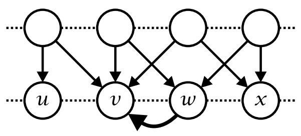
6 Constrained Crossing Reduction

In the previous chapter, we have presented algorithms for clustered crossing reduction. In one of them, the constraint method, constrained one-sided two-level crossing reduction appears as a subproblem. However, constrained crossing reduction is not only needed for clustered crossing reduction. It can also be used for the solution of other problems, such as the application of the Sugiyama algorithm to graphs with vertices of arbitrary size [97, 200], or for preserving the mental map when visualizing a sequence of related graphs [200]. The constraints may also be given by the semantics of the graph, e. g., if in UML diagrams all associations between classes should be drawn from left to right.

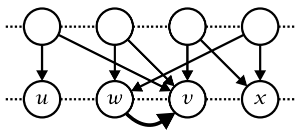
This chapter analyses the constrained one-sided two-level crossing reduction problem and gives a new heuristic based on the barycenter heuristic.

6.1 Problem Statement

In constrained one-sided two-level crossing reduction, not only the permutation of the first level is fixed, but also some pairs of vertices on the second level have a fixed relative position. Figure 6.1 shows a two-level graph with one *constraint* $r = (w, v)$, visualized by the bold arrow. The constraint means that its target vertex v must be positioned on the right of its source vertex w. In Fig. 6.1(a), the constraint is violated, and in Fig. 6.1(b) it is satisfied. Obviously, constraints may increase the minimum number of crossings, in this case from two to five.



(a) The constraint is violated



(b) The constraint is satisfied

Figure 6.1. The constrained crossing reduction problem

Formally, an instance of the constrained one-sided two-level crossing reduction problem consists of a two-level graph $G = (V_1 \cup V_2, E, \phi)$, $E \subseteq V_1 \times V_2$ with a fixed permutation π of the vertices on the first level V_1, and a set $R \subseteq V_2 \times V_2$ of constraints such that the *constraint graph* $G_R = (V_2, R)$ is acyclic. It is our objective to find a permutation of the vertices on the second level V_2 with a minimal number of edge crossings and all constraints satisfied. As a superset of the one-sided two-level crossing reduction problem, this problem is NP-hard as well.

Please note that in Chapter 5 only linear constraint graphs occurred, consisting of a simple path and isolated vertices. Nevertheless, we consider the general problem, because there are also applications for arbitrary constraint graphs, e. g., when preserving the mental map or in UML class diagrams. For cyclic constraint graphs not all constraints can be satisfied simultaneously. Then some constraints are deleted using a feedback arc set heuristic as described in Section 3.3.1.

6.2 Previous Results

The constrained crossing reduction problem has been considered several times. Sander [196] proposes a simple strategy to extend iterative two-level crossing reduction algorithms to handle constraints. Starting with an arbitrary admissible vertex permutation, updates are only executed if they do not violate a constraint. This can be applied directly to iterative heuristics like sifting [105, 161, 191], while with the barycenter heuristic a modified sorting algorithm is used: The positions of two vertices are only swapped, if no constraint is violated. Waddle [227] presents a similar algorithm: After the calculation of the barycenter values it is checked for each constraint whether its target has a lower barycenter value than its source. In that case the constraint would be violated after sorting the vertices by the barycenter values. To avoid this, the barycenter value of the source vertex is changed to the barycenter value of the target vertex plus some small value. The result of both heuristics is a vertex permutation that satisfies all constraints. However, the extensions are rather restrictive and often prevent the algorithm from finding a good permutation. Accordingly, the results are significantly worse than in graphs without constraints [200].

Schreiber [200] and Finnocchi [50, 51, 89] have independently presented an algorithm that considers constraints and crossing minimization simultaneously. The main idea of this algorithm is to reduce the constrained crossing reduction problem to the weighted feedback arc set problem. As an superset of the unweighted feedback arc set problem [55], this problem is also NP-hard. First the so-called is constructed. Its vertices are the vertices of the second level. For each pair (u, v) of vertices the number of crossings in the two relative orders of u and v is compared. Here, only crossings between edges incident to u or v are counted. If the number of crossings c_{uv} in the relative order $\dots, u, \dots, v, \dots$ is smaller than the number of crossings c_{vu} in the reverse order $\dots, v, \dots, u, \dots$, then an edge $e = (u, v)$ with weight $w(e) = c_{vu} - c_{uv}$ is inserted. Each constraint is added as an edge with infinite (or very large) weight. Figure 6.2 shows the penalty graph of the two-level graph in Fig. 6.1.

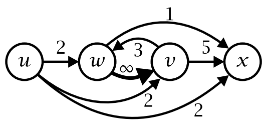


Figure 6.2. The penalty graph of Figure 6.1

Then a heuristic for the weighted feedback arc set problem is applied to the penalty graph. It is important that the used heuristic guarantees that the edges with infinite weight are not reversed, or constraints may be violated. Finally the vertices of the now acyclic penalty graph are sorted topologically, and the resulting permutation defines the order of the second level.

If no edges had to be reversed, then the number of crossings meets the obvious lower bound

$$c_{\min} = \sum_{\{u,v\} \in V^2} \min\{c_{uv}, c_{vu}\} \ .$$

Each reversed edge e increments the number of crossings by its weight. This implies that an optimal solution of the weighted feedback arc set problem is also optimal for the constrained crossing reduction problem.

A comparison of the approaches of Sander [196] and Waddle [227] with those of Schreiber [200] and Finnocchi [50, 51, 89] shows a direct trade-off between quality and execution time. Schreiber presents detailed experimental results which show that the penalty graph approach generates significantly less crossings than the barycenter heuristic extensions. This is especially evident, if there are many constraints. The running times, however, are considerably higher. This is not very surprising due to the $O(|V_2|^4 + |E|^2)$ time complexity of Schreiber's algorithm.

6.3 A Modified Barycenter Heuristic

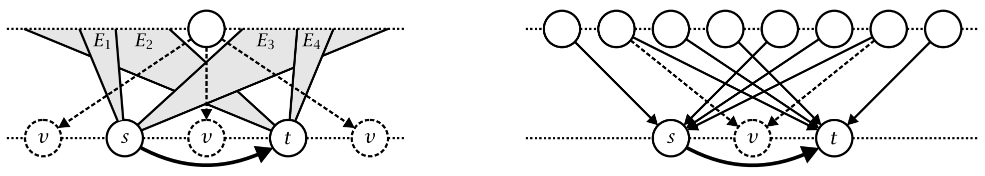
The goal of our research is to build an algorithm that is as fast as the existing barycenter extensions while delivering a quality comparable to the penalty graph approach. To achieve this we use a new extension of the barycenter heuristic. We could have used the median heuristic as well, but we did not, because it is experimentally worse, and in our algorithm median values are more difficult to handle and lead to a higher running time.

We describe our algorithm for the unweighted constrained two-level crossing reduction problem and assume a weight of 1 for all edges. An extension of the algorithm to two-level graphs with weighted edges is straight-forward by using the edge weights for calculating the barycenter values instead.

6.3.1 Idea

We start by computing the barycenter values of all vertices. As long as the source of each constraint has a lower barycenter value than the target, all constraints are satisfied automatically. In the reverse case the permutation has to be corrected. In this context, we call a constraint $r = (s, t)$ *satisfied* if $b(s) < b(t)$ and *violated* otherwise.

Our algorithm is based on a simple assumption: If a constraint is violated as in Fig. 6.3(a), the greater barycenter value of the source vertex indicates more edges "to the right" than "to the left", $|E_3| > |E_1|$. The converse is true for the target vertex, $|E_4| < |E_2|$. In this situation we assume that in the corrected permutation no other vertices should be positioned in-between. This seems plausible, because usually between s and t larger subsets of adjacent edges have to be crossed than beyond. For a vertex with only one incident edge there is always an optimal position beyond any violated constraint if using median values. This is not generally true, however, for vertices of higher degree or for the barycenter heuristic, as Fig. 6.3(b) shows. The optimal position for vertex v is in the middle, where its edges generate six crossings as opposed to eight crossings at the other two positions. Nevertheless, adopting the assumption is justified by good experimental results presented in Sect. 6.5.



(a) After the correction of a previously violated constraint ($b(s) > b(t)$), vertices with a single edge should not be positioned in-between.

(b) In general, the optimal position for a vertex may be between the vertices of a violated constraint.

Figure 6.3. The basic assumption of our algorithm

6.3.2 Main Algorithm

Our heuristic, shown in Algorithm 6.1, partitions the vertex set V_2 into totally ordered vertex lists. Initially there is one singleton list $L(v) = \langle v \rangle$ per vertex v. In the course of the algorithm these lists are pairwise concatenated into longer lists according to violated constraints. Concatenated lists are represented by new dummy vertices with associated barycenter values. As long as there are violated constraints, each violated constraint $r = (s, t)$ is removed one by one and the lists containing s and t are concatenated in the required order. They are then treated as a cluster of vertices. This guarantees that the constraint is satisfied but prevents other vertices from being placed between s and t. Following our assumption, this does no harm. A new vertex v_r replaces s and t to represent the concatenated list $L(v_r) = L(s) \circ L(t)$. The barycenter value of v_r is computed as if all edges that are incident to a vertex in $L(v_r)$ were incident to v_r. This can be done in constant time as demonstrated in lines 9 and 10 of the algorithm. Note that this efficient computation cannot be done for the median value. Therefore, and because of its experimental superiority, we only consider the barycenter heuristic.

When no violated constraints are left, the remaining vertices and vertex lists are sorted by their barycenter value as in the standard barycenter heuristic. The concatenation of all vertex lists results in a vertex permutation that satisfies all constraints. We claim that it has few crossings as well.

6.3.3 Constraint Processing Order

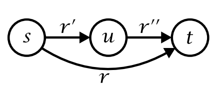
For the correctness of the algorithm, i. e., for satisfying all constraints, it is important to consider the violated constraints in the right order. In Fig. 6.4 the constraints are considered in the wrong order and r is processed first. This leads to a cycle in the resulting constraint graph which makes it impossible to satisfy all remaining constraints, although the original constraint graph was

Algorithm 6.1. CONSTRAINED-CROSSING-REDUCTION

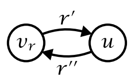
Input: A two-level graph $G = (V_1, V_2, E, \phi)$, a permutation π of V_1, and acyclic constraints $R \subseteq V_2 \times V_2$

Output: A permutation of V_2

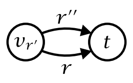
```
begin
    foreach $v \in V_2$ do
        $b(v) \leftarrow \sum_{u \in \mathrm{pred}(v)} \pi(u) / \deg(v)$        // barycenter of $v$
        $L(v) \leftarrow \langle v \rangle$                                      // new singleton list

    $V \leftarrow \{ s, t \mid (s, t) \in R \}$                                 // constrained vertices
    $V' \leftarrow V_2 - V$                                                       // unconstrained vertices

    while $(s, t) \leftarrow$ FIND-VIOLATED-CONSTRAINT$(V, R) \neq \perp$ do
        create new vertex $v_r$

        $\deg(v_r) \leftarrow \deg(s) + \deg(t)$                                  // update barycenter value
        $b(v_r) \leftarrow (b(s) \cdot \deg(s) + b(t) \cdot \deg(t)) / \deg(v_r)$

        $L(v_r) \leftarrow L(s) \circ L(t)$                                       // concatenate vertex lists

        foreach $r \in R$ do
            if $r$ is incident to $s$ or $t$ then
                make $r$ incident to $v_r$ instead of $s$ or $t$

        $R \leftarrow R - \{(v_r, v_r)\}$                                         // remove self loops
        $V \leftarrow V - \{s, t\}$

        if $v_r$ has incident constraints then $V \leftarrow V \cup \{v_r\}$
        else $V' \leftarrow V' \cup \{v_r\}$

    $V'' \leftarrow V \cup V'$
    sort $V''$ by $b()$

    $L \leftarrow \langle\rangle$                                                 // concatenate vertex lists
    foreach $v \in V''$ do
        $L \leftarrow L \circ L(v)$

    return $L$
end
```



(a) Before the merge all constraints are satisfiable by the given order. Let r be violated.



(b) After merging s and t the generated constraint cycle makes it impossible to satisfy all constraints.



(c) Starting with r' leads to a correct result.

Figure 6.4. Considering constraints in the wrong order

acyclic. If r is violated, at least one of the other constraints is also violated. Processing this constraint first leads to a correct result.

Thus, we must avoid generating constraint cycles. We use a modified topological sorting algorithm on the constraint graph. The constraints are considered sorted lexicographically by the topsort numbers of the target and source vertices in ascending and descending order, respectively. Using Algorithm 6.2 this traversal can be implemented in $O(|R|)$ time. The vertices are traversed in topological order. The incoming constraints of a vertex t are stored in an ordered list $L(t)$ that is sorted by the reverse traversal order of the source vertices. If a traversed vertex has incoming violated constraints, the topological sorting is cancelled and the first of them is returned. Note that the processing of a violated constraint can lead to newly violated constraints. Thus the traversal must be restarted for every violated constraint.

6.4 Theoretical Analysis

6.4.1 Correctness

In this section we analyse the correctness of our algorithm. We have to show that the vertex permutation computed by our algorithm satisfies all constraints. We start by analyzing Algorithm 6.2:

Lemma 6.1. *Let $r = (s, t)$ be a constraint returned by Algorithm 6.2. Then merging of s and t does not introduce a constraint cycle of two or more constraints.*

Proof. Assume that merging of s and t generates a cycle of at least two constraints. Because there was no cycle before, the cycle originates from a path p in G_R from s to t with a length of at least two. Because of the specified constraint traversal order, any constraint in p has already been considered, and

Algorithm 6.2. FIND-VIOLATED-CONSTRAINT

Input: An acyclic constraint graph $G_R = (V, R)$ without isolated vertices
Output: A violated constraint r, or $\perp$ if none exists

```
begin
    $S \leftarrow \emptyset$                          // active vertices
    foreach $v \in V$ do
        $L(v) \leftarrow \langle\rangle$                       // empty list of incoming constraints
        if indeg$(v) = 0$ then
            $S \leftarrow S \cup \{v\}$               // vertices without incoming constraints
    while $S \neq \emptyset$ do
        choose $v \in S$
        $S \leftarrow S - \{v\}$
        foreach $r = (s, v) \in L(v)$ in list order do
            if $b(s) \geq b(v)$ then
                return r
        foreach outgoing constraint $r = (v, t)$ do
            $L(t) \leftarrow \langle r\rangle \circ L(t)$
            if $|L(t)| =$ indeg$(t)$ then
                $S \leftarrow S \cup \{t\}$
    return $\perp$
end
```

thus is satisfied. This implies that $b(t) > b(s)$, and therefore contradicts the assumption. □

Lemma 6.2. *Let $G = (V_1 \cup V_2, E, \phi)$ be a two-level graph with a fixed permutation $\pi\colon V_1 \to \{1, \ldots, |V_1|\}$ of the first level, and an acyclic set $R \subseteq V_2 \times V_2$ of constraints. Then the permutation of G_2 computed by Algorithm 6.1 satisfies all constraints.*

Proof. Algorithm 6.1 maintains the invariant that the constraint graph is acyclic. Because of Lemma 6.1 no nontrivial cycles are introduced, and self loops are explicitly removed in line 15.

Next we analyse whether the removed self loop constraints are satisfied by the algorithm. Any such self loop r' has been generated by the lines 12–14 from a constraint between s and t. Because of the constraint $r = (s,t)$, the invariant implies that r' was not directed from t to s. Therefore $r' = (s,t)$ is explicitly satisfied by the list concatenation in line 11.

Each remaining constraint has not been returned by Algorithm 6.2. Thus, the barycenter value of its source vertex is less than that of its target vertex. Then the constraint is satisfied by line 20. □

Theorem 6.1. *Algorithm 6.1 and Algorithm 6.2 correctly solve the one-sided two-level crossing reduction problem.*

6.4.2 Complexity

This section analyses the running time of our algorithm. Again, we start with the analysis of Algorithm 6.2:

Lemma 6.3. *Let $G_R = (V, R)$ be an acyclic constraint graph without isolated vertices. Then Algorithm 6.2 runs on G_R in $O(|R|)$ time.*

Proof. The initialization of the algorithm in lines 2–6 runs in $O(|V|)$ time. The while-loop is executed at most $|V|$ times. The nested foreach-loops are both executed at most once per constraint. The sum of these time bounds is $O(|V| + |R|)$. Because the constraint graph does not contain isolated vertices, the overall running time of the algorithm is bounded by $O(|R|)$. □

Theorem 6.2. *Algorithm 6.1 runs in $O(|V_2| \log |V_2| + |E| + |R|^2)$ time.*

Proof. The initialization of the algorithm in lines 2–4 considers every vertex and edge once and therefore needs $O(|V_2| + |E|)$ time. The while-loop is executed at most once per constraint. It has an overall running time of $O(|R|^2)$ because the running time of one loop execution is bounded by the $O(|R|)$ running time of Algorithm 6.2. Finally the sorting in line 20 needs $O(|V_2|\log|V_2|)$ time. The sum of these time bounds is $O(|V_2|\log|V_2| + |E| + |R|^2)$. All other statements of the algorithm do not increase the running time. $\square$

6.5 Experimental Analysis

To analyse the performance of our heuristic, we have implemented both our algorithm and the penalty graph approach in Java. We have tested the implementations using a total number of 37,500 random graphs: 150 graphs for every combination of the following parameters:

$$\begin{aligned} |V_2| &\in \{50, 100, 150, 200, 250\}, \\ |E|/|V_2| &\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}, \\ |R|/|V_2| &\in \{0, 0.25, 0.5, 0.75, 1.0\}. \end{aligned}$$

Figure 6.5 displays a direct comparison of the two algorithms. The three diagrams show, how the results vary, when one of the three parameters is changed. Because the number of crossings grows very fast in the number of edges, we do not compare absolute crossing numbers, but the number of crossings divided by the number of crossings before the crossing reduction. As expected, the penalty graph approach gives strictly better results than our heuristic. But the graphs also show that the difference is very small. For a more detailed comparison, we have also analyzed the quotient of the crossing numbers in Figure 6.6. These graphs show that results of our algorithm are never more than 3% worse than the results of the penalty graph approach. For most graphs the difference is below 1%. Only for very sparse graphs there is a significant difference.

This is a very encouraging result, considering the running time difference of both algorithms: Figure 6.7 compares the running time of the algorithms. As expected, our algorithm is significantly faster than the penalty graph approach. Because of the high running time of the penalty graph approach we have not compared the algorithms on larger graphs, but our algorithm is certainly capable of processing larger graphs. For example, graphs with $|V_2| = 1000$, $|E| = 2000$, and $|R| = 500$ can be processed in less than a second, although our implementation is not highly optimized.

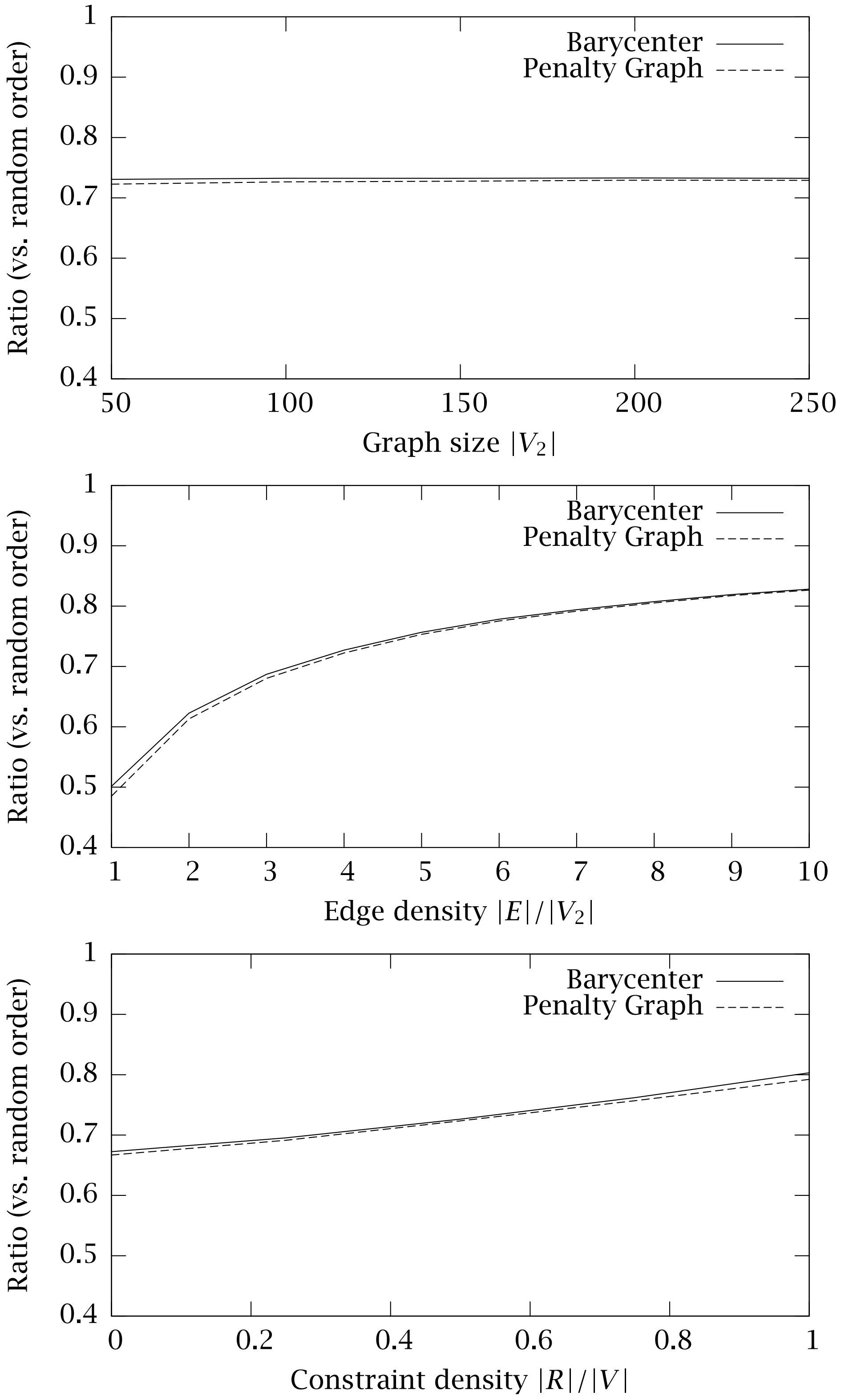


Figure 6.5. The number of crossings compared to random order. Smaller values are better.

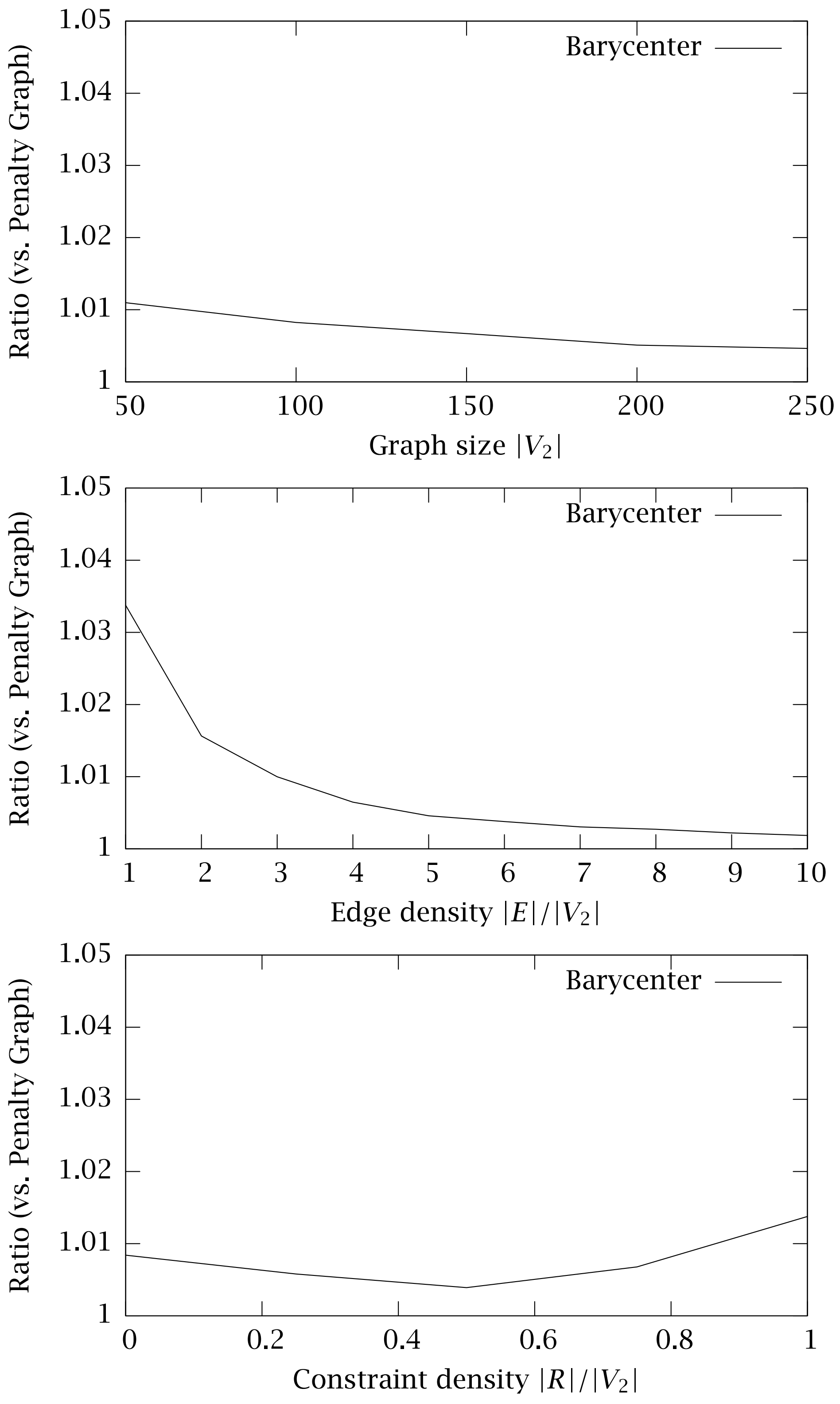


Figure 6.6. The number of crossings compared to the penalty graph approach. Smaller values are better.

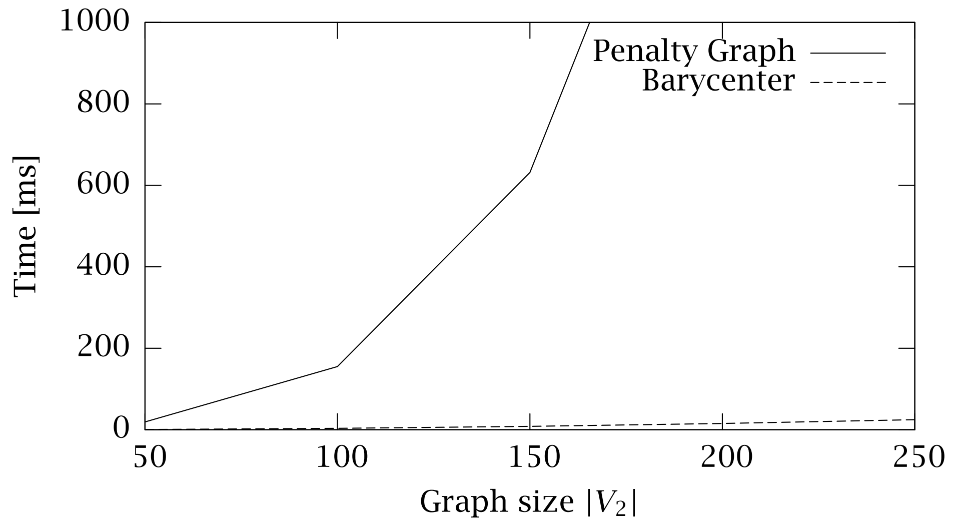


Figure 6.7. The running time in milliseconds

A mathematician is a device for turning coffee into theorems.

Paul Erdös

7 Clustered Level Planarity

In the previous chapters, our main objective was to compute drawings with few crossings. We will now consider the related question, whether a clustered level graph can be drawn without any crossings at all, i. e., whether it is clustered level planar. Despite the computational complexity of avoiding crossings, their needlessness is often very evident to a human viewer. Drawings with just a single edge crossing are often considered of poor quality if that crossing evidently could have been avoided. Many viewers regard such obviously avoidable crossings as a failure of the crossing reduction. Thus it is particularly important not to generate unnecessary crossings, if a planar drawing is possible. Unfortunately, crossing minimization heuristics may leave crossings even for planar graphs. Therefore, planar drawings need a different approach.

We will first summarize the state of the art regarding planarity testing of graphs, level graphs and clustered graphs. Afterwards, we will see how these known results may be extended to treat the new problem of clustered level planarity testing and embedding.

7.1 Previous Results

The theory of planarity has a long tradition. First results date back to Euler's research in the 18th century, such as his famous formula $|E| \leq 3|V|-6$ for planar simple graphs with at least three vertices. Since then planarity has been intensively studied for various drawing conventions and classes of graphs. There are many algorithms and theoretical results for graphs, level graphs, and clustered graphs, see for example [135, 175]. We will combine and extend these results for the investigation of clustered level planarity.

7.1.1 Planarity

A graph is planar if it admits a drawing in the plane without edge crossings. Interestingly enough, this is equivalent to the existence of a planar straight-line drawing as shown independently by Steinitz and Rademacher [212], Wagner [229], Fáry [98], and Stein [211]. Planar drawings of graphs are widely accepted as well readable. As a consequence, there are many algorithms for drawing planar graphs, see for example [39, 40, 47, 113, 135, 199, 223, 224].

Although the problems of minimizing crossings and testing for planarity seem similar, they are very different in terms of complexity. We have already seen that the crossing minimization problem is NP-hard for graphs and level graphs even in the one-sided two-level case. The *planarization problem*, i. e., determining a minimum set of edges whose removal eliminates all crossings, is NP-hard as well [76, 78, 157, 170, 222]. The planarity testing problem, however, can be solved efficiently for graphs and level graphs. This suggests that an efficient solution for planarity testing of clustered level graphs might exist as well. For clustered graphs, however, the complexity of planarity testing is one of the major open problems in graph drawing [19].

For testing planarity of graphs, there are many efficient algorithms, e. g., the LEC algorithm by Lempel, Even, and Cederbaum [83, 149]. This algorithm can be implemented in linear time using the PQ-tree data structure of Booth and Lueker [11]. It also has been extended by Chiba, Nishizeki, Abe, and Ozawa [38] to compute planar embeddings for planar graphs in linear time. Other efficient algorithms for planarity testing include the online planarity testing method of di Battista and Tamassia [58], and the path addition method of Hopcroft and Tarjan [120], which is based on work of Auslander and Parter [2] and Goldstein [107]. Several other algorithms [48, 49, 140, 141, 207, 208, 234] also solve the problem. Recently, Boyer and Myrvold [14, 15] have proposed a new linear time algorithm which is based on edge addition. This algorithm ist very interesting, because it is easy to understand and very fast in experimental results [12, 13].

7.1.2 Level Planarity

The level planarity problem [56, 116, 130] is the straightforward extension of planarity to level graphs. Given a level graph, we want to know whether there is a level planar embedding, i. e., whether the level graph can be drawn such that all vertices of the i-th level are placed on a horizontal line $l_i = \{ (x, i) \mid x \in \mathbb{R} \}$, and the edges are drawn as strictly y-monotone curves without crossings.

Note that the definition of a level graph includes a leveling of the vertices. Therefore, the leveling is given and need not be computed. Level planarity testing is substantially different from finding a level planar leveling. Heath and Rosenberg [118] have shown that deciding whether a planar graph has a proper level planar leveling is NP-hard. In contrast, if properness is not required, then every planar graph has a level planar leveling with up to $\mathcal{O}(|V|)$ levels and possibly long edges. This follows for example from straight-line grid drawings [46, 47, 55, 199] or visibility representations [55, 57, 59, 189, 219] of planar graphs. Both approaches, however, ignore the number of levels and the length of the edges. $\Omega(|V|)$ is also the lower bound for the number of levels, as a sequence of nested triangles [46] shows.

Similarly to planarity, there are several efficient algorithms for level planarity testing. The initially developed algorithms could only be applied to a restricted class of level graphs. The linear time algorithm of di Battista and Nardelli [56] is restricted to proper level graphs with a single source vertex, while the linear time algorithm of Chandramouli and Diwan [37] only works for triconnected DAGs.

The first linear time algorithm that correctly decides level planarity for arbitrary level graphs ist the JLM algorithm by Jünger, Leipert, and Mutzel [127–131, 148]. This algorithm is based on the work of Heath and Pemmaraju [116, 117], which in turn extends the algorithm of di Battista and Nardelli [56]. An extended version of the JLM algorithm is also able to compute level planar embeddings for level planar graphs. The JLM algorithm, however, is rather complex and difficult to implement. Therefore, Healy and Kuusik [115] presented a much simpler approach which runs in $\mathcal{O}(|V|^2)$ time for proper graphs and computes an embedding in $\mathcal{O}(|V|^3)$ time.

Finally, there are some interesting theoretical results about level planarity testing. Dujmović et al. [63] applied the concept of fixed parameter tractability and thereby obtained a linear running time algorithm for a bounded number of levels. Randerath et al. [187] presented a quadratic time reduction of level planarity of proper level graphs to the satisfiability problem of Boolean formulas in 2CNF, which is solvable in linear time.

7.1.3 Algorithm of di Battista and Nardelli

Our algorithm for testing planarity of clustered level graphs in Section 7.2 is an extension of the algorithm of di Battista and Nardelli [56] for testing level planarity of proper level graphs with a single source vertex. It is necessary to give a sketch of this algorithm before we can describe our extensions. We use a simplified notation for the description.

Let $G = (V, E, \phi)$ be a k-level graph. The basic idea of the algorithm is to perform a top down sweep over the graph, processing the levels in ascending order and for every level V_i to compute a set of vertex permutations that appear in some level planar embedding of G_i. G_i is the subgraph induced by the vertices of the first i levels $V_1 \cup V_2 \cup \cdots \cup V_i$. The input graph G is level planar if and only if the set of permutations of $G_k = G$ is not empty. Figure 7.1 illustrates the level sweep.

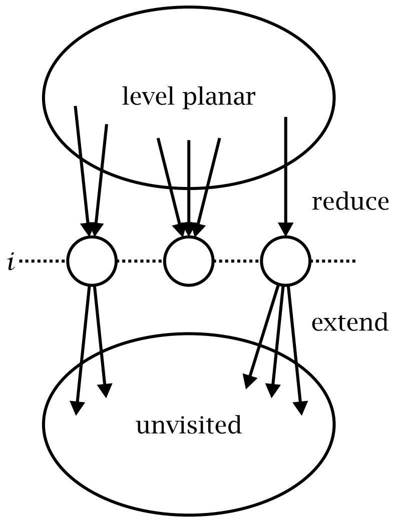


Figure 7.1. Vertices are reduced level by level

This is very similar to the LEC algorithm [83, 149] for testing planarity of graphs. The main difference is the processing order of the vertices. While for level graphs the vertices are processed in level order, a special order, called *st-numbering* [83, 84, 149] is used for graphs.

In order to store and manipulate sets of admissible vertex permutations efficiently, a data structure called PQ-tree is used. PQ-trees have been introduced by Booth and Lueker [11] for the linear time implementation of the LEC algorithm. Given a set S, a PQ-tree represents the set of those permutations of S in which the members of specified subsets of S occur consecutively. It is a rooted and ordered tree with leaves and two types of inner nodes, P- and Q-nodes, see Figure 7.2. P-nodes are drawn as circles, Q-nodes are drawn as rectangles. The leaves correspond to the elements of S and the possible permutations are encoded by the combination of the two types of inner nodes. The children of a P-node can be permuted arbitrarily, whereas the children of a Q-node are ordered and only reversion of the children is allowed.

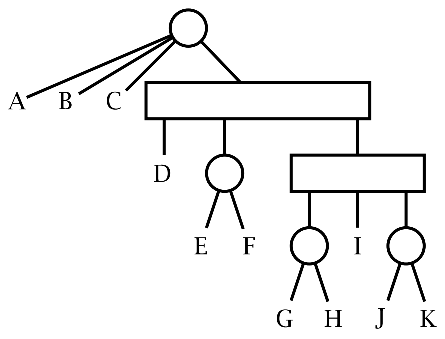


Figure 7.2. A PQ-tree

If PQ-trees are used in planarity tests, a P-node always represents a cut vertex and a Q-node represents a biconnected component of the visited part of the graph. The leaves represent edges to the unvisited part of the graph. Edges with the same target vertex on the next level that are represented by two distinct leaves of a PQ-tree impose a restriction on the admissible permutations, because there must not be a leaf between them that represents an edge ending at a different vertex. If there are no permutations with the given restrictions, the PQ-tree is empty.

The two most important operations on PQ-trees are REDUCE and REPLACE. The REDUCE operation restricts the encoded set of permutations such that all elements of a given subset $S' \subseteq S$ are consecutive in all remaining permutations. PQ-leaves representing elements of S' are called *pertinent*. After the reduction, the so-called *pertinent subtree* is the unique subtree of minimum height containing all pertinent PQ-leaves. Its root is called the *pertinent root*. All vertices on a path from the pertinent root to a pertinent leaf are also called pertinent. REDUCE is implemented on a PQ-tree by traversing bottom up the pertinent subtree, and performing local updates for all traversed nodes. The local updates are done according to a static set of rules called *templates*. In Figure 7.3 one of these templates is shown as an example. The triangles represent subtrees. Pertinent nodes and subtrees are drawn in grey. For a detailed description see [11].

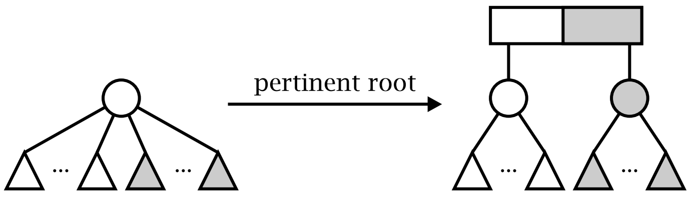


Figure 7.3. A PQ-tree template

The REPLACE operation replaces a set of elements $X = \{x_1, \dots, x_p\}$ by a set of new elements $Y = \{y_1, \dots, y_q\}$ in all stored permutations. REPLACE imposes the precondition that before an application the elements of X must be consecutive in all stored permutations. They can be in arbitrary order π, however:

$$(a_1, \dots, a_{i-1}, x_{\pi(1)}, \dots, x_{\pi(p)}, a_{i+1} \dots, a_q)$$

In practice this restriction is always satisfied, because REPLACE is always called directly after REDUCE. Then every permutation in S is replaced by a set of permutations in which the elements of X have been substituted by a permutation π' of the elements of Y:

$$(a_1, \dots, a_{i-1}, y_{\pi'(1)}, \dots, y_{\pi'(q)}, a_{i+1} \dots, a_q)$$

In a PQ-tree, REPLACE is implemented as follows: Because of the precondition, the leaves of the PQ-tree that represent the elements of X are the leaves of a subtree or of multiple sibling subtrees. These subtrees are removed from the tree and replaced by a new P-node whose children are new PQ-leaves for the elements of Y.

Algorithm 7.1 describes the algorithm of di Battista and Nardelli. Since the input graph has only a single source vertex, all intermediate graphs G_i are connected. Therefore, in contrast to the JLM algorithm, a single PQ-tree $T(G_i)$ is sufficient. It represents the set of those admissible permutations of the vertices in V_i that appear in some level planar embedding of G_i. At the beginning, the PQ-tree is initialized with all permutations of the first level, which in our case only contains the single source vertex, $V_1 = \{s\}$. Thus, there is exactly one permutation, and the initial PQ-tree consists of a single leaf with label s and no inner nodes. Then, the graph is traversed level by level, and for each level i the PQ-tree is updated to reflect the admissible permutations of the next level $i+1$. If for some level there are no admissible permutations, then the graph ist not level planar and the algorithm aborts. If after the traversal of all levels there are left some permutations of the last level, then the graph is level planar.

The update of the set of admissible permutations is done by the procedure CHECK-LEVEL in Algorithm 7.2. For efficiency reasons all operations are performed directly in $T(G_i)$ as opposed to in the graph. Define H_i to be the *extended form* of G_i. It consists of G_i and some new *virtual vertices* and *virtual edges*. For every edge (u, v) with $u \in V_i$ and $v \in V_{i+1}$, a new virtual vertex v' with label v and a virtual edge (u, v') are introduced into H_i. Note that there may be several virtual vertices with the same label, each with exactly one entering edge. The extension of $T(G_i)$ to $T(H_i)$ is called the *vertex addition step* and is accomplished by the PQ-tree operation REPLACE. For each vertex

Algorithm 7.1. LEVEL-PLANARITY-TEST

```
Input: A level graph $G = (V_1 \dot\cup V_2 \dot\cup \dots \dot\cup V_k, E, \phi)$
Output: A Boolean value indicating whether $G$ is level planar
1 begin
2     Initialize $T(G_1)$
3     for $i \leftarrow 1$ to $k - 1$ do
4         $T(G_{i+1}) \leftarrow$ CHECK-LEVEL($T(G_i)$, $V_{i+1}$)
5         if $T(G_{i+1}) = \emptyset$ then
6             return false
7     return true
8 end
```

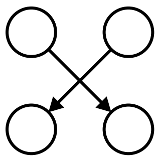
u on level i, the subtrees containing the leaves with label u are replaced by a P-node with new leaves for the adjacent vertices on the next level. Thereafter, using the PQ-tree operation REDUCE all PQ-leaves representing vertices in V_{i+1} with the same label are reduced to appear as a consecutive sequence in any permutation stored in the PQ-tree. The resulting *reduced extended form* of H_i is denoted by R_i. Finally, all PQ-leaves representing sinks v in V_{i+1} are removed from the PQ-tree and the tree is reconstructed such that it obeys the properties of a valid PQ-tree again.

Algorithm 7.2. CHECK-LEVEL

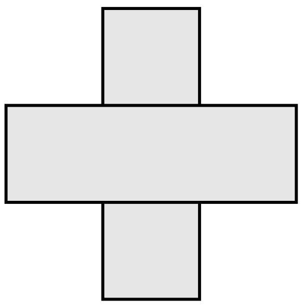
```
Input: PQ-tree $T(G_i)$ of the current level, Vertices $V_{i+1}$ of the next level
Output: PQ-tree $T(G_{i+1})$ of the next level
1 begin
2     extend $T(G_i)$ to $T(H_i)$
3     reduce $T(H_i)$ to $T(R_i)$
4     if $T(R_i) = \emptyset$ then
5         return $T(G_{i+1}) \leftarrow \emptyset$
6     remove sinks from $T(R_i)$
7     return $T(G_{i+1}) \leftarrow T(R_i)$
8 end
```

7.1.4 Clustered Planarity

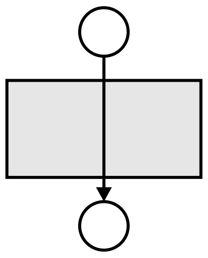
Planarity has also been investigated for clustered graphs. A clustered graph is *clustered planar* or *c-planar*, if it has a drawing without edge crossings, region intersections, or crossings between an edge and a region. An edge crosses a region if it crosses its border at least twice, see Figure 7.4.



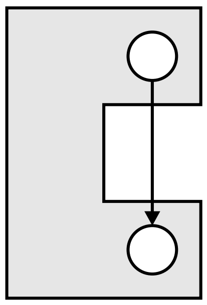
(a) Edge crossings



(b) Intersecting clusters



(c) Edges entering a cluster and leaving again

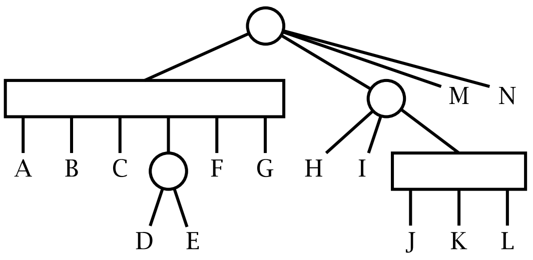


(d) Edges leaving a cluster and entering again

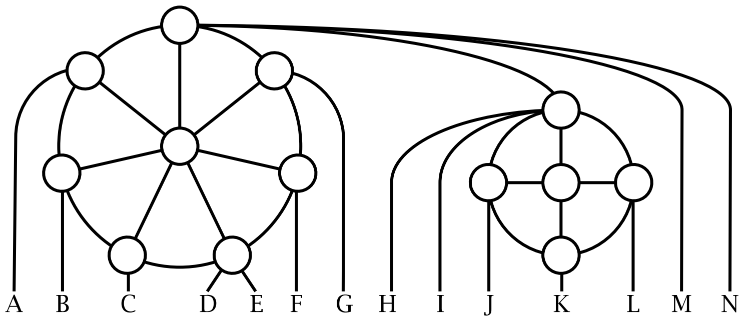
Figure 7.4. Forbidden situations in c-planar embeddings

In contrast to planarity and level planarity, c-planarity testing seems to be more difficult. Obviously, c-planarity of a clustered graph implies planarity of the underlying graph, but not vice versa. Here connectivity plays a crucial role. A clustered graph is *c-connected*, if each subgraph induced by a cluster is connected. For c-connected clustered graphs there is an algorithm by Feng, Cohen, and Eades [88] that tests c-planarity in quadratic time. This algorithm tests planarity of the contents of each cluster using a variant of the LEC algorithm. It starts with a cluster that is a leaf in the cluster tree and therefore does not contain other clusters. After the cluster has been tested, the resulting PQ-tree represents the admissible permutations of the edges leaving the cluster. This PQ-tree is then converted to a graph that encodes the same permutations by representing all Q-nodes with wheel subgraphs, see Figure 7.5. The cluster is then replaced by this graph, leading to a new graph that is c-planar if and only if the original graph was c-planar. The remaining clusters are processed in the same way, traversing the cluster tree bottom up.

Later, Dahlhaus [45] improved the algorithm of Feng, Cohen, and Eades to linear time. Gutwenger et al. [108] give a polynomial time algorithm for a some-



(a) In the resulting PQ-tree of a cluster...



(b) ...Q-nodes are converted to wheel subgraphs

Figure 7.5. Testing c-planarity

what larger class of clustered graphs. Cornelsen and Wagner [44] introduce a stronger form of connectivity for clustered graphs: A clustered graph is *completely connected*, if every cluster and each complement of a cluster induces a connected subgraph. They show that a completely connected clustered graph is c-planar if and only if the underlying graph is planar. For general clustered graphs, however, the complexity of c-planarity testing is still open [19]. Finally, di Battista, Didimo and Marcandalli give a planarization algorithm for clustered graphs [53].

7.2 Clustered Level Planarity Testing

We will now investigate planarity for clustered level graphs by combining and extending known results about planarity for graphs, level graphs and clustered graphs. We focus in particular on a new testing algorithm for clustered level planarity.

Remember that a clustered level graph is clustered level planar if it has a clustered level planar embedding, i. e., a level embedding, that satisfies the cluster/level, cluster/cluster, edge/edge, and cluster/edge restrictions, see Definition 4.13 on page 47. The restrictions ensure that any clustered level planar graph can be drawn without crossings such that all cluster regions are convex.

They can even be drawn as rectangles by using Sander's algorithm [195–197]. Clustered level planarity is the combination of level planarity and c-planarity.

Lemma 7.1. *If $G = (V, E, C, I, \phi)$ is a clustered k-level graph, then obviously*

1. *G is clustered level planar $\Rightarrow$ (V, E, ϕ) is level planar $\Rightarrow$ (V, E) is planar.*
2. *G is clustered level planar $\Rightarrow$ (V, E, C, I) is c-planar $\Rightarrow$ (V, E) is planar.*

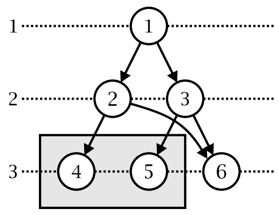
Note that a level planar and c-planar clustered level graph is not necessarily clustered level planar. The graph in Figure 7.6 is a counter-example. It cannot be drawn with all conditions of a clustered level planar embedding satisfied simultaneously. Either the embedding is level planar and the cluster/level restriction is violated, or the embedding is convex c-planar, but it is no level embedding.

Without loss of generality we only consider simple graphs without self loops and parallel edges. Because of Lemma 7.1, a simple input graph with $|E| > 3|V| - 6$ is rejected as not clustered level planar and we can assume that the number of edges is linear in the number of vertices.

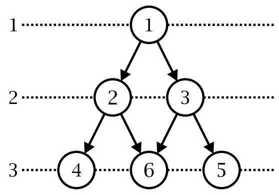
We give an $\mathcal{O}(k|V|)$ time algorithm for testing clustered level planarity of a large class of clustered k-level graphs, which we call *elementary* clustered k-level graphs. The restrictions of our algorithm are inherited from the underlying level planarity testing algorithm. The algorithms for testing level planarity of arbitrary level graphs cannot be easily extended for testing clustered level planarity. For example, see Section 7.4 for a description of why the JLM algorithm does not work for clusters. Therefore, our algorithm is based on the work of di Battista and Nardelli [56], and has similar restrictions on the input graphs. Both algorithms only work on proper clustered level graphs with a single source vertex. Similar to the c-planarity testing algorithm of Feng et al. [88], clusters with disconnected contents are also difficult to handle in clustered level graphs. Thus, similar restrictions are imposed. Instead of c-connectivity, however, a weaker form of connectivity is sufficient for clustered level graphs, which we call level connectivity:

Definition 7.1 (Level Connectivity). *A clustered level graph $G = (V, E, C, I, \phi)$ is* level connected, *if any two consecutive levels of the same cluster are spanned by an edge of the cluster, i. e., if $\forall c \in C\colon \forall i \in \{\phi_{\min}(c), \ldots, \phi_{\max}(c) - 1\}\colon \exists (u, v) \in E_c\colon \phi(u) \leq i \wedge \phi(v) \geq i + 1$.*

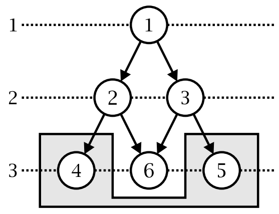
Level connectivity follows directly from c-connectivity. Every c-connected clustered level graph is level connected. The reverse direction is not generally true. Level connected clustered level graph need not be c-connected. See Figure 7.7 for a comparison of level connectivity and c-connectivity.



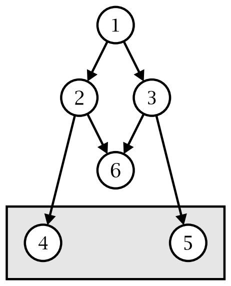
(a) A non-planar clustered level embedding



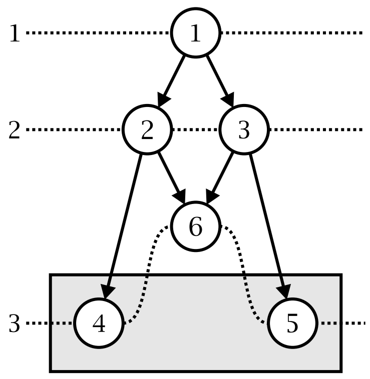
(b) A level planar embedding ...



(c) ... which violates the cluster/level restriction



(d) A c-planar embedding ...



(e) ... which is no level embedding

Figure 7.6. A clustered level graph that is level planar and c-planar but not clustered level planar

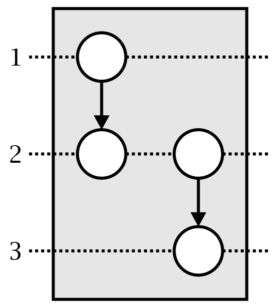


Figure 7.7. A level connected clustered level graph that ist not c-connected

Using the above descriptions of the restrictions, we define an elementary clustered level graph as follows:

Definition 7.2 (Elementary Clustered Level Graph). *A simple clustered k-level graph is* elementary *if it is proper, level connected, and has only a single source vertex.*

7.2.1 Idea

The algorithm of di Battista and Nardelli already ensures that the computed embedding satisfies the edge/edge restriction. It remains to show how the additional restrictions for clustered level planar embeddings can be maintained. We will see later that the cluster/cluster and cluster/edge restrictions are automatically satisfied if the graph is level connected, while an extension is necessary for the cluster/level restriction.

An analysis of the cluster/level restriction reveals a similarity to the ordering constraints of PQ-trees because the vertices of a cluster have to be placed next to each other. This corresponds directly to the semantics of the REDUCE operation which restricts the set of admissible permutations to those where the PQ-leaves given as an argument appear consecutively. We obtain the following idea: The level by level sweep of the level planarity testing algorithm remains the same. The admissible permutations are stored in a PQ-tree $T(G_i)$. We ensure the cluster/level restriction by additional applications of REDUCE. This is done by an extension of CHECK-LEVEL, see Algorithm 7.3. On each level a new method REDUCE-CLUSTERS is called, which ensures that the interior of each cluster is consecutive. Given a PQ-tree that encodes a set of permutations, the REDUCE-CLUSTERS operation removes all permutations that contradict the cluster/level restriction.

Note that this extension is designed to retain all invariants of the original algorithm. The main invariant is that $T(G_i)$ represents the set of currently admissible vertex permutations. This does not change with the extension of the algorithm, just the definition of "admissible" is changed. Only a subset of the previously admissible permutations is admissible now. Because no new permutations are introduced, this does not harm the correctness of the algorithm. The exact set of currently admissible permutations is insignificant. It is only important that it reflects the property for which the graph ist tested, and that it is correctly stored in a PQ-tree. In effect, the extended algorithm returns true if and only if the given graph is level planar and satisfies the cluster/level restriction.

Algorithm 7.3. EXTENDED-CHECK-LEVEL

Input: PQ-tree $T(G_i)$ of the current level, Vertices V_{i+1} of the next level
Output: PQ-tree $T(G_{i+1})$ of the next level

```
begin
    extend $T(G_i)$ to $T(H_i)$
    reduce $T(H_i)$ to $T(R_i)$
    if $T(R_i) = \emptyset$ then
        return $T(G_{i+1}) \leftarrow \emptyset$
    $T(R_i) \leftarrow$ REDUCE-CLUSTERS($T(R_i)$, $V_{i+1}$)        // new
    remove sinks from $T(R_i)$
    return $T(G_{i+1}) \leftarrow T(R_i)$
end
```

7.2.2 Efficient Cluster Reduction

A straightforward implementation of the REDUCE-CLUSTERS method is to call REDUCE for the PQ-leaves of each cluster. This leads to a running time of $\mathcal{O}(k|C||V|)$, i. e., up to $\mathcal{O}(k|V|^2)$, since for each of the k levels and for each of the |C| clusters the whole PQ-tree of size $\mathcal{O}(|V|)$ must be traversed. With the following approach this can be improved to $\mathcal{O}(k|V|)$ time.

First consider only two clusters c_1 and c_2 on the same level. There are two cases how c_1 and c_2 can interact, either they are disjoint or they are nested. In the former case c_1 and c_2 can be reduced independently. For each cluster only a subtree of the PQ-tree has to be considered. Because these subtrees are disjoint, in the worst case the whole PQ-tree has to be traversed once per level. In the latter case suppose that c_2 is nested in c_1. Then all descendants of c_2 are descendants of c_1 and the result of reducing c_2 can be used for reducing c_1. It is not necessary to traverse the pertinent subtree of c_2 again but we can start the second REDUCE at the pertinent root of c_2.

This result can be generalized to the whole cluster tree by using a simultaneous bottom up traversal of the cluster tree Γ and the PQ-tree $T(R_i)$. After a cluster c has been reduced, all PQ-leaves representing vertices contained in c are consecutive in any permutation stored in $T(R_i)$. They are exactly the leaves of a pertinent subtree. The pertinent root of this subtree can be a single node or a consecutive part of a Q-node, see Figure 7.8. We temporarily replace the pertinent subtree(s) by a new PQ-leaf X_c with label c. This avoids calling REDUCE for inner PQ-nodes which may not be supported by existing PQ-tree implementations. It is important that the replaced subtrees are reinserted later in the same order as they had before their removal. Reversions of their parent Q-node and other modifications have to be respected. Fortunately this can be done easily by remembering which sibling pointer of X_c represents the direction, w. l. o. g. the first stored pointer. This is similar to the *direction indicators* of [38].

Algorithm 7.4 shows the method REDUCE-CLUSTERS. The cluster tree Γ is traversed in a similar way as the REDUCE method traverses a PQ-tree. The cluster nodes are processed bottom up using a queue to ensure that nodes cannot be processed before all of their children that span the current level have been processed. This can be tested by comparing the number of processed children with $\mathit{child\_count}(c, i+1)$, the number of children of c spanning level $i+1$.

7.2.3 Computing an Embedding

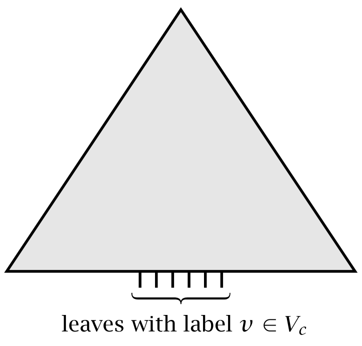
For computing a planar drawing of a clustered level planar graph, it is not sufficient to know about the existence of an embedding. We also need an algorithm for computing the embedding. Fortunately, our algorithm for clustered level planarity testing can easily be extended to an clustered level embedding algorithm that also runs in $\mathcal{O}(k|V|)$ time. The presented extensions to the level planarity testing algorithm of di Battista and Nardelli can be used without major modifications to extend the level planar embedding part of the JLM algorithm [116, 117, 129–131] to compute a clustered level planar embedding. The computed embedding can then be used as a basis for generating a drawing, e. g., with the algorithm of Sander [195–197].

7.3 Theoretical Analysis

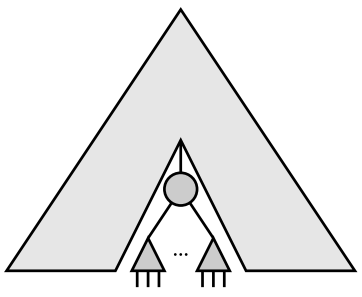
7.3.1 Correctness

Theorem 7.1. *Algorithm 7.1 with the EXTENDED-CHECK-LEVEL method shown in Algorithm 7.3 returns true if and only if the graph is clustered level planar.*

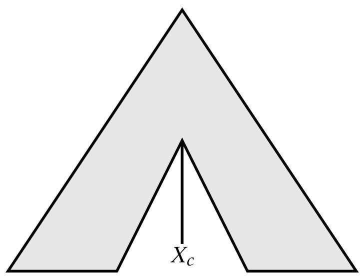
Proof. For the only if direction note that our extension does not modify the level planarity testing part of the algorithm, and it does not introduce any new level permutations. Any admissible permutation stored in the PQ-tree at a time also exists in the unmodified algorithm. Therefore, a positive result of our algorithm ensures that G is level planar. Thus, it remains to be shown that the remaining restrictions imposed by the definition of clustered level planarity are satisfied. The semantics of the cluster/level restriction for the intersection of



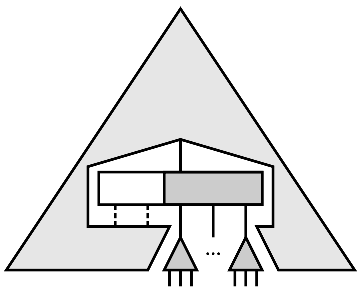
(a) Execute REDUCE with all leaves for vertices contained in the cluster



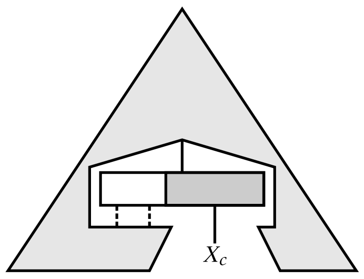
(b) If the pertinent root has only pertinent children…



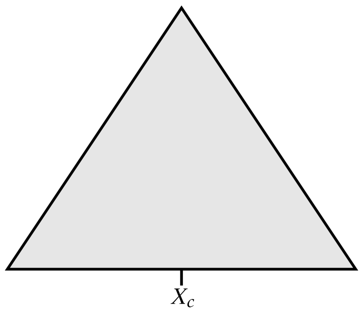
(c) …replace the whole pertinent subtree with X_c



(d) If the pertinent subtree(s) are a subsequence of a Q-node…



(e) …replace the corresponding children with X_c



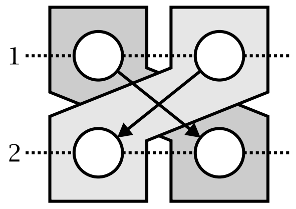
(f) The result is a valid PQ-tree where X_c represents the contents of c

Figure 7.8. Contracting pertinent subtrees

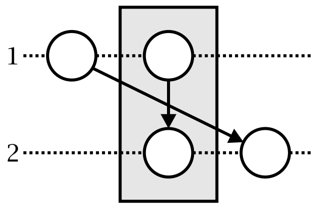
Algorithm 7.4. REDUCE-CLUSTERS

```
Input: PQ-tree $T(R_i)$, Vertices $V_{i+1}$ of the next level
Output: PQ-tree $T(R_i)$ with reduced clusters
begin
    foreach $c \in C \cup V$ do
        $children\_leaves[c] \leftarrow \emptyset$
    Initialize Queue $Q$ with $V_{i+1}$
    while $Q$ not empty do
        $c \leftarrow delete\_first(Q)$
        $p \leftarrow parent(c)$                    // parent in $\Gamma$
        // make cluster vertices consecutive
        $T(R_i) \leftarrow$ REDUCE($T(R_i)$, $children\_leaves[c]$)
        if $T(R_i) = \emptyset$ then
            return $T(R_i) \leftarrow \emptyset$
        // expand children
        foreach $X \in children\_leaves[c]$ do
            replace $X$ by $subtrees[X]$
        // contract pertinent subtree(s)
        $X_c \leftarrow$ new PQ-leaf with label $c$
        if the pertinent root has only pertinent children then
            $subtrees[X_c] \leftarrow \{pertinent\ root\}$
        else
            $subtrees[X_c] \leftarrow$ pertinent children of the pertinent root
        REPLACE($T(R_i)$, $children\_leaves[c]$, $X_c$)
        $insert(children\_leaves[p], X_c)$
        // ensure correct processing order
        if $|children\_leaves[p]| = child\_count(p, i+1)$ then
            $insert(Q, p)$
    return $T(R_i)$
end
```

a cluster $c \in C$ and a level line i are exactly the same as the semantics of a REDUCE operation applied to the children of c on level i. Since our algorithm explicitly calls REDUCE for every cluster on every level it is clear that the cluster/level restriction is satisfied. The cluster/cluster restriction is trivially satisfied for level connected graphs, because crossing clusters would imply crossing edges which are prohibited by the level planarity test. See Figure 7.9(a). The same is true for the cluster/edge restriction. The graph is proper and thus the crossing edge connects two adjacent levels. Between these two levels there is an edge in the cluster because of the level connectivity of the graph. Any intersection between these two edges is prohibited by level planarity. See Figure 7.9(b).



(a) Violation of the cluster/cluster restriction is not possible



(b) Violation of the cluster/edge restriction is not possible

Figure 7.9. Correctness of the algorithm

For the if direction consider a clustered level planar graph. We have to show that our algorithm returns true. Suppose the algorithm returns false. This means that a call of REDUCE failed, either in the level planarity test part or in REDUCE-CLUSTERS. In the former case this means that for some level $i+1$ there is no permutation such that the edges between the levels i and $i+1$ can be drawn without crossings. In the latter case there is no level planar permutation respecting the cluster/level restriction. In any case this contradicts the assumption. □

7.3.2 Complexity

The complexity of Algorithm 7.4 depends on the complexity of the *child_count* operation returning the number of cluster children on a given level. We assume that the used data structure for clustered level graphs provides $\mathcal{O}(1)$ time access to this information. This can be achieved for example by maintaining the level cluster trees or contracted level cluster trees as defined in Chapter 4. If this information is not available, an additional $\mathcal{O}(k|V|)$ size data structure can be pre-computed in $\mathcal{O}(k|V|)$ time. We use a two-dimensional matrix $M_{ci} = \mathit{child\_count}(c, i)$ with $c \in C \cup V$ and $i \in \{1, \ldots, k\}$ as indices.

M_{ci} is filled as shown in Algorithm 7.5 which traverses the cluster tree Γ in a similar way as Algorithm 7.4. Having this efficient *child_count* operation, the complexity of REDUCE-CLUSTERS derives as follows:

Algorithm 7.5. COMPUTE-CHILD-COUNT

```
Input: A clustered level graph $G = (V, E, C, I, \phi)$
Output: A matrix $M_{ci}$ containing the child_count values
begin
    Initialize $M_{ci}$ with zeros
    foreach $v \in V$ do $M_{v,\phi(v)} \leftarrow 1$
    foreach $c \in C$ do $processed\_children[c] \leftarrow 0$

    Initialize Queue $Q$ with $V$
    while $Q$ not empty do
        $c \leftarrow delete\_first(Q)$
        $p \leftarrow parent(c)$            // parent in $\Gamma$

        for $i \leftarrow 1$ to $k$ do
            if $M_{ci} > 0$ then
                $M_{pi} \leftarrow M_{pi} + 1$    // increase child count if $c$ spans level $i$

        $processed\_children[p] \leftarrow processed\_children[p] + 1$
        if $processed\_children[p] = |children(p)|$ then
            $insert(Q, p)$

    foreach $v \in V$ do $M_{v,\phi(v)} \leftarrow 0$

    return $M_{ci}$
end
```

Lemma 7.2. *The time complexity of REDUCE-CLUSTERS as described in Algorithm 7.4 is $\mathcal{O}(|V|)$.*

Proof. In REDUCE-CLUSTERS every cluster is considered exactly once. Since Γ is of linear size this can be done in $\mathcal{O}(|V|)$ time. Additionally every node of the PQ-tree is considered only once such that the time complexity of the REDUCE operations sum up to $\mathcal{O}(|V|)$. □

Theorem 7.2. *There is an $\mathcal{O}(k|V|)$ time algorithm for testing clustered level planarity of elementary clustered level graphs.*

7.4 Open Problems

The given algorithm solves the clustered level planarity problem only for elementary clustered level graphs. It would be desirable to extend it to general clustered level graphs, but a straightforward extension is difficult. In this section we will describe, why this is the case.

Level planarity testing has been extended to graphs with multiple sources in [129–131]. This is realized by the utilization of multiple PQ-trees, one for each connected component of G_i. If a vertex is common to more than one PQ-tree, these are merged into one. In a straightforward extension of our algorithm clusters can span multiple PQ-trees. This means that REDUCE-CLUSTERS cannot be applied directly. A possible solution would be to additionally merge the PQ-trees according to the contained clusters. It is not clear, however, how this could be done because in contrast to vertex merges there is no distinct position in the higher PQ-tree where the smaller one must be inserted.

An application of our algorithm to non-proper graphs leads to problems as well. A priori it is not clear whether long span edges entering or leaving a cluster have to be routed within or outside of the cluster. In Figure 7.10 it is not clear whether the reduction of the cluster nodes on level 2 has to include the dashed edge. When processing level 3 it becomes clear that this edge has to be routed within the cluster, but this is too late. On level 2 both routing alternatives would have to be stored in the PQ-tree. This is not possible, however, without major extensions of the data structure.

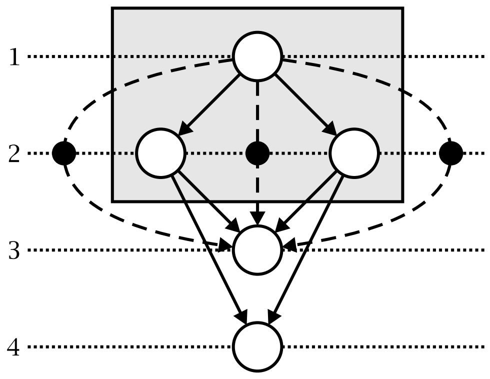


Figure 7.10. Problems with long span edges

The third remaining restriction of our algorithm is level connectivity. A straightforward idea to extend it to clustered level graphs that are not level connected would be to insert level connecting dummy edges for each cluster. It is difficult, however, to find the correct places for insertion without violating clustered level planarity. In Figure 7.11, there is only one clustered level planar embedding up to reflection. To make it level connected without destroying the planarity, an edge can only be inserted between the two grey vertices. This is not known in advance, however, so all combinations of possible edges in all clusters would have to be tried. This leads to exponential running time, however. The same problem occurs with c-planarity. There are some advances like in [108] but the general problem is unsolved. Apparently, the connectivity of the graph plays a major role for the detection of c-planarity and clustered level planarity.

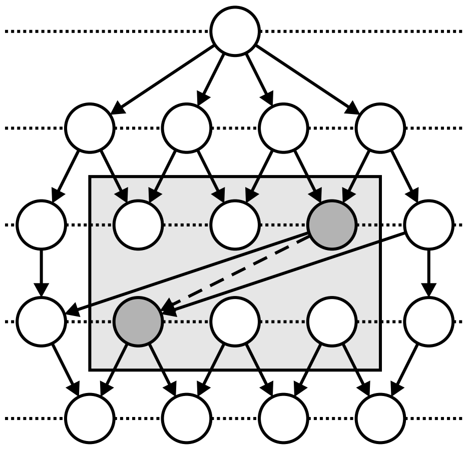


Figure 7.11. Making a clustered level graph level connected is difficult

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

Antoine de Saint Exupery

8 Conclusion

After motivating and introducing clustered graphs and level graphs, we have presented a comprehensive discussion of various problems that occur in clustered level graphs, i. e., in graphs that are clustered and leveled simultaneously. We have analyzed the interrelations between clusters and levels and their impact on edge crossings and cluster/edge crossings. Several algorithms and theoretical results have been presented. We have shown that clustered level embeddings and planarity of such embeddings can be characterized by four simple restrictions on the embedding. Further, we have shown that the problem of minimizing edge crossings and/or cluster/edge crossings is NP-hard for clustered level graphs.

Then we have demonstrated a new method for the application of crossing reduction heuristics to clustered level graphs. We have proven theoretically as well as experimentally that our heuristic improves previous heuristics in terms of crossings, with a negligible increase of running time. Our scheme for extending one-sided two-level crossing reduction algorithms to clustered level graphs is optimal in the sense that it does not introduce new unnecessary crossings. For the clustered one-sided two-level crossing reduction problem an optimal algorithm could be obtained by extending any optimal two-level crossing reduction algorithm for level graphs.

Afterwards, we analyzed the constrained one-sided two-level crossing reduction problem that occurred as a subproblem of clustered crossing reduction. We developed a new algorithm based on the barycenter heuristic that is fast and simple and has good quality as well. This algorithm leads to fewer crossings then previous simple extensions of one-sided two-level crossing reduction algorithms for constraints. It runs in quadratic time and is significantly faster than existing more complex heuristics, while in practice it delivers nearly the same quality. For further improvement, it would be desirable to reduce the running time of the algorithm to less than quadratic time. For this a more efficient traversal of violated constraints would be helpful.

Finally, we have investigated the new problem of clustered level planarity. We have presented a new algorithm for planarity testing of elementary clustered k-level graphs that runs in $\mathcal{O}(k|V|)$ time. Our algorithm also delivers a clustered level planar embedding, if one exists. Further investigations are desired for level graphs which are not elementary. It is not clear if this problem can be solved in polynomial time.

Bibliography

[1] J. Abello and J. Korn. MGV: A system for visualizing massive multidigraphs. *IEEE Transactions on Visualization and Computer Graphics*, 8:21–38, 2002.

[2] L. Auslander and S. V. Parter. On imbedding graphs in the plane. *Journal of Mathematics and Mechanics*, 10(3):517–523, 1961.

[3] W. Bachl. Entwicklung und Implementierung eines hierarchischen Spring Embedders. Diplomarbeit, Fakultät für Mathemtik und Informatik, Universität Passau, 1995.

[4] C. Bachmaier, F. J. Brandenburg, and M. Forster. Track planarity testing and embedding. In P. van Emde Boas, J. Pokorný, M. Bieliková, and J. Štuller, editors, *Proceedings Software Seminar: Theory and Practice of Informatics, SOFSEM 2004*, volume 2, pages 3–17. MatFyzPress, 2004.

[5] W. Barth, M. Jünger, and P. Mutzel. Simple and efficient bilayer cross counting. In Kobourov and Goodrich [142], pages 130–141.

[6] O. Bastert and C. Matuszewski. Layered drawings of digraphs. In Kaufmann and Wagner [138], chapter 5, pages 87–120.

[7] V. Batagelj, D. Keržič, and T. Pisanski. Automatic clustering of languages. *Computational Linguistics*, 18(3):60–74, 1992.

[8] M. Y. Becker and I. Rojas. A graph layout algorithm for drawing metabolic pathways. *Bioinformatics*, 17(5):461–467, 2001.

[9] B. Berger and P. Shor. Approximation algorithms for the maximum acyclic subgraph problem. In *Proceedings ACM-SIAM Symposium on Discrete Algorithms, SODA 1990*, pages 236–243, 1990.

[10] F. Bertault and M. Miller. An algorithm for drawing compound graphs. In Kratochvíl [144], pages 197–204.

[11] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.

[12] J. M. Boyer, P. F. Cortese, M. Patrignani, and G. di Battista. Stop minding your P's and Q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. Technical Report RT-DIA-83-2003, Dipartimento di Informatica e Automazione, Università degli Studi di Roma Tre, 2003.

[13] J. M. Boyer, P. F. Cortese, M. Patrignani, and G. di Battista. Stop minding your P's and Q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. In Liotta [156], pages 25–36.

[14] J. M. Boyer and W. Myrvold. Stop minding your P's and Q's: A simplified $\mathcal{O}(n)$ planar embedding algorithm. In *Proceedings ACM-SIAM Symposium on Discrete Algorithms, SODA 1999*, pages 140–146, 1999.

[15] J. M. Boyer and W. Myrvold. On the cutting edge: Simplified $\mathcal{O}(n)$ planarity by edge addition. Submitted for publication. Preprint available at http://www.pacificcoast.net/~lightning/planarity.pdf, October 2004.

[16] F. J. Brandenburg. Layout graph grammars: the placement approach. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 144–156. Springer, 1991.

[17] F. J. Brandenburg. Designing graph drawings by layout graph grammars. In Tamassia and Tollis [220], pages 416–427.

[18] F. J. Brandenburg, editor. *Proceedings Graph Drawing, GD 1995*, volume 1027 of *Lecture Notes in Computer Science*. Springer, 1996.

[19] F. J. Brandenburg, D. Eppstein, M. T. Goodrich, S. G. Kobourov, G. Liotta, and P. Mutzel. Selected open problems in graph drawing. In Liotta [156], pages 515–539.

[20] F. J. Brandenburg, M. Forster, A. Pick, M. Raitner, and F. Schreiber. BioPath – Visualization of biochemical pathways. In *Proceedings German Conference on Bioinformatics, GCB 2001*, pages 11–15, 2001.

[21] F. J. Brandenburg, M. Forster, A. Pick, M. Raitner, and F. Schreiber. BioPath. In Mutzel et al. [172], pages 455–456.

[22] F. J. Brandenburg, M. Forster, A. Pick, M. Raitner, and F. Schreiber. BioPath – Exploration and visualization of biochemical pathways. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 215–236. Springer, 2004.

[23] F. J. Brandenburg, B. Gruber, M. Himsolt, and F. Schreiber. Automatische Visualisierung biochemischer Information. In R. Hofestädt, editor, *Proceedings of the Workshop Molekulare Bioinformatik*, pages 24–38. GI Jahrestagung, Shaker Verlag, 1998.

[24] U. Brandes, T. Dwyer, and F. Schreiber. Visualizing related metabolic pathways in two and a half dimensions (long paper). In Liotta [156], pages 111–122.

[25] U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In Mutzel et al. [172], pages 31–44.

[26] R. Brockenauer and S. Cornelsen. Drawing clusters and hierarchies. In Kaufmann and Wagner [138], chapter 8, pages 193–227.

[27] C. Buchheim, M. Jünger, and S. Leipert. Improving Walker's algorithm to run in linear time. In Kobourov and Goodrich [142], pages 344–353.

[28] C. Buchheim, M. Jünger, and S. Leipert. A fast layout algorithm for k-level graphs. In Marks [158].

[29] A. L. Buchsbaum, M. T. Goodrich, and J. R. Westbrook. Range searching over tree cross products. In M. Paterson, editor, *Proceedings European Symposium on Algorithms, ESA 2000*, volume 1879 of *Lecture Notes in Computer Science*, pages 120–131. Springer, 2000.

[30] A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *Proceedings ACM-SIAM Symposium on Discrete Algorithms, SODA 2000*, pages 566–575, 2000.

[31] M. J. Carpano. Automatic display of hierarchized graphs for computer aided decision analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 10(11):705–715, 1980.

[32] R. Castelló, R. Mili, and I. G. Tollis. An algorithmic framework for visualizing statecharts. In Marks [158], pages 139–149.

[33] R. Castelló, R. Mili, and I. G. Tollis. A framework for the static and interactive visualization of statecharts. *Journal of Graph Algorithms and Applications*, 6(3):313–351, 2002.

[34] T. Catarci. The assignment heuristic for crossing reduction in biparite graphs. In *Proceedings Allerton Conference on Comunication, Control, and Computing*, 1988.

[35] T. Catarci. The assignment heuristic for crossing reduction. *IEEE Transactions on Systems, Man, and Cybernetics*, 25(3):515–521, 1995.

[36] C. Chambers, J. Dean, and D. Grove. A framework for selective recompilation in the presence of complex intermodule dependencies. In *Proceedings International Conference on Software Engineering, ICSE 1995*, pages 221–230, 1995.

[37] M. Chandramouli and A. A. Diwan. Upward numbering testing for triconnected graphs. In Brandenburg [18], pages 140–151.

[38] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *Journal of Computer and System Sciences*, 30:54–76, 1985.

[39] M. Chrobak and G. Kant. Convex grid drawings of 3-connected planar graphs. *International Journal of Computational Geometry and Applications*, 7(3):211–223, 1997.

[40] M. Chrobak and T. H. Payne. A linear-time algorithm for drawing a planar graph on a grid. *Information Processing Letters*, 54(4):241–246, 1995.

[41] J. O. Clark and D. A. Holton. *A First Look at Graph Theory*. World Scientific, 1991.

[42] E. G. Coffman and R. L. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, 1:200–213, 1972.

[43] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[44] S. Cornelsen and D. Wagner. Completely connected clustered graphs. In *Workshop on Graph-Theoretic Concepts in Computer Science, WG 2003*, volume 2880 of *Lecture Notes in Computer Science*, pages 168–179. Springer, 2003.

[45] E. Dahlhaus. A linear time algorithm to recognize clustered planar graphs and its parallelization. In C. L. Lucchesi and A. V. Moura, editors, *Theoretical Informatics: Latin American Symposium, LATIN '98*, volume 1380 of *Lecture Notes in Computer Science*, pages 239–248. Springer, 1998.

[46] H. de Fraysseix, J. Pach, and R. Pollack. Small sets supporting fáry embeddings of planar graphs. In *Proceedings ACM Symposium on Theory of Computing, STOC 1988*, pages 426–433, 1988.

[47] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[48] H. de Fraysseix and P. Rosenstiehl. A depth-first-search characterization of planarity. *Annals of Discrete Mathematics*, 13:75–80, 1982.

[49] H. de Fraysseix and P. Rosenstiehl. A characterization of planar graphs by trémaux orders. *Combinatorica*, 5(2):127–135, 1985.

[50] C. Demetrescu and I. Finocchi. Break the "right" cycles and get the "best" drawing. In B. E. Moret and A. V. Goldberg, editors, *Proceedings International Workshop on Algorithms Engineering and Experiments, ALENEX 2000*, pages 171–182, 2000.

[51] C. Demetrescu and I. Finocchi. Removing cycles for minimizing crossings. *ACM Journal of Experimental Algorithmics*, 6, 2001.

[52] G. di Battista, editor. *Proceedings Graph Drawing, GD 1997*, volume 1353 of *Lecture Notes in Computer Science*. Springer, 1997.

[53] G. di Battista, W. Didimo, and A. Marcandalli. Planarization of clustered graphs (extended abstract). In Mutzel et al. [172], pages 60–74.

[54] G. di Battista, P. Eades, H. de Fraysseix, P. Rosenstiehl, and R. Tamassia, editors. *Proceedings Graph Drawing, GD 1993*, 1993. ftp://ftp.cs.brown.edu/pub/gd94/gd-92-93/gd93-v2.ps.Z.

[55] G. di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[56] G. di Battista and E. Nardelli. Hierarchies and planarity theory. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(6):1035–1046, 1988.

[57] G. di Battista and R. Tamassia. Algorithms for plane representations of acyclic digraphs. *Theoretical Computer Science*, 61:175–198, 1988.

[58] G. di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25(5):956–997, 1996.

[59] G. di Battista, R. Tamassia, and I. G. Tollis. Constrained visibility representations of graphs. *Information Processing Letters*, 41:1–7, 1992.

[60] M. Dickerson, D. Eppstein, M. T. Goodrich, and J. Y. Meng. Confluent drawings: Visualizing non-planar diagrams in a planar way. In Liotta [156], pages 1–12.

[61] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir. A compound graph layout algorithm for biological pathways. In Pach and Shahrokhi [179]. To appear.

[62] S. Dresbach. A new heuristic layout algorithm for directed acyclic graphs. In U. Derigs, A. Bachem, and A. Drexl, editors, *Proceedings of the International Conference on Operations Research, OR'94*, pages 121–126. Springer, 1995.

[63] V. Dujmović, M. R. Fellows, M. T. Hallett, M. Kitching, G. Liotta, C. McCartin, N. Nishimura, P. Ragde, F. A. Rosamond, M. Suderman, S. H. Whitesides, and D. R. Wood. On the parameterized complexity of layered graph drawing. In F. M. auf der Heide, editor, *Proceedings European Symposium on Algorithms, ESA 2001*, volume 2161 of *Lecture Notes in Computer Science*, pages 488–499. Springer, 2001.

[64] T. Dwyer and P. Eckersley. Wilmascope – an interactive 3D graph visualization system. In Mutzel et al. [172], pages 442–443.

[65] P. Eades, Q. Feng, and H. Nagamochi. Drawing clustered graphs on an orthogonal grid. *Journal of Graph Algorithms and Applications*, 3(4):3–29, 1999.

[66] P. Eades and Q.-W. Feng. Drawing clustered graphs on an orthogonal grid. In di Battista [52], pages 146–157.

[67] P. Eades and Q.-W. Feng. Multilevel visualization of clustered graphs. In North [176], pages 101–112.

[68] P. Eades, Q.-W. Feng, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. Technical Report 96-2, University of Newcastle, 1996.

[69] P. Eades, Q.-W. Feng, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. In North [176], pages 113–128.

[70] P. Eades and M. L. Huang. Navigating clustered graphs using force-directed methods. *Journal of Graph Algorithms and Applications*, 4(3):157–181, 2000.

[71] P. Eades and D. Kelly. Heuristics for reducing crossings in 2-layered networks. *Ars Combinatoria*, 21(A):89–98, 1986.

[72] P. Eades, T. Lin, and X. Lin. Two tree drawing conventions. *International Journal of Computational Geometry and Applications*, 3(2):133–153, 1993.

[73] P. Eades and X. Lin. A new heuristic for the feedback arc set problem. *Australian Journal of Combinatorics*, 12:15–26, 1995.

[74] P. Eades, X. Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[75] P. Eades, X. Lin, and R. Tamassia. An algorithm for drawing a hierarchical graph. *International Journal of Computational Geometry and Applications*, 6(2):145–156, 1996.

[76] P. Eades, B. D. McKay, and N. C. Wormald. On an edge crossing problem. In *Proceedings Australian Computer Science Conference*, pages 327–334, 1986.

[77] P. Eades and K. Sugiyama. How to draw a directed graph. *Journal of Information Processing*, 13(4):424–437, 1990.

[78] P. Eades and S. H. Whitesides. Drawing graphs in two layers. *Theoretical Computer Science*, 131:361–374, 1994.

[79] P. Eades and N. C. Wormald. The median heuristic for drawing 2-layers networks. Technical Report 69, Department of Computer Science, University of Queensland, 1986.

[80] P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11:379–403, 1994.

[81] M. Eiglsperger, M. Siebenhaller, and M. Kaufmann. An efficient implementation of sugiyama's algorithm for layered graph drawing. In Pach and Shahrokhi [179]. To appear.

[82] D. Eppstein, M. Goodrich, and J. Meng. Confluent layered drawings. In Pach and Shahrokhi [179]. To appear.

[83] S. Even. *Graph Algorithms*, chapter 7, pages 148–191. Computer Science Press, 1979.

[84] S. Even and R. E. Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2:339–344, 1976.

[85] T. Feder, A. Meyerson, R. Motwani, L. O'Callaghan, and R. Panigrahy. Representing graph metrics with fewest edges. In *Proceedings Symposium on Theoretical Aspects of Computer Science, STACS 2003*, volume 2607 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 2003.

[86] Q.-W. Feng. *Algorithms for Drawing Clustered Graphs.* PhD thesis, Department of Computer Science and Software Engineering, University of Newcastle, 1997.

[87] Q.-W. Feng, R. F. Cohen, and P. Eades. How to draw a planar clustered graph. In *Proceedings Conference on Computing and Combinatorics, COCOON 1995*, volume 959 of *Lecture Notes in Computer Science*, pages 21–31. Springer, 1995.

[88] Q.-W. Feng, R. F. Cohen, and P. Eades. Planarity for clustered graphs (extended abstract). In P. Spirakis, editor, *Proceedings European Symposium on Algorithms, ESA 1995*, volume 979 of *Lecture Notes in Computer Science*, pages 213–226. Springer, 1995.

[89] I. Finocchi. Layered drawings of graphs with crossing constraints. In J. Wang, editor, *Proceedings Conference on Computing and Combinatorics, COCOON 2001*, volume 2108 of *Lecture Notes in Computer Science*, pages 357–367. Springer, 2001.

[90] A. Formella and J. Keller. Generalized fisheye views of graphs. In Brandenburg [18], pages 242–253.

[91] M. Forster. Applying crossing reduction strategies to layered compound graphs. In Kobourov and Goodrich [142], pages 276–284.

[92] M. Forster. A fast and simple heuristic for constrained two-level crossing reduction. In Pach and Shahrokhi [179]. To appear.

[93] M. Forster and C. Bachmaier. Clustered level planarity. In P. van Emde Boas, J. Pokorný, M. Bieliková, and J. Štuller, editors, *Proceedings Software Seminar: Theory and Practice of Informatics, SOFSEM 2004*, volume 2932 of *Lecture Notes in Computer Science*, pages 218–228. Springer, 2004.

[94] M. Forster, A. Pick, M. Raitner, F. Schreiber, and F. J. Brandenburg. The system architecture of the BioPath system. *In Silico Biology*, 2(3):415–426, 2002.

[95] M. Fröhlich and M. Werner. The graph visualization system daVinci – a user interface for applications. Technical Report 5/94, Department of Computer Science, University of Bremen, 1994.

[96] C. Friedrich and P. Eades. The Marey graph animation tool demo. In Marks [158], pages 396–406.

[97] C. Friedrich and F. Schreiber. Flexible layering in hierarchical drawings with nodes of arbitrary size. In *Proceedings of the 27th conference on Australasian computer science*, pages 369–376, 2004.

[98] I. Fáry. On straight line representing of planar graphs. *Acta Scientiarum Mathematicarum Szeged*, 11:229–233, 1948.

[99] G. W. Furnas. Generalized fisheye views. In *Proceedings ACM Conference on Human Factors in Computing Systems, CHI 1986*, pages 16–23, 1986.

[100] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[101] E. R. Gansner, S. C. North, and K.-P. Vo. DAG – a program that draws directed graphs. *Software – Practice and Experience*, 18(11):1047–1062, 1988.

[102] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[103] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[104] B. Genc and U. Dogrusoz. A constrained, force-directed layout algorithm for biological pathways. In Liotta [156], pages 314–319.

[105] W. Günther, R. Schönfeld, B. Becker, and P. Molitor. *k*-layer straightline crossing minimization by speeding up sifting. In Marks [158], pages 253–258.

[106] E. Godehardt. *Graphs as Structural Models*, volume 4 of *Advances in System Analysis.* Vieweg, 1988.

[107] A. J. Goldstein. An efficient and constructive algorithm for testing whether a graph can be embedded in a plane. Technical Report Contract No. NONR 1858-(21), Department of Mathematics, Princeton University, 1963.

[108] C. Gutwenger, M. Jünger, S. Leipert, P. Mutzel, M. Percan, and R. Weiskircher. Advances in c-planarity testing of clustered graphs. In Kobourov and Goodrich [142], pages 220–235.

[109] F. Harary. Determinants, permanents and bipartite graphs. *Mathematical Magazine*, 42:146–148, 1969.

[110] F. Harary and A. Schwenk. A new crossing number for bipartite graphs. *Utilitas Mathematica*, 1:203–209, 1972.

[111] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[112] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[113] D. Harel. An algorithm for straight-line drawings of planar graphs. *Algorithmica*, 20(2):119–135, 1998.

[114] R. Hassin and S. Rubinstein. Approximations for the maximum acyclic subgraph problem. *Information Processing Letters*, 51(3):133–140, 1994.

[115] P. Healy and A. Kuusik. The vertex-exchange graph: A new concept for multi-level crossing minimisation. In Kratochvíl [144], pages 205–216.

[116] L. S. Heath and S. V. Pemmaraju. Recognizing leveled-planar dags in linear time. In Brandenburg [18], pages 300–311.

[117] L. S. Heath and S. V. Pemmaraju. Stack and queue layouts of directed acyclic graphs: Part II. *SIAM Journal on Computing*, 28(5):1588–1626, 1999.

[118] L. S. Heath and A. L. Rosenberg. Laying out graphs using queues. *SIAM Journal on Computing*, 21(5):927–958, 1992.

[119] T. Hickl. *Rechtwinkliges Layout von hierarchisch strukurierten Graphen.* Dissertation, Fakultät für Mathemtik und Informatik, Universität Passau, 1995.

[120] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the JACM*, 21(4):549–568, 1974.

[121] M. L. Huang and P. Eades. A fully animated interactive system for clustering and navigating huge graphs. In Whitesides [233], pages 374–383.

[122] P. Hue, M. Schaefer, and D. Stefankovic. Train tracks and confluent drawings. In Pach and Shahrokhi [179]. To appear.

[123] J. W. II. A node-positioning algorithm for general trees. *Software – Practice and Experience*, 20(7):685–705, 1990.

[124] J. A. Illingworth. Integration & compartmentation of metabolism. http://www.bmb.leeds.ac.uk/illingworth/metabol/2120lec2.htm, November 2004.

[125] ILOG. JViews demos. http://www.ilog.de/products/jviews/demos/, November 2004.

[126] M. Jünger, E. K. Lee, P. Mutzel, and T. Odenthal. A polyhedral approach to the multi-layer crossing number problem. In di Battista [52], pages 13–24.

[127] M. Jünger and S. Leipert. Level planar embedding in linear time. In Kratochvíl [144], pages 72–81.

[128] M. Jünger and S. Leipert. Level planar embedding in linear time (full version). Technical Report 99.374, Mathematisch-Naturwissenschaftliche Fakultät der Universität zu Köln, 1999.

[129] M. Jünger and S. Leipert. Level planar embedding in linear time. *Journal of Graph Algorithms and Applications*, 6(1):67–113, 2002.

[130] M. Jünger, S. Leipert, and P. Mutzel. Level planarity testing in linear time. In Whitesides [233], pages 224–237.

[131] M. Jünger, S. Leipert, and P. Mutzel. Level planarity testing in linear time (full version). Technical Report 99.369, Mathematisch-Naturwissenschaftliche Fakultät der Universität zu Köln, 1999.

[132] M. Jünger and P. Mutzel. Exact and heuristic algorithms for 2-layer straightline crossing minimization. In Brandenburg [18], pages 337–348.

[133] M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1(1):1–25, 1997.

[134] R. Kaas. A branch and bound algorithm for the acyclic subgraph problem. *European Journal of Operations Research*, 8(335–362), 1981.

[135] G. Kant. *Algorithms for Drawing Planar Graphs*. PhD thesis, University of Utrecht, 1993.

[136] P. D. Karp and S. M. Paley. Automated drawing of metabolic pathways. In H. Lim, C. Cantor, and R. Bobbins, editors, *Proceedings of the 3rd International Conference on Bioinformatics and Genome Research*, pages 225–238, 1994.

[137] R. Karp. Reducibility among combinatorical problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, 1972.

[138] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer, 2001.

[139] K. Kaugars, J. Reinefelds, and A. Brazma. A simple algorithm for drawing large graphs on small screens. In Tamassia and Tollis [220], pages 278–281.

[140] P. N. Klein and J. H. Reif. An efficient parallel algorithm for planarity. In *Proceedings IEEE Symposium on Foundations of Computer Science, FOCS 1986*, pages 465–477. IEEE Computer Society Press, 1986.

[141] P. N. Klein and J. H. Reif. An efficient parallel algorithm for planarity. *Journal of Computer and System Sciences*, 37(2):190–246, 1988.

[142] S. G. Kobourov and M. T. Goodrich, editors. *Proceedings Graph Drawing, GD 2002*, volume 2528 of *Lecture Notes in Computer Science*. Springer, 2002.

[143] S. G. Kobourov and R. Yusufov. Visualizing large graphs with compound-fisheye views and treemaps. In Pach and Shahrokhi [179]. To appear.

[144] J. Kratochvíl, editor. *Proceedings Graph Drawing, GD 1999*, volume 1731 of *Lecture Notes in Computer Science*. Springer, 1999.

[145] M. R. Laguna and R. Martí. GRASP and path relinking for 2-layer straight line crossing minimization. *INFORMS Journal on Computing*, 11(1):44–52, 1999.

[146] M. R. Laguna, R. Martí, and V. Valls. Arc crossing minimization in hierarchical digraphs with tabu search. *Computers and Operations Research*, 24(12):1175–1186, 1997.

[147] W. Lai. *Building Interactive Diagram Applications*. PhD thesis, Department of Computer Science, University of Newcastle, 1993.

[148] S. Leipert. *Level Planarity Testing and Embedding in Linear Time*. Dissertation, Mathematisch-Naturwissenschaftliche Fakultät der Universität zu Köln, 1998.

[149] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Theory of Graphs, International Symposium, Rome*, pages 215–232. Gordon Breach, 1967.

[150] T. Lengauer. Hierarchical planarity testing algorithms. *Journal of the ACM*, 36:474–509, 1989.

[151] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley & Sons, 1990.

[152] P. M. Lester. Digital hegemony: The clash between words and pictures. `http://commfaculty.fullerton.edu/lester/writings/murspeech.html`, 1996.

[153] T. Lin and P. Eades. Integration of declarative and algorithmic approaches for layout creation. In Tamassia and Tollis [220], pages 376–387.

[154] X. Lin. *Analysis of Algorithms for Drawing Graphs*. PhD thesis, Department of Computer Science, University of Queensland, 1992.

[155] X. Lin. On the computational complexity of edge concentration. *Discrete Applied Mathematics*, 101(1-3):197–205, 2000.

[156] G. Liotta, editor. *Proceedings Graph Drawing, GD 2003*, volume 2912 of *Lecture Notes in Computer Science*. Springer, 2004.

[157] P. C. Liu and R. C. Geldmacher. On the deletion of nonplanar edges of a graph. In *Proceedings Southeastern Conference on Combinatorics, Graph Theory, and Computing*, pages 727–738, 1977.

[158] J. Marks, editor. *Proceedings Graph Drawing, GD 2000*, volume 1984 of *Lecture Notes in Computer Science*. Springer, 2001.

[159] R. Martí and M. Laguna. Heuristics and meta-heuristics for 2-layer straight line crossing minimization. Technical report, Department of Statistics and Operations Research, University of Valencia, 1997.

[160] R. Martí and M. Laguna. Heuristics and meta-heuristics for 2-layer straight line crossing minimization. *Discrete Applied Mathematics*, 127(3):665–678, 2003.

[161] C. Matuszewski, R. Schönfeld, and P. Molitor. Using sifting for *k*-layer straightline crossing minimization. In Kratochvíl [144], pages 217–224.

[162] C. L. McCreary, C. L. Combs, D. H. Gill, and J. V. Warren. An automated graph drawing system using graph decomposition. In di Battista et al. [54], pages 119–120. ftp://ftp.cs.brown.edu/pub/gd94/gd-92-93/gd93-v2.ps.Z.

[163] C. L. McCreary, F.-S. Shieh, and H. Gill. GQ: A graph drawing system using graph-grammar parsing. In Tamassia and Tollis [220], pages 270–273.

[164] G. Melancon and I. Herman. DAG drawing from an information visualization perspective. Technical Report INS-R9915, Centrum voor Wiskunde en Informatica, Amsterdam, 1999.

[165] P. Mendes. Advanced visualization of metabolic pathways in PathDB. In *Proceedings of the 8th Conference on Plant and Animal Genome*, 2000.

[166] E. B. Messinger, L. A. Rowe, and R. R. Henry. A divide-and-conquer algorithm for the automatic layout of large directed graphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(1):1–12, 1991.

[167] E. Mäkinen. Experiments of drawing 2-level hierarchical graphs. Technical Report A-1988-1, Department of Computer Science, University of Tampere, 1988.

[168] E. Mäkinen. A note on the median heuristic for drawing bipartite graphs. Technical Report A-1988-4, Department of Computer Science, University of Tampere, 1988.

[169] E. Mäkinen and M. Sieranta. Genetic algorithms for drawing bipartite graphs. Technical Report A-1994-1, Department of Computer Science, University of Tampere, 1994.

[170] P. Mutzel. *The Maximum Planar Subgraph Problem*. Dissertation, Mathematisch-Naturwissenschaftliche Fakultät der Universität zu Köln, 1994.

[171] P. Mutzel. An alternative method to crossing minimization on hierarchical graphs. In North [176], pages 318–333.

[172] P. Mutzel, M. Jünger, and S. Leipert, editors. *Proceedings Graph Drawing, GD 2001*, volume 2265 of *Lecture Notes in Computer Science*. Springer, 2002.

[173] H. Nagamochi and K. Kuroya. Convex drawing for c-planar biconnnected clustered graphs. In Liotta [156], pages 369–380.

[174] F. J. Newbery. Edge concentration: A method for clustering directed graphs. In *Proceedings 2nd International Workshop on Software Configuration Management*, pages 76–85, 1989.

[175] T. Nishizeki and N. Chiba. *Planar Graphs: Theory and Algorithms*, volume 32 of *Annals of Discrete Mathematics*. North Holland, 1988.

[176] S. North, editor. *Proceedings Graph Drawing, GD 1996*, volume 1190 of *Lecture Notes in Computer Science*. Springer, 1997.

[177] S. C. North. Drawing ranked digraphs with recursive clusters. In di Battista et al. [54]. ftp://ftp.cs.brown.edu/pub/gd94/gd-92-93/gd93-v2.ps.Z.

[178] S. C. North and G. Woodhull. Online hierarchical graph drawing. In Mutzel et al. [172], pages 232–246.

[179] J. Pach and F. Shahrokhi, editors. *Proceedings Graph Drawing, GD 2004*, Lecture Notes in Computer Science. Springer, 2004. To appear.

[180] F. N. Paulisch. *The Design of an Extendible Graph Editor*, volume 704 of *Lecture Notes in Computer Science*. Springer, 1993.

[181] F. N. Paulisch and W. Tichy. EDGE: An extendible graph editor. *Software – Practice and Experience*, 20(S1):63–88, 1990.

[182] H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In di Battista [52], pages 248–261.

[183] H. C. Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing*, 13:501–516, 2002.

[184] M. Raitner. Dynamic tree cross products. In *Proceedings International Symposium on Algorithms and Computation, ISAAC 2004*, LNCS. Springer, 2004. To appear.

[185] M. Raitner. Maintaining hierarchical graph views for dynamic graphs. Technical Report MIP-0403, Fakultät für Mathemtik und Informatik, Universität Passau, 2004.

[186] M. Raitner. Visual navigation of compound graphs. In Pach and Shahrokhi [179]. To appear.

[187] B. Randerath, E. Speckenmeyer, E. Boros, P. Hammer, A. Kogan, K. Makino, B. Simeone, and O. Cepek. A satisfiability formulation of problems on level graphs. Rutcor Research Report RRR 40-2001, Rutgers Center for Operations Research, Rutgers University, 2001.

[188] E. Reingold and J. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981.

[189] P. Rosenstiehl and R. E. Tarjan. Rectilinear planar layouts and bipolar orientations of planar graphs. *Discrete & Computional Geometry*, 1(4):343–353, 1986.

[190] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Software – Practice and Experience*, 17(1):61–76, 1987.

[191] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings IEEE/ACM International Conference on CAD*, pages 42–47, 1993.

[192] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2nd edition, 2004.

[193] G. Sander. Graph layout through the VCG tool. In Tamassia and Tollis [220], pages 194–205.

[194] G. Sander. A fast heuristic for hierarchical manhattan layout. In Brandenburg [18], pages 447–458.

[195] G. Sander. Layout of compound directed graphs. Technical Report A/03/96, Universität Saarbrücken, 1996.

[196] G. Sander. *Visualisierungstechniken für den Compilerbau*. Dissertation, Universität Saarbrücken, 1996.

[197] G. Sander. Graph layout for applications in compiler construction. *Theoretical Computer Science*, 217:175–214, 1999.

[198] M. Sarkar and M. H. Brown. Graphical fisheye views. *Communications of the ACM*, 37(12):73–84, 1994.

[199] W. Schnyder. Embedding planar graphs on the grid. In *Proceedings ACM-SIAM Symposium on Discrete Algorithm, SODA 1990*, pages 138–148, 1990.

[200] F. Schreiber. *Visualisierung biochemischer Reaktionsnetze*. Dissertation, Fakultät für Mathemtik und Informatik, Universität Passau, 2001.

[201] F. Schreiber. High quality visualization of biochemical pathways in BioPath. *In Silico Biology*, 2(2):59–73, 2002.

[202] F. Schreiber. Comparison of metabolic pathways using constraint graph drawing. In Y.-P. P. Chen, editor, *APBC*, volume 19 of *CRPIT*, pages 105–110, Adelaide, Australia, 2003. Australian Computer Society.

[203] F. Schreiber. Visual comparison of metabolic pathways. *Journal of Visual Languages and Computing*, 14(4):327–340, 2003.

[204] F. Shahrokhi, O. Sýkora, L. A. Székely, and I. Vrt'o. On bipartite drawings and the linear arrangement problem. In *Proceedings of the International Workshop in Implementation of Functional Languages, IFL'97*, volume 1467 of *Lecture Notes in Computer Science*, pages 55–68. Springer, 1997.

[205] F.-S. Shieh and C. L. McCreary. Directed graphs drawing by clan-based decomposition. In Brandenburg [18], pages 472–482.

[206] F.-S. Shieh and C. L. McCreary. Clan-based incremental drawing. In Marks [158], pages 384–395.

[207] W.-K. Shih and W.-L. Hsu. A simple test for planar graphs. In *Proceedings Workshop on Discrete Mathematics and Algorithms*, pages 110–122, 1993.

[208] W.-K. Shih and W.-L. Hsu. A new planarity test. *Theoretical Computer Science*, 223(1–2):179–191, 1999.

[209] M. Sirava, T. Schäfer, M. Eiglsperger, M. Kaufmann, O. Kohlbacher, E. Bornberg-Bauer, and H.-P. Lenhof. Biominer—modeling, analyzing, and visualizing biochemical pathways and networks. *Bioinformatics*, 18(Suppl. 2):219–230, 2002.

[210] M. Stallman, F. Brglez, and D. Ghosh. Heuristics and experimental design for bigraph crossing number minimization. In M. T. Goodrich and C. C. McGeoch, editors, *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation, ALENEX 1999*, volume 1619 of *Lecture Notes in Computer Science*, pages 74–93. Springer, 1999.

[211] S. K. Stein. Convex maps. In *Proceedings American Mathematical Society*, volume 2, pages 464–466, 1951.

[212] E. Steinitz and H. Rademacher. *Vorlesungen über die Theorie der Polyeder*. Springer, 1934.

[213] K. Sugiyama. *Graph Drawing and Applications for Software and Knowledge Engineers*, volume 11 of *Software Engineering and Knowledge*. World Scientific, 2002.

[214] K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):876–892, July 1991.

[215] K. Sugiyama and K. Misue. A generic compound graph visualizer/manipulator: D-ABDUCTOR. In Brandenburg [18], pages 500–503.

[216] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[217] Sun Microsystems. Java 2 platform standard edition 5.0 API specification. `http://java.sun.com/j2se/1.5.0/docs/api/`, 2004.

[218] R. Tamassia, G. di Battista, and C. Batini. Automatic graph drawing and readybility of diagrams. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):61–79, 1988.

[219] R. Tamassia and I. G. Tollis. A unified approach to visibility representations of planar graphs. *Discrete & Computional Geometry*, 1(4):321–341, 1986.

[220] R. Tamassia and I. G. Tollis, editors. *Proceedings Graph Drawing, GD 1994*, volume 894 of *Lecture Notes in Computer Science*. Springer, 1995.

[221] Tom Sawyer Software. Image gallery. `http://www.tomsawyer.com/gallery/gallery.php`, November 2004.

[222] N. Tomii, Y. Kambayashi, and S. Yajima. On planarization of 2-level graphs. *Papers of Technical Group on Electronic Computers, IECEJ*, EC77-38:1–12, 1977.

[223] W. T. Tutte. Convex representations of graphs. In *Proceedings London Mathematical Society, Third Series*, volume 10, pages 304–320, 1960.

[224] W. T. Tutte. How to draw a graph. In *Proceedings London Mathematical Society, Third Series*, volume 13, pages 743–768, 1963.

[225] V. Valls, R. Martí, and P. Lino. A branch and bound algorithm for minimizing the number of crossing arcs in bipartite graphs. *Journal of Operational Research*, 90:303–319, 1996.

[226] J. van Helden. Cholesterol biosynthesis. Personal Communication, Université Libre de Bruxelles, 2004.

[227] V. Waddle. Graph layout for displaying data structures. In Marks [158], pages 241–252.

[228] V. Waddle and A. Malhotra. An $e \log e$ line crossing algorithm for levelled graphs. In Kratochvíl [144], pages 59–71.

[229] K. Wagner. Bemerkungen zum Vierfarbenproblem. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 46:26–32, 1936.

[230] J. N. Warfield. Crossing theory and hierarchy mapping. *IEEE Transactions on Systems, Man, and Cybernetics*, 7(7):502–523, 1977.

[231] M. E. Watkins. A special crossing number for bipartite graphs: A research problem. *Annals of New York Academy of Sciences*, 175:405–410, 1970.

[232] C. Wetherell and A. Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, 5(5):514–520, 1979.

[233] S. H. Whitesides, editor. *Proceedings Graph Drawing, GD 1998*, volume 1547 of *Lecture Notes in Computer Science*. Springer, 1998.

[234] S. G. Williamson. Embedding graphs in the plane – algorithmic aspects. *Annals of Discrete Mathematics*, 6:349–384, 1980.

[235] A. Yamaguchi and A. Sugimoto. An approximization algorithm for the two-layered graph drawing problem. In T. Asano, H. Imai, D. T. Lee, S. Nakano, and T. Tokuyama, editors, *Proceedings of the 5th International Conference on Computing and Combinatorics, COCOON'99*, volume 1627 of *Lecture Notes in Computer Science*, pages 81–91. Springer, 1999.

[236] yWorks. yEd gallery. http://www.yworks.com/en/products_yfiles_practicalinfo_gallery.htm, November 2004.

List of Figures

List of Definitions

List of Algorithms

Index