UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Dissertation

Efficient Visual Navigation of Hierarchically Structured Graphs

Marcus Raitner

February 9, 2006

Supervisor
Prof. Dr. Franz J. Brandenburg

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

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

Abstract

Visual navigation of hierarchically structured graphs is a technique for interactively exploring large graphs that possess an additional hierarchical structure. This structure is expressed in form of a recursive clustering of the nodes: in call graphs of telephone networks, for instance, the nodes are identified with phone numbers; they are clustered recursively through the implicit structure of the numbers, e. g., nodes with the same area code belong to a cluster. In order to reduce the complexity and the size of the graph, only those subgraphs that are currently needed are shown in detail, while the others are collapsed, i. e., represented by meta nodes. In such a *graph view* the subgraphs in the areas of interest are expanded furthest, whereas those on the periphery are abstracted. As the areas of interest change over time, clusters in a view need to be expanded or contracted.

First and foremost, there is need for an efficient data structure for this *graph view maintenance problem*. Depending on the admissible modifications of the graph and its hierarchical clustering, three variants have been discussed in the literature: in the *static* case, everything is fixed; in the *dynamic graph* variant, only edges of the graph can be inserted and deleted; finally, in the *dynamic graph and tree* variant the graph additionally is subject to node insertions and deletions and the clustering may change through splitting and merging of clusters. We introduce a new variant, *dynamic leaves*, which is based on the dynamic graph variant, but additionally allows insertion and deletion of graph nodes, i. e., leaves of the hierarchy.

So far efficient data structures were known only for the static and the dynamic graph variant, i. e., neither the nodes of the graph nor the clustering could be modified. As this is unsatisfactory in an interactive editor for hierarchically structured graphs, we first generalize the approach of Buchsbaum et. al (*Proc. 8th ESA*, vol. 1879 of *LNCS*, pp. 120–131, 2000), in which graph view maintenance is formulated as a special case of *range searching over tree cross products*, to the new dynamic leaves variant. This generalization builds on a novel technique of superimposing a search tree over an ordered list maintenance structure. With an additional factor of roughly $\mathcal{O}(\log n/\log\log n)$, this is the first data structure for the problem of graph view maintenance where the node set is dynamic.

Visualizing the expanding and contracting appropriately is the second challenge. We propose a local update scheme for the algorithm of Sugiyama and Misue (IEEE Trans. on Systems, Man, and Cybernetics **21** (1991) 876–892) for drawing compound digraphs. The layered drawings that it produces have many applications ranging from biochemical pathways to UML diagrams. Modifying the intermediate results of every step of the original algorithm locally, the update scheme is more efficient than re-applying the entire algorithm after expansion or contraction. As our experimental results on randomly generated graphs show, the average time for updating the drawing is around 50 % of the time for redrawing for dense graphs and below 20 % for sparse graphs. Also, the performance gain is not at the expense of quality as regards the area of the drawing, which increases only insignificantly, and the number of crossings, which is reduced. At the same time, the locality of the updates preserves the user's mental map of the graph: nodes that are are not affected stay on the same level in the same relative order and expanded edges take the same course as the corresponding contracted edge; furthermore, expansion and contraction are visually inverse.

Finally, our new data structure and the update scheme are combined into an interactive editor and viewer for compound (di-)graphs. A flexible and extensible software architecture is introduced that lays the ground for future research. It employs the well-known Model-View-Controller (MVC) paradigm to separate the abstract data from its presentation. As a consequence, the purely combinatorial parts, i. e., the compound (di-)graph and its views, are reusable without the editor front-end. A proof-of-concept implementation based on the proposed architecture shows its feasibility and suitability.

Contents

Abstract v

1 Introduction 1
- 1.1 Terminology . . . 11
- 1.2 Graph View Maintenance . . . 18
  - 1.2.1 Problem Definition . . . 19
  - 1.2.2 Previous Solutions . . . 20
  - 1.2.3 Dynamic Tree Cross Products . . . 23
- 1.3 Visual Navigation . . . 23
  - 1.3.1 Drawing Ordinary Graphs . . . 23
  - 1.3.2 Drawing Hierarchically Structured Graphs . . . 25
  - 1.3.3 Dynamic Aspects of Graph Drawing . . . 26
  - 1.3.4 Dynamic Layered Drawings of Compound Digraphs 30
- 1.4 Interactive Editor and Viewer . . . 31
  - 1.4.1 Key Features . . . 32
  - 1.4.2 Previous Solutions . . . 33

2 Dynamic Tree Cross Products 35
- 2.1 Tree Cross Products . . . 36
- 2.2 Naive Approach . . . 39
- 2.3 Static Trees . . . 39
  - 2.3.1 The Two-Dimensional Case . . . 40
  - 2.3.2 Higher Dimensions . . . 46
- 2.4 Dynamization: Inserting and Deleting Leaves . . . 48
  - 2.4.1 The Two-Dimensional Case . . . 48
  - 2.4.2 Higher Dimensions . . . 57
- 2.5 Application to Graph View Maintenance . . . 57
  - 2.5.1 Modeling . . . 57
  - 2.5.2 Complexity . . . 59
  - 2.5.3 Comparison . . . 63
- 2.6 Summary . . . 65

3 Visualization **67**
3.1 Static Layered Drawings of Compound Digraphs 68
3.1.1 Step I: Hierarchization 69
3.1.2 Step II: Normalization 71
3.1.3 Step III: Crossing Reduction 72
3.1.4 Step IV: Metric Layout 75
3.2 Expansion 80
3.2.1 Step I: Hierarchization 81
3.2.2 Step II: Normalization 84
3.2.3 Step III: Crossing Reduction 86
3.2.4 Step IV: Metric Layout 92
3.3 Contraction 93
3.4 Experimental Results 95
3.5 Summary 105

4 Architecture **107**
4.1 Design Goals 108
4.2 High-Level Architecture 110
4.3 Low-Level Architecture 112
4.3.1 Model 113
4.3.2 MVC-View 117
4.3.3 Controller 120
4.4 Use Cases 122
4.4.1 Expansion 122
4.4.2 Adding a Leaf 125
4.5 Summary 127

5 Conclusion **129**
5.1 Results 129
5.2 Open Problems and Future Work 130

Bibliography **142**

List of Figures **145**

List of Tables **147**

Index **149**

1

Introduction

Over the last decades, information technology progressively pervaded nearly all areas of everyday life. Nowadays, every phone call, every item bought in a store, every credit card transaction, every bank transfer, every visit of a web page, and every e-mail produces some sort of data. In many companies, this leads to massive amounts of data, which need to be analyzed in order to support management decisions, for instance. This data often is of relational nature, i. e., it consists of links between elements of certain sets: a phone call links two phone numbers, an item bought in a store is linked with the customer, a credit card transaction or a bank transfer links the two bank accounts, a web page is linked with the visitor's IP address, and an e-mail links the sender with the receiver. In other words, the data in any of these examples can be interpreted as a graph.

In order to get an overview of such a graph and to identify its interesting parts, a good visualization is extremely useful. In fact, the human brain can analyze a (good) picture much faster than textual information.[1] Just imagine to sit in the New York Metro and the links would be given in textual form, e. g., "line A connects W. 4th St. with Spring St." instead of the usual map; see Figure 1.1 for a clipping of it. It is, however, not visualizations per se that are superior to other forms of representation. It all depends on the quality of the drawing, which is, of course, a highly subjective measure and thus is difficult to formalize in general. For specific problems like the New York Metro map, some criteria of a good drawing are evident. Unnecessary

[1] Or, as the well-known proverb expresses it: "One picture is worth ten thousand words.", which, incidentally, often is wrongly attributed to Confucious or some other (ancient) far-east philosopher, but actually has its source in a 1927 ad by Fred R. Barnard, National Advertising Manager for the Street Railways Advertising Company; see (Hepting, 1999).

Figure 1.1: Clipping of the New York Metro map.

crossings, for instance, should be avoided, as they are misleading. Also, links should not be too long, as they are hard to follow. And the drawing should fit on a given medium, e. g., a letter-sized piece of paper.

As the amount of data increases, many visualization techniques in general and graph drawing algorithms in particular reach their limits. The telephone call graph of Germany, for instance, consists of calls between nearly 90 million phone numbers (counting both the conventional telephone network and cellular phones) (Statistisches Bundesamt Deutschland, 2004). Considering that high-end consumer displays nowadays have a resolution of $1,920 \times 1,200 = 2,304,000$ pixels, it is obvious that this graph cannot be visualized as a whole. Even if every phone number would be depicted with only one pixel, 40 such displays would have to be joined, but still there would be no space for visualizing the phone calls.

This example admittedly is extreme. Nevertheless, it shows that the "screen real estate" (Abello and Korn, 2002) is a bottleneck. This becomes an imporatant issue even for much smaller graphs such as the biochemical reaction network in an organism. In fact, Michal (1993) depicts approximately 1,500 biochemical reactions with about as many chemical substances (Schreiber, 2001, p. 4) on his famous, mostly hand-drawn wall chart; see Figure 1.2 for a small clipping of it. Although his drawing already is very compact, the chart still measures 1.4 m × 1 m. Many more reactions and substances, however, have been discovered since 1993 when the chart was published. By the time of writing, databases such as KEGG or EMP list over 20,000 entries; see (SRS). Nevertheless, it is estimated that this is only

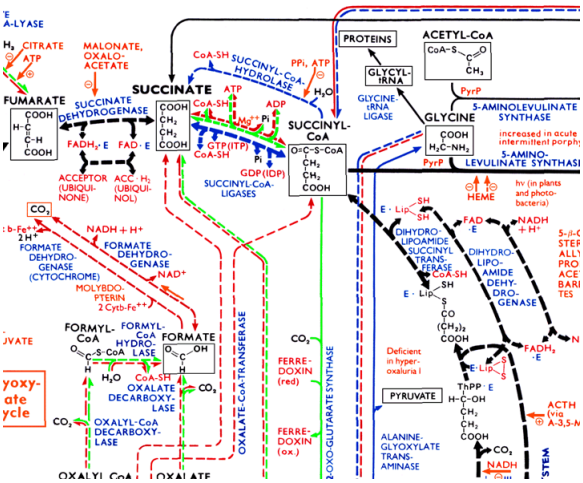

Figure 1.2: Clipping of the biochemical pathway wall chart of Michal (1993) showing part of the citric acid cycle.

a small part of all biochemical reactions in organisms (Schreiber, 2001, p. 5).

The obvious solution for visualizing such large graphs is to show only a clipping of the whole drawing and to provide a scrolling mechanism to move the clipped area. However, as only a small portion of the graph is visible at any time, exploring a large graph bears certain resemblance with an odyssey. The situation can be improved to some extent by varying also the scale of the clipped area: one can choose a large scale for an overview and then zoom into the interesting areas. In order to avoid excessive zooming, often a permanently visible, large scale overview of the whole drawing is provided additionally and the clipped area is highlighted within this overview. The problem, however, is that of losing context. Even with an additional overview, it is cumbersome to map the elements of the detailed area to their counterparts in the overview and vice versa. Incidentally, the difficulty of this task can be observed in everyday life when switching from a large scale map to a detailed one (of a particular city, for instance).

The main drawback with using different scales is that each drawing has a uniform scale. Thus, a view either focuses on some area of interest or shows the overall structure of the drawing, i. e., the context. Fisheye views (Furnas, 1986; Sarkar and Brown, 1992, 1994; Formella and Keller, 1995) or hyperbolic views (Lamping and Rao, 1996) combine both, focus and context, into one drawing. These approaches relinquish the paradigm of one uniform scale for the whole view: some areas of the drawing are shown at a large scale while others are shown in detail. More precisely, fisheye views

mimic the imaging behavior of an extremely wide-angle lens, a so-called *fisheye lens*. The entire drawing can be explored by moving the focused area. Although this non-uniform scale leads to distorted images, it has the advantage that the details are shown within their context.

Such distortion techniques, however, reach their limits if there are simply too many details. In order to fit the telephone call graph of Germany onto a contemporary computer display the scale would have to be chosen such large that everything outside the area of focus would be indistinguishable. Therefore, one major drawback of these purely geometric approaches is that they do not reduce the number of details; they only scale them differently.

Instead of choosing a larger scale for a currently uninteresting part, the overall structure of the graph usually is conveyed better by displaying this part in abstract form. On a large scale road map of Germany, for instance, even major cities like Munich or Berlin are depicted as small circles and all the streets within a particular city are hidden. For the telephone call graph of Germany, a similar abstraction can be derived from the structure of a phone number, which basically consists of an area code and the number itself, e. g., 0851 5090. If, for instance, the subgraph representing the calls between two numbers in Passau (area code 0851) is not needed in detail, the entire subgraph can be collapsed into a single meta node labeled 0851. Calls from a number with a different area code to a number in Passau (or vice versa) are no longer shown in detail. Rather, they are combined into so-called derived edges that are attached directly to the respective meta node, e. g., calls from 07531 884263 to both 0851 5093034 and 0851 5093030 are represented as a single derived edge from 07531 884263 to the collapsed meta node 0851.

Even if all of the over 5,000 area codes are collapsed, the resulting abstract call graph of Germany is still too large. Therefore, this abstraction technique can be applied recursively, for instance, by grouping the area codes according to common prefixes. The meta node for the prefix 085 thus contains, among others, the meta nodes 0851 (Passau) and 08551 (Freyung). On the highest level of abstraction, the prefix 08 comprises large parts of Bavaria, e. g., Munich with all its outskirts, Landshut, and Passau. Considering that Passau, for instance, has over 50,000 citizens, the subgraphs consisting of all phone numbers with the same area code are also too large in general and thus need to be subdivided. This could be done by grouping the phone numbers with the same area code according to their geographical location, e. g., all numbers within the same district. Also, the phone numbers of many institutions or companies often consist of some base number followed by an extension, which can be used to group them.

While for telephone call graphs this recursive partitioning is given implicitly by the structure of the phone numbers, it has to be specified explicitly for other large graphs. Biochemists already have grouped the reactions in an organism into larger units, so-called pathways, e. g., the citric acid cycle, a clipping of which is shown in Figure 1.2, or the biosynthesis of valine, which is depicted in Figure 1.3. Like in the telephone call graph example above, this grouping is carried on recursively, e. g., the biosynthesis of valine belongs to the metabolism of branched chain amino acids; furthermore, pathways are classified as anabolic, catabolic, or amphibolic. On the other hand, an elementary reaction often consists of many partial reactions (Schreiber, 2001, p. 23).

If a pathway like the biosynthesis of valine, which converts pyruvate into valine in a sequence of four elementary reactions, is not needed in full detail, it is often abbreviated as one overview reaction (Schreiber, 2001, p. 18) connecting the two key substances pyruvate and valine directly; see Figure 1.3. In BioPath (Brandenburg et al., 2003), which, incidentally, was used to create the images in Figure 1.3, this abstraction mechanism plays an important role: the user can explore the biochemical reactions stored in a database by iteratively refining overview reactions. If the present drawing shows the biosynthesis of valine in abbreviated form, the full reaction sequence can be obtained by clicking the overview reaction.[2]

In both examples, a large graph is partitioned recursively into a hierarchy of meaningful subgraphs. In order to reduce the complexity and the size of the graph, only those subgraphs that are currently needed are shown in detail, while the others are collapsed, i. e., represented by meta nodes. In such a *graph view* the subgraphs in the area of interest are expanded furthest, whereas those on the periphery are abstracted. This resembles the fisheye views as regards the concept of providing focus and context simultaneously. The technique for locally increasing or decreasing the level of detail, however, is fundamentally different: in a fisheye view this is done by non-uniform scaling, whereas in a graph view non-uniform levels of abstraction are used. A lower scale for the focused area in a fisheye

[2] Although it seems that in this example a subgraph is replaced with a meta *edge* instead of a meta *node* like in the telephone call graph example, this is not the case in the data model of BioPath. A biochemical reaction in general has more than one reactant and more than one product and thus essentially is a hyperedge. The standard technique for modeling hypergraphs as ordinary (bipartite) graphs is to replace every hyperedge with a new node and ordinary edges from each source to the new node and from the new node to each target. Therefore, the reactions in the data model of BioPath are also nodes and are replaced with meta nodes representing the overview reaction; see (Schreiber, 2001, Chap. 5) for a more detailed description of this data model.

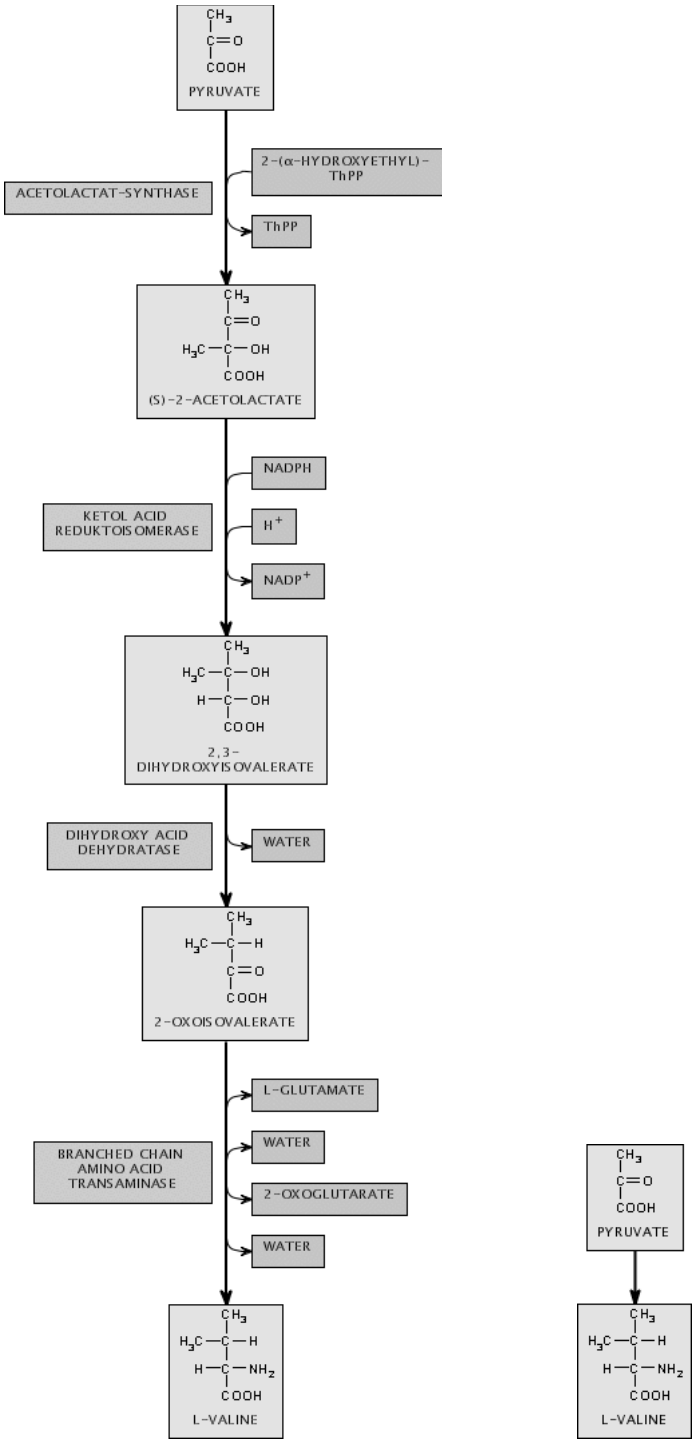

Figure 1.3: This example of a biochemical pathway (Schreiber, 2001) shows the biosynthesis of valine both as sequence elementary reactions (left) and as overview reaction (right). Both drawings have been produced with BioPath (Brandenburg et al., 2003).

view corresponds to lower level of abstraction in a graph view, i. e., the subgraphs in this area are expanded further. On the other hand, a larger scale for the surroundings corresponds to a higher level of abstraction, i. e., large subgraphs are collapsed into meta nodes.[3]

The focus of this work is the visual exploration and navigation of large hierarchically structured graphs and its hub is the concept of a graph view. Starting from an abstract view, in which large subgraphs are contracted into meta nodes, the areas of interest are explored by expanding the corresponding meta nodes *within* the context of the whole graph, i. e., the level of abstraction is decreased locally by showing the subordinate subgraphs of the expanded meta node. Expanding the meta node 08 in a view of the telephone call graph, for instance, increases the level of detail locally by showing its subordinate meta nodes, e. g., 085 and 089. Conversely, subgraphs that are no longer needed in detail can be collapsed. Compared to fisheye or hyperbolic views this approach also shows the currently interesting areas in detail, but the surroundings are abstracted, not just scaled down. Especially for larger graphs, this abstraction mechanism is superior to scaling because it emphasizes the overall structure of the periphery instead of cluttering it with unnecessary details. On the other hand, visual navigation with graph views involves some novel, challenging problems that will be addressed in this work.

First and foremost, there is need for a data structure for hierarchically structured graphs that efficiently supports the expanding and contracting of subgraphs in a graph view. Depending on the admissible modifications of the hierarchy and the graph, Buchsbaum and Westbrook (2000) differentiate between three variants of this *graph view maintenance problem*: in the *static* case, the graph and the hierarchy are both fixed; in the *dynamic graph* variant, edges can be inserted and deleted; in the *dynamic graph and tree* variant, the graph additionally is subject to node insertions and deletions and the hierarchy may change through splitting and merging of subgraphs. Splitting groups part of the subordinate subgraphs of some meta node into a new meta node; merging is the inverse operation. Suppose, for instance, that in the aforementioned biochemical pathways example so far only the whole metabolism of branched chain amino acids has been grouped as a pathway and now shall be refined further by splitting off the biosynthesis of valine. This means that the substances and reactions involved in the biosynthesis of valine are assigned to a new meta node that replaces

[3] Abello et al. (2004) combine both approaches by automatically expanding and collapsing subgraphs as the user changes the focused area.

them in the metabolism of branched chain amino acids. Buchsbaum and Westbrook (2000) and Buchsbaum et al. (2000) propose data structures for both the static and the dynamic graph variant.[4] Efficient data structures that additionally support modifications of the node set were left as an open problem (Buchsbaum and Westbrook, 2000), which is solved partially in this work.

In Chapter 2, an efficient data structure for a new variant, *dynamic leaves*, is introduced; it is based on the dynamic graph variant, but additionally allows insertion and deletion of graph nodes, i. e., leaves of the hierarchy. In contrast to the dynamic graph and hierarchy variant, it lacks splitting and merging of clusters. Thus it is adequate for dynamic graphs for which the hierarchical structure is fixed or at least changes infrequently.[5] The hierarchy of our telephone call graph example, for instance, is fixed because it is derived from the structure of telephone numbers. At the same time, this graph is highly dynamic as regards insertion and deletion of both nodes and edges because phone numbers become active or inactive and phone calls start and end. The classification of biochemical reactions into pathways is another example: new reactions and substances are discovered frequently, but it happens relatively seldom that part of the reaction network is grouped into a new pathway. Previous solutions (Buchsbaum et al., 2000; Buchsbaum and Westbrook, 2000) are no adequate model for such applications because they cannot handle graphs with a dynamic node set at all.

A data structure is a necessary precondition for the efficient visual navigation of large hierarchically structured graphs with graph views. On the other hand, it is equally important that the visual representation of the current view is adjusted efficiently after expanding and contracting. Efficiency usually refers to the time for computing the new drawing, which is indeed critical in an interactive setting. Measuring only computation time, however, is not enough, as it neglects the user's effort to recognize the new drawing. This effort is influenced not only by the traditional aesthetic criteria of graph drawing, e. g., small area, short edges, or few edge crossings, but also by dynamic aspects. It is commonly assumed that the user builds some sort of mental representation of a drawing and that a modification of the graph is easier to follow if as much as possible is preserved of this so-called *mental map* (Misue et al., 1995). The goal of our visualization therefore is not only to produce aesthetically pleasing drawings but also to adjust

[4] In Section 1.2.2, both approaches are described in more detail.

[5] Note that in theory every hierarchy can be constructed by adding and removing leaves in the correct order. Therefore, the split and merge operations could be emulated through a series of leaf operations, but this would not be very efficient.

the drawings in a way that makes it is easy to follow the expanding and contracting of subgraphs.

The dynamic aspects of graph drawing so far have been researched mostly for ordinary graphs and elementary operations like adding or removing nodes or edges (Branke, 2001).[6] The few exceptions (Huang and Eades, 1998; Eades and Huang, 2000; Dwyer and Eckersley, 2003) that try to preserve the user's mental map in the context of expanding and contracting meta nodes in graph views employ force-directed approaches. A force-directed layout algorithm mimics a physical system where the nodes of the graph repel each other and edges are springs exerting attracting forces. Starting from some initial drawing, the nodes are moved according the overall forces acting on them until some reasonably stable configuration is found. Therefore, adjusting a drawing after expanding or contracting with a force-directed algorithm is straightforward: the old drawing simply is used as start configuration. If the old drawing corresponded to some relatively stable state of the physical system and the graph was modified only locally, it is likely that the iterative movement converges against a drawing that resembles the old one.

Because of the symmetric nature of the forces, such algorithms are well suited for *undirected* graphs. For *directed* graphs, however, it is often the case that the majority of edges follows some overall direction that shall be conveyed in the drawing. In biochemical reaction networks, for instance, the directed edges encode the succession of the chemical substances and thus it is desirable to draw the edges in one preferred direction, e. g., from top to bottom like in Figure 1.3; other examples are PERT (CPM) charts in project management or UML diagrams in software engineering. The method of choice for such directed graphs is *layered drawing* based on the framework of Sugiyama et al. (1981). It consists of four phases: first, nodes are distributed on horizontal levels; second, the edges are normalized; third, the order of the nodes on each level is improved as to reduce the number of edge crossings; finally, exact coordinates are assigned to the nodes without changing the order. Both Sugiyama and Misue (1991) and Sander (1996a,b, 1999) extend and adapt this framework for hierarchically structured graphs, but the dynamic aspects of expanding and contracting in this context have not been researched so far. Sugiyama and Misue (1991) mention these operations, though it appears that they implement them by re-applying their algorithm to the entire graph.

In contrast to this naive approach, our solution, presented in Chapter 3, is an update scheme for the algorithm of Sugiyama and Misue (1991). It

[6] In Section 1.3.3 these dynamic aspects of graph drawing are described in more detail.

works on the intermediate results and auxiliary structures of the four phases of this algorithm with particular emphasis on the locality of the updates: every expand and contract operation has only a local effect. The drawing of the new graph essentially is computed on the manipulated subgraph only. This is a significant improvement for the time consuming phases of the original algorithm, such as level assignment or crossing reduction. The update scheme thus is faster than a complete relayout. In fact, experimental results on randomly generated graphs show that the running time on the average improves by a factor between two and ten depending on the density of the graph; see Section 3.4. The user's mental map is preserved by keeping all nodes not involved in the expanding or contracting on their former levels and in the same relative order.

In Chapter 4, both our data structure and the update scheme for the algorithm of Sugiyama and Misue (1991) are combined into an appropriate software architecture for the visual navigation of hierarchically structured graphs. This architecture already has been put into practice in the form of a proof-of-concept implementation (Pfeiffer, 2005; Pröpster, 2005)

In order to ensure its reusability, the data structure is coupled only loosely with the visualization and the user interface using the well-known Model-View-Controller (MVC) pattern (Buschmann et al., 1996). For each hierarchically structured graph, the proposed architecture supports arbitrarily many views, which are notified about modifications of the base graph through an Observer pattern (Gamma et al., 1995, pp. 293–303). The user thus can have many windows showing different views of the graph.

Although our update scheme already preserves the user's mental map well, the transition between the drawings before and after expanding and contracting is much easier to follow if it is animated. Therefore, the proposed architecture also supports animation, though only the straightforward, yet remarkably effective, linear interpolation has been implemented so far. Since the data structure, the drawing style, and the animation style probably will be subject to variations and improvements, particular emphasis is laid on the flexibility and extensibility in these regards.

After this motivation and high-level overview of efficient visual navigation of hierarchically structured graphs and the challenges linked with it, we first put the discussion on a more formal basis in Section 1.1. In Section 1.2, we state the problem of graph view maintenance formally and briefly describe its previous solutions. Section 1.3 introduces into graph drawing in general and layered drawing in particular. Furthermore, our notion of preserving the mental map for layered drawings of hierarchically structured graphs is presented in the context of previous approaches. Fi-

nally, we briefly describe other interactive editing and exploration tools and compare them to our architecture in Section 1.4.

1.1 Terminology

graph

Definition 1.1. A *directed graph*, short *digraph*, $G = (V, E)$ consists of a set of *nodes* V and a set of *directed edges* $E \subseteq V \times V$. In an *undirected graph*, the edges are unordered sets instead of ordered pairs, i. e., $E \subseteq \mathcal{P}_2(V)$, where $\mathcal{P}_2(V)$ denotes the set of all subsets of V with exactly two elements.

An edge $(u, v) \in E$ of a digraph $G = (V, E)$ is an *incoming edge* of node v and an *outgoing edge* of node u; both u and v are *incident* with the edge (u, v). A *path* is a sequence of nodes $u_1, u_2, \ldots, u_k$ such that $(u_i, u_{i+1}) \in E$ for $1 \leq i < k$. For $U \subseteq V$, the set of *induced edges* is defined as $E[U] = \{(u, v) \in E \mid u, v \in U\}$; the digraph $G[U] = (U, E[U])$ is called a *(node) induced subgraph* of G.

tree

Definition 1.2. A *rooted tree* is a digraph $T = (V, E)$ with one distinguished node $\text{root}(T)$, the *root* of T, such that for every other node $u \in V \setminus \{\text{root}(T)\}$ there is an unique path from $\text{root}(T)$ to u.

From the above definition it follows that every non-root node $v \in V \setminus \{\text{root}(T)\}$ in a tree $T = (V, E)$ has exactly one incoming edge (u, v); u is called the *parent* of v and denoted $\text{parent}(v)$; conversely, v is a *child* of u. Let $\text{children}(v)$ denote the set of all *children* of a node $v \in V$; a node without children is a *leaf*, whereas the others are referred to as *internal nodes*. The *descendants* of v, $\text{desc}(v)$, are all nodes u such that there is a path from v to u. Conversely, a node u on the unique path from $\text{root}(T)$ to v is said to be an *ancestor* of v. Note that v is both an ancestor and a descendant of itself. If neither u is a descendant of v nor v is a descendant of u, i. e., $u \notin \text{desc}(v)$ and $v \notin \text{desc}(u)$, the nodes u and v are termed *unrelated*. The *subtree* rooted at a node v is the subgraph of T induced by $\text{desc}(v)$, i. e., $T[\text{desc}(v)] = (\text{desc}(v), E[\text{desc}(v)])$. The *depth* of v, $\text{depth}(v)$, is the number of edges on the path from $\text{root}(T)$ to v. This implies that $\text{depth}(\text{root}(T)) = 0$. The depth of the entire tree T is defined as $\text{depth}(T) = \max_{v \in V} \text{depth}(v)$. The *height* of v, $\text{height}(v)$, is the depth of the subtree rooted at v.

compound (di-)graph

Definition 1.3. A *compound digraph* $D = (V, E, F)$ consists of nodes V, directed *inclusion edges* E, and directed *adjacency edges* F. The *inclusion digraph* $T = (V, E)$ is a rooted tree, and no adjacency edge connects a node to one of its descendants or ancestors, i. e., for every adjacency edge

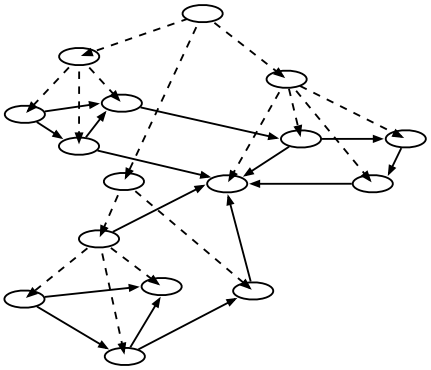

Figure 1.4: A compound digraph; the dashed edges form the inclusion tree; the solid ones are adjacency edges.

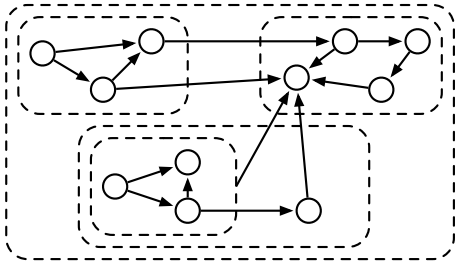

Figure 1.5: The same compound digraph as in Figure 1.4, but the inclusion tree is depicted by the inclusion of the dashed rectangles.

$(u, v) \in F$, u and v are unrelated in T. If the adjacency edges are undirected, D is called a *compound graph*.

Figure 1.4 shows an example of a compound digraph: the dashed edges form the inclusion tree and the solid ones are adjacency edges. Figure 1.5 shows an alternative method for drawing compound (di-)graphs: the leaves and the internal nodes are drawn as circles and rectangles and the inclusion tree is depicted through their geometric inclusion.

Besides compound (di-)graphs, there are other concepts for extending graphs with a hierarchical structure. Lengauer and Wanke (1988) describe a large graph in form of a special context-free *graph grammar*[7]. They define a *hierarchical graph* $\Gamma = (G_1, \ldots, G_k)$ as a sequence of ordinary graphs. The nodes of each G_i are subdivided into three classes: *pins*, *terminals*, and *non-terminals*. Each non-terminal v in G_i is associated with exactly one graph G_j with $1 \leq j < i$, which represents the production replacing v with G_j. The neighbors of each non-terminal v are labeled with the pins of the associated graph G_j, which encodes how to embed G_j within G_i after v has been re-

[7] Graph grammars (Rozenberg, 1997) can be seen as the extension of context-free word grammars to graphs. In a word grammar non-terminal symbols are replaced with a word consisting of terminal and non-terminal symbols, whereas in a graph grammar a non-terminal node is replaced with a graph that contains terminal and non-terminal nodes. The main difference, however, is that a production in a graph grammar needs additional *embedding rules*: The non-terminal node on the left side of the production in general has some incident edges and the embedding rules describe how these edges are inherited to the nodes of the right side.

placed. As every non-terminal has only one production, the language of this graph grammar has only one word, i. e., only one graph.[8] Thus, the derivation tree can also be interpreted as a hierarchical decomposition of this graph similar to the inclusion tree of a compound (di-)graph. As identical subgraphs need only be listed once, a hierarchical graph is a succinct representation of an ordinary, large graph, which can be exploited to speed-up algorithms for classical graph problems such as connectivity (Lengauer and Wanke, 1988).

Layout graph grammars (Brandenburg, 1990, 1994; Hickl, 1994) extend ordinary graph grammars by local layout specifications for the right sides of each production. The drawing of a graph is produced by applying the appropriate productions together with the specified layout instructions. Thus, the purpose of layout graph grammars is the layout of a (possibly infinite) class of graphs through the local layout of the (finite) set of productions.

Harel (1988) introduces the general concept of a *higraph* for hierarchically structuring, for instance, entity-relationship diagrams used in the conceptual specification of databases or state-transition diagrams for modeling the behaviour of reactive systems. The latter appears to be the most important application of higraphs considering that according to Harel (1988) higraphs actually evolved in the process of trying to formulate the underlying graphical principles used in *statecharts*, one particular concept for the design and specification of complex reactive systems introduced earlier by Harel (1987). A higraph consists of recursively nested nodes, so-called *blobs*, and edges connecting them. The nesting has to be acyclic, but not necessarily tree-like as in compound (di-)graphs, i. e., a blob may be contained in more than one superordinate blob. The edges may connect arbitrary blobs and thus are not restricted to unrelated nodes as in compound (di-)graphs.

Cigraphs as introduced by Lai and Eades (1996) are closely related to compound (di-)graphs. A cigraph $C = (r, I, W)$ consists of a single root node r, a possibly empty set of sub-cigraphs I, and a set of (adjacency) edges W. This recursive definition results in a tree-like nesting of cigraphs that is similar to the inclusion hierarchy of compound (di-)graphs. Also, edges are allowed only between unrelated nodes. Compared to compound (di-)graph essentially the only difference is that the set of edges is scattered across the various sub-cigraphs. More precisely, an edge connecting cigraphs C_v and C_u always belongs to the cigraph that is the nearest common ancestor of C_v and C_u in the inclusion hierarchy.

[8] A general graph grammar can produce more than one graph; its language can even be infinite.

Clustered graphs, which were introduced by Feng et al. (1995), are another popular concept for extending graphs with a hierarchical structure. A clustered graph $C = (G, T)$ consists of an ordinary base graph G and a tree T such that the leaves of T are exactly the nodes of the base graph. Each node v of T thus represents a subset $V(v)$ of the nodes of the underlying graph G defined by the leaves of the subtree rooted at v. Hence, T describes the inclusion relation between these clusters. Since the nodes of the underlying graph are the leaves of the inclusion tree T, clustered graphs can have adjacency edges only between leaves, whereas the other models allow edges between any pair of unrelated nodes (compound (di-)graphs and cigraphs) or even between any pair of nodes (higraphs).

Schreiber (2001, Sect. 5.3) adapts the general concept of a clustered graph to the special case of hierarchically structured biochemical reaction networks, which are essentially hypergraphs, i. e., each reaction is a hyperedge connecting the participating chemical substances. The standard technique for modeling a hypergraph as a bipartite graph is the following: every hyperedge is replaced with a new node and all sources and targets of the hyperedge are connected appropriately with it. As this results in two different types of nodes, the inclusion tree built on top of such a bipartite reaction network has to meet additional requirements in order to ensure that abstractions of the network (defined in terms of the clustering given by the inclusion tree) are again correct reaction networks; see Schreiber (2001, Sect. 5.3).

Shieh and McCreary (1995), McCreary et al. (1998), and Shieh and McCreary (2000) use a *clan-based decomposition* for drawing directed acyclic graphs (DAG). They show that every DAG can be decomposed into a tree of special subgraphs, so-called *clans*, such that the leaves are the nodes of the DAG and the internal nodes are complex clans, which are termed either *series* or *parallel*. Although this *parse tree* corresponds to the inclusion tree of other models, it is the result of a graph-theoretic decomposition and as such it is determined by the underlying graph as opposed to the independent inclusion trees in compound (di-)graphs or clustered graphs. Messinger et al. (1991) also partition a large digraph into subgraphs as part of a divide-and-conquer strategy for automatically drawing digraphs. This decomposition, however, is not carried on recursively, i. e., the implicitly generated inclusion tree has only depth 1.

The hub of visual navigation in this work is the concept of a graph view, which can be seen as an abstraction of a compound (di-)graph that is too large to be displayed as a whole. Since a graph view also will be a compound (di-)graph, we first introduce the notion of an upper subtree, which describes the requirements of a view's inclusion tree.

Definition 1.4. An *upper subtree* $T[U]$ of a tree $T = (V, E)$ is a subgraph induced by $U \subseteq V$ such that **upper subtree**

- $\text{root}(T) \in U$,
- for all $u \in U \setminus \text{root}(T)$: $\text{parent}(u) \in U$,
- and for all $u \in U$: either $\text{children}(u) \cap U = \emptyset$ or $\text{children}(u) \subseteq U$.

Essentially, the first two conditions imply that an upper subtree $T[U]$ of a tree $T = (V, E)$ is indeed a tree with the same root as T. The third condition is important only for our notion of a graph view, which will be an upper subtree of the inclusion tree together with appropriate adjacency edges. The darker shaded nodes of the compound digraph in Figures 1.6 and 1.8 induce an upper subtree of the inclusion tree.

Alternatively, an upper subtree of $T = (V, E)$ can be characterized by pruning T. For a node $v \in V$, *pruning* T at v means to remove all subtrees rooted at children of v. In other words, after pruning T at v, v is a leaf.

Lemma 1.5 Let $T = (V, E)$ be a tree and $U \subseteq V$ such that $T[U]$ also is a tree. Then $T[U]$ is an upper subtree of T if and only if $T[U]$ can be constructed from T by a sequence of pruning operations.

Proof. Since T is an upper subtree of itself and pruning an upper subtree of T does not violate any of the conditions in Definition 1.4, it follows by induction that any subtree that is constructed from T by a sequence of pruning operations is an upper subtree of T. Conversely, let $T[U]$ be an upper subtree of T according to Definition 1.4. Obviously, $T[U]$ is the result of pruning T at each leaf of $T[U]$. □

For $v \in V$ and $U \subseteq V$ such that $\text{root}(T) \in U$, let $\text{anc}_U(v)$ denote the *nearest ancestor* of v in U, i. e., the ancestor with largest depth in T. Note that $\text{anc}_U(v)$ is well-defined for any $v \in V$, because $\text{root}(T) \in U$. The following lemma gives yet another characterization of an upper subtree.

Lemma 1.6 Let $T = (V, E)$ be a tree and $U \subseteq V$ such that $T[U]$ also is a tree. The $T[U]$ is an upper subtree of T if and only if $\text{root}(T) \in U$ and $\text{anc}_U(v)$ is a leaf in $T[U]$ for every $v \in V$.

Proof. If $T[U]$ is an upper subtree of T, then $\text{root}(T) \in U$ by definition. Suppose that $\text{anc}_U(v)$ is an internal node of $T[U]$ for some $v \in V$; hence, it has at least one child in U and thus all children are in U because of the third condition of Definition 1.4. However, this is not possible because $\text{anc}_U(v)$ is the nearest ancestor of v in U. Conversely, the same argument shows that if $\text{root}(T) \in U$ and $\text{anc}_U(v)$ is a leaf in $T[U]$ for every $v \in V$ then

either children$(u) \subseteq U$ or children$(u) \cap U = \emptyset$. Since $T[U]$ is a tree and root$(T) \in U$, it also follows that parent$(u) \in U$ for each $u \in U$. □

As already mentioned, we define a graph view as an upper subtree of the inclusion tree $T = (V, E)$ of a compound digraph[9] $D = (V, E, F)$ with "appropriate" adjacency edges. The adjacency edges in a view should also represent adjacency edges in D between nodes that have been cut off in the course of pruning T. This means that adjacency edges in a view are not necessarily edges of D, but rather *derived edges* in the following sense:

derived edge

Definition 1.7. For a compound digraph $D = (V, E, F)$, two unrelated nodes $u, v \in V$ are connected by a *derived edge* if and only if there are nodes $u' \in \text{desc}(u)$ and $v' \in \text{desc}(v)$ such that u' and v' are connected by an adjacency edge $(u', v') \in F$.

A derived edge thus can be seen as a representative of all the edges between descendants of u and v.

Now all necessary concepts have been introduced in order to define our notion of a *graph view*:

view

Definition 1.8. A *view* of a compound digraph $D = (V, E, F)$ is a compound digraph $D[U] = (U, E[U], F\langle U \rangle)$ determined by the nodes $U \subseteq V$ such that $T[U] = (U, E[U])$ is an upper subtree of the inclusion tree $T = (V, E)$. Two nodes $u, v \in U$, $u \neq v$, are connected with a derived (adjacency) edge $(u, v) \in F\langle U \rangle$ if and only if there exists an adjacency edge $(u', v') \in F$ such that $u = \text{anc}_U(u')$ and $v = \text{anc}_U(v')$.

Intuitively, a view is constructed by collapsing the subtree (in T) of every leaf (in $T[U]$) into a single meta node. As our above definition allows derived adjacency edges only between different nodes, an adjacency edge $(u', v') \in F$ such that $\text{anc}_U(u') = \text{anc}_U(v')$, i. e., both nodes lie in the same collapsed subtree, is simply hidden. On the other hand, an adjacency edge (u', v') with $u = \text{anc}_U(u') \neq \text{anc}_U(v') = v$ is represented by the derived edge $(u, v) \in F\langle U \rangle$. Note that a derived adjacency edge $(u, v) \in F\langle U \rangle$ stands for all original adjacency edges connecting the subtrees (in T) rooted at u and v. Figure 1.7 shows an example of a view induced by the darker shaded nodes of the compound digraph in Figure 1.6. Figure 1.9 shows the same view as an inclusion diagram.

Views were first introduced by Eades and Feng (1996): for a clustered graph, a view at level i is an ordinary graph consisting of all nodes at

[9] We restrict the discussion to compound digraphs; unless explicitly mentioned, the presented results can easily be transferred to undirected compound graphs.

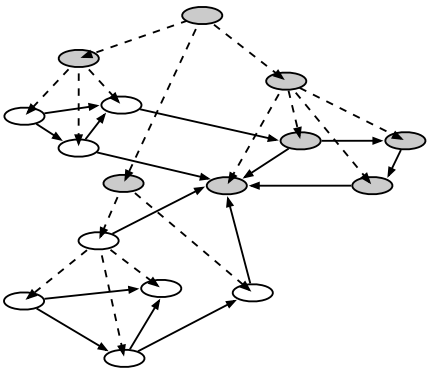

Figure 1.6: The darker shaded nodes of this compound digraph induce an upper subtree of the inclusion tree.

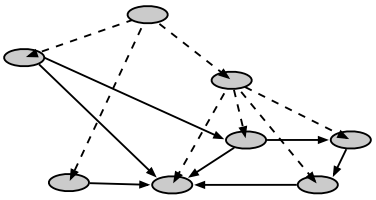

Figure 1.7: The view consisting of the darker shaded nodes of the compound digraph in Figure 1.6.

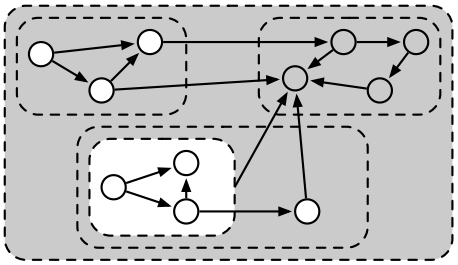

Figure 1.8: The same compound digraph as in Figure 1.6, but the inclusion tree is depicted by the inclusion of the dashed rectangles.

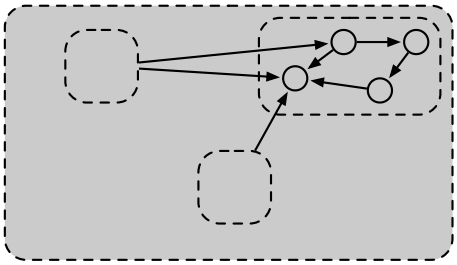

Figure 1.9: The same view as in Figure 1.7 with the inclusion hierarchy depicted as an inclusion diagram.

height i in the inclusion tree and their derived edges. The inclusion tree has to be normalized such that each inclusion edge $(\text{parent}(u), u)$ satisfies $\text{height}(\text{parent}(u)) = \text{height}(u) + 1$, which can easily be done by inserting dummy nodes. The *abridgments* of Huang and Eades (1998) and later the views of Buchsbaum and Westbrook (2000) generalize this notion: the nodes of a view no longer need to have the same height as long as the corresponding subtrees partition the set of leaves. This makes expanding and contracting of single nodes possible, whereas with the views of Eades and Feng (1996) these operations were possible only for the entire layer. Schreiber (2001, Sect. 5.3) uses the views of Buchsbaum and Westbrook (2000) for exploring hierarchically structured reaction networks through expanding and contracting of nodes.

Definition 1.8 generalizes the one of Buchsbaum and Westbrook in two ways: first, it uses compound digraphs instead of clustered graphs as underlying model, and second, the views of Buchsbaum and Westbrook are ordinary graphs, whereas ours are again compound digraphs. Incidentally, this makes them similar to the abridgments of Huang and Eades (1998) that are clustered graphs again. In other words, Definition 1.8 restricted to clustered graphs is equal to the one Huang and Eades and the leaves of one of our views together with the derived edges yield the corresponding view of Buchsbaum and Westbrook.

1.2 Graph View Maintenance

Views can be seen as an abstract representation of the underlying compound digraph. By Definition 1.8, the level of abstraction is non-uniform: some areas of interest can be displayed in more detail than the remainder. Although this abstraction is useful by itself, its true power is unleashed when the areas of interest can be chosen interactively. In other words, the view can be refined or coarsened locally when areas become more or less interesting. In this regard, a view can be compared to the well known *tree views* of file systems: initially only the topmost layer of folders is shown and the folders of interest can be expanded within the view.[10]

[10] Of course, there are no edges between the elements of a tree view, but the same metaphor of expanding and contracting locally is used.

1.2.1 Problem Definition

Given a compound digraph $D = (V, E, F)$ and a view $D[U]$, the *graph view maintenance problem* is to efficiently perform the following operations on $D[U]$:

`expand`(v) where v is a leaf in $T[U]$: refines the view at v, i. e., the result is the view $D[U']$ given by the nodes $U' = U \cup \text{children}(v)$; see Figures 1.10 and 1.11 for an example.

`contract`(v) where $\text{children}(v)$ are leaves in $T[U]$: coarsens the view at v, i. e., the result is the view $D[U']$ given by the nodes $U' = U \setminus \text{children}(v)$; this is the inverse of `expand`(v) in $D[U']$.

Note that expanding and contracting are well-defined, i. e., the set U' indeed defines a correct view (cf. Definition 1.8). This is the case if and only if $T[U']$ is an upper subtree of T. Since $D[U]$ is a correct view of D, i. e., $T[U]$ is an upper subtree of T, appending $\text{children}(v)$ to a leaf v (in $T[U]$) does not violate any of the conditions for an upper subtree (cf. Definition 1.4). Contracting is the same as pruning $T[U]$ at v, which always results in an upper subtree by Lemma 1.5.

Apart from this *static* case, there are three more dynamic variants of the *graph view maintenance problem*: in the *dynamic graph* variant, only edges can be inserted and deleted; in the *dynamic leaves* variant, also leaves of the inclusion tree can be inserted and deleted; in the *dynamic graph and tree* variant the hierarchy additionally may change through splitting and merging of subgraphs. In other words, these variants support some or all of the following operations; see Table 1.1.

`newEdge`(u, v) where $u, v \in V$ are unrelated: adds a new adjacency edge (u, v) to F.

`deleteEdge`(u, v) where $(u, v) \in F$: removes adjacency edge (u, v) from F.

`newLeaf`(u) where $u \in V$: adds a new node v to V and a new inclusion edge (u, v) to E, i. e., v becomes a child of u in the inclusion tree.

`deleteLeaf`(v) where v is a leaf in the inclusion tree: removes v from V and the inclusion edge $(\text{parent}(v), v)$ from E.

`split`(u, N) where $u \in V$ and $\text{children}(u) = N \dot\cup R$: inserts a new node v between u and N into the inclusion tree, i. e., $\text{children}(v) = N$, $\text{parent}(v) = u$, and $\text{children}(u) = R \cup v$; see Figures 1.12 and 1.13 for an example.

`merge`(v) where $v \in V$ and v is not the root of the inclusion tree T: v is deleted from V and its children are attached directly to $\text{parent}(v)$; see

Figures 1.14 and 1.15 for an example. Note that this is the inverse of split(parent(v), N), where N are v's designated children.

Any compound digraph can be constructed from scratch with an appropriate sequence of newLeaf and newEdge operations. Using also the inverse operations, i. e., deleteLeaf and deleteEdge, it follows that any compound digraph can be transformed in any other by an appropriate sequence of these four primitives. From this point of view, the operations split and merge are unneccessary. In fact, both can be simulated by the four primitive operations: for merging a node v, first the whole subtree of v with all its incident edges is deleted bottom-up and then rebuilt top-down without v, i. e., the former children of v are attached directly to parent(v); split is simulated analogously. Since this amounts to quite a lot operations and thus tends to be inefficient, it seems to be worthwhile to consider split and merge independently of the other operations.[11]

contracted edge
expanded edges

Definition 1.9. Let $(u, v) \in F\langle U\rangle$ be a derived edge in a view $D[U] = (U, E[U], F\langle U\rangle)$ such that v is a leaf in $T[U]$ and let $D[U'] = (U', E[U'], F\langle U'\rangle)$ be the view after expanding v. The *expanded edges of* (u, v) *with respect to* v are denoted by $E((u, v), v) = \{(u, v') \in F\langle U'\rangle \mid v' \in \text{children}(v)\}$. For an expanded edge $(u, v') \in E((u, v), v)$, the edge (u, v) is called the corresponding *contracted edge*. The definition for expanding u instead of v is symmetrical.

In Figure 1.10, for instance, only one derived edge is incident with the node to be expanded. As shown in Figure 1.11, this edge generates two expanded edges.[12] Chapter 2 will show that the problem of efficiently performing an expand(v) operation on a view essentially is dominated by the problem of finding the expanded edges of every derived edge (u, v) or (v, u).

1.2.2 Previous Solutions

Although the operations expand and contract have been introduced together with compound digraphs by Sugiyama and Misue (1991), data structures to support them efficiently have been considered only recently. Buchsbaum and Westbrook (2000) stated the problem (for clustered graphs) formally and described its variations except for the dynamic leaves variant,

[11] Note that conversely newLeaf(u) and deleteLeaf(u) can be implemented as split($u, \emptyset$) and merge(u), respectively.

[12] We will omit the "with respect to v" whenever the expanded node is clear from the context.

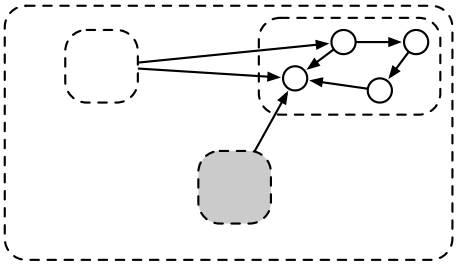

Figure 1.10: View of the compound graph in Figure 1.8 before expanding the highlighted node.

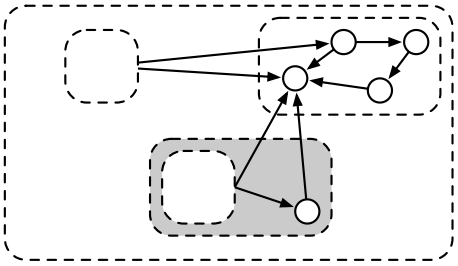

Figure 1.11: Expanding means to insert children of the highlighted node with appropriate derived edges.

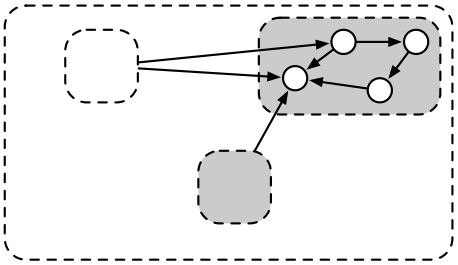

Figure 1.12: The compound digraph before splitting off the highlighted nodes.

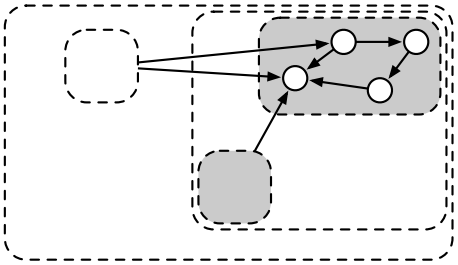

Figure 1.13: Splitting inserts a new node between the selected nodes and their former parent.

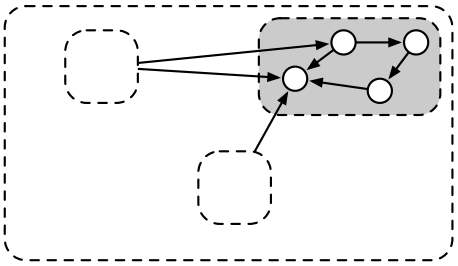

Figure 1.14: The compound digraph before merging the highlighted node.

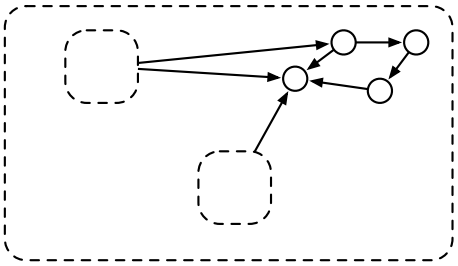

Figure 1.15: Merging removes the selected node and attaches its children directly to its parent.

Table 1.1: Operations supported in the four variants of the graph view maintenance problem.

	`expand`	`contract`	`newEdge`	`deleteEdge`	`newLeaf`	`deleteLeaf`	`split`	`merge`
Static	×	×						
Dynamic graph	×	×	×	×				
Dynamic leaves	×	×	×	×	×	×		
Dynamic graph and hierarchy	×	×	×	×	×	×	×	×

which has been introduced in (Raitner, 2004c). The solution proposed by Buchsbaum and Westbrook (2000), however, is rather eager: all possible derived edges are calculated in advance and linked appropriately in order to provide both `expand` and `contract` in optimal time, i. e., linear in the number of nodes and edges that are modified. As this approach is quite space-intensive, they improve it by employing a well-known tree compression technique of Harel and Tarjan (1984) to reduce the depth of the inclusion tree and thus the number of potential derived edges. Insertion and deletion of edges is also supported, yet an efficient data structure for the more dynamic variants was left as an open problem.

Another solution for both the static and the dynamic graph variant is described in Buchsbaum et al. (2000). They express the graph view maintenance problem (for clustered graphs) as a special case of *range searching over tree cross products*. A tree cross product consists of hyperedges connecting the nodes of d disjoint trees $T_1, \ldots, T_d$. In this context, range searching means to determine all hyperedges connecting the subtrees of a given set of tree nodes $(u_1, \ldots, u_d)$. Using two identical copies of the inclusion tree of a clustered graph as T_1 and T_2, the problem of graph view maintenance can be described as a two-dimensional tree cross product. This solution also allows contracting in optimal time. With an additional factor of $\log \log n$ (where n is the number of nodes of the clustered graph), expanding becomes slightly more inefficient compared to the eager approach of Buchsbaum and Westbrook (2000). Again, inserting and deleting edges is supported efficiently, but the set of nodes is fixed, leaving the more dynamic variants as open problems.

1.2.3 Dynamic Tree Cross Products

In Chapter 2, the solution of Buchsbaum et al. (2000) for range searching over tree cross products is generalized as regards insertion and deletion of leaves in any of the trees $T_1, \ldots, T_d$ that form the tree cross product. Using basically the same modeling as Buchsbaum et al. (2000), this yields a solution for the dynamic leaves variant of graph view maintenance (for compound digraphs). The extra cost for this dynamization is roughly a factor of $\mathcal{O}(\log n / \log\log n)$, where n is the number of nodes of the compound digraph. Our approach thus partially solves an open problem of Buchsbaum and Westbrook (2000), but an efficient data structure for the dynamic graph and hierarchy variant, i. e., with splitting and merging of clusters, still remains an open problem.

1.3 Visual Navigation

As our goal is an interactive editor for compound (di-)graphs that supports visual exploration through expanding or contracting, we also have to be concerned with the adequate visualization of compound (di-)graphs. The field of graph drawing, to which this problem naturally belongs, has matured over the last decades, which is manifested in several books (Kaufmann and Wagner, 2001; Di Battista et al., 1998) and in a wealth of graph drawing software (see the book by Jünger and Mutzel (2004)). Although the majority of results in this field consider only ordinary graphs, the more complex graph models such as clustered graphs or compound (di-)graphs have been studied as well.

1.3.1 Drawing Ordinary Graphs

In its most basic formulation, graph drawing means to map the nodes of a graph to points in the two-dimensional plane and to connect them with straight lines representing the edges, which is why this *drawing convention* is referred to as *straight-line*. An obvious generalization is to draw the edges as a sequences of straight-line segments, i. e., as *polylines*.

While drawing conventions specify the boundary conditions that must be fulfilled always, a "good" drawing also complies with certain *aesthetic criteria* such as small area, short edges, few edge crossings, or few edge bends; see (Fleischer and Hirsch, 2001) for a more detailed overview. Since most of these criteria conflict, graph drawing algorithms have to find a suitable balance between them. To this end, it has been shown empirically

(Purchase et al., 1997; Purchase, 1998) that both minimizing edge crossings and minimizing edge bends significantly improve human understanding.

Various *drawing styles* have been proposed in the literature, among which methods using some sort of physical analogy have become relatively popular for three reasons according to Brandes (2001, p. 71):

> First of all, they are very intuitive because layout is related to the everyday experience of the surrounding physical world. Secondly, their basic instances are comparatively easy to understand and to program. The threshold to get started is thus very low. And finally, they often yield fairly satisfactory results on medium-sized graphs up to around 50 vertices.

In force-directed algorithms (Eades, 1984), for instance, nodes repel each other while edges are interpreted as springs[13] exerting attracting forces on the nodes they connect. The algorithm then iteratively moves some nodes according to the overall force acting on each of them until a reasonably stable configuration is found. Instead of moving the nodes to reduce the forces locally, it has also been attempted to reduce the internal energy of the entire physical system (Kamada and Kawai, 1988). See (Brandes, 2001) for a more detailed discussion of graph drawing methods based on physical analogies.

Due to the symmetric nature of the forces along the edges, force-directed algorithms are best suited for undirected graphs. For general digraphs, on the other hand, *layered drawings* have become popular[14]. In this drawing style, it is assumed that the majority of the edges follows some overall direction that shall be emphasized by drawing as many edges as possible in one specific direction, e. g., form top to bottom. Nearly all algorithms for layered drawings are somehow based on the classical four-step method described in the seminal paper of Sugiyama et al. (1981). In its first step, the *cycle removal*, the digraph is made acyclic by temporarily reversing as few edges as possible. (At the end of the whole algorithm the affected edges are reversed again.) It follows the *layer assignment* in which the nodes are distributed on horizontal layers such that all edges point downward. Often the following steps assume that edges occur only between adjacent layers, which is why edges spanning more layers, so-called *long-span edges*, are broken up by inserting dummy nodes on the spanned layers. In the *crossing reduction* step, the order of the nodes on every layer is determined such that

[13] This is why such algorithms often are referred to as *spring embedders*.

[14] As Bastert and Matuszewski (2001) point out, most graph drawing tools provide an implementation of this drawing style.

the number of edge crossings is kept small. The final *coordinate assignment* step calculates real coordinates within this ordering such that, for instance, nodes are distributed evenly and balanced among their neighbors and long span edges are kept as straight as possible. For a more comprehensive overview of layered drawings, see (Bastert and Matuszewski, 2001).

Besides these general purpose graph drawing methods, there also exist algorithms to satisfy the specific needs of restricted graph classes such as trees or planar graphs; see (Müller-Hahnemann, 2001) and (Weiskirchner, 2001), respectively. Especially for planar graphs, not only straight-line drawings but also *orthogonal drawings* have been studied a lot. One major motivation for this drawing style has been to improve the *angular resolution* of a drawing, which is defined as the smallest angle that two edges form at a common node. In an orthogonal drawing, a good angular resolution is achieved by drawing all edges as axis-parallel paths; see (Eiglsperger et al., 2001) for a more detailed overview.

1.3.2 Drawing Hierarchically Structured Graphs

Drawing clustered graphs or compound (di-)graphs is more complicated because of the additional inclusion hierarchy. One viable solution would be to use an additional dimension. The three-dimensional drawing style for clustered graphs proposed by Eades and Feng (1996), for instance, draws the underlying ordinary graph in two dimensions and uses the third one for the hierarchy.

In contrast to these *multilevel drawings*, in which both the inclusion tree and the underlying graph are equally visible, most drawing algorithms for compound (di-)graphs or clustered graphs—especially two-dimensional ones—emphasize the underlying graph by drawing the inclusion tree as nested regions. In other words, the inclusion edges are not drawn explicitly; rather, they are visualized through the geometric nesting of the corresponding regions. As this complicates the recognition of the inclusion hierarchy, most drawing algorithms use very simple cluster regions such as rectangles or circles.

More formally, the drawing conventions for the inclusion hierarchy of this *nested drawing* style are the following:

(ND 1) Every node is drawn as a two-dimensional region, e. g., as an axis-parallel rectangle.

(ND 2) The region of every non-root node is entirely contained within the region of its parent.

(ND 3) The regions of unrelated nodes are disjoint.

Nearly all graph drawing styles for ordinary graphs have been extended to nested drawings of clustered graphs or compound (di-)graphs. Besides force-directed approaches (Huang and Eades, 1998; Huang et al., 1998; Eades and Huang, 2000), which basically add additional forces to keep a node within its parent's region, the notion of planarity also has been extended to clustered graphs and various drawing algorithms for such *clustered planar* graphs have been proposed (Feng et al., 1995; Eades et al., 1996a; Eades and Feng, 1997; Eades et al., 1999; Nagamochi and Kuroya, 2003). Layered drawings also have been considered: Sugiyama and Misue (1991) and later Sander (1996a,b, 1999) adapt the classical four-step algorithm of Sugiyama et al. (1981) to *general* compound digraphs, whereas the framework of Castelló et al. (2000, 2002), which is also based on (Sugiyama et al., 1981), is specifically tailored to *statecharts*. Bertault and Miller (1999) present a general framework for drawing compound digraphs that can be parameterized with drawing algorithms for ordinary graphs: for each internal node, a suitable drawing algorithm for the subgraph induced by its children can be chosen. Brockenauer and Cornelsen (2001) give a more detailed overview of drawing hierarchically structured graphs.

1.3.3 Dynamic Aspects of Graph Drawing

In our chosen scenario the user explores a large compound (di-)graph by iteratively refining the interesting nodes in an initially abstract view. Conversely, the subgraphs that are not—or no longer—needed in detail can be contracted. An efficient data structure for the problem of graph view maintenance, doing all the work in the background, is a necessary precondition for this approach, but in our interactive setting its visual aspects must not be neglected.

These dynamic aspects of graph drawing have been researched so far mostly for ordinary graphs and elementary operations like adding or removing nodes or edges. Even in this simpler setting, the obvious solution of re-applying a standard layout algorithm does not yield the desired result as Misue et al. (1995) point out:

> Most automatic layout facilities take a purely combinatorial description of a graph and produce a layout of the graph; these methods are called 'layout creation' methods. For interactive systems, another kind of layout is needed: a facility which can adjust a layout after a change is made by the user or by the application. (...) The use of a layout creation method for layout adjustment may totally rearrange the layout (...).

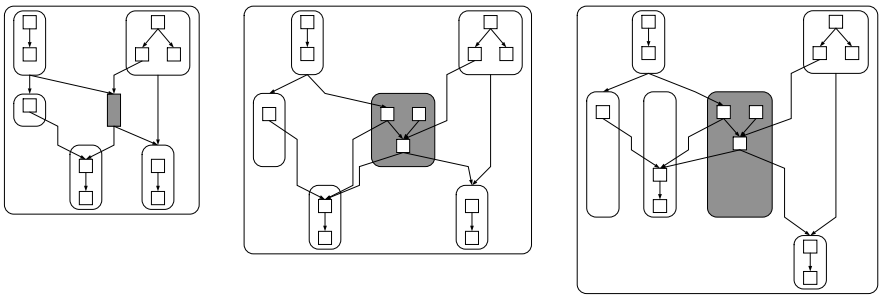

Figure 1.16: Both the middle and the right drawing are possible after expanding the darker shaded node in the left drawing. It is, however, much easier to keep track of the expanding if the middle one follows left drawing instead of the right one.

The middle and the right drawing[15] in Figure 1.16, for instance, are both possible layouts after the shaded node in the left drawing has been expanded. It is, however, much easier to keep track of the common nodes and edges if the left drawing of Figure 1.16 is followed by the middle one instead of the right one. This holds especially when the drawings are not presented side by side as in Figure 1.16, but sequentially as in an interactive exploration tool.

Besides the traditional aesthetic criteria for drawing graphs nicely, e. g., few edge crossings, few bends, small area, or short edges, it is therefore important that the user is able to follow the expansion or contraction of a node (or any other modification) visually. This is only possible if the effort to re-familiarize with the new drawing is reasonably small. More precisely, it is assumed that the user builds some sort of mental representation of the drawing, a so-called *mental map*. Since this is a demanding task, taking a considerable amount of time for larger graphs, the expanding or contracting should preserve as much of the user's mental map as possible. In other words, we have to find a good compromise between the usual aesthetic criteria and the preservation of the user's mental map.

Various approaches have been made to capture the nebulous concept of a mental map more formally. One straightforward, but also drastic, method for preserving the mental map is to prohibit any movement of nodes (Branke, 2001). Although this almost perfectly preserves the mental map, it is not flexible enough: it necessarily leads to unpleasant drawings

[15] As opposed to the majority of drawings in this work, those in Figure 1.16 are not hand-drawn. Rather, they have been created automatically with a proof-of-concept implementation of our layout algorithm (Pröpster, 2005), which also is an important component of an interactive editing and exploration tool (Pfeiffer, 2005); see Chapter 4.

with, for instance, many edge crossings. For expanding a node, fixing the node positions is impracticable altogether because of the additional space needed for the new children. Even if the nodes in the vicinity of the change are allowed to move, as suggested in Böhringer and Paulisch (1990), this approach still is too inflexible in general.

Therefore, most authors measure the similarity of the two drawings with some sort of distance metric and then try to find a reasonable balance between the common aesthetic criteria and this additional similarity measure. Such a metric can be based on the absolute coordinates of the nodes. The similarity then is measured as the sum of the Eucledian distances (or some other metric for the two-dimensional plane) that the nodes were moved as, for instance, in (Bridgeman and Tamassia, 1998) or (Lyons et al., 1998). However, Branke (2001) mentions the following drawback:

> A general problem (...) is that operations like translation, rotation or scaling will clearly yield large dissimilarity values and indicate a large change in the layout, while the user would easily recognize the old drawing.

Although this problem can be alleviated to some extent by aligning the drawings with a point set matching algorithm before computing the metric (Bridgeman and Tamassia, 1998), it seems to be more promising to preserve only the *relative positions* instead of the absolute coordinates. With their *orthogonal ordering* model Misue et al. (1995) take into account the relative ordering of every pair of nodes: if u lies, for instance, north-east of v in the previous drawing, this should be the case in the new drawing too. Instead of just counting how many pairs of nodes violate this condition, Bridgeman and Tamassia (1998) suggest to measure the similarity more gradually as the angle formed by the straight lines through the nodes u and v in both drawings (and sum over all pairs of nodes u and v); they further refine this measure with a weighted version.

Alternatively, the similarity can be expressed in terms of *proximity relations*, i. e., nodes close together should stay close together. Misue et al. (1995), for instance, formalize proximity as the preservation of the *nearest neighbor graph* of a drawing, which has an edge from every node to its nearest neighbor. Also many other proximity relations have been suggested; see (Branke, 2001) for a detailed overview.

For the most part, dynamic graph drawing has been researched only for ordinary graphs and the supported operations are restricted to insertion and deletion of nodes and edges; see (Branke, 2001). For visual navigation, however, it is required to expand and contract nodes in compound (di-)graphs,

which is different in both regards. To this end, the most flexible dynamic graph drawing algorithm probably is the client-server model of North and Woodhull (2001), which provides a dynamic variant of the layered drawing algorithm of Sugiyama et al. (1981). The clients are allowed to insert, delete, or modify subgraphs of the shared graph. After each operation (or after a sequence of them) a client can request a new layout of the modified graph from the server. Although inserting and deleting subgraphs could be useful for simulating the expanding and contracting in views, the approach of North and Woodhull (2001) works only for ordinary DAGs and not for compound digraphs.

Schreiber (2001) tries to preserve the user's mental map using constraints such as node u lies left of node v for visualizing the expanding and contracting of nodes in views of hierarchically structured biochemical reaction networks. These views, however, are ordinary graphs as defined by Buchsbaum and Westbrook (2000) and thus this approach is of limited use for our views, which are compound digraphs.

Although there are several algorithms for statically drawing compound (di-)graphs or clustered graphs (cf. Section 1.3.2), the dynamic aspects of visual navigation have been neglected in most cases. This is even more surprising when considering that the basic operations `expand` and `contract` already have been introduced by Sugiyama and Misue (1991), yet they seem to be implemented through a complete relayout. Up to now, only Huang and Eades (1998) briefly describe a (force-directed) graph drawing algorithm for the visual navigation of clustered graphs; it also redraws the entire view, but tries to preserve the user's mental map by animating the transition. WILMASCOPE of Dwyer and Eckersley (2003) provides *three-dimensional* visualization of compound (di-)graphs, also with a force-directed algorithm with animated expanding and contracting of nodes.

Note that force-directed algorithms are already equipped with an intrinsic animation: since the nodes are moved iteratively, one can start from the previous layout and simply display every step of the algorithm instead of the final result. With other drawing styles, animation is not so easy, yet more advanced approaches have been proposed for animating the transition between two drawings of the same graph (Friedrich and Eades, 2002). Though helpful, such animation techniques reach their limits when the two drawings differ too much.

1.3.4 Dynamic Layered Drawings of Compound Digraphs

Among the various graph drawing styles, we focus on *layered drawings* for two major reasons. First, layered drawing is well investigated not only for ordinary graphs, but also for compound digraphs. Second, it is the method of choice for digraphs; it is used for graphs as diverse as biochemical pathways in bioinformatics, PERT (CPM) charts in project management, UML diagrams in software engineering, or entity relationship diagrams in database design.

As already mentioned in Section 1.3.2, two major approaches exist for layered drawings of compound digraphs: the algorithm of Sugiyama and Misue (1991), and the more recent approach of Sander (1999, 1996a,b), which differs from the former, among other things, by its method for distributing the nodes into layers. Although the so-called *global layering* of Sander produces more compact (and supposedly more pleasant) layouts it is more difficult to update after an expand or contract than the *local layering* of Sugiyama and Misue. Therefore, the approach of Sugiyama and Misue (1991) has been chosen as basis.

Considering that the operations `expand` and `contract` are semantically inverse, the following property is highly desirable for a good user experience in our intended interactive setting:

(MM 0) The drawing after expanding (contracting) a node and immediately contracting (expanding) it again is the same as before expanding (contracting) it, i. e., `expand` and `contract` are *visually inverse*.

While this property is not specific to any particular drawing style, we now state more precisely our notion of preserving the mental map in the chosen context of layered drawings. Consider a compound digraph $D = (V, E, F)$ and a view $D[U]$ that already has a layered drawing; assume that the leaf v is expanded resulting in the new view $D[U']$, where $U' = U \cup \text{children}(v)$. The degree to which a layered drawing for $D[U']$ preserves the user's mental map is measured on the basis of the following properties:

(MM 1) Each node $u \in U$, remains on its layer.

(MM 2) Two nodes $u, w \in U$ on the same layer do not change their order.

(MM 3) For each edge (u, v) (or (v, u)), the expanded edges with respect to v take the "same course" as the original contracted edge. This means that for each layer that the contracted edge intersects, all nodes and edges that are left (right) of this intersection stay left (right) of all expanded edges in the drawing of $D[U']$.

Note that (MM 1) and (MM 2) can be seen as an adaptation of the orthogonal ordering model of Misue et al. (1995) for layered drawings: (MM 1) assures that the above-below relations of all pairs of common nodes are preserved, while (MM 2) takes care of the left-right relations within the same layer. [16]

For contracting a node, (MM 1) and (MM 2) are formulated symmetrically by switching the roles of $D[U]$ and $D[U']$. The symmetric version of (MM 3), however, states that each contracted edge takes the same course as the corresponding expanded edges, which is only well-defined if all expanded edges take the same course before contracting. Unfortunately, this is not true in general for the drawings the algorithm of Sugiyama and Misue (1991) produces. Nevertheless, it turns out that our method of updating the drawing after expanding a node always will fulfill condition (MM 3), which is why we will allow contracting only for nodes that have been expanded with our method before.

The solution, presented in Chapter 3, is an update scheme for the intermediate results and auxiliary graphs of the algorithm of Sugiyama and Misue (1991). The initial view is drawn with the original algorithm in order to initialize all these intermediate results. After each `expand` or `contract` operation, the intermediate results of the previous run are adjusted according to our update scheme. This has two major advantages. First, it is more efficient than redrawing the entire view because the expensive steps of the algorithm are effectively restricted to the modified part; in fact, only the last step, the coordinate assignment, affects the entire view. Second, it preserves the user's mental map by satisfying (MM 0)–(MM 3), i. e., all common nodes stay on their layers with their relative order unchanged, expanded edges take the same course as the corresponding contracted edge, and expanding and contracting are visually inverse.

1.4 Interactive Editor and Viewer

These two major aspects of visual navigation of hierarchically structured graphs, namely an efficient, dynamic data structure for the problem of graph view maintenance and a drawing algorithm that appropriately visualizes the expanding and contracting, finally shall be combined in an interactive editor and viewer for compound (di-)graphs. Therefore, a suit-

[16] Although (MM 2) does not make any statement for nodes on *different* layers, their left-right relations will be preserved implicitly in most cases due to the way the coordinate assignment works.

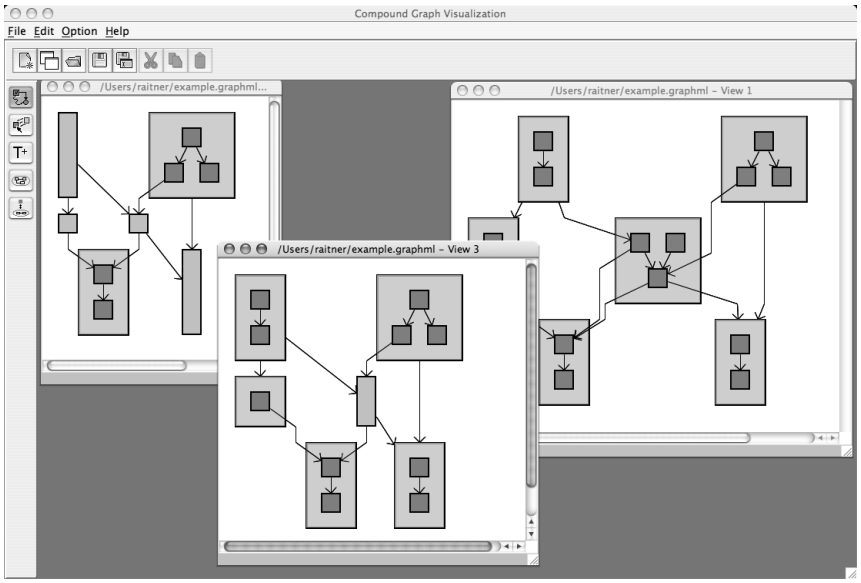

Figure 1.17: Screenshot of the proof-of-concept implementation by Pfeiffer (2005) and Pröpster (2005) based on the proposed software architecture. Three different views of the graph in Figure 1.16 are shown.

able software architecture for exploring and editing compound (di-)graphs through graph views is presented in Chapter 4. The proposed architecture already has been put into practice in form of a proof-of-concept implementation (Pfeiffer, 2005; Pröpster, 2005); see Figure 1.17 for a screenshot of this implementation showing the compound digraph of Figure 1.16 in three different views.

1.4.1 Key Features

The key features of our architecture are as follows. First, a compound (di-)graph can have arbitrarily many views. This is indispensable, as we want to show different views of the same compound (di-)graph simultaneously, e. g., one window with an abstract overview and another one for the details. This is, incidentally, a common feature of many applications with a graphical user interface, e. g., in most text editors it is possible to view and work on different parts of the document simultaneously.

Second, other data structures than the one we present in Chapter 2 already exist or will be developed in the future. The architecture therefore is kept flexible, i. e., it allows the choice between various data structures, and extensible, i. e., new ones can be integrated easily. The same holds for layout

algorithms, as our update scheme for the algorithm of Sugiyama and Misue (1991) is just one possible solution; a force-directed approach, for instance, would be another option.

Third, the transition between the drawings before and after expanding and contracting can be animated. Although our update scheme already preserves the user's mental map well, animation makes it easier to follow the transition. The concrete realization of the animation, whether to chose linear interpolation or a more sophisticated approach like the one of Friedrich and Eades (2002), is beyond the scope of this work. As the concrete animation style therefore will be subject to variations, particular emphasis is laid on the flexibility and extensibility in this regard.

Finally, compound (di-)graphs and associated views are employed not only in our interactive editor, but conceivably also in a command line interface or a web-based front end. In order to assure maximum reusability, the application is divided into multiple parts that are coupled only loosely following the Model-View-Controller (MVC) pattern (Buschmann et al., 1996).

1.4.2 Previous Solutions

There are various libraries implementing data structures for ordinary graphs, e. g., LEDA, GTL, or Boost Graph Library. None of these incorporates clustered graphs or compound (di-)graphs. A thorough description of the coherence of graph, hierarchies, and views from a software-engineering perspective can be found in (Raitner, 2002), in which an architecture for clustered graphs with arbitrarily many different inclusion hierarchies per graph and arbitrarily many different views per hierarchy is presented. Although it is possible to adapt the paradigm of arbitrarily many views to compound (di-)graphs, having arbitrarily many hierarchies is not possible for compound (di-)graphs because they do not feature the same rigid distinction between graph and inclusion hierarchy as clustered graphs.

Interactive systems for manipulating and exploring clustered graphs or compound (di-)graphs have a long tradition. Sugiyama and Misue (1991) implemented their drawing algorithm in a system called SKETCH-II. It seems that not only `expand` and `contract` but also `split` and `merge` are supported in SKETCH-II, albeit no special care is taken to preserve the user's mental map; apparently, the entire graph is redrawn. Later, Sugiyama and Misue (1993, 1995) devised D-ABDUCTOR, a generic compound digraph editor. Besides the functionality of SKETCH-II, it also provides fisheye views and animation. DA-TU of Huang and Eades (1998) features navigation and manipulation of clustered graphs through abridgments, which are

similar to our views (cf. Definition 1.8). It uses a force-directed drawing style and animates the transitions by showing the intermediate steps of the force-directed algorithm. HIGRES of Lisitsyn and Kasyanov (1999) also is an editor for clustered graphs working with views that resemble the abridgments of Huang and Eades; it is extensible with custom drawing styles, which may consist of several animation steps.

Besides these two-dimensional systems, there are others that make use of three dimensions. Parker et al. (1998) employ expanding and contracting in their system NESTEDVISION3D as one solution to the problem of focus and context, i. e., to show both an overview and the necessary details of a large graph. WILMASCOPE of Dwyer and Eckersley (2003) is a framework for three-dimensional visualization, navigation, and manipulation of compound (di-)graphs. Different drawing styles, mostly force-directed ones, are available and the expanding and contracting of nodes is animated, at least for the force-directed styles. WILMASCOPE also follows the MVC paradigm; it is designed to be extensible with regard to drawing styles.

Aiming at the visualization of huge graphs with hundreds of millions of nodes, MGV of Abello and Korn (2002) employs a different concept for exploring a hierarchically structured graph: not only nodes, also edges can be expanded. Expanding an edge (u, v) yields a graph consisting of u's and v's children and all derived edges between them; expanding a node u replaces it with its children and their derived edges. Both operations, however, lose the context, but multiple views are used to compensate for this. Various visualization styles for the selected subgraphs are described and it seems that the system can easily be extended with new styles.

Although all of these systems somehow need to solve the problem of graph view maintenance, none provides a thorough description of the underlying data structure used for representing clustered graphs or compound (di-)graphs and their views. Consequently, they do not seem to be flexible and extensible in this respect, which becomes increasingly important, as various different data structures for graph view maintenance have been introduced recently.

2

Dynamic Tree Cross Products

First and foremost, a suitable data structure for the problem of graph view maintenance is needed. Although there already exist applications that apparently feature expanding and contracting in views of clustered graphs or compound (di-)graphs, we cannot choose from a wide range of such data structures. Only recently a formal definition of the graph view maintenance problem (for clustered graphs) has been given by Buchsbaum and Westbrook (2000). The data structure they propose solves the dynamic graph variant (for clustered graphs), i. e., edges may be added and removed, but the node set is fixed. A different solution for the dynamic graph variant is given by Buchsbaum et al. (2000): they formulate it as a special case of a more general problem: *range searching over tree cross products*. So far these two are the only known solutions to the problem of graph view maintenance.

A tree cross product consists of d distinct trees $T_1 = (V_1, E_1), \ldots, T_d = (V_d, E_d)$ and a set of d-dimensional hyperedges $\boldsymbol{u} = (u_1, \ldots, u_d)$ with $u_i \in V_i$ for $i = 1, \ldots, d$. In this context, range searching means to decide for given tree nodes $u_1, \ldots, u_d$ whether there exists a hyperedge connecting the subtrees rooted at these nodes and to report all those hyperedges if necessary. In (Buchsbaum et al., 2000) the set of hyperedges is dynamic, but the trees are static. In the following, we generalize this approach as regards insertion and deletion of leaves of the trees $T_1, \ldots, T_d$.

As mentioned in Buchsbaum et al. (2000), tree cross products have more applications than just graph view maintenance, e. g., *text indexing with one error*: given a text $T = x_1 \ldots x_n$ and a pattern P of length m the task is to find all locations i in T such that P matches the substring $x_i, \ldots x_{i+m-1}$ with exactly one error. Another important application is finding *hammocks* in a

network. In a directed graph $G = (V, E)$ with a source node s and a sink node t, a node u *dominates* a node v if every path from s to v goes through u. Symmetrically, a node v *post-dominates* a node u if every path from u to t goes through v. The *hammock* between two nodes u and v is defined as the set of nodes dominated by u and post-dominated by v. Buchsbaum et al. formulate both problems as tree cross products and solve them with their range searching data structure.

After a formal definition of range searching over tree cross products in Section 2.1, a first impression of the complexity of this problem is given in Section 2.2 by means of analyzing the most straightforward solution, namely using no special data structures at all. Since we generalize the approach of Buchsbaum et al. (2000), we first recapitulate their solution in Section 2.3. The data structure they propose supports only insertion and deletion of hyperedges, whereas the trees are fixed. In Section 2.4, we therefore extended this solution to allow insertion and deletion of leaves. Both the description of the original data structure and our extension are subdivided into two parts: we first examine the data structure for the two-dimensional case in Section 2.3.1 and Section 2.4.1, respectively; then the two-dimensional case is used as basis for the recursive description of the higher dimensional cases in Section 2.3.2 and Section 2.4.2, respectively. We conclude the excursion into range searching over tree cross products with Table 2.4.2 comparing the results for our new data structure to the approach of Buchsbaum et al. (2000) and to the naive solution. In Section 2.5, we finally formulate the new dynamic leaves variant of the graph view maintenance problem as a two-dimensional tree cross product and thus solve it efficiently with our new data structure.[1]

2.1 Tree Cross Products

d-partite hypergraph

Definition 2.1. A *d-partite hypergraph* $G = (V, E)$ consists of nodes $V = V_1 \dot\cup \ldots \dot\cup V_d$ and edges $E \subseteq \prod_{i=1}^{d} V_i = V_1 \times \cdots \times V_d$; $\boldsymbol{u} = (u_1, \ldots, u_d) \in E$ is called a *d-dimensional hyperedge*. For $d = 2$, a d-partite hypergraph is a *bipartite graph*.

tree cross product

Definition 2.2. A *tree cross product* $C = (\{T_1, \ldots, T_d\}, E)$ consists of disjoint rooted trees $T_1 = (V_1, E_1), \ldots, T_d = (V_d, E_d)$ and hyperedges E such that $G = (V_1 \dot\cup \ldots \dot\cup V_d, E)$ is a d-partite hypergraph. Let $n = |V_1 \dot\cup \ldots \dot\cup V_d|$, $m = |E|$, and $\Delta = \max_{i=1}^{d} \text{depth}(T_i)$.

[1] A preliminary version of the material in this chapter appears in (Raitner, 2004a,c).

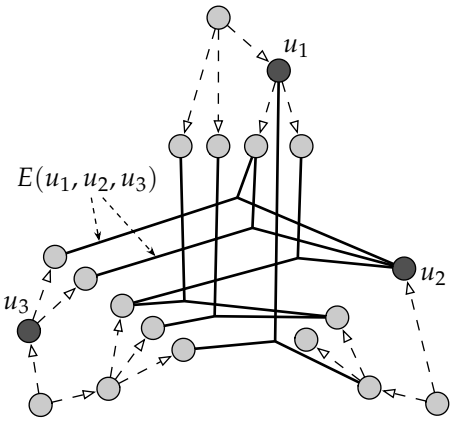

Figure 2.1: Example of a three-dimensional tree cross product. The dashed edges belong to the trees; the solid ones are the hyperedges E.

In other words, a tree cross product consists of d-dimensional hyperedges connecting the nodes of d disjoint trees. Figure 2.1, for instance, shows a three-dimensional tree cross product; the dashed edges form the trees and the solid ones are the three-dimensional hyperedges.

Let $C = (\{T_1, \ldots, T_d\}, E)$ be a tree cross product consisting of the trees $T_1 = (V_1, E_1), \ldots, T_d = (V_d, E_d)$. For a tuple $\boldsymbol{u} = (u_1, \ldots, u_d) \in \prod_{i=1}^{d} V_i$, let

$$E(\boldsymbol{u}) = \{\boldsymbol{x} = (x_1, \ldots, x_d) \in E \mid \forall\, 1 \leq i \leq d : x_i \in \operatorname{desc}(u_i)\}$$

be the set of hyperedges between descendants of $\boldsymbol{u}$'s elements, i. e., the set of hyperedges connecting the subtrees rooted at the nodes u_i ($1 \leq i \leq d$). In Figure 2.1 the set $E(u_1, u_2, u_3)$, for instance, contains two edges.

Definition 2.3. A tuple $\boldsymbol{u} = (u_1, \ldots, u_d) \in \prod_{i=1}^{d} V_i$ is called a *derived hyperedge* if $E(\boldsymbol{u}) \neq \emptyset$. The *derived hypergraph* $\mathfrak{G} = (\mathfrak{V}, \mathfrak{E})$ consists of the node set $\mathfrak{V} = V_1 \dot\cup \ldots \dot\cup V_d$ and all derived hyperedges: $\mathfrak{E} = \{\boldsymbol{u} \in \prod_{i=1}^{d} V_i \mid E(\boldsymbol{u}) \neq \emptyset\}$.

derived hyperedge, hypergraph

In the example shown in Figure 2.1, the two edges in the set $E(u_1, u_2, u_3)$ give rise to a derived edge $(u_1, u_2, u_3) \in \mathfrak{E}$. Note that a hyperedge $\boldsymbol{u} = (u_1, \ldots, u_d) \in E$ contributes to every set $E(u'_1, \ldots, u'_d)$ where u'_i is an ancestor of u_i ($1 \leq i \leq d$). The number of derived hyperedges therefore is at most $m\Delta^d$, i. e., $|\mathfrak{E}| \leq m\Delta^d$.

As defined in (Buchsbaum et al., 2000), the problem of range searching over tree cross products is to perform the following queries on tuples $\boldsymbol{u} \in \prod_{i=1}^{d} V_i$:

`edgeQuery`$(\boldsymbol{u})$ determines whether $E(\boldsymbol{u}) \neq \emptyset$, i. e., whether $\boldsymbol{u}$ is a derived hyperedge;
`edgeReport`$(\boldsymbol{u})$ determines the set $E(\boldsymbol{u})$;
`edgeExpand`$(\boldsymbol{u}, j)$ determines the set of all derived hyperedges of the form $(u_1, \ldots, u_{j-1}, x, u_{j+1}, \ldots, u_d) \in \mathfrak{E}$ with $x \in \text{children}(u_j)$ (*precondition:* $1 \leq j \leq d$).

In the example in Figure 2.1, `edgeQuery`(u_1, u_2, u_3) returns true, while `edgeReport`(u_1, u_2, u_3) returns the two edges in $E(u_1, u_2, u_3)$. Both children of u_3 are returned by `edgeExpand`$((u_1, u_2, u_3), 3)$, whereas only the left child of u_1 is returned by `edgeExpand`$((u_1, u_2, u_3), 1)$.

Note that the definition of a derived hyperedge in a tree cross product intentionally bears some resemblance to a derived edge in a compound digraph (cf. Definition 1.7). Indeed, we will utilize this fact in Section 2.5 and transform a compound digraph into a tree cross product such that the question whether there is a derived edge between two nodes of the graph can be answered with an appropriate `edgeQuery`. The `edgeExpand` operation turns out to be useful for expanding a node u in view of a compound digraph: it yields the expanded edges of each derived edge that is incident to u before expanding.

Buchsbaum et al. (2000) describe both a static variant supporting only these queries and a more dynamic variant providing additionally the following updates:

`newEdge`$(\boldsymbol{u})$ adds $\boldsymbol{u}$ to the set of hyperedges E;
`deleteEdge`$(\boldsymbol{u})$ removes $\boldsymbol{u}$ from E.

Note that for our application, i. e., an interactive editor that works with views of compound (di-)graphs, adding and removing edges surely is indispensable. The trees, however, are fixed, which is counterintuitive in this context. Therefore, we consider a more dynamic variant by adding the following operations:

`newLeaf`(u) adds a new leaf to T_j as child of node u (*precondition:* $u \in V_j$);
`deleteLeaf`(u) removes u and all hyperedges incident to it from T_j (*precondition:* $u \in V_j$ is a leaf, but not the root of T_j).

This gives us the possibility to construct any tree from scratch, though the required operations have to be ordered appropriately. The data structure we propose in Section 2.4 extends the approach of Buchsbaum et al. (2000) and thus solves the problem of range searching over tree cross products in this more dynamic variant, i. e., with adding and removing leaves. As we will see in Section 2.5, a compound digraph can be modeled as a tree

cross product. For this reason, our data structure also solves the problem of graph view maintenance in the dynamic leaves variant. Before discussing our approach as well as the one it extends in detail, we first take a brief look at the most naive solution in order to get an impression of the complexity of these operations and the trade-offs between query and update operations.

2.2 Naive Approach

The most straightforward solution to the problem of range searching over tree cross product probably is to be lazy, i. e., to use no additional data structure at all. The trees are stored canonically: in every node a reference to its parent and a list of references to its children is kept; additionally, a list of all hyperedges E is maintained.

Since no complicated data structures have to be adjusted, this approach favors updates: `newLeaf` and `deleteLeaf` take only constant time[2], as do `newEdge` and `deleteEdge`. Note that deleting a leaf first requires locating it in its parent's list of children. Since every node is in exactly one such list, a reference to the corresponding list entry can be stored with each node. Likewise, an edge is linked with its position in the list of all edges to facilitate its deletion.

Queries, however, require inspecting all edges in the worst case. For instance, `edgeQuery`$(u_1, \ldots, u_d)$ can be implemented by checking for each edge $(u'_1, \ldots, u'_d) \in E$ whether $u'_i \in \operatorname{desc}(u_i)$. Without special data structures for these ancestor queries, `edgeQuery` takes $\mathcal{O}(md\Delta)$ time. Analogously, all original edges that are represented by a derived edge can be reported in $\mathcal{O}(md\Delta)$ time. Finally, expanding a derived edge $(u_1, \ldots, u_d)$ at index j, i. e., `edgeExpand`$((u_1, \ldots, u_d), j)$, also takes $\mathcal{O}(md\Delta)$ in this naive approach: each edge $(u'_1, \ldots, u'_d) \in E$ is checked whether it contributes to the derived edge; whenever such an edge is found, the corresponding child of u_j is determined by navigating upwards from u'_j in T_j.

2.3 Static Trees

Before we describe its extension, we recapitulate the original approach of Buchsbaum et al. (2000), who propose a data structure for d-dimensional tree cross products where the set of hyperedges E is dynamic, but the trees are fixed. This data structure is defined recursively: the two-dimensional

[2] Unless explicitly stated, all time bounds are worst-case.

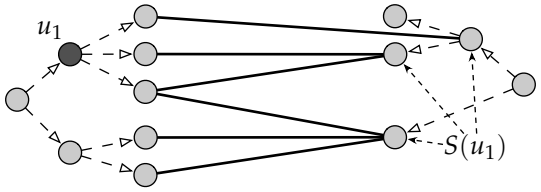

Figure 2.2: A two-dimensional tree cross product illustrating the set $S(\cdot)$.

case, presented in Section 2.3.1, serves as basis for the recursive description of the higher dimensional data structure, which is given in Section 2.3.2.

2.3.1 The Two-Dimensional Case

In the two-dimensional case, the tree cross product consists of two trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ connected by edges $E \subseteq V_1 \times V_2$. In an $\mathcal{O}(n)$ preprocessing phase, both trees are traversed in post-order and each node is assigned its post-order number. In order to simplify the following description, we will abuse notation and identify a node $u \in V_1 \cup V_2$ with its post-order number in the respective tree. Let $\min(u) = \min\{v \mid v \in \text{desc}(u)\}$ denote the smallest and $\max(u) = \max\{v \mid v \in \text{desc}(u)\}$ the largest node in the subtree rooted at u. Note that these two bounds fully determine the set of nodes in the subtree; more precisely: $\text{desc}(u) = \{v \in \mathbb{N} \mid \min(u) \leq v \leq \max(u)\}$.

For a node $u_1 \in V_1$, the set

$$S(u_1) = \{u_2 \in V_2 \mid \exists (u_1', u_2) \in E : u_1' \in \text{desc}(u_1)\}$$

contains all nodes of the tree T_2 that are connected to a node in the subtree of u_1; see Figure 2.2. For a node $u_2 \in T_2$ this set is defined analogously:

$$S(u_2) = \{u_1 \in V_1 \mid \exists (u_1, u_2') \in E : u_2' \in \text{desc}(u_2)\}.$$

The following lemma shows how the sets $S(\cdot)$ and $\text{desc}(\cdot)$ can be utilized for an `edgeQuery` operation:

Lemma 2.4 For all $u_1 \in V_1$ and all $u_2 \in V_2$, $E(u_1, u_2) \neq \emptyset$ if and only if $S(u_1) \cap \text{desc}(u_2) \neq \emptyset$. (Symmetrically, $E(u_1, u_2) \neq \emptyset$ if and only if $S(u_2) \cap \text{desc}(u_1) \neq \emptyset$.)

Proof. By definition, $S(u_1) \cap \text{desc}(u_2) \neq \emptyset$ if and only if there exists an edge $(u_1', u_2') \in E$ such that $u_1' \in \text{desc}(u_1)$ and $u_2' \in \text{desc}(u_2)$, i. e., if and only if $E(u_1, u_2) \neq \emptyset$. □

In other words, $\texttt{edgeQuery}(u_1, u_2)$ is true if and only if $S(u_1) \cap \text{desc}(u_2) \neq \emptyset$ (or symmetrically if and only if $S(u_2) \cap \text{desc}(u_1) \neq \emptyset$).

For every tree node $u \in V_1 \cup V_2$, Buchsbaum et al. store $\min(u)$, $\max(u)$, and the set $S(u)$. Furthermore, the children of every node are stored in a list sorted in ascending order. Being a subset of a fixed integer universe[3], each set $S(\cdot)$ is maintained as a separate contracted stratified tree (CST) (Preparata et al., 1992). A CST stores a subset A of a fixed integer universe $U = \{0, 1, \ldots, n-1\}$ in $\mathcal{O}(|A| \log\log n)$ space such that inserting, deleting, and looking up an element takes $\mathcal{O}(\log\log n)$ time.[4] Also, the successor operation is supported in $\mathcal{O}(\log\log n)$ time: for $x \in U$, $\texttt{successor}(A, x)$ finds the smallest element of A that is greater or equal than x or null if no such element exists.

As the following lemma shows, the successor operation plays a key role in the implementation of edgeQuery:

Lemma 2.5 For all $u_1 \in V_1$ and all $u_2 \in V_2$, $E(u_1, u_2) \neq \emptyset$ if and only if $\texttt{successor}(S(u_1), \min(u_2)) \leq \max(u_2)$. (Symmetrically, $E(u_1, u_2) \neq \emptyset$ if and only if $\texttt{successor}(S(u_2), \min(u_1)) \leq \max(u_1)$.)

Proof. The claim follows from Lemma 2.4, as $S(u_1) \cap \text{desc}(u_2) \neq \emptyset$ if and only if $\texttt{successor}(S(u_1), \min(u_2)) \leq \max(u_2)$. □

Therefore, Buchsbaum et al. implement $\texttt{edgeQuery}(u_1, u_2)$ in $\mathcal{O}(\log\log n)$ time by checking whether

$$\texttt{successor}(S(u_1), \min(u_2)) \leq \max(u_2)$$

or, symmetrically, whether

$$\texttt{successor}(S(u_2), \min(u_1)) \leq \max(u_1).$$

In order to facilitate the edgeReport operation, Buchsbaum et al. link the leaves in each CST. Therefore, $\texttt{edgeReport}(u_1, u_2)$ is just a matter of

[3] Remember that the nodes have been identified with their post-order numbers and that the trees are static, i. e., the node set is fixed.

[4] The time bounds for these operations are similar to those of the more popular *van Emde Boas trees* (van Emde Boas et al., 1977). The space bound of a CST, however, depends on the size of the subset A it represents, whereas the space bound of a van Emde Boas tree is independent of $|A|$: it always uses $\mathcal{O}(n \log\log n)$ space. Although this bound can be improved easily to $O(n)$ (van Emde Boas, 1977), it is still independent of the size of A, which is a considerable overhead for small subsets A.

finding the first edge with $\text{successor}(S(u_1), \min(u_2))$ and reporting all subsequent leaves as long as they are less or equal than $\max(u_2)$. This takes $\mathcal{O}(\log\log n + k)$ time where k is the number of edges reported.

Remark. Although not mentioned explicitly in Buchsbaum et al. (2000), storing the sets $S(\cdot)$ as defined above is not sufficient for `edgeReport`. First and foremost, each entry in a set $S(\cdot)$ needs to be associated with the edge that created it. Consider two edges $(u_1, u_2) \in E$ and $(v_1, u_2) \in E$ and the nearest common ancestor n_1 of u_1 and v_1; then, $S(n_1)$ contains only one entry u_2. Of course, the sets $S(\cdot)$ could be defined as multisets instead of sets, which would yield one entry per edge. However, the each set $S(\cdot)$ is maintained as a CST, which can only store a *subset* of a fixed integer universe and no *multiset*. Even if it were possible (as it is, for instance, for ordinary balanced search trees), it would unnecessarily complicate the deletion of an edge. If we wanted to delete the edge (u_1, u_2) in the above example, we cannot know which of the two entries with key u_2 in $S(n_1)$ belongs to this edge and thus has to be deleted; in the worst case, we would have to check all of them.

Instead of maintaining $S(\cdot)$ as multisets each node $u_2 \in S(u_1)$ for $u_1 \in V_1$ therefore has to be associated with the set of edges that created this entry, i. e., with the set $\{(u_1', u_2) \in E \mid u_1' \in \text{desc}(u_1)\}$. (Symmetrically, each node $u_1 \in S(u_2)$ for $u_2 \in V_2$ is associated with the set $\{(u_1, u_2') \in E \mid u_2' \in \text{desc}(u_2)\}$.) Storing these sets as doubly linked lists results in the same problem as for the multiset solution described above: the entire list has to be scanned for finding the correct entry when an edge is deleted. Dynamic perfect hashing (Dietzfelbinger et al., 1994) would yield expected constant time for inserting and deleting, but it does not support a sequential scan of the whole set, which is needed for `edgeReport`. Using an ordinary balanced search tree would be possible, but would require additional $\mathcal{O}(\log n)$ time for deleting and inserting an edge and thus would violate the time bounds stated in (Buchsbaum et al., 2000).

The problem with any of the aforementioned representations for the set of edges associated with an entry in a set $S(\cdot)$ is that the edge has to be searched before it can be deleted. In other words, we only know the edge and not its respective position in the data structure. If we knew instead, for instance, the edge's container in a doubly linked list, we could delete the edge in constant time. This gives rise to the following solution: we maintain the set of edges for an entry in $S(\cdot)$ as a doubly linked list and keep track of all occurrences of an edge with a list of references to the respective containers. We store this list of occurrences in an additional field in the record representing an edge such that it can be accessed in constant time given the edge. Although this

does not facilitate the deletion of an edge in any *particular* set $S(\cdot)$, as the list of occurrences has to be scanned for the one to delete, it accelerates the deletion of an edge in *all* sets $S(\cdot)$ because all occurrences have to be deleted anyway. Incidentally, this also offers the possibility of storing multiple edges, which indeed is an issue in the generalization to higher dimensions presented in Section 2.3.2.

Buchsbaum et al. could have implemented `edgeExpand` with an appropriate collection of `edgeQuery` operations. In general, however, this is less efficient than Algorithm 2.1.[5] Since it simplifies the description, Algorithm 2.1 treats the expansion of the first element of an edge only, i. e., $\texttt{edgeExpand}((u_1, u_2), 1)$; expanding the second element works analogously.

Algorithm 2.1: $\texttt{edgeExpand}((u_1, u_2), 1)$

input : a derived edge $(u_1, u_2) \in \mathfrak{E}$
output: all children u_1' of u_1 such that $(u_1', u_2) \in \mathfrak{E}$

let $v_1, \ldots, v_l$ be the list of children of u_1 in ascending order
$R \leftarrow \emptyset$, $t \leftarrow v_1$
repeat
  $s \leftarrow \text{successor}(S(u_2), \min(t))$
  if $s \leq \max(v_l)$ **then**
6   **let** v_i be the ancestor of s among children(u_1)
    $R \leftarrow R \cup \{v_i\}$
    if $i \neq l$ **then**
      $t \leftarrow v_{i+1}$
    end
  end
until $s > \max(v_l)$ *or* $i = l$
return R

Since the complexity of Algorithm 2.1 hinges on the implementation of line 6, Buchsbaum et al. additionally use a separate *level ancestor* data structure for each tree (Buchsbaum, 2003). Such a data structure provides *level ancestor queries*, i. e., determining the ancestor of a given node on a given depth of the tree. This *level ancestor problem* is well studied both in the static variant (Dietz, 1991; Berkman and Vishkin, 1994; Bender and Farach-Colton, 2002), which is sufficient here, and the dynamic variant (Dietz, 1991; Alstrup and Holm, 2000), which we will use in our dynamization of this approach

[5] The original description of this algorithm in Buchsbaum et al. (2000) is rather short and omits essential details. A detailed description similar to Algorithm 2.1 has been presented in (Raitner, 2004c).

because it additionally allows inserting and deleting of leaves. For the static variant, the solutions of Dietz (1991), Berkman and Vishkin (1994), and Bender and Farach-Colton (2002) are all optimal[6], i. e., preprocessing a tree with n nodes takes $\mathcal{O}(n)$ and a level ancestor query takes $\mathcal{O}(1)$ time. With such a data structure for both trees Algorithm 2.1 takes $\mathcal{O}(k \log\log n)$ time, because every `successor` operation except the last yields a new result.

Inserting or deleting an edge (u_1, u_2) is straightforward: for each ancestor u_1' of u_1, we first look up u_2 in the CST for the set $S(u_1')$. This yields the doubly linked list stored under the key u_2 in which we insert the edge (u_1, u_2). If u_2 is not found in $S(u_1')$, a new doubly linked list containing only the edge (u_1, u_2) is inserted into the CST for the set $S(u_1')$ under the key u_2. Either way, the reference to the list container holding the edge (u_1, u_2) is added to the list of occurrences stored in the edge record. For the ancestors u_2' of u_2 the updates are symmetric. These updates take $\mathcal{O}(\log\log n)$ time at each ancestor and there are at most $\mathcal{O}(2\Delta)$ of them.[7].

Altogether, Buchsbaum et al. (2000) get the following results for the two-dimensional case:

Theorem 2.6 (Buchsbaum and Westbrook (2000))
With $\mathcal{O}(2\Delta m \log\log n)$ space, we can insert or delete edges in $\mathcal{O}(2\Delta \log\log n)$ time[8] and perform `edgeQuery` in $\mathcal{O}(\log\log n)$, `edgeExpand` in $\mathcal{O}(k \log\log n)$, and `edgeReport` in $\mathcal{O}(\log\log n + k)$ time, where k is the number of edges reported.

Buchsbaum et al. (2000) use *compressed trees* (Harel and Tarjan, 1984) to reduce both the space bound and the time bounds for inserting and deleting edges. For a tree T, an edge $(\text{parent}(u), u)$ is termed *light* if $2|\text{desc}(u)| \leq |\text{desc}(\text{parent}(u))|$ and *heavy* otherwise. At most one edge from u to a child can be heavy because $|\text{desc}(u)| = 1 + \sum_{v \in \text{children}(u)} |\text{desc}(v)|$. This means that the heavy edges partition the nodes of the tree into a collection of *heavy paths*.[9] The *apex* of a heavy path is its topmost node, i. e., the one of smallest depth. The compressed tree $C(T)$ as defined by Buchsbaum et al. (2000) evolves from T by contracting every heavy path into its respective apex.

[6] From a practical point of view, however, the solution of Bender and Farach-Colton (2002) is preferable; its description is much easier and requires no "heavy" machinery making it more suitable for an implementation.

[7] Recall that $\Delta = \max\{\text{depth}(T_1), \text{depth}(T_2)\}$.

[8] The constant factor 2 could be omitted here. However, in the recursive description of the higher dimensional data structure, for which this case will serve as basis, we will get a factor 2 for *every* dimension. This factor can thus be neglected here only because the dimension is fixed.

[9] A node with no incident heavy edges is a trivial heavy path.

In the original definition of Harel and Tarjan (1984), however, the compressed tree is constructed differently. For any node v, let $\text{apex}(v)$ denote the apex of the heavy path containing v. The compressed tree has the same set of nodes as T, but a different set of edges: each node $v \neq \text{root}(T)$ is connected to $\text{apex}(\text{parent}(v))$. Therefore, every node of a heavy path except the apex is attached as leaf to the respective apex in the compressed tree, whereas in the definition used by Buchsbaum et al. (2000) these nodes are merged into the apex.

Note that every edge in $C(T)$ corresponds to a light edge in T. With the original definition of Harel and Tarjan, in fact, every edge in $C(T)$ is light. This is not true for the definition of Buchsbaum et al. because contracting a heavy path into its apex reduces the number of nodes in the subtree rooted at the apex. According to (Harel and Tarjan, 1984, Lemma 7), $C(T)$ has the convenient property that it is more balanced than T: its depth is at most $\lfloor \log n \rfloor$, where n is the number of nodes in T. This holds also for a compressed tree as defined by Buchsbaum et al. which can be constructed from the one according to the definition of Harel and Tarjan by removing some leaves.

The key idea of Buchsbaum et al. (2000) is to maintain the sets $S(\cdot)$ for the nodes of $C(T)$ instead of T. Since a node μ of $C(T)$ represents a heavy path in T, a set $S(\mu)$ conceptually combines the sets $S(u)$ according to the former definition for all nodes u on the heavy path corresponding to μ. In order to differentiate between these sets again, the entries of $S(\mu)$ are enriched with a second parameter, i. e., they are now pairs of the form (u, i). As a consequence, the query operations become more complicated, more precisely, they are implemented as *three-sided range queries* on such a set $S(\mu)$. Thus, Buchsbaum et al. (2000) maintain these sets no longer as CSTs, but as *priority search trees (PST)* (McCreight, 1985; Willard, 2000). Although this improves the space bound and the time bounds for inserting and deleting edges, the time bounds of all query operations increase slightly:

Theorem 2.7 (Buchsbaum and Westbrook (2000))
Let $s = \min\{\Delta, \log n\}$. With $\mathcal{O}(2sm)$ space, we can insert or delete edges in $\mathcal{O}(2s \log n / \log\log n)$ time[10], perform `edgeQuery` in $\mathcal{O}(\log n / \log\log n)$, `edgeExpand` in $\mathcal{O}(k \log n / \log\log n)$, and `edgeReport` in $\mathcal{O}(\log n / \log\log n + k)$ time, where k is the number of edges reported.

Remark. In order to reduce the space bound further Buchsbaum et al. (2000) stratify T_1 into $\sqrt{\text{depth}(T_1)}$ strata of $\sqrt{\text{depth}(T_1)}$ levels each (similarly for

[10] Again, the factor 2 can be omitted here because the dimension is fixed, but it is no longer negligible if the dimension is part of the input.

T_2). An edge (u, v) contributes an entry only to those sets $S(x)$ where x is an ancestor of u (or v) in the *same* stratum. Additionally, each node x at the *top* of stratum, i. e., parent(x) is in a higher stratum, maintains a set $S'(x)$ with all entries corresponding to edges incident with descendants in deeper strata. Buchsbaum et al. (2000) claim that a query on a set $S(x)$ (as defined before) can be implemented by uniting the corresponding queries on the new $S(x)$ and $S'(\text{top}(x))$, where top(x) is the ancestor of x at the top of x's stratum. Unfortunately, this is not correct (Buchsbaum, 2003): consider x and y such that top$(x) =$ top(y); any edge incident with a descendant of y will be placed into $S'(\text{top}(x))$ and will thus be a possible answer to a query on $S(x)$.

2.3.2 Higher Dimensions

Since the generalization to higher dimensions is very concise and partly incorrect in (Buchsbaum et al., 2000), we give a more detailed and corrected description in the following.

The two-dimensional data structure described so far maintains a dynamic set of (hyper-)edges $(u_1, u_2) \in V_1 \times V_2$ between the nodes of two trees. It provides the retrieval operations `edgeQuery`, `edgeReport`, and `edgeExpand`. The recursive definition of the higher dimensional data structure uses this two-dimensional case as basis.

Suppose that there is already such a data structure for the case d, i. e., for maintaining hyperedges between nodes of d trees that provides all the operations described above. The $(d+1)$-dimensional data structure stores at each node $u_1 \in V_1$ the set

$$S_{d+1}(u_1) = \{(u'_1, u'_2, \ldots, u'_{d+1}) \in E \mid u'_1 \in \text{desc}(u_1)\},$$

i. e., all hyperedges incident with descendants of u_1. If we ignore the first element of these $(d+1)$-dimensional hyperedges, each set $S_{d+1}(\cdot)$ can be interpreted as a (multi-)set of d-dimensional hyperedges and thus it can be stored in a separate d-dimensional data structure. In other words, the $(d+1)$-dimensional hyperedges become entries in a d-dimensional data structure according to their projections onto the last d elements. In Figure 2.3, for instance, the dark edges form the set $S_3(u_1)$, which is stored as the two-dimensional tree cross product shown in Figure 2.4.

Obviously, `edgeQuery`$(u_1, u_2, \ldots, u_{d+1})$ can be forwarded to the d-dimensional data structure stored at u_1 as `edgeQuery`$(u_2, \ldots, u_{d+1})$; `edgeReport` is forwarded similarly. Note that the returned edges are already the correct $(d+1)$-dimensional result because the d-dimensional data structure contains the original $(d+1)$-dimensional hyperedges; their projections onto

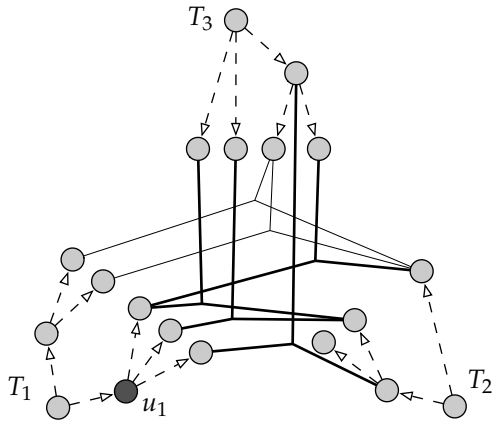

Figure 2.3: A three-dimensional tree cross product; the darker edges form the set $S_3(u_1)$.

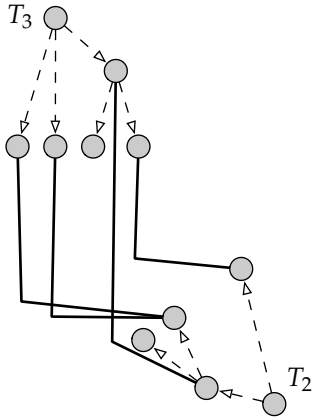

Figure 2.4: The two-dimensional tree cross product as which the set $S_3(u_1)$ is stored.

the last d elements are used only as keys. `edgeExpand`$((u_1, u_2, \ldots, u_{d+1}), j)$ for $j \neq 1$ is implemented as `edgeExpand`$((u_2, \ldots, u_{d+1}), j)$ on the d-dimensional data structure at the node u_1. For expanding a hyperedge at its first element ($j = 1$), the same data structure is built with some other tree in the role of T_1, for instance, T_2.

Theorem 2.8
With $\mathcal{O}((2\Delta)^{d-1} m \log\log n)$ space, we can insert or delete edges in $\mathcal{O}((2\Delta)^{d-1} \log\log n)$ time, perform `edgeQuery` in $\mathcal{O}(d + \log\log n)$, `edgeExpand` in $\mathcal{O}(d + k \log\log n)$, and `edgeReport` in $\mathcal{O}(d + \log\log n + k)$ time, where k is the number of edges reported.

Proof. The base case, $d = 2$, is exactly Theorem 2.6. For $d > 2$, an edge $(u_1, \ldots, u_d)$ contributes an entry to the lower dimensional data structure at every ancestor of u_1 and at every ancestor of u_2 (assuming that T_2 is the tree designated to be T_1 for the second data structure). These are $\mathcal{O}(2\Delta)$ entries; by induction each entry uses $\mathcal{O}((2\Delta)^{(d-1)-1} \log\log n)$ space in the lower dimensional data structure, which gives a total of $\mathcal{O}(m(2\Delta)^{d-1} \log\log n)$ space. Inserting or deleting an edge is implemented as one insert or delete operation in a lower dimensional data structure for every ancestor in the two dimensions, each of which takes $\mathcal{O}((2\Delta)^{(d-1)-1} \log\log n)$ by induction. All retrieval operations `edgeQuery`, `edgeReport`, and `edgeExpand` are forwarded to an appropriate lower dimensional data structure; this recursion ends at some two-dimensional data structure with the time bounds of Theorem 2.6. Hence, we get $d - 1$ additional steps for the recursion. □

Theorem 2.9
Let $s = \min\{\Delta, \log n\}$. With $\mathcal{O}((2s)^{d-1}m)$ space, we can insert or delete an edge in $\mathcal{O}((2s)^{d-1}\log n/\log\log n)$ time, perform `edgeQuery` in $\mathcal{O}(d + \log n/\log\log n)$, `edgeExpand` in $\mathcal{O}(d + k\log n/\log\log n)$, and `edgeReport` in $\mathcal{O}(d + \log n/\log\log n + k)$ time, where k is the number of edges reported.

Proof. This follows similar to the proof of Theorem 2.8 with Theorem 2.7 instead of Theorem 2.6 as base case. □

Buchsbaum et al. (2000) include neither the additive d terms in the time bounds of `edgeQuery`, `edgeReport`, and `edgeExpand` nor the factor 2^{d-1} in the space bound and in the time bounds for inserting or deleting an edge. However, both seem unavoidable given their description (Buchsbaum, 2003).

2.4 Dynamization: Inserting and Deleting Leaves

After having introduced the approach of Buchsbaum et al. (2000), we will combine it with a novel technique of using search trees superimposed over order maintenance data structures. This makes the previous data structure more dynamic in regard to insertion and deletion of leaves at the cost of a slight slow-down for the other operations. Our extension primarily affects the two-dimensional case. The recursive data structure for the general case is built essentially as in the original approach.

2.4.1 The Two-Dimensional Case

As in Section 2.3.1, the tree cross product consists of two trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ connected by edges $E \subseteq V_1 \times V_2$.

The approach of Buchsbaum et al. (2000) hinges on the fact the nodes of both trees are ordered linearly such that for each tree node v the set $\text{desc}(v)$ forms a contiguous block from $\min(u)$ to $\max(u)$ in this order. Since `newLeaf`(u) was defined to insert the new leaf as a child of u, insertions occur at any point in the linear order. Therefore, simply assigning consecutive integers to the nodes, as in (Buchsbaum et al., 2000), is inefficient: all nodes following the new one would have to be renumbered. In other words, for fixed trees a simple mapping of tree nodes to consecutive integers is sufficient, but adding new leaves requires a more sophisticated solution. Besides comparing two nodes with respect to their position in the order, an appropriate data structure also needs to support insertion and deletion at any point.

Maintaining Order in a List

The *order maintenance problem* is that of maintaining a linearly ordered set of elements under a sequence of the following three operations (Dietz and Sleator, 1987):

`insert`(x, y) inserts a new record y immediately after record x into the order;
`delete`(x) deletes record x;
`order`(x, y) returns true if and only if x comes before y in the order.

The trivial solution, of course, is to store the records in a linked list. Inserting and deleting thus take $\mathcal{O}(1)$ time, while an `order` query requires traversing the list in $\mathcal{O}(n)$ time, where n denotes the number of records. Another straightforward solution is to use a balanced search tree in which the records are stored in the nodes in symmetric order. Insertion and deletion are implemented as the respective operations on the search tree, which take $\mathcal{O}(\log n)$ time. An `order`(x, y) query is performed in $\mathcal{O}(\log n)$ time as follows: we walk up the tree from both x and y to their nearest common ancestor and determine whether x lies in the left subtree.

The more advanced solutions all have one thing in common: the records are maintained as a linked list and each record is assigned a number, its *label*.[11] The labels are monotonically increasing from the beginning of the list to its end, which reduces an `order` query to a simple comparison of the two records' labels. In order to maintain this invariant during insertions it may be necessary to renumber certain nodes in the vicinity of the inserted record.

Dietz (1982) propose an algorithm in which each insertion causes $\mathcal{O}(\log n)$ renumberings in the amortized sense[12]; deletions are not considered. Tsakalidis (1984) eliminates this restriction: insertions and deletions take $\mathcal{O}(\log n)$ amortized time while determining the order still is possible in $\mathcal{O}(1)$ time. He improves this result to $\mathcal{O}(1)$ amortized time for both insertion and deletion and $\mathcal{O}(1)$ time for `order`. Dietz and Sleator (1987, Section 2) give a much simpler algorithm yielding the same time bounds for all three operations. Furthermore, Dietz and Sleator (1987, Section 3) describe an optimal algorithm, i. e., all operations take $\mathcal{O}(1)$ *worst-case* time. Using a technique of

[11] This is why the problem of order maintenance often is referred to as the problem of *list labeling*.

[12] Amortized analysis means that the cost of a sequence of operations is averaged over all operations performed. It is useful to show that the *average* cost of an operation is small, even though a single operation in the sequence might be expensive. The techniques of amortized analysis are surveyed in Tarjan (1985) and Cormen et al. (2001, Chap. 17).

Willard (1986) for insertions and deletions in dense sequential files on the top level of a four-level data structure, it is, however, rather complicated. Bender et al. (2002) propose two algorithms that match the bounds of Dietz and Sleator (1987), but are much simpler.

Replacing the Fixed Order

The key idea of our approach is to replace the simple numbering of the tree nodes in the solution of Buchsbaum et al. (2000) with an order maintenance data structure. Since we will treat the order maintenance component as a black box, any implementation will do. For the theoretical results of Theorem 2.16, however, we assume an optimal, i. e., $\mathcal{O}(1)$ worst-case, solution (Dietz and Sleator, 1987; Bender et al., 2002).

Each tree uses its own, separate order maintenance data structure. In an $\mathcal{O}(n)$ preprocessing phase, both trees are traversed in post-order and their nodes are stored in this order in the respective order maintenance data structure. Note that whenever in the following two nodes belonging to the same tree are compared, this is implemented as an `order` query on the order maintenance data structure for this tree. To simplify the description, we will use $<$, $\leq$, $\geq$, and $>$ to compare two nodes (of the same tree) with respect to this order instead of the corresponding `order` query. Let again $\min(u)$ denote the smallest and $\max(u)$ and the largest node in the subtree rooted at u. As in the original approach of Buchsbaum et al. (2000), $\min(u)$, $\max(u)$, and $S(u)$ are stored at every node $u \in V_1 \cup V_2$; the children of u are kept in a list sorted in ascending order.

Section 2.3.1 shows that the data structure chosen for the sets $S(\cdot)$ determines the complexity of all query operations.[13] Buchsbaum et al. (2000) interpret the nodes of their static trees as a fixed integer universe. Therefore, a set $S(\cdot)$, which is a subset thereof, can be maintained as a CST. This is impractical here because the set of tree nodes now is dynamic. Apart from the universe not being fixed, using the (integer) labels of the order maintenance structures in a CST is complicated: during an `insert` operation, nodes in the vicinity of the inserted one are possibly renumbered and thus all sets $S(\cdot)$ containing a shifted node need to be updated.

Therefore, each set $S(\cdot)$ is maintained as an ordinary height-balanced search tree. Basically, any height-balanced search tree can be used, e. g., red-black trees (Cormen et al., 2001, Chap. 13), AVL trees (Goodrich and Tamassia, 2003, Sect. 9.2), or 2-3 trees (Aho et al., 1983, Sect. 5.4). However,

[13] More precisely, the complexity of the `successor` operation essentially determines the bounds for all query operations.

the trees additionally have to support sequential traversals. More precisely, for the `edgeReport` operation we need to access all elements following a given one in order. For this the 2-3 trees in the variant described by Aho et al. (1983, Sect. 5.4) are well suited: in contrast to red-black trees or AVL trees, the elements are stored only at the leaves and the internal nodes serve just as signposts to guide the other operations. The leaves[14] are already linked because it is needed for rebalancing the tree after insertion and deletion.[15] Provided that the comparison of two nodes, i. e., the `order` operation, takes constant time, insertion, deletion, and determining the successor take $\mathcal{O}(\log n)$ time for a 2-3 tree.

As mentioned in Section 2.3.1, the sets $S(\cdot)$ do not provide enough information for the `edgeReport` operations. Each node $u_2 \in S(u_1)$ for $u_1 \in V_1$ has to be associated with the set of edges that created this entry, i. e., with the set $\{(u_1', u_2) \in E \mid u_1' \in \operatorname{desc}(u_1)\}$. (Symmetrically, a node $u_1 \in S(u_2)$ for $u_2 \in V_2$ has to be associated with the set $\{(u_1, u_2') \in E \mid u_2' \in \operatorname{desc}(u_2)\}$.) As already pointed out in Section 2.3.1, these sets are maintained as doubly linked lists. In order to facilitate the deletion of edges, all occurrences of an edge in such lists are stored (in a linked list) within the record representing the edge.

Level Ancestors in Dynamic Trees

For the `edgeExpand` operation, as shown in Algorithm 2.1, the ancestor of a node at a given depth of the tree needs to be determined efficiently. As both trees are fixed in the approach of Buchsbaum et al. (2000), a data structure for the static variant of this *level ancestor problem* (Berkman and Vishkin, 1994; Bender and Farach-Colton, 2002) is sufficient. But we want to add and remove leaves and thus we need a level ancestor data structure supporting these updates. Dietz (1991) proposes a complicated two-level data structure that provides both level ancestor queries and adding leaves in $\mathcal{O}(1)$ amortized time. The solution of Alstrup and Holm (2000), which we will use here, is much simpler. Moreover, it improves the bounds for both queries and updates to $\mathcal{O}(1)$ *worst-case* time.

In a preprocessing phase, Alstrup and Holm compute for each node v of the tree its depth, the size of the subtree rooted at v, denoted by $s(v)$, and the *rank* of v, denoted by $r(v)$. The rank of a node v is defined as the

[14] In fact, on every level of a 2-3 tree the nodes are linked, but for the leaves we explicitly need it.

[15] Red-black trees or AVL trees can be used as well, but their nodes have to be linked additionally.

maximum integer i such that $2^i | \text{depth}(v)$ and $s(v) \geq 2^i$. Then the following tables are constructed:

`levelanc`$[v][x]$ contains the x's ancestor of v for $0 \leq x \leq 2^{r(v)}$ and
`jump`$[v][i]$ contains the first proper ancestor of v whose depth is divisible by 2^i for $0 \leq i < \log(\text{depth}(v) + 1)$.

Implemented naively, this preprocessing requires $\mathcal{O}(n \log n)$ time and space, where n is the number of nodes in the tree. However, Alstrup and Holm show that both the time and the space can be improved to $\mathcal{O}(n)$. Their algorithm for determining level ancestors in constant time is remarkably simple: the `jump`$[][]$ table is used a constant number of times until a node w is found that has a rank sufficiently large to have the answer precomputed in its `levelanc`$[w][]$ table.

Adding a leaf essentially means to update the two tables, which is easily possible in $\mathcal{O}(\log n)$ time and can be improved to $\mathcal{O}(1)$ time. Deleting a leaf is not explicitly mentioned in (Alstrup and Holm, 2000), but it can be implemented in $\mathcal{O}(1)$ time (Alstrup, 2003). Obviously, we could simply delete the leaf in constant time. The space bound, however, would no longer be linear in the number of tree nodes. This can be solved by rebuilding the data structure whenever, for instance, half of the nodes have been deleted. This takes $\mathcal{O}(1)$ amortized time per delete, but using the standard doubling technique we can distribute it over a sequence of operations such that each operation is $\mathcal{O}(1)$ worst-case.[16]

Complexity

In summary our data structure for the two-dimensional tree cross product consists of the following building blocks:

Adjacency lists: For every node, the set of incident edges (both incoming and outgoing) is stored as a doubly linked list. In order to facilitate the deletion of an edge the (two) positions in such lists are stored within the edge record.

Order maintenance data structures: For each tree, a separate order maintenance data structure stores the total order of the nodes. At any time, this order corresponds to a post-order traversal of the respective tree. This assures that the descendents of a node u are uniquely determined by the two extremal nodes $\min(u)$ and $\max(u)$. All comparisons of

[16] For our application, it is actually not necessary do be able to add or remove a leaf in $\mathcal{O}(1)$ because updating the other parts of our tree cross product data structure already takes $\mathcal{O}(\Delta)$.

two nodes (of the same tree) are implemented as the corresponding `order` query.

Tree representation: Each node contains a reference to its parent, which is `null` for the root, and a list of children, which is empty for a leaf. The list of children is sorted in ascending order. Each node u stores references to the extremal nodes $\min(u)$ and $\max(u)$. In order to delete a node in $\mathcal{O}(1)$ time from its parent's list of children, every node explicitly stores its position as reference to the respective list container.

Level ancestor data structures: For each tree, a separate level ancestor data structure capable of adding and deleting leaves is maintained.

Sets $S(\cdot)$: At each node u, the set $S(u)$ is maintained as a 2-3 tree. Under a key $u_2 \in S(u_1)$ for $u_1 \in V_1$ (and symmetrically under a key $u_1 \in S(u_2)$ for $u_2 \in V_2$) the set of edges that created this entry is stored in a doubly linked list. In order to facilitate the deletion of an edge all occurrences in such lists are stored within the edge record in a doubly linked list.

Lemma 2.10 The data structure described above uses $\mathcal{O}(m2\Delta)$ space; it can be built in $\mathcal{O}(m2\Delta \log n)$ preprocessing time.[17]

Proof. An edge $e = (u_1, u_2) \in E$ can contribute an entry only to those sets $S(w)$ where w is an ancestor of either u_1 or u_2. Therefore, the space needed for all lists stored in the 2-3 trees together is $\mathcal{O}(m2\Delta)$.

The data structure is built incrementally: each edge $e = (u_1, u_2) \in E$ is inserted at each ancestor u_2' of u_2 into the list stored under the key u_1 in the 2-3 tree for the set $S(u_2')$. If the 2-3 tree for $S(u_2')$ does not exist (because it is empty), it is created; if the key u_1 is not found in $S(u_2')$, it is inserted with an empty list. For the ancestors u_1' of u_1 the insertion is performed symmetrically. Since inserting the edge in the list and storing its position in the edge record take only constant time, the update at each of the $\mathcal{O}(2\Delta)$ ancestors takes $\mathcal{O}(\log n)$ time, which gives a total of $\mathcal{O}(m2\Delta \log n)$.

The additional space for any of the order maintenance data structures (Bender et al., 2002; Dietz and Sleator, 1987) is linear in the number of elements they contain, i. e., $\mathcal{O}(n)$; preprocessing takes $\mathcal{O}(n)$ time. The level ancestor structure also can be preprocessed in linear time and space (Alstrup and Holm, 2000, Theorem 6). □

[17] As in the approach of Buchsbaum et al. (2000) (cf. Theorem 2.6), the factor 2 can be omitted here because the dimension is fixed, but it is no longer negligible if the dimension is part of the input.

Lemma 2.11 For all $u_1 \in V_1$, $u_2 \in V_2$, we can perform `edgeQuery`(u_1, u_2) in $\mathcal{O}(\log n)$ and `edgeReport`(u_1, u_2) in $\mathcal{O}(\log n + k)$ time, where k is the number of edges reported.

Proof. As described in Lemma 2.5, `edgeQuery`(u_1, u_2) is implemented by checking whether

$$\texttt{successor}(S(u_1), \min(u_2)) \leq \max(u_2).$$

Since each set $S(\cdot)$ is stored as a 2-3 tree, the `successor` operation, and thus the `edgeQuery`, takes $\mathcal{O}(\log n)$ time. Note that the lists stored under the keys of the sets $S(\cdot)$ are not needed for `edgeQuery`; the existence of a key between $\min(u_2)$ and $\max(u_2)$ in $S(u_1)$ is sufficient, as it exists only if the associated set of edges is not empty.

The first result of an `edgeReport` is found similar to an `edgeQuery`: we determine $s = \texttt{successor}(S(u_1), \min(u_2))$; if s is less or equal than $\max(u_2)$, we report all associated edges by sequentially scanning the entries in the list stored under the key s. Since the leaves of the 2-3 trees for the sets $S(\cdot)$ are linked, we find the keys following s in $S(u_1)$ in constant time each; as long as they are less or equal than $\max(u_2)$, we report all edges in their lists. □

Lemma 2.12 For all $u_1 \in V_1$, all $u_2 \in V_2$, and all $j \in \{1, 2\}$, we can perform `edgeExpand`$((u_1, u_2), j)$ in $\mathcal{O}(k \log n)$ time, where k is the number of edges reported.

Proof. We implement `edgeExpand` as described in Algorithm 2.1. Let $j = 1$; the case $j = 2$ is symmetric. Let $v_1, v_2, \ldots, v_l$ denote the children of u_1, in ascending order, i. e., $v_1 < v_2 < \cdots < v_l$.[18] We start with v_1 and determine whether it is connected to u_2 by calculating $s = \texttt{successor}(S(u_2), \min(v_1))$. If $s \leq \max(v_1)$, v_1 is reported; if $\max(v_1) < s \leq \max(v_l)$, the ancestor s' of s among the children of u_1 is reported by way of the level ancestor data structure. This procedure is iterated until v_l has been added or $s > \max(v_l)$; see Algorithm 2.1. Each `successor` operation except the last yields a new result. Since determining the level ancestor takes constant time, we get $\mathcal{O}(k \log n)$ time. □

After adding a new edge (u_1, u_2) to E, all sets $S(w)$ where w is an ancestor of either u_1 or u_2 have to be updated. We describe the procedure only for an ancestor u_1' of u_1; for the ancestors of u_2, it is performed symmetrically. We

[18] Note that the children are stored in this order and do not need to be sorted.

first search the key u_2 in the 2-3 tree for $S(u_1')$. This yields the list of edges between the subtree of u_1' and the node u_2. (If the key is not found, this list is empty and we insert a new list under the key u_2.) Into this list, we insert the edge (u_1, u_2) and store the reference to the respective list container within the edge record (in the list of occurrences).

For deleting an edge (u_1, u_2), we first delete it from all lists (adjacency lists and all lists stored in the 2-3 trees) in which it occurs by means of traversing the list of occurrences stored within the edge record. For every ancestor u_1' of u_1 (and symmetrically for every ancestor u_2' of u_2), we retrieve the list stored under the key u_2; if it has become empty, we delete the key u_2.

Since the edge knows its positions within every list, deletion takes only constant time. However, there are $\mathcal{O}(2\Delta)$ ancestors and thus $\mathcal{O}(2\Delta)$ such deletions and, what is more, $\mathcal{O}(2\Delta)$ lookups and deletions in 2-3 trees, which immediately yields the following lemma:

Lemma 2.13 For all $u_1 \in V_1$, $u_2 \in V_2$, inserting a new edge (u_1, u_2) takes $\mathcal{O}(2\Delta \log n)$ time. Deleting an edge $(u_1, u_2) \in E$ takes $\mathcal{O}(2\Delta \log n)$ time.[19] □

For deleting a leaf u, all incident edges are deleted with `deleteEdge`, which implicitly updates all affected sets $S(\cdot)$. Next, the order maintenance and the level ancestor data structures are updated. Finally, the leaf is removed from its tree, which takes only constant time, as every node knows its position in its parent's list of children. If at an ancestor u' of u either $\min(u') = u$ or $\max(u') = u$, then $\min(u')$ can be set to u's successor and $\max(u')$ to u's predecessor in the ordered list of tree nodes.

When inserting a new leaf u' as a child of node u, we insert u' right before $\max(u)$ into the order maintenance structure, add u' to the level ancestor data structure, and insert it into the tree as a child of u; all this takes $\mathcal{O}(1)$ time because of the special data structures used. Note that by placing the new node between $\min(u)$ and $\max(u)$ these values stay valid for the node u and thus for all ancestors of u if u already has been an internal node before. But if u has been a leaf, i. e., $\max(u) = \min(u)$, we have to set $\min(u) = u'$. This may cause further updates of the $\min(\cdot)$ values at ancestors of u. Since every node has at most Δ ancestors, we get the following lemma:

Lemma 2.14 Deleting a leaf (without any incident edges) and inserting a leaf take $\mathcal{O}(\Delta)$ time each. □

[19] Of course, the factor 2 can be omitted here, as the dimension is fixed. If the dimension is part of the input, however, it is no longer negligible.

Putting Lemmas 2.10, 2.11, 2.12, 2.13, and 2.14 together, we obtain the following results for the two-dimensional tree cross product problem with adding and removing leaves:

Theorem 2.15
With $\mathcal{O}(2\Delta m)$ space, we can insert or delete an edge in $\mathcal{O}(2\Delta \log n)$ time, insert a new leaf or delete a leaf (without incident edges) in $\mathcal{O}(\Delta)$ time, and perform `edgeQuery` in $\mathcal{O}(\log n)$, `edgeExpand` in $\mathcal{O}(k \log n)$, and `edgeReport` in $\mathcal{O}(\log n + k)$ time, where k is the number of edges reported. □

Compressed Trees

As already mentioned in Section 2.3.1, Buchsbaum et al. (2000) use *compressed trees* to improve the space bound and the time bounds for inserting and deleting edges. This technique could be employed here as well, but maintaining $C(T)$ subject to insertion and deletion of leaves into the original tree T is not straightforward. The problem is that these modifications change the size of the subtree rooted at an ancestor of the affected node. Therefore, any tree edge incident to such an ancestor may change its status from light to heavy and vice versa. In the compressed tree, this results in adding or removing an internal node. Especially for a new internal node this is expensive because it has to be equipped with appropriate data structures, e. g., the set $S(\cdot)$, which is of size $\mathcal{O}(m)$.

Gabow (1990) shows how to maintain an approximation $C'(T)$ of the compressed tree (in the definition of Harel and Tarjan) under `newLeaf` operations. This approximation still has the convenient property that its depth is $\mathcal{O}(\log n)$, where n is the number of nodes in the tree. Initially, $C'(T)$ is equal to $C(T)$; as long as for any node w the number of nodes in the subtree rooted at w has grown by less than a fixed factor $\alpha > 1$, a new node is simply attached appropriately. After adding a node v, this condition may be violated for some ancestors of v; in this case, the one having lowest depth is recompressed, i. e., its subtree is replaced with the correct compressed tree. The cost for recompressing a subtree is linear in the number of nodes it contains. It is easy to see that, whenever recompressing in necessary, enough nodes will have been added to the respective subtree to charge the cost of the recompression to them at a constant rate. This leads to $\mathcal{O}(1)$ amortized time for `newLeaf`.

The way Buchsbaum et al. (2000) use compressed trees would also work with the approximation of Gabow (1990). The only thing they rely on is the fact that the compressed tree has logarithmic depth, which is true for the

approximation as well. Recompressing a subtree, however, is too expensive, because the sets $S(\cdot)$ for all nodes in the subtree have to be recomputed.

2.4.2 Higher Dimensions

The two-dimensional data structure we just have introduced can be generalized using the same recursive scheme as described in Section 2.3.2. Inserting or deleting a leaf in some tree T_i does not affect the data stored at nodes of other trees. Therefore, these two operations are independent of the dimension; they are implemented like in the two-dimensional case. Following the proof of Theorem 2.8, we get the following result for the general d-dimensional tree cross product problem with adding and removing leaves:

Theorem 2.16
With $\mathcal{O}((2\Delta)^{d-1}m)$ space, we can insert or delete an edge in $\mathcal{O}((2\Delta)^{d-1}\log n)$ time, insert a new leaf or delete a leaf (without incident edges) in $\mathcal{O}(\Delta)$ time, and perform `edgeQuery` in $\mathcal{O}(d+\log n)$, `edgeExpand` in $\mathcal{O}(d+k\log n)$, and `edgeReport` in $\mathcal{O}(d+\log n+k)$ time, where k is the number of edges reported. □

Table 2.4.2 finally summarizes our results for range searching over tree cross products with adding and deleting leaves and compares them to the naive solution and the results of Buchsbaum et al. (2000) (with and without compressed trees).

2.5 Application to Graph View Maintenance

After having described the solution to the general tree cross product problem with adding and removing leaves, we apply it to the dynamic leaves variant of the graph view maintenance problem for a compound digraph $D = (V, E, F)$. Our technique for modeling compound digraphs as tree cross products is basically the same as Buchsbaum et al. (2000) used for clustered graphs. For compound digraphs, however, expanding a node gets more complicated.

2.5.1 Modeling

Let $T = (V, E)$ denote the inclusion tree of D. We set $T_1 = T_2 = T$ and interpret an adjacency edge $(u, v) \in F$ as an edge connecting $u \in T_1$ and $v \in T_2$; see Figure 2.5.

Table 2.1: Summary and comparison of the results for range searching over d-dimensional tree cross products. Let $\Delta = \max_{i=1}^{d} \operatorname{depth}(T_i)$ and $s = \min\{\Delta, \log n\}$. For `edgeReport` and `edgeExpand`, k denotes the size of the output. The `edgeQuery` and `edgeReport` bounds stated in (Buchsbaum et al., 2000) do neither include the additive d terms nor the extra factor 2^{d-1}, but they seem unavoidable given their description.

	Naive solution	Buchsbaum et al. (2000)		This work
		without compressed trees	with compressed trees	
Additional Space	$\mathcal{O}(1)$	$\mathcal{O}(m(2\Delta)^{d-1} \log\log n)$	$\mathcal{O}(m(2s)^{d-1})$	$\mathcal{O}(m(2\Delta)^{d-1})$
`edgeQuery`(u)	$\mathcal{O}(md\Delta)$	$\mathcal{O}(d + \log\log n)$	$\mathcal{O}(d + \log n/\log\log n)$	$\mathcal{O}(d + \log n)$
`edgeReport`(u)	$\mathcal{O}(md\Delta)$	$\mathcal{O}(d + \log\log n + k)$	$\mathcal{O}(d + \log n/\log\log n + k)$	$\mathcal{O}(d + \log n + k)$
`edgeExpand`(u, j)	$\mathcal{O}(md\Delta)$	$\mathcal{O}(d + k\log\log n)$	$\mathcal{O}(d + k\log n/\log\log n)$	$\mathcal{O}(d + k\log n)$
`newEdge`(u)	$\mathcal{O}(1)$	$\mathcal{O}((2\Delta)^{d-1} \log\log n)$	$\mathcal{O}((2s)^{d-1} \log n/\log\log n)$	$\mathcal{O}((2\Delta)^{d-1} \log n)$
`deleteEdge`(u)	$\mathcal{O}(1)$	$\mathcal{O}((2\Delta)^{d-1} \log\log n)$	$\mathcal{O}((2s)^{d-1} \log n/\log\log n)$	$\mathcal{O}((2\Delta)^{d-1} \log n)$
`newLeaf`(u)	$\mathcal{O}(1)$	n/a	n/a	$\mathcal{O}(\Delta)$
`deleteLeaf`(u)	$\mathcal{O}(1)$	n/a	n/a	$\mathcal{O}(\Delta)$

Furthermore, we store at each internal node $v \in V$ the set

$$N(v) = \{(u', v') \in \text{children}(v)^2 \mid (u', v') \text{ is a derived edge}\},$$

i. e., the set of derived edges between two children of v. We maintain each such set as balanced search tree with respect to the lexicographic order of edges interpreted as pairs of nodes. Furthermore, we store with every edge $(u', v') \in N(v)$ a counter for the number of adjacency edges that are represented by the derived edge (u', v').

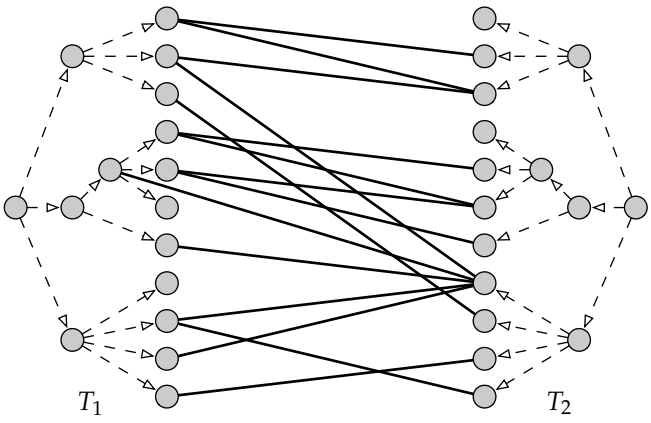

Figure 2.5: The compound digraph in Figure 1.6 modeled as a tree cross product.

2.5.2 Complexity

Determining whether there is a derived edge between two nodes u and v becomes an `edgeQuery`; `newEdge` and `deleteEdge` directly map to the corresponding operations for tree cross products. Inserting or deleting leaves in the inclusion tree results in a `newLeaf` or `deleteLeaf` operation on both T_1 and T_2.

Contracting a view at a node v is straightforward: all children of v are removed and v is connected to all former neighbors of children of v.

For expanding a view $D[U]$ at a leaf v of $T[U]$, we would like to use the `edgeExpand` operation to determine all children of v inheriting an edge $(v, u) \in F\langle U \rangle$.[20] If u also is a leaf in $T[U]$, `edgeExpand`$((v, u), 1)$ indeed yields the correct result, but if u is an internal node, it may happen that too many children are reported.

[20] We restrict the discussion to an outgoing edge $(v, u) \in F\langle U \rangle$; the arguments for an incoming edge $(u, v) \in F\langle U \rangle$ are symmetric.

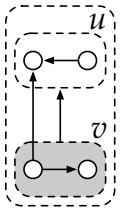

Figure 2.6: Initial view

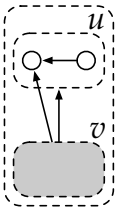

Figure 2.7: View after contracting the highlighted node v

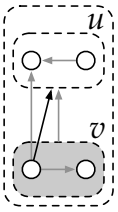

Figure 2.8: Expanding the edge (v, u) in Figure 2.7 leads to the darker shaded incorrect edge

Consider, for instance, the compound digraph $D = (V, E, F)$ in Figure 2.6, which at the same time acts as our initial view $D[V] = D$. Contracting the highlighted node v yields the view shown in Figure 2.7. Expanding v again by simply applying `edgeExpand` to both incident edges of v in Figure 2.7 results in the set of edges depicted in Figure 2.8. The darker shaded edge in Figure 2.8 is not correct according to our notion of a view (cf. Definition 1.8). It emerged from the operation `edgeExpand`$((v, u), 1)$, which was defined to return all children x of v such that (x, u) is a derived edge. In this respect, the darker shaded edge in Figure 2.8 is a perfectly correct result of the `edgeExpand` operation. However, it also shows that simply expanding all incident edges yields too many edges.

Note that Buchsbaum et al. (2000) avoid this problem because they consider the graph view maintenance problem for clustered graphs only. As opposed to the more general compound digraphs, clustered graphs have adjacency edges between leaves only. Therefore, every derived edge in a view of a clustered graph connects two leaves of the view.[21]

Lemma 2.17 Let $D[U] = (U, E[U], F\langle U\rangle)$ be a view of a compound digraph $D = (V, E, F)$ and let $D[U'] = (U', E[U'], F\langle U'\rangle)$ be the view after expanding a leaf v of $T[U]$, i. e., $U' = U \cup \text{children}(v)$. A derived edge (u, x) (or (x, u)) with $x \in \text{children}(v)$ is in $F\langle U'\rangle$ if and only if u is a leaf in $T[U']$ or there exists an edge $(u, x') \in F$ (or $(x', u) \in F$) such that $x' \in \text{desc}(x)$.

Proof. We proof the claim only for an edge (u, x); the arguments for an edge (x, u) are symmetric. If $(u, x) \in F\langle U'\rangle$, then there exists some edge $(u', x') \in F$ with $u' \in \text{desc}(u)$ and $x' \in \text{desc}(x)$ such that u and x are the nearest ancestors of u' and x' in U'. Therefore, u is a leaf or $u' = u$.

[21] In fact, a view of a clustered graph according to Buchsbaum et al. (2000) is an ordinary graph consisting only of the leaves and the derived edges between them (cf. Section 1.1).

Conversely, let (u, x) be a derived edge. If u is a leaf, then (u, x) obviously is in $F\langle U' \rangle$. If u is an internal node in $T[U']$ and there exists an edge $(u, x') \in F$ such that $x' \in \text{desc}(x)$, then x is the nearest ancestor of x' in U' and u itself is in U'. Therefore, $(u, x') \in F\langle U' \rangle$. □

Lemma 2.17 provides a sufficient criterion that a derived edge (resulting from expanding an edge incident to the node we want to expand) has to satisfy in order to be incorporated into the expanded view. Therefore, we implement `expand`$(D[U], v)$ as shown in Algorithm 2.2. After adding the children of v and all derived edges $N(v)$ connecting them (see line 3), every edge (u, v) and (v, u) in $D[U]$ is expanded. For each child v' that these `edgeExpand` operations yield, it is checked whether the corresponding expanded edge satisfies the criterion of Lemma 2.17. Note that there is an adjacency edge in D connecting a node in the subtree of v' to u if and only if $u \in S(v')$; see lines 6 and 12 in Algorithm 2.2. Finally, the edges (u, v) and (v, u) are deleted unless they are also adjacency edges in D[22]; see lines 8 and 14 in Algorithm 2.2.

Algorithm 2.2: `expand`$(D[U], v)$

input : view $D[U] = (U, E[U], F\langle U \rangle)$ and leaf v of $T[U]$
output: view $D[U'] = (U', E[U'], F\langle U' \rangle$ with $U' = U \cup \text{children}(v)$

$U' \leftarrow U \cup \text{children}(v)$
$E[U'] \leftarrow E[U] \cup \{(v, v') \mid v' \in \text{children}(v)\}$
3 $F\langle U' \rangle \leftarrow F\langle U \rangle \cup N(v)$
foreach $(v, u) \in F\langle U \rangle$ **do**
  foreach $v' \in$ `edgeExpand`$((v, u), 1)$ **do**
6 **if** *u is a leaf in $T[U']$ or $u \in S(v')$* **then** $F\langle U' \rangle \leftarrow F\langle U' \rangle \cup \{(v', u)\}$
  end
8 **if** $(v, u) \notin F$ **then** $F\langle U' \rangle \leftarrow F\langle U' \rangle \setminus \{(v, u)\}$
end
foreach $(u, v) \in F\langle U \rangle$ **do**
  foreach $v' \in$ `edgeExpand`$((u, v), 2)$ **do**
12 **if** *u is a leaf in $T[U']$ or $u \in S(v')$* **then** $F\langle U' \rangle \leftarrow F\langle U' \rangle \cup \{(u, v')\}$
  end
14 **if** $(u, v) \notin F$ **then** $F\langle U' \rangle \leftarrow F\langle U' \rangle \setminus \{(u, v)\}$
end

[22] This can easily be checked by traversing the adjacency list of v.

For a node $v \in U$ such that its children are leaves in $T[U]$, we define

$$\mathrm{Opt}(U, v) = \sum_{v' \in \mathrm{children}(v)} 1 + |\{(u, v') \in F\langle U \rangle\} \cup \{(v', u) \in F\langle U \rangle\}|.$$

In other words, $\mathrm{Opt}(U, v)$ counts all children of v and all derived edges incident to one of them. As the following lemma shows, $\mathrm{Opt}(U, v)$ is a lower bound for the number of elements (nodes or edges) that have to be added in order to expand v in the view $D[U \setminus \mathrm{children}(v)]$.

Lemma 2.18 Expanding a node v in a view $D[U]$ takes $\Omega(\mathrm{Opt}(U', v))$ time, where $D[U']$ denotes the view after expanding.

Proof. The number of nodes and edges that have to be added in the transition from $D[U]$ to $D[U']$ are a lower bound for expanding v in $D[U]$. Every added edge is incident to at least one child of v and thus counted in $\mathrm{Opt}(U', v)$. Since the 1 terms in the definition of $\mathrm{Opt}(\cdot, \cdot)$ account for the added nodes, $\Omega(\mathrm{Opt}(U', v))$ nodes and edges have to be added. □

Since contracting a node is the inverse of expanding, we get the following lower bound for `contract`:

Corollary 2.19 Contracting a node v in a view $D[U]$ takes $\Omega(\mathrm{Opt}(U, v))$ time.

Proof. The nodes and edges that have to be removed in the transition from $D[U]$ to $D[U']$ are identical to those that have to be added in the inverse `expand` operation. Therefore, the claim follows from Lemma 2.18. □

Altogether, we get the following results for the new dynamic leaves variant of graph view maintenance:

Theorem 2.20
Let $\Delta = \mathrm{depth}(T)$. With $\mathcal{O}(m\Delta)$ additional space, we can insert a new leaf or delete a leaf (without incident edges) in $\mathcal{O}(\Delta)$ time, insert or delete an edge in $\mathcal{O}(\Delta \log n)$ time, and perform `contract`(v) in a view $D[U]$ in $\mathcal{O}(\mathrm{Opt}(U, v))$ and `expand`(v) in $\mathcal{O}(\mathrm{Opt}(U', v)\Delta \log n)$ time, where $D[U']$ is the view after expanding v in $D[U]$.

Proof. By traversing all edges incident to children of v, we can find the neighbors of v after contraction. Hence, contracting v takes $\mathcal{O}(\mathrm{Opt}(U, v))$ time.

As regards expanding a node v in a view $D[U]$ with Algorithm 2.2, adding the children of v and the edges connecting two of them, which are stored

explicitly in the set $N(v)$, is linear in the number of elements added. Every `edgeExpand` operation takes $\mathcal{O}(\log n)$ per result; see Theorem 2.15. The additional checks according to Lemma 2.17 in lines 6 and 12 of Algorithm 2.2 take $O(\log n)$ time each for the lookup in a set $S(\cdot)$. It remains to set the total number of these results in relation to the optimal number, i. e., to the number of edges that actually lead to a valid derived edge in the expanded view $D[U']$. Let $(v', u) \in F\langle U'\rangle$ with $v' \in \text{children}(v)$ be such a valid edge. It follows immediately from Lemma 2.18 that this edge can cause an invalid result (v', w) only when expanding an edge (v, w) where w is an ancestor of u. Every valid result therefore leads to at most Δ unnecessary results. Since there are at most $\mathcal{O}(\text{Opt}(U', v))$ valid results, expanding v takes $\mathcal{O}(\text{Opt}(U', v)\Delta \log n)$ time.

Each adjacency edge $(u, v) \in F$ is stored exactly once in a set $N(\cdot)$; more precisely, it is stored at the nearest common ancestor of u and v. Hence, all sets $N(\cdot)$ together use $\mathcal{O}(m)$ additional space, which does not violate the $\mathcal{O}(m\Delta)$ space bound. Updating these sets after adding or removing an adjacency edge (u, v) takes additional $\mathcal{O}(\Delta)$ time for obtaining the nearest common ancestor w of u and v and $\mathcal{O}(\log n^2) = \mathcal{O}(\log n)$ time for finding the corresponding entry (u', v') in the set $N(w)$ where u' and v' are the ancestors of u and v among the children of w. The counter associated with this entry then is incremented or decremented accordingly. (If no entry was found, a new one is inserted and its counter set to 1; if the counter is 0 after decrementing, the entry is deleted.) Altogether inserting or deleting an edge takes $\mathcal{O}(\Delta + \log n)$, which is within the $\mathcal{O}(\Delta \log n)$ bound for inserting and deleting edges. All other bounds follow immediately from Theorem 2.16.□

2.5.3 Comparison

Table 2.2 compares our results to the approaches of Buchsbaum and Westbrook (2000) and Buchsbaum et al. (2000). Note that both define $\text{Opt}(\cdot, \cdot)$ without the 1 terms, i. e., they neglect all nodes that have to be added or removed. This still gives a lower bound for expanding and contracting. The time bounds they claim for these operations, however, hold only if most children of v have an incident edge in the expanded view $D[U']$, i. e., if there are at most $\mathcal{O}(\text{Opt}(U', v) - |\text{children}(v)|)$ isolated children in $D[U']$.

Our solution for the general tree cross product problem with adding and removing leaves (cf. Theorem 2.16) extends the one of Buchsbaum et al. (2000). For most operations, the extra cost for this dynamization is roughly a factor of $\log n / \log\log n$.[23] This accounts for *almost* all differences between

[23] Compared to their solution without compressed trees; with compressed trees it is roughly

the results of Buchsbaum et al. and ours shown in Table 2.2, because they basically use the same technique for modeling the problem of graph view maintenance as a tree cross product. The extra factor Δ for `expand`, however, is owed to the fact that we generalized the problem of graph view maintenance from clustered graphs to compound digraphs (cf. Definition 1.8). The former have adjacency edges only between leaves in the inclusion hierarchy and thus the extra condition of Lemma 2.17 for the results of `edgeExpand` is trivially satisfied, i. e., every result of an `edgeExpand` operation in Algorithm 2.2 yields a valid edge. Note that we could eliminate this factor in our solution by restricting our graph model to clustered graphs. Adding a leaf in a clustered graph, however, would be less natural and more complicated than it is in a compound digraph: only if an existing leaf does not have any incident edges, a new leaf may be attached to it. In a compound digraph this is no problem at all: the former leaf simply becomes an internal node with incident edges.

For clustered graphs, Buchsbaum and Westbrook (2000) describe a different solution for the problem of graph view maintenance in the dynamic graph variant, i. e., with a dynamic set of adjacency edges, but without adding or removing leaves. In their basic solution, they explicitly store every possible derived edge and link them appropriately to facilitate `expand` and `contract` operations. More precisely, they keep at each derived edge (u, v) references to all derived edges between children of u and v in a *left expansion set* $L(u, v)$ and references to those between u and children of v in a *right expansion set* $R(u, v)$. $L(u, v)$ and $R(u, v)$ thus represent the precomputed results of `edgeExpand`$((u, v), 1)$ and `edgeExpand`$((u, v), 2)$, respectively. As in our solution, the set of derived edges between children of an internal node v are stored in a set $N(v)$.

Contracting and expanding are similar to our approach. The main difference is that the results of the `edgeExpand` operations in Algorithm 2.2 are already precomputed and stored explicitly in the left and right expansion sets. This gives optimal time not only for contracting like in our approach but also for expanding a node. After inserting a new edge (u, v), all $\mathcal{O}(\Delta^2)$ derived edges between ancestors of u and v are generated if they do not already exist. In order to check this efficiently, the set of derived edges is maintained as dictionary storing for each derived edge (u, v) the number of adjacency edges it represents in a counter $c(u, v)$. For deleting an edge, all $\mathcal{O}(\Delta^2)$ combinations of ancestors are looked up in the dictionary and their counters are decremented; if a counter becomes 0 the corresponding derived

a factor of $\log \log n$.

edge is deleted. If the dictionary is implemented with dynamic perfect hashing (Dietzfelbinger et al., 1994), inserting and deleting an edge takes $\mathcal{O}(\Delta^2)$ expected time, whereas with an implementation as balanced search tree these operations take $\mathcal{O}(\Delta^2 \log n)$ time. Either way, the additional space is $\mathcal{O}(m\Delta^2)$. Buchsbaum and Westbrook (2000, Theorem 4.1) then improve this naive variant by compressing the inclusion tree as already discussed in Section 2.3.1. Essentially, this replaces the factor Δ with $s = \min\{\log n, \Delta\}$ in the above bounds; see Table 2.2.

2.6 Summary

In this chapter, an efficient data structure for range searching over tree cross products has been presented. It supports insertion and deletion of leaves and thus is more dynamic than existing solutions. As summarized in Table 2.4.2, it can compete with the approach of Buchsbaum et al. (2000), which it extends. So far it is the only data structure for range searching over tree cross products with a dynamic node set. Its application to graph view maintenance has partially solved the dynamic graph and tree variant, an open problem in (Buchsbaum and Westbrook, 2000). Table 2.2 shows that our solution can compare with the more static ones (Buchsbaum and Westbrook, 2000; Buchsbaum et al., 2000), but additionally supports insertion and deletion of graph nodes.

Table 2.2: Results and comparison for the graph view maintenance problem. Let $\Delta = \text{depth}(T)$, $s = \min\{\Delta, \log n\}$, and $\text{Opt}(U, v) = \sum_{v' \in \text{children}(v)} 1 + |\{(u, v') \in F\langle U\rangle\} \cup \{(v', u) \in F\langle U\rangle\}|$. For $\texttt{expand}(v)$, $D[U']$ denotes the view after expanding v in $D[U]$. The bounds labeled with $^{\text{exp}}$ are expected, all others are worst-case.

	Buchsbaum and Westbrook (2000)	Buchsbaum et al. (2000)		This work
		without compressed trees	with compressed trees	
Space	$\mathcal{O}(ms^2)$	$\mathcal{O}(m\Delta \log\log n)$	$\mathcal{O}(ms)$	$\mathcal{O}(m\Delta)$
$\texttt{expand}(v)$	$\mathcal{O}(\text{Opt}(U', v))$	$\mathcal{O}(\text{Opt}(U', v) \log\log n)$	$\mathcal{O}(\text{Opt}(U', v)^{\log n/\log\log n})$	$\mathcal{O}(\text{Opt}(U', v)\Delta \log n)$
$\texttt{contract}(v)$	$\mathcal{O}(\text{Opt}(U, v))$	$\mathcal{O}(\text{Opt}(U, v))$	$\mathcal{O}(\text{Opt}(U, v))$	$\mathcal{O}(\text{Opt}(U, v))$
$\texttt{newEdge}(u, v)$	$\mathcal{O}^{\text{exp}}(s^2 \log n)$	$\mathcal{O}(\Delta \log\log n)$	$\mathcal{O}(s^{\log n/\log\log n})$	$\mathcal{O}(\Delta \log n)$
$\texttt{deleteEdge}(u, v)$	$\mathcal{O}^{\text{exp}}(s^2 \log n)$	$\mathcal{O}(\Delta \log\log n)$	$\mathcal{O}(s^{\log n/\log\log n})$	$\mathcal{O}(\Delta \log n)$
$\texttt{newLeaf}(u)$	n/a	n/a	n/a	$\mathcal{O}(\Delta)$
$\texttt{deleteLeaf}(u)$	n/a	n/a	n/a	$\mathcal{O}(\Delta)$

3

Visualization

For the efficient visual navigation of compound digraphs, the data structure described in Chapter 2 is a vital building block because it ensures the responsiveness of an interactive application. Another aspect, which should not be underestimated, is the adequate *visualization* of expanding and contracting. It is not only important how efficient the underlying data structure is, but also how "efficiently" the user can follow these operations visually. With a poor visualization, it takes the user quite a long time to become familiar with the drawing again (cf. Section 1.3.4).

Our scenario is as follows: the user starts with an initial layout of some view $D[U]$, and then iteratively applies `expand` or `contract` operations. The challenge is to produce a new layout of the view after each such operation efficiently as regards both the time for the calculation and the time for the user to re-familiarize with it. The obvious solution is redrawing the entire graph after each operation; unfortunately, it is neither efficient nor will it preserve the user's mental map as already mentioned in Section 1.3.3 (cf. Figure 1.16). The basic idea therefore is to update the drawing locally after expanding or contracting, which is more efficient and preserves the user's mental map better.

Section 1.3.2 mentions various algorithms for nested drawings of hierarchically structured graphs in general and compound digraphs in particular. Although devising updates for any of them basically is possible, we have chosen the popular layered drawing style. As already described in Section 1.3.1, nearly all algorithms for layered drawings of digraphs are based on the classical approach of Sugiyama et al. (1981) consisting of four basic steps. First, the node set is partitioned into horizontal layers such that as many edges as possible point from top to bottom. Second, the edges are nor-

malized such that every edge connects only nodes on adjacent layers. Third, the nodes on each layer are reordered in order to minimize the number of crossings. Finally, the nodes' exact positions within this order are determined with respect to common aesthetic criteria such as few edge bends or balancing nodes among their neighbors.

Among the two prominent generalizations of this classical approach to compound digraphs (Sugiyama and Misue, 1991; Sander, 1999), we have chosen the one of Sugiyama and Misue (1991) as basis for an update scheme. This algorithm, briefly recalled in Section 3.1, also consists of four steps, corresponding to those of the classical algorithm for layered drawings of digraphs (Sugiyama et al., 1981). Besides the standard conventions for layered drawings (nodes are placed on horizontal levels and edges point from top to bottom), it adopts the nested drawing conventions (ND 1)–(ND 3) introduced in Section 1.3.2: nodes are drawn as axis-parallel *rectangles*, an inclusion edge (u, v) is depicted by the *geometrical inclusion* of the rectangle for v within the one for u, and the rectangles of unrelated nodes are disjoint. The user starts with a reasonably abstract initial view that is drawn with the original algorithm. After every `expand` or `contract` operation, the intermediate results and auxiliary structures used in the four steps of the previous run are adjusted with our new update scheme described in Section 3.2.

In this context of expanding and contracting nodes in nested, layered drawings, we already have introduced the four properties (MM 0)–(MM 3) (see Section 1.3.4) that constitute our notion of preserving the mental map. Essentially, we demand that all nodes that are not affected by the `expand` or `contract` operation stay on their levels (MM 1) with their relative order unchanged (MM 2) and that expanded edges take the same course as the corresponding contracted edge (MM 3). Since `expand` and `contract` are semantically inverse, we demand that they are also visually inverse, i. e., that a drawing does not change upon expanding a node and immediately contracting it again (or vice versa) (MM 0). Because of its locality, our new update scheme for the algorithm of Sugiyama and Misue (1991) preserves the user's mental map perfectly as regards these four properties.[1]

3.1 Static Layered Drawings of Compound Digraphs

This section briefly describes the original algorithm of Sugiyama and Misue (1991) for layered drawings of compound digraphs because the proposed update scheme works on its intermediate results.

[1] A preliminary version of the material in this chapter appears in (Raitner, 2004b).

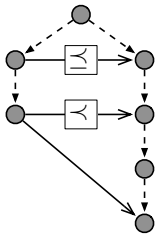

Figure 3.1: Small example of derived edges in a compound digraph D (the inclusion hierarchy is depicted as tree consisting of the dashed edges). The two edges labeled $\prec$ and $\preceq$ are derived from the solid adjacency edge between the two leaves.

3.1.1 Step I: Hierarchization

Input of this step is the original compound digraph $D = (V, E, F)$; the result is the *assigned compound digraph* $D_\mathrm{A} = (V, E, F_\mathrm{A}, \mathrm{clev})$. The layer assignment function $\mathrm{clev} : V \to \bigcup_{i \in \mathbb{N}} \mathbb{N}^i$ maps each node $v \in V$ to its *compound layer* $\mathrm{clev}(v)$, i. e., the layers are sequences of integers. For $\mathrm{clev}(v) = (n_1, \ldots, n_i) \in \mathbb{N}^i$, let $\mathrm{tail}(\mathrm{clev}(v)) = n_i$ denote the last integer in the sequence and $\mathrm{head}(\mathrm{clev}(v)) = (n_1, \ldots, n_{i-1})$ the subsequence up to but not including $\mathrm{tail}(\mathrm{clev}(v))$. For a correct layer assignment, it is required that $\mathrm{clev}(\mathrm{root}(T)) = (1)$, where $T = (V, E)$ is the inclusion tree of D, and that $\mathrm{head}(\mathrm{clev}(v)) = \mathrm{clev}(u)$ for every inclusion edge $(u, v) \in F$. The adjacency edges F_A of the assigned compound digraph result from orienting the original adjacency edges F from lower to higher layer with respect to the lexicographical order.

This step uses the *derived compound digraph* $D_\mathrm{D} = (V, E, F_\mathrm{D}, \mathrm{type})$ as an auxiliary graph; it is identical to D except for the adjacency edges F_D with their types $\mathrm{type} : F_\mathrm{D} \to \{\prec, \preceq\}$. Both the adjacency edges F_D and their types are derived from the original adjacency edges F: let $(u, v) \in F$ and let a be the nearest common ancestor of u and v in the inclusion tree; let $a = u_0, u_1, \ldots, u_k = u$ and $a = v_0, v_1, \ldots, v_l = v$ denote the unique paths (in the inclusion tree) from a to u and from a to v. In D_D this results in derived adjacency edges $(u_1, v_1) \in F_\mathrm{D}, \ldots, (u_m, v_m) \in F_\mathrm{D}$, where $m = \min(k, l)$. The edge (u_m, v_m) has $\mathrm{type}(u_m, v_m) = \prec$; all others are of type $\preceq$. In other words, every original adjacency edge $(u, v) \in F$ is replaced with edges between those ancestors of u and v having equal depth. The deepest such edge is of type $\prec$; all others of type $\preceq$; see Figure 3.1. Note that multiple edges between the same pair of nodes are unnecessary; instead, only one edge is stored and its type is updated to reflect the most restrictive condition, where $\prec$ is more restrictive than $\preceq$.

The different types of edges are considered appropriately in the layer assignment algorithm: if an edge $(u', v') \in F_D$ has type $\prec$, the layer of u' is chosen strictly *less than* the layer of v', whereas u' and v' may also reside on the *same layer* if (u', v') has type $\preceq$. For the original compound digraph D, this means that every edge $(u, v) \in F$ leads from a lower to a strictly higher layer because of the one $\prec$ edge; the ancestors of u and v, however, are kept on the same layer as long as possible. Such an assignment is possible only if all cycles of the derived compound digraph D_D entirely consist of $\preceq$ edges. Otherwise, some edges of D_D have to be deleted. Finding the minimal number of such edges, often referred to as the *feedback arc set problem* (Garey and Johnson, 1979, p. 192), is an NP-complete problem. Consequently, we cannot efficiently determine the minimal number of edges breaking all cycles in D_D and thus have to employ any of the heuristics for this well studied problem; see (Bastert and Matuszewski, 2001, pp. 91–96). Sugiyama and Misue (1991) do not use any elaborate heuristic[2], yet they take into account the different types of edges: edges of type $\preceq$ are eliminated before those of type $\prec$ and, among edges of type $\prec$, original edges, i. e., edges that are present in D as well, are eliminated before derived edges. The resulting cycle-free compound digraph is $D_F = (V, E, F_F, \text{type})$.

Compound layers are assigned to D_F as follows: the root is placed on layer (1); then children of already placed nodes are treated recursively. Note that, unlike other algorithms on trees, the children of all nodes on the same layer are always processed in a common recursive call of the layer assignment procedure. Since the parents' layers of the nodes processed in such a recursive call are already fixed, it suffices to determine the *local layer* of each child v, i. e., $\text{tail}(\text{clev}(v))$. Therefore, the subgraph of D_F induced by these children is built; the local layers of the children then are determined by a standard layer assignment algorithm that takes into account the two types of edges. The compound layer of a child v, $\text{clev}(v)$, is built by appending its local layer to its parent's compound layer, $\text{clev}(\text{parent}(v))$.

After having assigned all nodes to layers, the adjacency edges F are oriented from lower to higher layer and thus yield the adjacency edges F_A of the assigned compound digraph $D_A = (V, E, F_A, \text{clev})$; see Figure 3.2 for an example of such a layer assignment.

Let the complexity of this step be $\mathcal{O}(f_I(n))$ for some function $f_I(n)$ of the size n of the input D.

[2] It is no problem to substitute their heuristic by a more advanced.

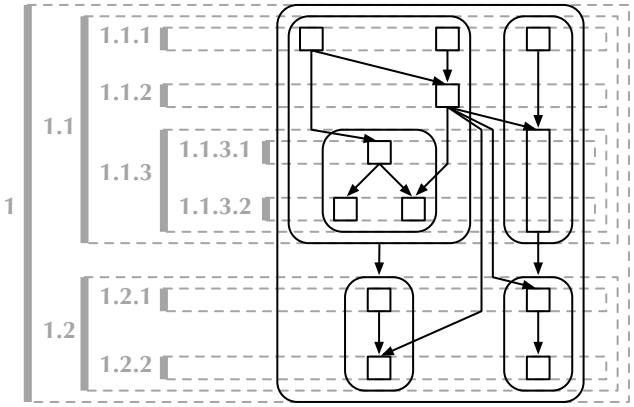

Figure 3.2: Example of a layer assignment.

3.1.2 Step II: Normalization

In this step, all adjacency edges F_A of the assigned compound digraph D_A are made *proper*. The result is the *proper assigned compound digraph* $D_P = (V_P, E_P, F_P, \mathrm{clev})$.

Definition 3.1. An adjacency edge $(u, v) \in F_A$ is proper if **proper edge**

$$\mathrm{clev}(\mathrm{parent}(u)) = \mathrm{clev}(\mathrm{parent}(v)) \quad \text{and} \tag{P1}$$

$$\mathrm{tail}(\mathrm{clev}(v)) = \mathrm{tail}(\mathrm{clev}(u)) + 1, \tag{P2}$$

i. e., the parents lie on the same layer and the children's local layers differ by one.

If $\mathrm{parent}(u) = \mathrm{parent}(v)$, condition (P1) is satisfied trivially and thus condition (P2) can be established by replacing the edge with a path of dummy nodes that are siblings of u and v; see Figure 3.3.[3]

If $\mathrm{parent}(u) \neq \mathrm{parent}(v)$ but $\mathrm{clev}(\mathrm{parent}(u)) = \mathrm{clev}(\mathrm{parent}(v))$, this path of dummy nodes is put into another dummy node p lying on the same layer as $\mathrm{parent}(u)$ and $\mathrm{parent}(v)$; see Figure 3.4. Note that this determines the depth of p in the inclusion tree, but not its place in the inclusion hierarchy, i. e., p's parent. Neither $\mathrm{parent}(u)$ nor $\mathrm{parent}(v)$ is practical because in the final drawing the edge would be closer to either u or v, which would not be justified. The nearest common ancestor of u and v is the best choice, but in general it does not have the right depth to act as the direct ancestor of p; therefore, ancestors of p are inserted as needed.

[3] Incidentally, this is similar to replacing long edges in layered drawings of ordinary digraphs; see (Sugiyama et al., 1981; Bastert and Matuszewski, 2001).

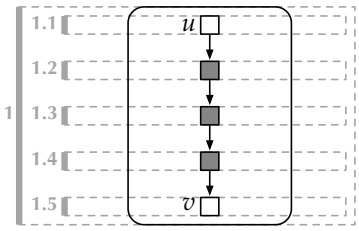

Figure 3.3: An improper edge for which $\text{parent}(u) = \text{parent}(v)$ is made proper with a path of dummy nodes that are siblings of u and v.

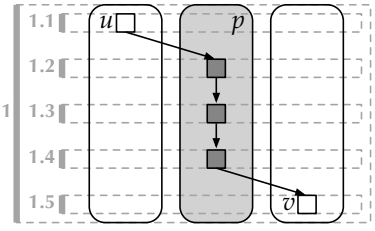

Figure 3.4: If $\text{parent}(u) \neq \text{parent}(v)$ but $\text{clev}(\text{parent}(u)) = \text{clev}(\text{parent}(v))$, the path is placed into another dummy node p on the layer of $\text{parent}(u)$ and $\text{parent}(v)$.

If $\text{clev}(\text{parent}(u)) \neq \text{clev}(\text{parent}(v))$, as depicted in Figure 3.5, a *dummy node complex* consisting of dummy nodes p and c is inserted such that p is the parent of c and p is a sibling of $\text{parent}(u)$, i. e., $\text{parent}(c) = p$ and $\text{parent}(p) = \text{parent}(\text{parent}(u))$. The edge (u, v) is split into edges (u, c) and (p, v). The layers of p and c are chosen such that the edge (u, c) is proper, i. e., $\text{clev}(p) = \text{clev}(\text{parent}(u))$ and $\text{clev}(c) = \text{clev}(u) + 1$ (introducing a new bottom layer if necessary); see Figure 3.6. At the target node v the replacement works analogously (introducing a new top layer if necessary). The edge (p, v) still may not be proper, but by applying this technique iteratively all edges (u, v) violating condition (P1) are finally reduced to an improper edge (u', v') with $\text{clev}(\text{parent}(u')) = \text{clev}(\text{parent}(v'))$ (or even $\text{parent}(u') = \text{parent}(v')$); see Figure 3.7. As described above, this improper edge is replaced with a path of dummy nodes; see Figure 3.8.

The complexity of this step is $\mathcal{O}(k)$, where k is the number of dummy nodes added.

3.1.3 Step III: Crossing Reduction

Given the proper assigned compound digraph D_P, this step calculates the relative order of the nodes on each layer. The goal is to minimize the number of edge crossings, which is a NP-hard problem (Eades and Whitesides, 1994; Eades and Wormald, 1994) even for an ordinary DAG with only two layers one of which is fixed (*one sided crossing minimization*).

Since inclusion edges shall be drawn by the inclusion of the corresponding rectangles, the children of an internal node must form a contiguous block on all of their layers. Therefore, instead of a global order of all nodes on a layer, for every internal node, only the local orders of its children on

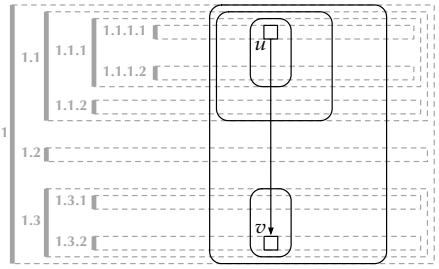

Figure 3.5: An improper edge that violates condition (P1).

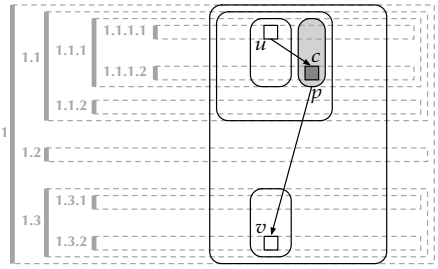

Figure 3.6: A dummy node complex consisting of parent p and child c is inserted and the improper edge is replaced with two edges the first of which is proper.

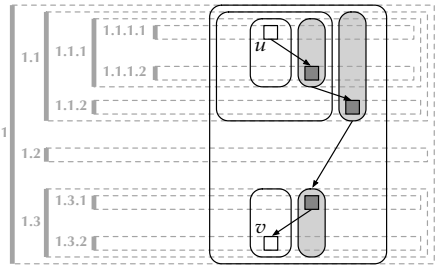

Figure 3.7: Iterating this procedure ends with an improper edge that fulfills condition (P1).

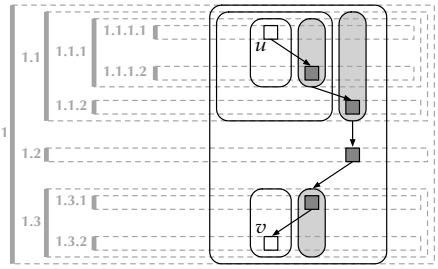

Figure 3.8: The complete sequence of dummy nodes and proper edges.

their layers are determined. The result of this step is the *ordered compound digraph* $D_O = (V_P, E_P, F_P, \text{clev}, \sigma)$, where σ describes for each internal node $u \in V_P$ the order of children(u) on their layers.

The node ordering algorithm starts at the root and traverses the inclusion tree depth-first towards the leaves. For an internal node u, the compound digraph induced by the descendants of u is reduced to an ordinary layered digraph, the *local hierarchy* for u, by shrinking each child of u into a single node; see Figure 3.9. Because all edges of D_P are proper, this leads to two types of edges in the local hierarchy: edges between nodes on adjacent layers and edges connecting nodes on the same layer. Note that for the latter multiple edges between two nodes are avoided by keeping only one of them, which is annotated with the degree of multiplicity.

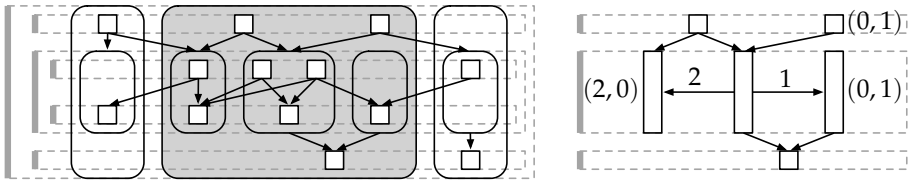

Figure 3.9: Local hierarchy (right) for the highlighted node (left). The edge labels are the multiplicities of the edges; the node labels (l, r) of a node u denote the λ and ρ values, where $\lambda(u) = l$ and $\rho(u) = r$.

A descendant u' of u may be adjacent to a node v' that is no descendant of u. By the definition of a proper edge, it follows that parent(u') and parent(v') lie on the same layer. Note that by construction the layer of an ancestor of some node w always is an initial subsequence of w's layer. Therefore, the respective ancestors of u' and v' also lie on the same layer. As a consequence, there exists an ancestor v of v' such that v and u lie on the same layer. Since the algorithm works depth-first, it has already ordered the children of all ancestors of u. In other words, it is known whether v—and therefore v'—lies to the left or to the right of u. In order to consider this in the crossing reduction of the local hierarchy every child u' is annotated with two values, $\lambda(u')$ and $\rho(u')$ counting the edges going to the left and to the right, respectively; see Figure 3.9.

The crossing reduction of the local hierarchy starts with a preprocessing step—the so-called *splitting method*—pinning the children u' with $\lambda(u') - \rho(u') > 0$ to the left end and those with $\lambda(u') - \rho(u') < 0$ to the right end of their layers; the larger $|\lambda(u') - \rho(u')|$, the nearer to the end u' is placed. For the remaining nodes the crossings are minimized with a modified barycenter heuristic (Sugiyama and Misue, 1991) that takes into account the horizontal edges. The barycenter heuristic is one of the standard heuristics for the NP-hard problem of one sided crossing minimization. It moves each node on the variable layer to the barycenter of its neighbors on the fixed layer. Usually, it is part of a layer-by-layer—top-down or bottom-up—sweep that restricts the graph to the current layer, which is variable, and the previous layer, which is fixed. See (Bastert and Matuszewski, 2001, Chap. 5.4) for a detailed description of this technique and (Forster, 2004) for an extension of it to clustered graphs.

Let the complexity of this step be $\mathcal{O}(f_{\mathrm{III}}(n))$ for some function $f_{\mathrm{III}}(n)$ of the size n of the input D_{P}.

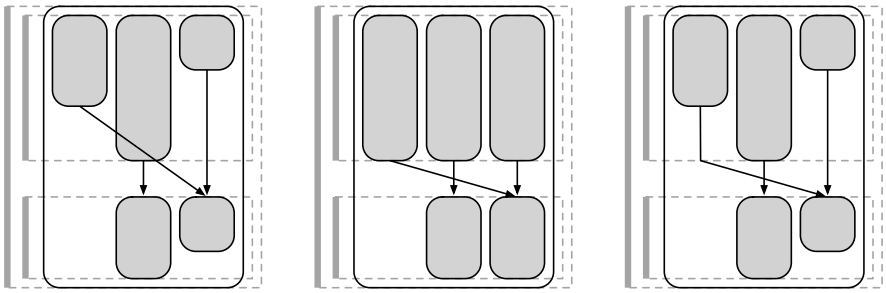

Figure 3.10: Determining the height of a node individually (based on the heights of its children) leads to edge-node crossings (left). These crossings can be avoided by choosing the same height for all nodes on the same layer (middle). With an additional bend at the boundary of a layer, the advantages of both style can easily be combined (right).

3.1.4 Step IV: Metric Layout

The purpose of this step is to assign coordinates (x and y) and dimensions (width and height) to the nodes of the ordered compound digraph D_O. So far the nodes and dummy nodes have been assigned to horizontal layers and their relative order within every layer has been fixed. Note that the vertical position essentially is determined by the layer assignment. The dimensions of a rectangle can easily be calculated as the bounding box of its children once their coordinates are fixed. The effect of this, however, is that nodes lying on the same layer may have different heights[4], which can lead to the unpleasant situation of an edge crossing a node; see Figure 3.10 (left). Therefore, instead of individual heights only the height of the entire layer is calculated, i. e., all nodes on the same layer have uniform height determined by the tallest node on the layer. It is also possible to combine both styles with additional bends on the boundaries of the layer; see Figure 3.10 (right). Although Sugiyama and Misue (1991) do not describe this part in detail, their drawings suggest that they also use this combination. Since the routing of an edge is completely determined by the position of its dummy nodes, the only degree of freedom remains the exact horizontal position of every node within the given relative order.

The horizontal positions are computed in two phases: first, the children receive *local coordinates* relative to their parent's position; second, a depth first traversal of the inclusion tree sums up the local coordinates to *absolute*

[4]They also have different widths which has to be considered appropriately in the horizontal coordinate assignment.

coordinates. The first phase is a recursive algorithm; for an internal node u, it is applied to all children of u first, thus determining their widths and heights. Starting with all children as far to the left as possible within the relative order, the local coordinates then are optimized by applying the so-called *priority layout method* to the *metrical local hierarchy*, which is the local hierarchy from the previous step without the horizontal edges.

Since the metrical local hierarchy is an ordinary layered digraph, this two-phase approach reduces the horizontal coordinate assignment problem for compound digraphs to the analogous problem for layered digraphs. More precisely, given a layered, ordered digraph the problem is to find x-coordinates for all nodes, such that the *minimum separation constraint* is satisfied, i. e., the coordinates of two nodes on the same layer differ by at least some given value δ. According to (Brandes and Köpf, 2001) a horizontal coordinate assignment should additionally satisfy the following criteria, which seem to have great influence on the readability of the layered drawing with a given ordering:

- edges should have small length,
- the position of a node should be balanced between upper and lower neighbors,
- and long edges should have few bends, i. e., should be as straight as possible.

Originally, Sugiyama et al. (1981) formulate this problem as a quadratic program, for which it can be time-consuming to find a solution on large instances. Therefore, Sugiyama and Misue (1991) propose a faster, iterative heuristic, the priority layout method. Similar to the barycenter heuristic for minimizing crossings, the priority layout method tries to improve the nodes' positions by moving them—as far as possible without changing the order on the layers—to their respective (metrical) barycenters. Each node has a certain *priority* reflecting the importance of this nodes being placed at the barycenter of its neighbors. When a node is moved, it may displace all nodes with lower priority to increase the available space. By giving the highest priority to dummy nodes they are more likely to end up at their optimal position, which favors the straightness of long edges.

As summarized by Brandes and Köpf (2001), there are also other optimization approaches, apart from the quadratic program of Sugiyama et al. (1981). Eades et al. (1996b), for instance, use a system of linear equations that places each node at the mean coordinate of its neighbors averaging the influence of upper and lower neighbors. Some implementations (Gutwenger et al.,

2001; Ellson et al., 2001) minimize the length of the edges with a piecewise linear objective function subject to the minimum separation constraint. In order to straighten long edges, each edge gets a weight reflecting the importance of drawing that edge vertically. The more dummy nodes are incident with the edge, the more weight it gets, i. e., edges connecting two dummy nodes, which are the inner segments of a long edge, have highest priority. Also, some variations and improvements of the priority layout method have been proposed. Sander (1999), for instance, considers the average coordinates of all neighbors instead of treating upper and lower neighbors separately; rather than shifting nodes according to their priority, they are grouped and their common movement is determined as the average of the individual shifts.

Brandes and Köpf (2001) present a linear time, non-iterative heuristic approach that can compare well with the above mentioned in terms of assignment quality. It guarantees that an edge connecting two dummy nodes is vertical and thus that a long edge has at most two bends. This heuristic consists of two phases. First, four *vertical alignments* are calculated each prescribing which edges are drawn vertically. Note that the optimal, i. e., the most balanced, position of a node u with upper (or lower) neighbors $u_1, \ldots, u_k$ is to align it vertically with its median neighbor. An alignment encodes these restrictions for all nodes with respect to upper (or lower) neighbors. Whenever conflicts in an alignment occur, they are resolved either in leftmost or rightmost fashion. In either case, the conflict resolution favors inner segments, i. e., edges connecting two dummy nodes are always vertical.[5] This results in four different alignments: one for each combination of upper and lower neighbors with leftmost and rightmost conflict resolution.

A horizontal coordinate assignment then is determined for each vertical alignment separately, which inherently has a certain bias toward either upper or lower neighbors with either leftmost or rightmost conflict resolution. We describe only the case of upward alignment to the left; the other three cases are symmetric. For this *horizontal compaction*, the nodes are partitioned into maximal sets of vertically aligned nodes called *blocks*. The *block graph* is obtained by inserting a directed edge from a node to its predecessor on the same layer and by contracting blocks into single nodes. Note that the block graph is acyclic. Brandes and Köpf (2001) subdivide it into classes, defined by the sinks, and apply a variant of *longest path layering* within each class,

[5] The case that two inner segments are in conflict, which prohibits one of them being vertical, can be avoided by swapping their lower neighbors until the crossing is no longer between two inner segments.

i. e., the relative coordinate of a block with respect to the sink in this class is the maximum coordinate of the preceding blocks in the same class.

In the second phase, the four separate assignments are combined in order to average out their individual directional bias. The layouts are aligned with the one of smallest width among them. Then, the *average median*[6] of the four candidate coordinates of each node is chosen as its final coordinate.

Its linear running time and its simplicity together with its appealing results make the approach of Brandes and Köpf (2001) a good candidate for replacing the priority layout method suggested by Sugiyama and Misue (1991). Our modified algorithm for the metrical layout therefore works as follows: for each node v we determine recursively the width and height of each child; then, the local coordinates of the metrical local hierarchy of v are determined with the algorithm of Brandes and Köpf (2001). Note that in (Brandes and Köpf, 2001) all nodes have uniform dimensions, whereas in the metrical local hierarchies the sizes of the nodes differ. This generalization, however, is straightforward. Finally, the absolute coordinates are summed up in a top-down traversal of the inclusion tree.

In the final drawing the dummy nodes have to be replaced with bends of the corresponding edge. Sugiyama and Misue (1991) suggest to derive the bends from the positions of the dummy nodes by setting the width of dummy nodes to zero in the first phase of the above algorithm. In our setting, i. e., with the guarantees of the horizontal coordinate assignment of Brandes and Köpf, this leads to the following tight lower and upper bound on the number of bends of a single edge.

Lemma 3.2 Let $u, v \in V$ with $\mathrm{clev}(u) = (x_1, \ldots, x_r, u_1, \ldots, u_k)$ and $\mathrm{clev}(v) = (x_1, \ldots, x_r, v_1, \ldots, v_l)$ such that r is maximal, i. e., $x_1, \ldots, x_r$ is the longest common initial subsequence of $\mathrm{clev}(u)$ and $\mathrm{clev}(v)$. If the width of the dummy nodes is set to zero, an adjacency edge $(u, v) \in F_A$ has between $2(k+l)-4$ and $2(k+l)-2$ bends.

Proof. Consider the case $k = 1$ and $l = 1$, which means that the parents of u and v lie on the same layer. Hence, (u, v) is made proper with a path of dummy nodes that are either siblings of u and v (if u and v have the same parent) or reside in another dummy node on the same layer as the parents of u and v (cf. Figures 3.3 and 3.4, respectively). In either case, the whole path of dummy nodes belongs to the same metrical local hierarchy. Since the horizontal coordinate assignment of Brandes and Köpf (2001) guarantees a

[6] For k values $x_1 \leq \cdots \leq x_k$ the average median is $(x_{\lfloor (k+1)/2 \rfloor} + x_{\lceil (k+1)/2 \rceil})/2$.

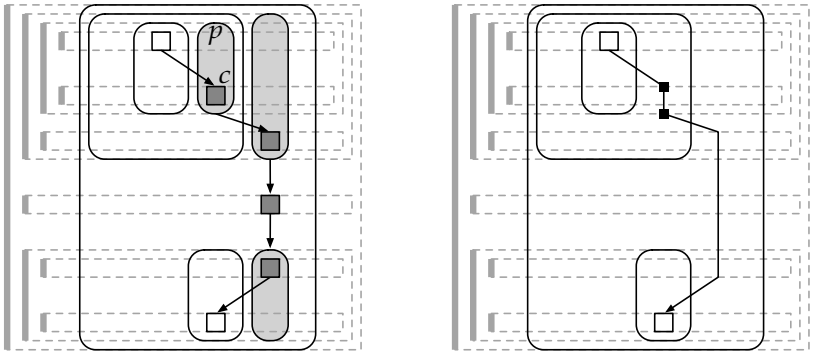

Figure 3.11: Setting the width of the dummy node complex consisting of p and c (left) to zero necessarily leads to the two bends highlighted with black squares (right).

maximum of two bends, the edge (u, v) has between $0 = 2(k + l) - 4$ and $2 = 2(k + l) - 2$ bends.

Suppose that the claim already holds for $k - 1$ ($k > 1$) and l. Since $k > 1$, the parents of u and v cannot lie on the same layer. As described in Section 3.1.2, a dummy node complex consisting of a node p with one child c is inserted such that p lies on the same layer as parent(u); the edge (u, v) is split into edges (u, c) and (p, v). Since clev$(p) = (x_1, \ldots, x_r, u_1, \ldots, u_{k-1})$, we know by induction that the edge (p, v) has between $2((k - 1) + l) - 4 = 2(k + l) - 6$ and $2((k - 1) + l) - 2 = 2(k + l) - 4$ bends. Shrinking the width of the dummy nodes p and c to zero necessarily leads to two bends; see Figure 3.11. Therefore, the edge (u, v) has between $2(k + l) - 4$ and $2(k + l) - 2$ bends. For the replacement at the target v, the arguments are symmetrical. □

Theorem 3.3
The metrical layout algorithm of Sugiyama and Misue (1991) with the horizontal coordinate assignment of Brandes and Köpf (2001) takes $\mathcal{O}(n)$ time, where n is the size of the input D_O. The total number of edge bends is $\mathcal{O}(m\Delta)$.

Proof. The number of bends follows immediately from Lemma 3.2 because every edge can have at most $\mathcal{O}(\Delta)$ bends. Since every node is contained in exactly one local hierarchy and the horizontal coordinate assignment of Brandes and Köpf (2001) is linear in the size of the respective local hierarchy, the complexity of this step is linear in the size of the input D_O. □

The two bends for the dummy node complex in Figure 3.11 appear to be unnecessary. In this special case they indeed could (and should) be replaced with a straight line. In general, however, some sibling of p might be lying right of it. The bends then assure that no such sibling is crossed by the edge (u, v). As regards readability, avoiding such crossings seems to be at least as important as minimizing the number of bends. Thus, the seemingly unnecessary bends should be removed only in special cases like the one shown in Figure 3.11.

Other bends, however, can always be avoided. Recall that if u already lies on the bottom layer within its parent, an extra bottom layer is introduced for the child c of a dummy node p. (Symmetrically, an extra top layer is created if v lies on the top layer.) As such an extra layer contains only dummy nodes, its height can be chosen freely. In particular, it can be set to zero, which means that the two bends for the respective dummy node complexes overlap.

3.2 Expansion

Let $D[U] = (U, E[U], F\langle U\rangle)$ be a view of a compound digraph $D = (V, E, F)$ and assume that $D[U]$ already has been drawn with the static algorithm of Sugiyama and Misue (1991) (or has been updated with the proposed update scheme). Now consider a leaf $v \in U$ that shall be expanded, resulting in a new view $D[U'] = (U', E[U'], F\langle U'\rangle)$, with $U' = U \cup \text{children}(v)$. In order to simplify the notation in the description of the update scheme, let $\mathcal{D} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ and $\mathcal{D}' = (\mathcal{V}', \mathcal{E}', \mathcal{F}')$ denote the views $D[U]$ and $D[U']$, respectively.

It is assumed that the combinatorial structure of $\mathcal{D}'$ can be determined efficiently from $\mathcal{D}$, which is where the data structures for maintaining hierarchical graph views, described in Chapter 2, come into play. In the following, our novel method for updating the *drawing* of $\mathcal{D}$ to a drawing of $\mathcal{D}'$ by locally adjusting the intermediate results of steps I–IV is described.

There are various goals that the proposed update scheme tries to achieve: first, it shall be efficient; second, the user's mental map measured in terms of the properties (MM 0)–(MM 3) introduced in Section 1.3.4 shall be preserved; and third, the updated drawing shall be "nice" in terms of common aesthetic criteria such as small area, short edges, few edge crossings, or few edge bends. All these goals are measured in relation to redrawing the new view entirely with the original algorithm because this is the straightforward alternative to our local updates of the old drawing. This means that the

update should be more efficient and better preserving the mental map than redrawing, yet the updated drawing shall resemble a complete redrawing as close as possible.

3.2.1 Step I: Hierarchization

In this step, the assigned compound digraph $\mathcal{D}_A = (\mathcal{V}, \mathcal{E}, \mathcal{F}_A, \text{clev})$ for the old view $\mathcal{D}$ has to be updated to the assigned compound digraph $\mathcal{D}_A{}' = (\mathcal{V}', \mathcal{E}', \mathcal{F}_A{}', \text{clev}')$ for the new view $\mathcal{D}'$. For preserving the mental map, property (MM 1) demands that the layers of all old nodes $u \in \mathcal{V}$ are not changed. Hence, only for the children of the expanded node v appropriate layers have to be found, i. e., $\text{clev}(u) = \text{clev}'(u)$ for all $u \in \mathcal{V}$.

As shown in Algorithm 3.1, updating the assigned compound digraph starts in line 1 with expanding v in $\mathcal{D}_A$ with Algorithm 2.2. Note, that $\mathcal{D}_A$ is not necessarily a correct view of D because of the reversed adjacency edges, which do not correspond to original adjacency edges in D. Algorithm 2.2, however, is defined only for views of D because it relies on the fact that all edges incident to v are derived and thus `edgeExpand` can be applied to them. Therefore, line 1 in Algorithm 3.1 should not be taken literally; it rather means that v is expanded as described in Algorithm 2.2, but instead of expanding a reversed adjacency edge the original edge is expanded and the resulting expanded edges are reversed again. The result is the same as if for each reversed edge in $\mathcal{D}_A$ all corresponding adjacency edges in D also had been reversed, i. e., as if $\mathcal{D}_A$ was a correct view of D.

After v has been expanded, all old nodes receive the same layer as in $\mathcal{D}_A$. For an edge $(u, v) \in \mathcal{F}_A$ either $(u, v) \in \mathcal{F}$ or $(v, u) \in \mathcal{F}$ because adjacency edges in $\mathcal{D}_A$ are always oriented from lower to higher layer regardless of their original direction in $\mathcal{D}$. Since all old nodes stay on their layers, each expanded edge necessarily inherits the direction of the respective contracted edge (regardless of its original direction in $\mathcal{D}'$). Nevertheless, the edges between the children of v can introduce new cycles, but any such cycle consists entirely of these edges. Therefore, the cycle removal in $\mathcal{D}_A{}'$ can be restricted to the subgraph induced by the children of v, which is an ordinary graph; see line 5 in Algorithm 3.1.

After the new cycles have been removed, the local layers of the new children are determined on the subgraph induced by the children of v; see line 6 in Algorithm 3.1. This subgraph is an ordinary directed acyclic graph, just like the subgraphs that act as input for the original recursive layer assignment algorithm in Section 3.1.1; therefore, the new children's local layers are calculated on this subgraph using an ordinary layer assignment

Algorithm 3.1: updateLevels($\mathcal{D}_A, v$)

input : $\mathcal{D}_A = (\mathcal{V}, \mathcal{E}, \mathcal{F}_A, \text{clev})$: assigned compound digraph,
v: leaf to be expanded
output: $\mathcal{D}_A{}' = (\mathcal{V}', \mathcal{E}', \mathcal{F}_A{}', \text{clev}')$: assigned compound digraph with v expanded

1 $\mathcal{D}_A{}' \leftarrow \texttt{expand}(\mathcal{D}_A, v)$
forall $u \in \mathcal{V}$ **do** $\text{clev}'(u) = \text{clev}(u)$

let $E_c = \{(u', v') \in \mathcal{F}_A{}' \mid \{u', v'\} \subseteq \text{children}(v)\}$
let $G = (\text{children}(v), E_c)$

5 determine edges $E_r \subseteq E_c$ such that $G' = (\text{children}(v), E_c \setminus E_r)$ is acyclic
6 determine a local layer assignment lev : $\text{children}(v) \to \mathbb{N}$ for G'
forall $v' \in \text{children}(v)$ **do** $\text{clev}'(v') = \text{append}(\text{clev}(v), \text{lev}(v'))$

forall $(u, w) \in E_r$ **do**
  if $\text{clev}'(w) < \text{clev}'(u)$ **then** $\mathcal{F}_A{}' \leftarrow \mathcal{F}_A{}' \cup \{(w, u)\} \setminus \{(u, w)\}$
end

method as described in Section 3.1.1. This assures that every edge between two children of v that survived the cycle removal points from lower to higher layer. That this is also true for expanded edges is shown by the following lemma:

Lemma 3.4 Let $v' \in \text{children}(v)$ and $u \notin \text{children}(v)$. For an expanded edge (v', u) (or (u, v')), $\text{clev}'(v') < \text{clev}'(u)$ (or $\text{clev}'(v') > \text{clev}'(u)$) regardless of the local layer assigned to v'.

Proof. We prove the claim for the edge (v', u); for (u, v') the arguments are symmetric. Before expanding v the edge (v', u) was represented by the contracted edge (v, u); therefore, $\text{clev}(v) < \text{clev}(u)$. Since all old nodes stay on their layers, i. e., $\text{clev}(u) = \text{clev}'(u)$ for all $u \in \mathcal{V}$, we get $\text{clev}'(v) < \text{clev}'(u)$. Because of the lexicographical order of the layers, it follows that $\text{clev}'(v') < \text{clev}'(u)$ regardless of the local layer of v'. □

Lemma 3.4 proves that local layer assignment is sufficient in the chosen setting where all old nodes stay on their layers. In order to evaluate the quality of this update, it remains to inspect the dissimilarity of the updated assigned compound digraph $\mathcal{D}_A{}'$ compared to the assigned compound digraph resulting from re-applying the hierarchization algorithm of Section 3.1.1 to the new view $\mathcal{D}'$. The differences can be seen best by comparing the two derived compound digraphs: the one that re-applying the hierarchization internally uses and the one that conceptually stands behind

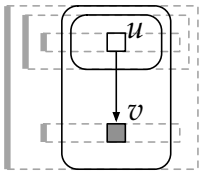

Figure 3.12: Situation before expanding v: the adjacency edge (u, v) leads to a type $\prec$ derived edge $(\text{parent}(u), v)$; therefore, $\text{parent}(u)$ is being placed on a layer above v.

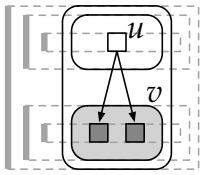

Figure 3.13: Result of updating the layer assignment: all old nodes stay on their layers.

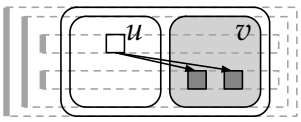

Figure 3.14: Result of re-applying the hierarchization algorithm: nodes $\text{parent}(u)$ and v are placed on the same layer.

our update, although we did not construct it explicitly because the assigned compound digraph is updated directly in Algorithm 3.1.

Consider, for instance, an expanded edge (u, v') with $v' \in \text{children}(v)$ and $u \notin \text{children}(v)$ such that $\text{depth}(v) < \text{depth}(u)$; see Figure 3.12 for an example. If the derived compound digraph for $\mathcal{D}'$ is built entirely anew, as described in Section 3.1.1, this edge leads to a derived edge (u', v') of type $\prec$, where u' is the ancestor of u with $\text{depth}(u') = \text{depth}(v')$. Before expanding v, the edge (u, v') was represented by the contracted edge (u, v), which in turn led to a derived edge $(\text{parent}(u'), v)$ of type $\prec$. After expanding v, the edge (u', v') becomes the deepest derived edge; therefore, the type of the edge $(\text{parent}(u'), v)$ changes from $\prec$ to $\preceq$. In other words, if the derived compound digraph for $\mathcal{D}'$ is built entirely anew, nodes v and $\text{parent}(u')$ are placed on the same layer—given that the other edges permit this—, whereas they are placed on different layers before expanding v. Since the update keeps all old nodes on their former layers, v and $\text{parent}(u')$ remain on different layers, although they could have been placed on the same. The effect of this can be seen by comparing Figure 3.13, which shows the layer assignment produced by the update, and Figure 3.14, which is the result of re-applying the hierarchization: the adjacency edge (u, v) in Figure 3.12 leads to a type $\prec$ derived edge $(\text{parent}(u), v)$, which results in $\text{parent}(u)$ being placed on a layer above v; if the edge has type $\preceq$, $\text{parent}(u)$ and v are placed on the same layer, as shown in Figure 3.14.

Locality Property 1 Let k denote the number of nodes and edges added to $\mathcal{D}$ by expanding v. The complexity of updating the assigned compound digraph $\mathcal{D}_A$ is $\mathcal{O}(f_I(k))$, compared to $\mathcal{O}(f_I(n+k))$ for re-applying step I to $\mathcal{D}'$, where n is the size of $\mathcal{D}$. All old nodes $\mathcal{V}$ stay on their former layer (MM 1).

3.2.2 Step II: Normalization

The *proper* assigned compound digraph $\mathcal{D}_\mathrm{P} = (\mathcal{V}_\mathrm{P}, \mathcal{E}_\mathrm{P}, \mathcal{F}_\mathrm{P}, \mathrm{clev})$ for the old assigned compound digraph $\mathcal{D}_\mathrm{A}$ is updated as shown in Algorithm 3.2. In line 1, Algorithm 3.1 first is applied to expand node v in $\mathcal{D}_\mathrm{P}$ and to determine the layers of v's children. As already defined in Section 3.2.1, this step must not be taken literally because $\mathcal{D}_\mathrm{P}$ is no correct view of D: compared to the original view $\mathcal{D}$, the adjacency edges of $\mathcal{D}_\mathrm{P}$ could be reversed or replaced with a sequence of dummy nodes and edges or both. Expanding v in $\mathcal{D}_\mathrm{P}$ with Algorithm 2.2, which is the first step of Algorithm 3.1, therefore is not well-defined because the edges incident to v in $\mathcal{D}_\mathrm{P}$ in general do not correspond to derived edges in D and thus cannot be expanded. The intended result of expanding an edge incident to v in $\mathcal{D}_\mathrm{P}$, however, is obvious. Each edge between v and a dummy node u uniquely corresponds to an adjacency edge in $\mathcal{D}$, which can be expanded and thus yields the children inheriting this edge. In $\mathcal{D}_\mathrm{P}{}'$, we simply insert an edge between each of these children and the dummy node u in the right direction.

After expanding v and assigning layers to the new children of v with Algorithm 3.1, every improper adjacency edge is necessarily incident to at least one child of v because the layers of the old nodes $\mathcal{V}$ are unchanged. The edges that are incident with *two* children of v are easy: they are made proper with a simple sequence of dummy nodes because both children trivially satisfy the condition (P1), namely that their parents, which is v in this case, lie on the same layer.

By expanding v in the old proper assigned compound digraph $\mathcal{D}_\mathrm{P}$ we intentionally deviate from what re-applying the normalization to the updated assigned compound digraph $\mathcal{D}_\mathrm{A}$ would have yielded. Consider, for instance, an (improper) edge $(v, u) \in \mathcal{F}_\mathrm{A}$ with its corresponding (improper) expanded edges $(v_1, u), \ldots, (v_k, u) \in \mathcal{F}_\mathrm{A}{}'$. In $\mathcal{D}_\mathrm{P}$ the edge (v, u) has been made proper with a sequence of dummy nodes as in the example shown in Figure 3.15. Applying the normalization to $\mathcal{D}_\mathrm{A}{}'$ would replace *each* expanded edges (v_i, u) with a distinct sequence of dummy nodes; see Figure 3.16. Note that every such sequence is—except for an extra dummy node complex on the layer of v—identical to the one for edge (v, u).

One goal of the proposed update scheme is that expanded edges take the same course as the corresponding contracted edge (MM 3), which means that during the crossing reduction the dummy nodes for the expanded edges have to be treated as blocks. Therefore, it is unnecessary to normalize each expanded edge separately in this step. Instead, we expand v within $\mathcal{D}_\mathrm{P}$, which does not expand the edge (v, u) entirely, but only up to its first

Algorithm 3.2: updateProper$(\mathcal{D}_P, v)$

input : $\mathcal{D}_P = (\mathcal{V}_P, \mathcal{E}_P, \mathcal{F}_P, \text{clev})$: proper assigned compound digraph,
v: leaf to be expanded
output: $\mathcal{D}_P{}' = (\mathcal{V}_P{}', \mathcal{E}_P{}', \mathcal{F}_P{}', \text{clev}')$ proper assigned compound digraph with v expanded

1 $\mathcal{D}_P{}' \leftarrow$ updateLevels$(\mathcal{D}_P, v)$

foreach *improper edge* $(u', v') \in \mathcal{F}_P{}'$ *such that* $\{u', v'\} \subseteq \text{children}(v)$ **do**
  make (u', v') proper
end

foreach *contracted edge* $(v, u) \in \mathcal{F}_P$ **do**
  // insert dummy node p_u on v's layer
  $\mathcal{V}_P{}' \leftarrow \mathcal{V}_P{}' \cup \{p_u\}$, $\mathcal{E}_P{}' \leftarrow \mathcal{E}_P{}' \cup \{(\text{parent}(v), p_u)\}$, $\text{clev}'(p_u) = \text{clev}'(v)$
  $\mathcal{F}_P{}' \leftarrow \mathcal{F}_P{}' \cup \{(p_u, u)\}$
  foreach *corresponding expanded edge* (v', u) **do**
    let $l = \text{append}(\text{clev}'(p_u), \text{tail}(\text{clev}'(v')) + 1)$
    if *there is not already a child c of* p_u *on layer* l **then**
      $\mathcal{V}_P{}' \leftarrow \mathcal{V}_P{}' \cup \{c\}$, $\mathcal{E}_P{}' \leftarrow \mathcal{E}_P{}' \cup \{(p_u, c)\}$
      $\text{clev}'(c) = l$
    end
    $\mathcal{F}_P{}' \leftarrow \mathcal{F}_P{}' \cup \{(v', c)\} \setminus \{(v', u)\}$
  end
end

foreach *contracted edge* $(u, v) \in \mathcal{F}_P$ **do**
  // symmetric to $(v, u) \in \mathcal{F}_P$
  $\mathcal{V}_P{}' \leftarrow \mathcal{V}_P{}' \cup \{p_u\}$, $\mathcal{E}_P{}' \leftarrow \mathcal{E}_P{}' \cup \{(\text{parent}(v), p_u)\}$, $\text{clev}'(p_u) = \text{clev}'(v)$
  $\mathcal{F}_P{}' \leftarrow \mathcal{F}_P{}' \cup \{(u, p_u)\}$
  foreach *corresponding expanded edge* (u, v') **do**
    let $l = \text{append}(\text{clev}'(p_u), \text{tail}(\text{clev}'(v')) - 1)$
    if *there is not already a child c of* p_u *on layer* l **then**
      $\mathcal{V}_P{}' \leftarrow \mathcal{V}_P{}' \cup \{c\}$, $\mathcal{E}_P{}' \leftarrow \mathcal{E}_P{}' \cup \{(p_u, c)\}$
      $\text{clev}'(c) = l$
    end
    $\mathcal{F}_P{}' \leftarrow \mathcal{F}_P{}' \cup \{(c, v')\} \setminus \{(u, v')\}$
  end
end

dummy node; see Figure 3.17. The dummy nodes for the old edge (v, u) thereby become representatives for a block of dummy nodes for the expanded edges. At the end of the crossing reduction this "mistake" will be corrected by splitting up each such representative into the corresponding set of dummy nodes, i. e., after the crossing reduction each expanded edge is represented by a distinct sequence of dummy nodes as in Figure 3.16.

Now, let d_u be the first dummy node of the edge (v, u) (the procedure is symmetric for an edge (u, v)). Expanding the proper edge $(v, d_u) \in \mathcal{F}_P$ yields expanded edges $(v_1, d_u), \ldots, (v_k, d_u) \in \mathcal{F}_P{}'$ that are all improper; see Figure 3.17. Making these edges proper as described in Section 3.1.2 would generate for each edge (v_i, d_u) one dummy node p_i on the layer of v with one child c_i whose local layer is one greater than the local layer of v_i. This is, however, unnecessary, as the remainder of all edges (v_i, u) is not expanded, but rather treated as one block. Therefore, all p_i are conceptually merged into a single block, i. e., only *one* dummy node p_u is inserted on the layer of v together with an edge (p_u, d_u). Then, each edge (v_i, d_u) is replaced with an edge (v_i, c_i), where c_i is a child of p_u on a local layer one greater than the local layer of v_i; see Figure 3.18. Since the dummy nodes have to be split anyway after the crossing reduction, one node c_i per local layer is sufficient; see Algorithm 3.2 for a detailed description.

Locality Property 2 The complexity of making $\mathcal{D}_P{}'$ proper again after expanding v in $\mathcal{D}_P$ is $\mathcal{O}(l)$, where l is the number of *additional* dummy nodes needed. On the other hand, the complexity of normalizing the assigned compound digraph $\mathcal{D}_A{}'$ as a whole is $\mathcal{O}(k + l)$ where k is the number of dummy nodes in $\mathcal{D}_P$.

3.2.3 Step III: Crossing Reduction

In this step the ordered compound digraph $\mathcal{D}_O = (\mathcal{V}_P, \mathcal{E}_P, \mathcal{F}_P, \text{clev}, \sigma)$ is updated. Note that the only difference between $\mathcal{D}_O$ and $\mathcal{D}_P$ is σ that encodes for each internal node the order of its children on their layers. Therefore, the actual challenge in this step is to update σ to σ'. As preserving the mental map demands preserving the order of all old nodes (MM 2), only for the nodes that have been added during the update from $\mathcal{D}_P$ to $\mathcal{D}_P{}'$ appropriate positions on their layers have to be determined. These nodes are either children of v—including dummy nodes for edges between two children of v—or dummy nodes that belong to an expanded edge between a child of v and a node $u \notin \text{children}(v)$. As described in Section 3.1.3, the node ordering algorithm recursively calculates a local order for the children of

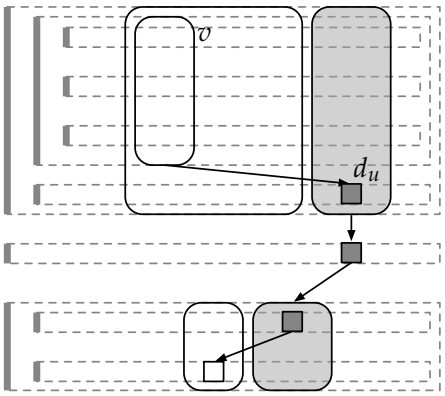

Figure 3.15: Dummy nodes and edges for the contracted edge before expanding v.

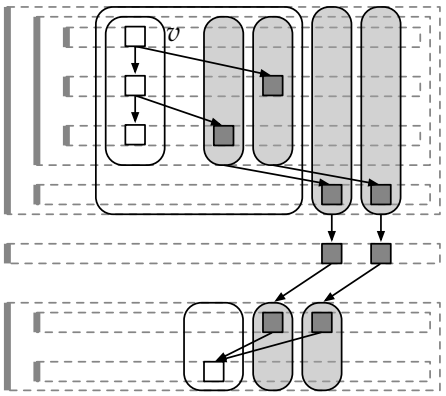

Figure 3.16: Dummy nodes and edges for the corresponding expanded edges.

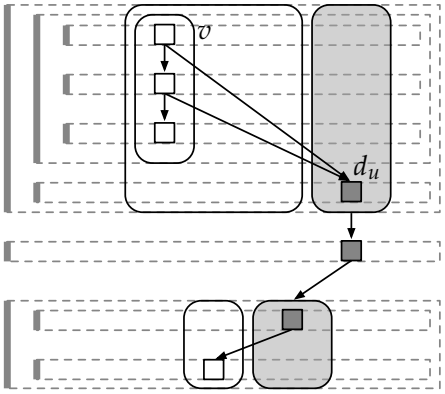

Figure 3.17: Improper expanded edges after expanding the contracted edge up to the first dummy node d_u.

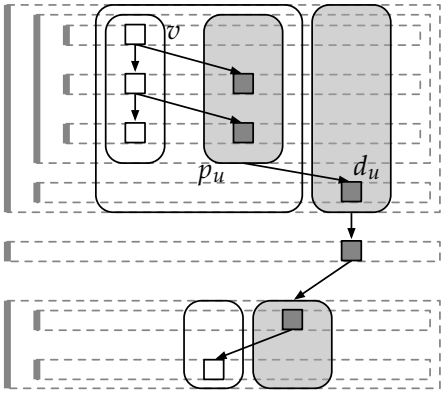

Figure 3.18: Dummy nodes for the expanded edges grouped into p_u.

an internal node. Hence, determining the order of the children of v is just a matter of applying this algorithm to the subtree rooted at v. A precondition, however, is that the children of all ancestors of v already have been ordered. Therefore, the positions of dummy nodes that are not children of v have to be fixed prior to ordering the children of v.

Consider an edge $(v, u) \in \mathcal{F}_A$ (the arguments for an edge $(u, v) \in \mathcal{F}_A$ are symmetric) such that children $v_1, \ldots, v_k$ inherit this edge after expanding v. Remember that in the update from $\mathcal{D}_P$ to $\mathcal{D}_P'$ (cf. Algorithm 3.2) (v, u) is expanded only up to its first dummy node d_u and that the expanded edges are made proper with a single dummy node p_u (with some children on appropriate layers). Being not expanded, the old dummy nodes of the edge (v, u) are reused for representing blocks of dummy nodes of the expanded edges (v_i, u); compare Figures 3.15 and 3.16. Since they already have valid positions and the relative order of all old nodes is preserved, all expanded edges (v_i, u) inherit the course of the edge (v, u).

The dummy node p_u, however, is new and lacks a valid position.[7] In other words, the set of nodes that have to be arranged and that are not children of v consists of one dummy node p_u for every neighbor u of v in $\mathcal{D}_A$. They are all siblings of v and lie on the same layer as v. Therefore, the original node ordering algorithm (cf. Section 3.1.3) applied to parent(v) and modified such that only the new dummy nodes and the children of v are allowed to move could be used. In order to avoid the overhead of processing a lot of fixed nodes, however, a closer examination of the node ordering algorithm as regards the positions of the new dummy nodes is necessary.

Since an edge $(v, u) \in \mathcal{F}_A$ is expanded only up to its first dummy node d_u, we can simplify the following description and assume without loss of generality that (v, u) is proper.

local and external

Definition 3.5. A new dummy node p_u for the expanded edges of an edge (v, u) or (u, v) is *local* if parent(v) = parent(u); otherwise it is *external*.

Consider an external dummy node p_u first. Since the edge (v, u) is proper, parent(v) and parent(u) must lie on the same layer. In the local hierarchy induced by parent(v)'s children, p_u has $\lambda(p_u) - \rho(p_u) = \pm 1$, depending on whether parent(u) lies to the left $(+1)$ or to the right (-1) of parent(v). The splitting method, as described in Section 3.1.3, places p_u to the left or right end of the layer, with the exact position determined by the $\lambda(p_u) - \rho(p_u)$ value; see line 3 in Algorithm 3.3.

[7] Its children are also new, but as there is at most one child per layer, ordering them is trivial.

Algorithm 3.3: `updateOrdered`$(\mathcal{D}_O, v)$

input : $\mathcal{D}_O = (\mathcal{V}_P, \mathcal{E}_P, \mathcal{F}_P, \text{clev}, \sigma)$: ordered proper assigned compound digraph,
v: leaf to be expanded
output: $\mathcal{D}_O{}' = (\mathcal{V}_P{}', \mathcal{E}_P{}', \mathcal{F}_P{}', \text{clev}', \sigma')$ ordered proper assigned compound digraph with v expanded

$\mathcal{D}_O{}' \leftarrow$ `updateProper`$(\mathcal{D}_O, v)$, $\sigma' \leftarrow \sigma$

foreach *external dummy node* p_u **do**
  insert p_u into σ' according to $\lambda(p_u) - \rho(p_u)$
end

let L be the set of all local dummy nodes
insert $L \cup \{v\}$ sorted according to the barycenter values at v's position into σ'
apply ordering algorithm of Section 3.1.3 to the local hierarchy of v

foreach *new (local or external) dummy node* p_u **do**
  // p_u represents a block of expanded edges $(u, v_1), \ldots, (u, v_k)$
  // or $(v_1, u), \ldots, (v_k, u)$ with $v_i \in \text{children}(v)$
  split p_u into dummy nodes $p_1, \ldots, p_k$
  let $v_{\pi^\uparrow(1)}, \ldots, v_{\pi^\uparrow(k)}$ be the order of $v_1, \ldots, v_k$ from bottom to top layer and from left to right within a layer
  let $v_{\pi^\downarrow(1)}, \ldots, v_{\pi^\downarrow(k)}$ be the order of $v_1, \ldots, v_k$ from top to bottom layer and from left to right within a layer

  if *p_u represents an incoming edge (u, v)* **then**
    if *p_u lies left of v in σ* **then**
      insert the split dummy nodes at the position of p_u into σ' in left to right order $p_{\pi^\uparrow(1)}, \ldots, p_{\pi^\uparrow(k)}$
    else
      insert the split dummy nodes at the position of p_u into σ' in left to right order $p_{\pi^\downarrow(1)}, \ldots, p_{\pi^\downarrow(k)}$
    end
  else
    if *p_u lies left of v in σ* **then**
      insert the split dummy nodes at the position of p_u into σ' in left to right order $p_{\pi^\downarrow(1)}, \ldots, p_{\pi^\downarrow(k)}$
    else
      insert the split dummy nodes at the position of p_u into σ' in left to right order $p_{\pi^\uparrow(1)}, \ldots, p_{\pi^\uparrow(k)}$
    end
  end

  split the remaining dummy nodes of the contracted edge (u, v) or (v, u) and propagate the order of the p_i to them.
end

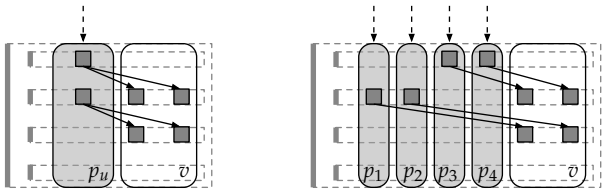

Figure 3.19: Dummy node p_u for an incoming edge (u, v) lying left of v (left). The left to right order of $p_1, \ldots, p_k$ for the expanded edges $(u, v_1), \ldots, (u, v_k)$ ($k = 4$ in this example) is derived from the order of the children v_i from bottom to top layer and from left to right within a layer (right).

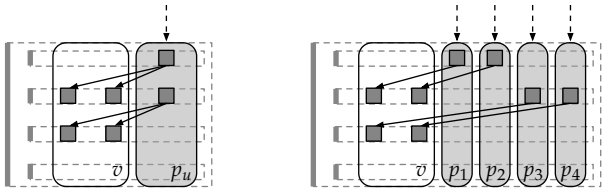

Figure 3.20: Dummy node p_u for an incoming edge (u, v) lying right of v (left). The left to right order of $p_1, \ldots, p_k$ for the expanded edges $(u, v_1), \ldots, (u, v_k)$ ($k = 4$ in this example) is derived from the order of the children v_i from top to bottom layer and from left to right within a layer (right).

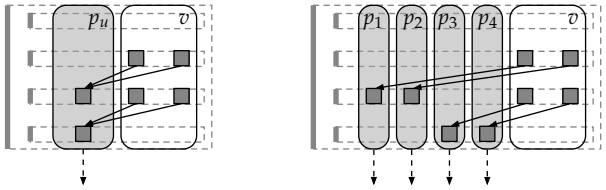

Figure 3.21: Dummy node p_u for an outgoing edge (v, u) lying left of v (left). The left to right order of $p_1, \ldots, p_k$ for the expanded edges $(v_1, u), \ldots, (v_k, u)$ ($k = 4$ in this example) is derived from the order of the children v_i from top to bottom layer and from left to right within a layer (right).

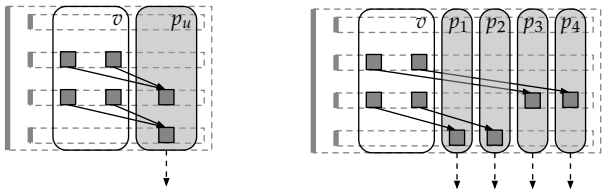

Figure 3.22: Dummy node p_u for an outgoing edge (v, u) lying right of v (left). The left to right order of $p_1, \ldots, p_k$ for the expanded edges $(v_1, u), \ldots, (v_k, u)$ ($k = 4$ in this example) is derived from the order of the children v_i from bottom to top layer and from left to right within a layer (right).

Remark. Let p_w be another external dummy node for the proper edge (v, w) such that $\lambda(p_u) - \rho(p_u) = \lambda(p_w) - \rho(p_w)$. Then p_u and p_w are indistinguishable in the splitting method; they are pinned to one end in *arbitrary* relative order. This order, however, should be the same as for the nodes u and w, which both lie on the same layer. This problem, incidentally, is immanent to the original algorithm of Sugiyama and Misue (1991); it is not specific to the proposed update scheme. However, it can be alleviated by taking the relative order of the end nodes as secondary sorting criterion in the splitting method.

After the splitting method, all external dummy nodes are fixed; it remains to do the same for the *local* ones. Let p_u and p_w now denote two local dummy nodes inserted after expanding the proper edges (v, u) and (v, w). Then u and w lie on the same layer and thus determine the relative order of p_u and p_w. Essentially, the only degree of freedom is whether to place p_u or p_w right or left of v, as it makes no sense to have some old node x between p_u and v: otherwise the edges p_u represents would cross x. In other words, the new local dummy nodes and v itself must form a contiguous block. In the local hierarchy induced by children(parent(v)), a dummy node p_u has only one outgoing edge; hence, its barycenter is identical to the position of u. Therefore, it is sufficient to sort the new local dummy nodes together with v according to their barycenters and insert them as contiguous block at v's old position in σ; see line 6 in Algorithm 3.3.

It remains to correct the mistake we made by not fully expanding the edges incident with v in the previous step; see Algorithm 3.2. Up to this point, all the expanded edges (v_i, u) of the edge (v, u) are represented by *one* sequence of dummy nodes as shown, for instance, in Figure 3.18. Each dummy node is split such that finally there is a *distinct* sequence for every expanded edge (v_i, u) as shown, for instance, in Figure 3.16. For the order σ this essentially means that a block of dummy nodes takes the position of the dummy node it replaces. Nevertheless, we need to determine the relative order of the dummy nodes for the expanded edges *within* these blocks. This order, however, depends on the positions of v's children, which can easily be determined with the node ordering algorithm of Section 3.1.3 applied to the local hierarchy of v; see line 7 in Algorithm 3.3.

Let $p_1, \ldots, p_k$ denote the dummy nodes that emanate from splitting p_u. It suffices to determine their relative order because the remaining dummy nodes in the sequence for the expanded edges $(v_1, u), \ldots, (v_k, u)$ need to be ordered accordingly to avoid unnecessary crossing. The relative order of $p_1, \ldots, p_k$ is derived from the positions of the children $v_1, \ldots, v_k$. If p_u lies right of v and if $v_{\pi(1)}, \ldots, v_{\pi(k)}$ is the order of v's children from *bottom to top*

and within the same layer from *left to right*, then $p_{\pi(1)}, \ldots, p_{\pi(k)}$ is the order of the dummy nodes from left to right; see Figure 3.22. The case that p_u lies left of v as well as the two cases for an incoming edge (u, v) are symmetric; see Figures 3.19, 3.20, and 3.21. Finally, the remaining old dummy nodes of the edge (v, u) are split accordingly, i. e., they inherit the order of the p_i; see Algorithm 3.3.[8]

Locality Property 3 Let k denote the number of elements added to $\mathcal{D}_P$ by expanding v including the dummy nodes that are generated by splitting the old dummy nodes of contracted edges used as representatives. The complexity of updating the local order σ is roughly $\mathcal{O}(f_{III}(k))$, compared to $\mathcal{O}(f_{III}(n+k))$ for re-applying step III to $\mathcal{D}_P{}'$, where n is the size of $\mathcal{D}_P$. The relative order of all old nodes $\mathcal{V}$ is preserved and expanded edges take the same course as the corresponding contracted edge, i. e., properties (MM 2) and (MM 3) for preserving the mental map are satisfied.

3.2.4 Step IV: Metric Layout

Although the results of all other steps could be updated locally, this is not entirely possible for the metric layout. The reason is that expanding v changes the width and height of v, which leads to adjustments of the local coordinates for v's siblings; this changes the width and height of parent(v), and so on up to the root. The children of v, however, are not the only nodes that have been added due to expanding; there are also new dummy nodes for the expanded edges, which may be scattered across the whole graph. On the other hand, if the subtree rooted at some node u contains neither children of v nor new dummy nodes, the local coordinates in the entire subtree rooted at u are not affected and thus can be reused; see Figure 3.23.

As described in Section 3.1.4, the metric layout consists of two steps: computing local coordinates followed by a depth-first traversal of the inclusion tree to sum them up to absolute coordinates. Therefore, the update of the metric layout also has two steps: first, the local coordinates of all new nodes and their ancestors are adjusted; then, the second step is used unalteredly. For the updates of the local coordinates basically the same recursive proce-

[8] Note that it can happen that the corresponding contracted edge (v, u) also is part of the expanded view, because there exists an original adjacency edge $(v, u') \in F$ such that $u' \in \text{desc}(u)$. In this case, which is not mentioned explicitly in Algorithm 3.3, a separate sequence of dummy nodes for the edge (v, u) needs to be split off. The best position of these dummy nodes is either before or after the block of dummy nodes for the expanded edges. For an outgoing edge (v, u) with p_u lying right of v, for instance, these dummy nodes obviously have to be placed before the blocks.

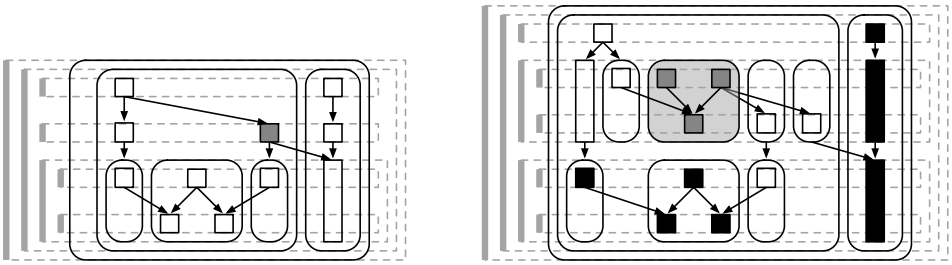

Figure 3.23: The local coordinates for the black nodes can be reused (right) after expanding the lighter shaded node (left).

dure as in Section 3.1.4 is used; the only difference is that recursive calls are made only for subtrees that need to be adjusted, i. e., those containing either children of v or new dummy nodes. This can easily be done by first traversing the tree bottom-up from every new node and marking all reached nodes and then restricting the recursive calls to the marked nodes.

Locality Property 4 Let n denote the size of $\mathcal{D}_O{}'$. In the worst case, the complexity of updating is the same as for re-applying this step to $\mathcal{D}_O{}'$ as a whole, i. e., $\mathcal{O}(n)$. The final depth-first traversal to sum up the absolute coordinates is completely applied in any case. The local coordinates, however, are adjusted only for ancestors of new nodes.

3.3 Contraction

Again, let $\mathcal{D} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ be a view of a compound digraph $D = (V, E, F)$. Contracting a node $v \in \mathcal{V}$ that has been expanded with the above update scheme is straightforward: after contraction, i. e., in the view $\mathcal{D}' = (\mathcal{V}', \mathcal{E}', \mathcal{F}')$, all nodes are old because $\mathcal{V}' = \mathcal{V} \setminus \text{children}(v)$. Hence, the layer assignment clev and the node order σ just need to be restricted to $\mathcal{V}'$, which ensures that properties (MM 1) and (MM 2) for preserving the mental map are satisfied. The position of the dummy nodes for a derived edge incident to v is given by the position of the blocks of the corresponding expanded edges. Thus, the contracted edge takes the same course as the expanded edges as required by property (MM 3). Since the width and height of v has changed, the metric layout has to be updated as described in Section 3.2.4.

If we assume that v has been expanded in a view $\mathcal{D}''$ *just before* this `contract` operation, it is obvious that $\mathcal{D}'$ and $\mathcal{D}''$ contain the same nodes. As both expanding and contracting satisfy properties (MM 1) and (MM 2),

these nodes stay on their respective layers in unchanged relative order during both operations. Also, the reusing of dummy node positions and the grouping of expanded edges ensures property (MM 3) during both operations. In other words, before the metric layout all nodes of the contracted view $\mathcal{D}'$—even dummy nodes—reside on the same layer in the same relative order as in $\mathcal{D}''$. Therefore, the input for the metric layout before and after both operations is identical, and thus its output, i. e., the final drawing, will be identical. This means that expanding and contracting are also visually inverse as required by property (MM 0).

Locality Property 5 Let k denote the number of elements that are removed from $\mathcal{D}_\mathrm{P}$ during the update to $\mathcal{D}_\mathrm{P}{}'$ as a result of the contraction of node v. The number of elements removed from the other (intermediate) compound digraphs is at most k. Therefore, updating the drawing after contracting a node v that has been expanded as described in Section 3.2 takes $\mathcal{O}(k)$ for steps I to III. Step IV is the same as after expanding; see Locality Property 4. The user's mental map is preserved by satisfying (MM 1)–(MM 3). Furthermore, expanding and contracting are visually inverse as defined by (MM 0).

Contraction is more complicated for nodes v that have *not* been expanded with our update scheme for two reasons. First, the dummy nodes for edges $(v_1, u), \ldots, (v_k, u)$ incident with children $v_i \in \mathrm{children}(v)$ $(1 \leq i \leq k)$ need not form contiguous blocks on their layers, but after contraction all these edges are represented by one edge (v, u). This is a minor problem that can be solved by declaring one of the edges $(v_1, u), \ldots, (v_k, u)$ as representative and thus reusing its dummy nodes for the contracted edge (v, u). Expanding and contracting, however, are no longer visually inverse then.

The main reason why contracting a node that has not been expanded before is more complicated is the following. Consider a child v' of v with an edge (u, v') such that v and $\mathrm{parent}(u)$ lie on the same layer, e. g., as in Figure 3.14. As pointed out in Section 3.2.1, our method of updating the layer assignment prevents this situation, yet it is possible in the layout of the initial view. Note that, as the deepest derived edge always is of type $\prec$, the layer assignment assures that for every edge (x, y) the compound layers $\mathrm{clev}(x)$ and $\mathrm{clev}(y)$ differ—after a common start sequence—by at least one position (cf. Section 3.1.1). The derived edge (u, v), representing (u, v') after contracting v, would violate this invariant, because $\mathrm{clev}(v)$ would be a subsequence of $\mathrm{clev}(u)$. Unfortunately, the normalization, described in Section 3.1.2, relies on this invariant. This problem also can be observed in the derived compound digraph: before contracting v the deepest edge, the one of type $\prec$, was adjacent to v' and is removed; therefore, the type of the

derived edge between v and the ancestor of u at the same depth as v would have to be adjusted from $\preceq$ to $\prec$, which in general leads to substantial changes in the layer assignment and thus destroys the user's mental map; compare Figures 3.13 and 3.14.

The easiest way to deal with this problem is to allow contraction only for nodes that have been expanded before, i. e., no node of the initial view can be contracted. Another way is to modify the algorithm of Sugiyama and Misue (1991) used for the initial view as regards the way the derived compound digraph is constructed: instead of generating derived edges for an edge (u, v) between all those ancestors of u and v that have equal depth, only the highest derived edge, i. e., the one between the children of the nearest common ancestor of u and v, is created and assigned the type $\prec$. The consequence is that nodes with descendants that are connected never lie on the same layer and thus the layout is less compact.

3.4 Experimental Results

The static layout algorithm of Sugiyama and Misue (1991) with the metric layout of Brandes and Köpf (2001) and its update scheme described above have been implemented by Pröpster (2005) as part of the interactive compound (di-)graph editing framework described in Chapter 4. The following experimental results have been created with a stand-alone, i. e., non-interactive, variant of this implementation.

Since the alternative to the proposed update scheme is redrawing the entire graph with the static algorithm, its efficiency and quality are analyzed competitively, e. g., the computation time of the update is measured in relation to the time needed for redrawing the entire graph. As regards the quality of the drawing, the area and the number of crossings have been chosen as two indicators. On the one hand, the area is an important factor in the context of exploring large compound (di-)graphs, where the screen real estate often is a bottleneck. On the other hand, experimental results (Purchase et al., 1997; Purchase, 1998) suggest that crossings have great influence on the readability of drawings.

As described in (Pröpster, 2005), a randomly generated compound digraph first is completely contracted, i. e., the view contains only the root. Then, all nodes are expanded successively (in breadth-first order of the inclusion tree) and the drawing is adjusted with our update scheme after each `expand` operation. Finally, the thus generated drawing is compared to the result of applying the static algorithm once to the fully expanded compound

digraph.

The following parameters were varied during the random generation of the compound digraphs: the number of nodes n, the average number of children of internal nodes γ, and the density δ, which is defined as the number of adjacency edges divided by the maximum number of pairs of unrelated nodes.

The inclusion tree is generated breadth-first starting with the root, which is put into a queue. As long as the total number of nodes does not exceed n, children for the next node in the queue are inserted. Their number is determined between 1 and 2γ in a random experiment with a binomial distribution having γ as mean. After the inclusion tree has been generated, the adjacency edges are inserted randomly with probability δ between the pairs of unrelated nodes.

In our experiments, 1,500 compound digraphs were thus created; ten for every combination of the following values for the three parameters: $n \in \{20, 35, 50, 75, 100\}$, $\gamma \in \{2, 4, 6, 8, 10, 15\}$, and $\delta \in \{0.01, 0.05, 0.1, 0.2, 0.3\}$.

As far as the area of the drawing is concerned, it can be said that our update scheme performs nearly as well as the static algorithm. On the average, the iterative expanding as described above needs about 2 % more area than the static algorithm applied directly to the fully expanded graph. This small increase in area is almost independent of the number of nodes, as shown in Figure 3.24. Also, neither the density δ nor the average number of children γ has great influence on it; see Figures 3.25 and 3.26, respectively.

Remark. All diagrams in this section are of the same type. The grey bar is a 70 %-quantile centered at the median, which is depicted as a small horizontal line dividing the grey bar. This means that the grey bar represents 70 % of all data, 35 % on each side of the median. The vertical line is the 90 %-quantile centered at the median.

Although the values are scattered over a wide range (from -25 % to $+50$ %), it is astonishing that the average increase in area is not very significant. One would actually expect that the update scheme needs more area than the static algorithm because updating the layer assignment often needs additional layers, as (MM 1) demands to keep all old nodes on their former layer during the expanding; see Figures 3.13 and 3.14 in Section 3.2.1. This effect indeed is existent, though barely noticeable. The more adjacency edges the graph has, the more likely it is that two arbitrary nodes (or their predecessors) are connected by a type $\prec$ edge in the derived compound digraph and thus are placed on different layers anyway. This means that the denser the graph gets, the smaller is the increase in area, which can

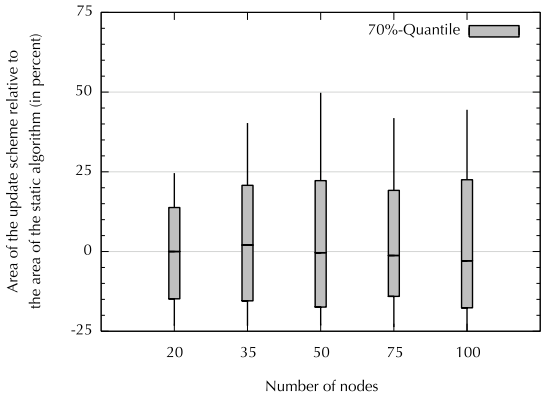

Figure 3.24: Area needed by our update scheme relative to the area of the static algorithm as affected by the number of nodes n.

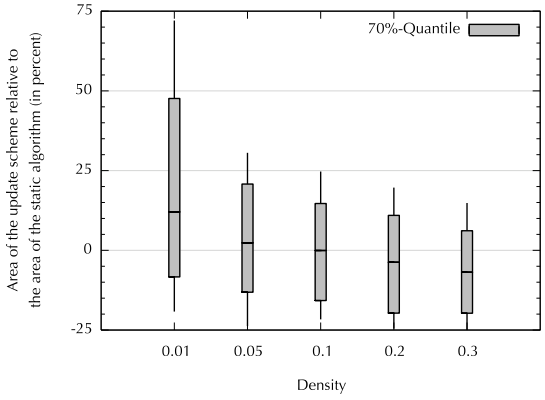

Figure 3.25: Area needed by our update scheme relative to the area of the static algorithm as affected by the density δ.

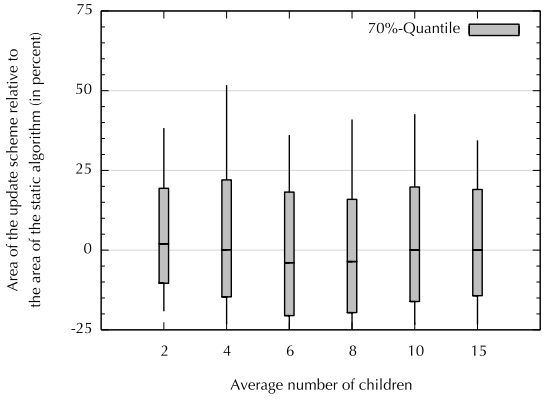

Figure 3.26: Area needed by our update scheme relative to the area of the static algorithm as affected by the average number of children γ.

be observed in Figure 3.25. Furthermore, if the hierarchy tree is relatively deep, i. e., if γ is small, many `expand` operations are needed until the graph is fully expanded and thus the increase in area accumulates as shown in Figure 3.26.

As regards the relative number of crossings[9], i. e., the number of crossings in the drawing after fully expanding the graph with our update scheme divided by the number of crossings in the static layout of the entire graph, our update scheme produces approximately 12 % fewer crossings on the average. Moreover, neither the number of nodes n, nor the density δ, nor the average number of children γ has significant influence on this decrease; see Figures 3.27, 3.28, and 3.29, respectively. Also, the scatter is relatively small in all three diagrams: for nearly all parameters, the 70 %-quantile centered at the median entirely lies below 0 %, which means that for 85 % of the graphs[10] our update scheme produces less crossings. One reason for this improvement appears to be the way the update scheme expands edges. Recall that the overall course of the expanded edges is inherited from the corresponding contracted edge and that the relative order among the expanded edges is derived such that they cannot cross; see Algorithm 3.3 and Figures 3.19, 3.20, 3.21, and 3.22 in Section 3.2.3. In the static layout, all expanded edges are treated alike whether they belong to the same contracted edge or not.[11] This can lead to unnecessary crossing that our update scheme effectively avoids.

The performance of our update scheme is measured as the time updating the view compared to the time for re-applying the static algorithm. On the average (both over all graphs and all `expand` operations), our update scheme is about twice as fast. The larger a graph gets, the more local are the updates and thus the larger are the reusable parts of the intermediate results. Therefore, our update scheme performs better as the number of nodes increases; see Figure 3.30. As regards the density of the graph, an opposite trend in recognizable in Figure 3.31: the performance gain decreases with increasing density. The reason appears to be that more adjacency edges imply more expanded edges and thus a larger fraction of the graph is affected, which counteracts the locality of our updates. Similarly, more children result in less locality and thus slightly less performance gain; see Figure 3.32.

[9] This includes both ordinary crossings among edges and crossings between an edge and a node that is no predecessor of either end of the edge.

[10] Note that the upper end of the vertical bar indicates the value below which 85 % of our data lie, as the 70 % quantile is centered at the median.

[11] In fact, the term "contracted edge" does not make much sense in the static layout. This information is not available and thus cannot be exploited.

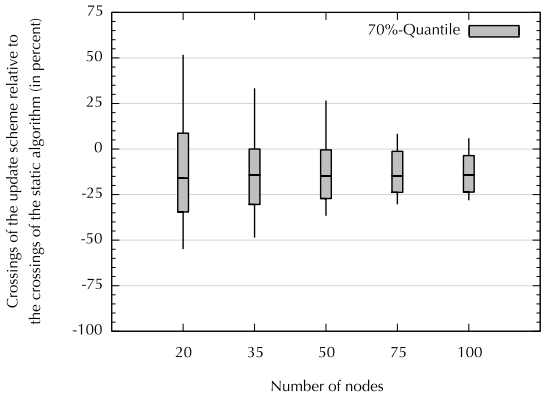

Figure 3.27: Number of crossings produced by our update scheme relative to the number of crossings in the static algorithm as affected by the number of nodes n.

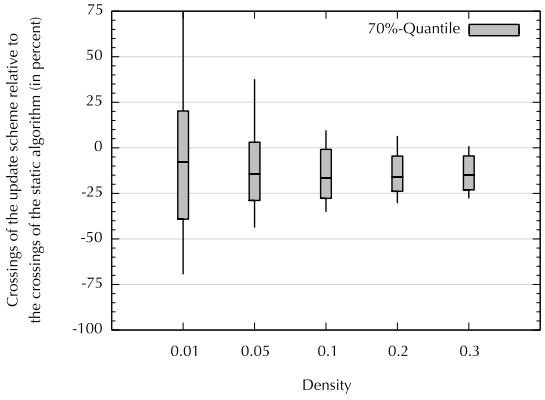

Figure 3.28: Number of crossings produced by our update scheme relative to the number of crossings in the static algorithm as affected by the density δ.

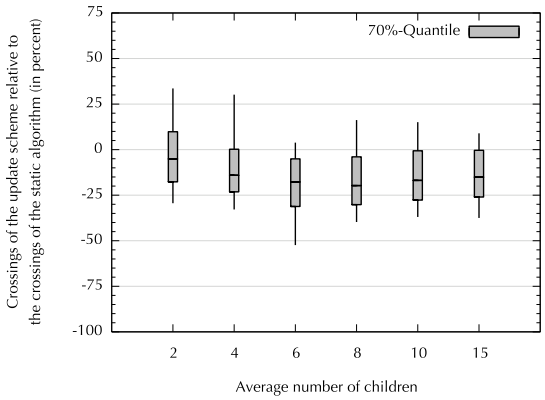

Figure 3.29: Number of crossings produced by our update scheme relative to the number of crossings in the static algorithm as affected by the average number of children γ.

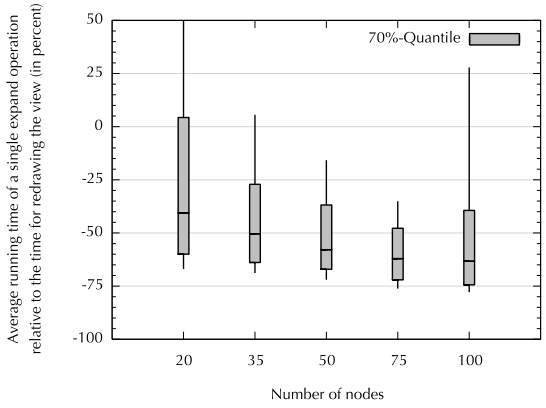

Figure 3.30: Average running time of a single `expand` operation relative to the time for re-applying the static algorithm to the respective view as affected by the number of nodes n.

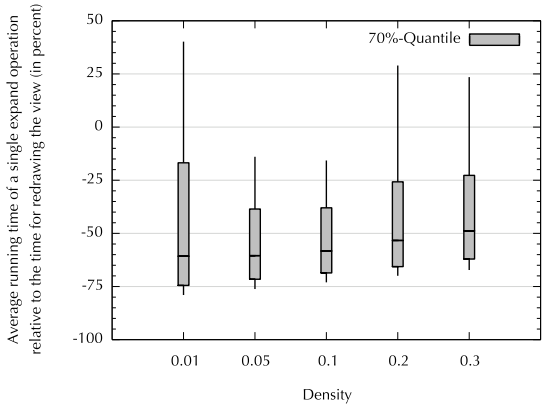

Figure 3.31: Average running time of a single `expand` operation relative to the time for re-applying the static algorithm to the respective view as affected by the density δ.

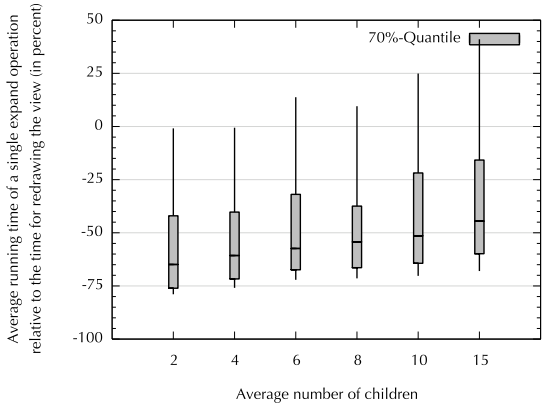

Figure 3.32: Average running time of a single `expand` operation relative to the time for re-applying the static algorithm to the respective view as affected by the average number of children γ.

There are some outliers in all three diagrams, but the majority of the data is scattered not very widely; almost all 70 %-quantiles entirely lie below −25 %, which means that for 85 % of the graphs our update scheme decreases the running time by at least 25 %. Moreover, as the median always lies below 50 %, the performance gain is at least a factor of two for at least half of the graphs.

Note that even for small values of the density δ the randomly generated graph get rather dense, as the number of unrelated nodes grows quadratically in the number of nodes.[12] Furthermore, these graphs are quite unnatural: every pair of unrelated nodes may be connected by an adjacency edge with the same probability. This means that an edge between two siblings is just as likely as between two nodes for which the nearest common ancestor is the root.

In order to measure the degree of relationship of two nodes u and v in the inclusion tree $T = (V, E)$ of a compound digraph $D = (V, E, F)$, Pröpster (2005) defines the *complexity* of u and v as the length of the unique path connecting them in T.[13] The complexity of two siblings, for instance, is 2, which is the smallest possible value. The higher the complexity of two nodes, the lower is their degree of relationship.

Our modified algorithm for randomly generating compound digraphs builds on the assumption that closely related nodes are more likely to be connected by an edge.[14] The hierarchy tree is built as before with the number of nodes n and the average number of children γ as parameters. But now also the average complexity ρ and not only the chosen density influences the creation of adjacency edges. In a preprocessing step, the maximum number of pairs of unrelated nodes is calculated for each possible complexity. Then the total number of needed edges, which is essentially determined by the density δ and the number of nodes n, is distributed over the range of possible complexities, using a binomial distribution with mean ρ. For each possible complexity i, this yields the number of needed edges and dividing it by the maximum number of possible edges with complexity i gives the probability p_i. The algorithm for creating adjacency edges then checks each

[12] In fact, $n(n - \lceil \log_\gamma n \rceil) = n^2 - n\lceil \log_\gamma n \rceil$ is a lower bound for the number of unrelated nodes if the hierarchy is created as balanced as described above because a node cannot have an edge to any of its predecessors, which are at most $\lceil \log_\gamma n \rceil$ many.

[13] More precisely, the unique path in the underlying *undirected* counterpart of T, which is created by ignoring the direction of the edges.

[14] Basically the same assumption, namely that dense subgraphs form a cluster, is made by many clustering techniques for partitioning ordinary graphs into a hierarchy of meaningful subgraphs; see Alpert and Kahng (1995) for a detailed overview.

pair of unrelated nodes u and v and inserts an edge with probability p_i, where i is the complexity of u and v.

As the above results already cover dense graphs, this refined method was used to generate sparse compound digraphs, i. e., the number of adjacency edges equals the number of nodes. In order to get an impression of the quality and performance of our update scheme on such supposedly more natural graphs, we chose two combinations of the average complexity and the average number of children, $(\rho, \gamma) \in \{(2.2, 5), (2.3, 15)\}$, and varied only the number of nodes n. In this second test, 1,400 compound digraphs, 100 for each value $n \in \{50, 100, 200, 400, 600, 800, 1000\}$, were generated. Figure 3.33, for instance, shows such a randomly generated compound digraph with 100 nodes and edges.

The test method was the same as in our first test: the graph is contracted completely; then all nodes are expanded successively and the drawing is adjusted with our update scheme after every `expand` operation. As regards the area and the number of crossings, the final drawing is compared to a drawing of the static algorithm applied to the fully expanded graph. The performance of the updates again is measured as the time for updating compared to the time for redrawing the respective view with the static algorithm.

Even for these relatively sparse graphs, our update scheme uses only approximately 3 % more area on the average. As for the denser graphs in our first test, the additional area is almost independent of the number of nodes n; see Figure 3.34. Again, the data is scattered widely (from -25 % to $+30$ %), but symmetrically around the median, which is close to 0. This means that roughly half of the graphs need more area, while the others need less.

As regards the number of crossings, our update scheme produces on the average approximately 6 % less crossings than the static algorithm; see Figure 3.35. This is not as remarkable as the 12 % before, but plausible, as there is less room for improvements in sparse graphs. For small graphs, the data is scattered widely and unlike the diagram in Figure 3.27 it happens relatively often that our update scheme leads to more crossings. Nevertheless, as the number of nodes increases, the scatter reduces remarkably and the decrease in crossings approaches its former value.

The performance gain is impressive on these sparse graphs. The time for a single `expand` operation on the average is approximately 20 % of the time for redrawing the respective view. The larger the graph, the greater is the performance gain, as larger parts of the graph can be reused; see Figure 3.36. This effect is more noticeable than for the denser graphs in our first test; see

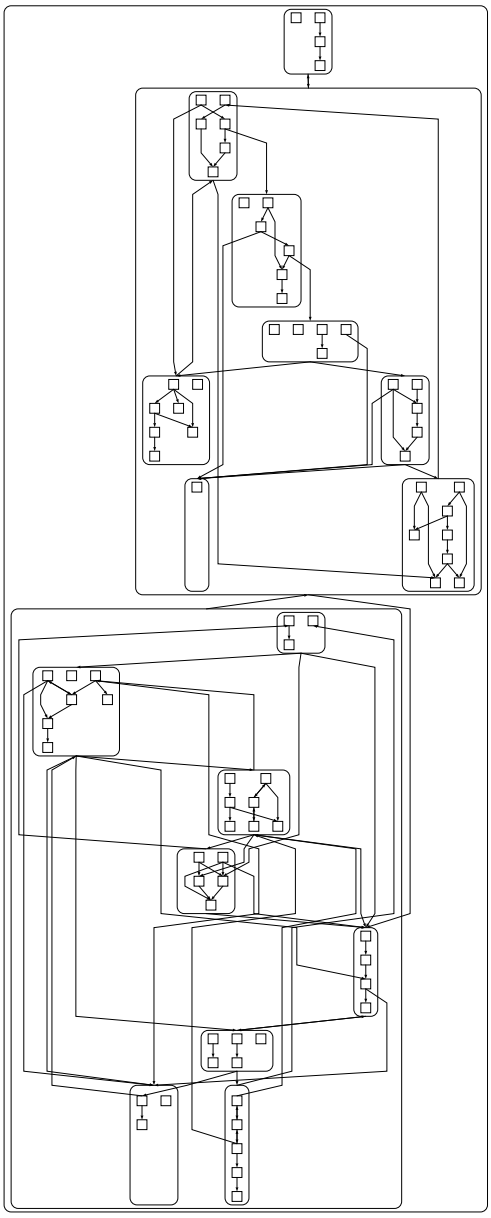

Figure 3.33: Example of a randomly generated compound digraph with 100 nodes and about as many edges; average number of children is 5 and the average complexity of the edges is 2.2.

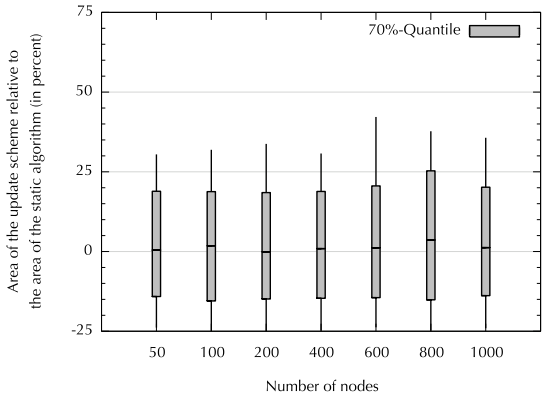

Figure 3.34: Area needed by our update scheme relative to the area of the static algorithm as affected by the number of nodes n.

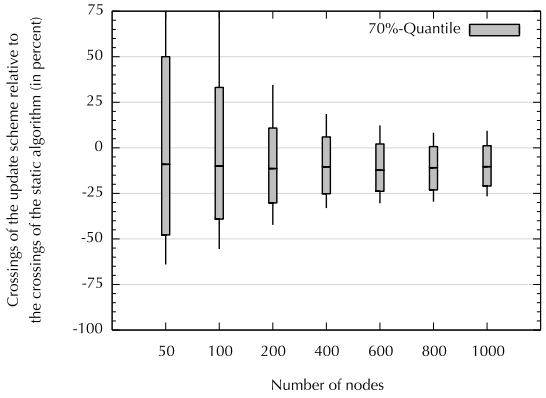

Figure 3.35: Number of crossings produced by our update scheme relative to the number of crossings in the static algorithm as affected by the number of nodes n.

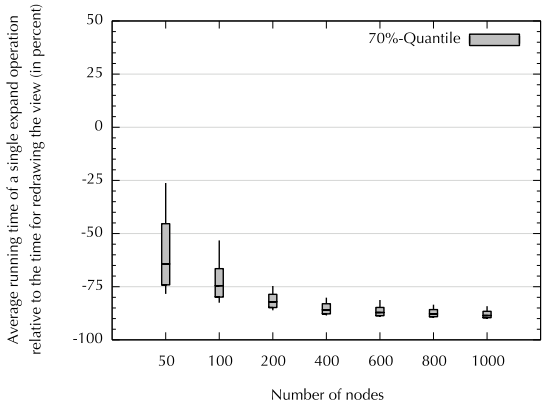

Figure 3.36: Average running time of a single `expand` operation relative to the time for re-applying the static algorithm to the respective view as affected by the average number of nodes n.

Figure 3.30. Moreover, as the number of nodes increases, the scatter of the values decreases significantly: for over 90 % of the graphs with more than 200 nodes, the average `expand` operation takes significantly less than 20 % of the time for redrawing the respective view.

3.5 Summary

The proposed update scheme for the algorithm of Sugiyama and Misue (1991) visualizes the expanding and contracting of nodes in views of compound digraphs. With the particular emphasis laid on the locality of the updates, it is more efficient than redrawing the entire view. For expanding a node, the complexity of updating the drawing is determined largely by applying each of the steps I to III to the *modified part* of the compound digraph, followed by step IV for adjusting the coordinates. As our experimental results in Section 3.4 show, the average time for updating the drawing is around 50 % of the time for redrawing for dense graphs and below 20 % for sparse graphs. Moreover, the performance gain is not at the expense of quality, at least not as regards the area of the drawing, which increases only insignificantly, and the number of crossings, which is reduced.

The user's mental map is preserved well: old nodes stay on their layers in the same relative order and expanded edges take the same course as the corresponding contracted edge, i. e., properties (MM 1)–(MM 3) are fulfilled. Moreover, expanding and contracting are visually inverse as defined by property (MM 0) in Section 1.3.4.

With the help of the novel idea of reusing the positions of dummy nodes of a contracted edge for the corresponding expanded edges (cf. Section 3.2.3), a modified version of (Sugiyama and Misue, 1991) supporting constraints could possibly yield similar results, yet it would have to be applied to the whole compound digraph and thus would be not as efficient as our update scheme.

4

Architecture

So far we have discussed the two main aspects of visual navigation in compound (di-)graphs separately: in Chapter 2, an efficient data structure for expanding and contracting nodes in views has been introduced, whereas in Chapter 3 a scheme for updating a layered drawing of a view after these operations has been proposed. Now we shall combine these aspects into a software architecture for a compound (di-)graph viewer and editor.

Particular emphasis must be laid on the extensibility with respect to both the underlying data structure and the drawing style because both are likely to be researched further and thus will experience variations. Another aspect, which is not covered in detail in this work, but fits nicely into this framework, is *animation*. This means to smooth the transition from one drawing to another, e. g., after expanding or contracting a node, by animating the movement of the (common) nodes. There exist various animation styles, ranging from simple linear interpolation to modeling the transition as a composition of translation, rotation, and scaling (Friedrich and Eades, 2002). Therefore, the architecture is kept extensible in this regard as well.

Employing the *Observer* design pattern (Gamma et al., 1995, pp. 293–303) allows for arbitrarily many, dynamically attachable views of the same compound (di-)graph. This is convenient in an editor because the user can have multiple windows showing different views of the same compound (di-)graph. *Extensibility* and *flexibility* is accomplished with the *Strategy* design pattern (Gamma et al., 1995, pp. 315–323), which makes it possible to switch the drawing style and the animation style of each view on the fly. Also, this pattern accounts for the extensibility as regards the various data structures for the graph view maintenance problem. *Reusability* is given

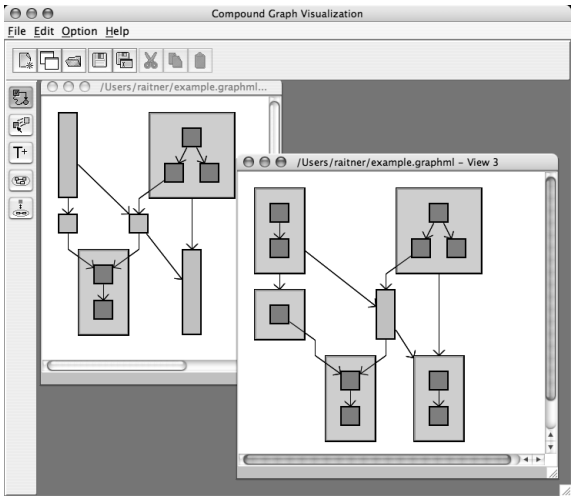

Figure 4.1: Screenshot of the proof-of-concept implementation by Pfeiffer (2005) and Pröpster (2005).

through coupling the main parts only loosely via the *Model-View-Controller (MVC)* design pattern (Buschmann et al., 1996). The model, i. e., the compound (di-)graph and its associated views, for instance, do not have any knowledge of the GUI used to manipulate them and thus could be reused unalteredly elsewhere.

The proposed architecture already has demonstrated its feasibility and solidity in a proof-of-concept implementation by Pfeiffer (2005) and Pröpster (2005), a screenshot of which is shown in Figure 4.1.

4.1 Design Goals

Before discussing the software architecture in detail, we have to explain its underlying objectives:

(DG 1) Arbitrarily many, dynamically attachable views
A view conceptually represents one perspective of looking at a compound (di-)graph. Especially for a large graph it is convenient to examine it from different perspectives simultaneously. Therefore, the number of views of a single compound (di-)graph must not be restrained. As a consequence, views must be attachable and removable dynamically.

(DG 2) Interchangeable data structures
The implementation of the compound (di-)graph is where the efficient data structures for the graph view maintenance problem come into play. Except

for `expand` and `contract`, which are performed on the view, the compound (di-)graph must provide all methods of the fully dynamic graph and hierarchy variant described in Chapter 2. There already exist various data structures for the graph view maintenance problem and, what is more important, new ones probably will be researched in the future. Therefore, the architecture must be flexible and extensible in this respect.

(DG 3) Interchangeable presentation styles
Many drawing styles, including the one presented in Chapter 3, are based on the paradigm that the inclusion hierarchy is depicted through the geometric inclusion of the corresponding objects in two dimensions. There are, however, other presentation styles that follow other paradigms, e. g., drawing the inclusion hierarchy as tree with or without the adjacency edges or three-dimensional drawings. Although these alternative styles are beyond the scope of this work, it should be possible to integrate them easily into the proposed architecture.

(DG 4) Dynamically interchangeable drawing styles
For each presentation style, many drawing style are conceivable. The layered drawing style presented in Chapter 3, for instance, is one among many possible for the chosen two-dimensional presentation style where the inclusion is depicted through geometric inclusion. Also, it is likely that alternative drawing styles will be researched in the future. The user must be allowed to choose a drawing style for each view and to switch it on the fly. Again, the architecture needs to be flexible and extensible in this respect.

(DG 5) Dynamically interchangeable animation styles
Animating the changes between two drawings, e. g., before and after expanding or contracting, helps the user in better keeping track of the change. Many different animation styles are conceivable; the user must be able to choose one for each view and to switch it dynamically. Note that the animation styles are independent of the drawing styles, i. e., any animation style can be combined with any drawing style.

(DG 6) Reusability
An interactive editor and viewer with drawing styles and animation is just one application for the purely combinatorial data structures modeling compound (di-)graphs and their views. Other applications are easily conceivable, e. g., web-based ones using a web-browser as front-end like BioPath (Brandenburg et al., 2003). Therefore, the purely combinatorial part of our architecture may not be coupled too tightly with the remainder in order to maximize reusability.

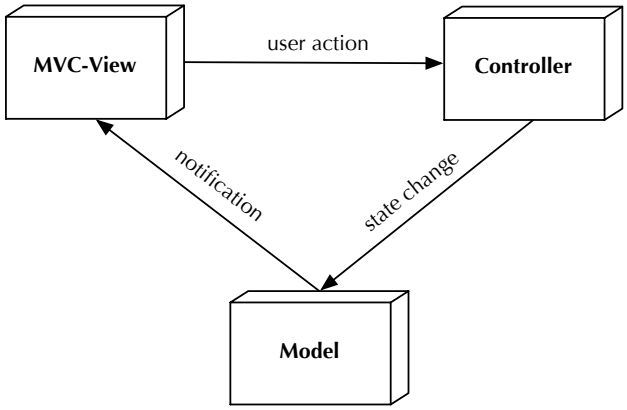

Figure 4.2: Schematic overview of the Model-View-Controller paradigm

4.2 High-Level Architecture

Since our goal is an *interactive* editor and viewer for compound (di-)graphs, the Model-View-Controller (MVC) paradigm is a good and approved choice as high-level architecture. The MVC pattern divides the application into three components: the *model* encapsulates the abstract data and the core functionality; *views* display this data to the user; the *controller* handles the user input. In order to avoid the ambiguous use of the term "view" for the ones of Definition 1.8 on the one hand and for the ones of the MVC pattern on the other, we call the view of the MVC pattern in the following *MVC-view*.

There may be arbitrarily many MVC-views showing the same data in different ways; therefore, a *change-propagation* mechanism is provided to ensure the consistency of the data and its presentation. Figure 4.2 schematically shows the interplay of the three components: the user interacts with the model through its representation in a MVC-view; the controller receives any user actions and changes the state of the model accordingly; the model then notifies all registered MVC-views via the change-propagation mechanism. See Buschmann et al. (1996, pp. 125–143) for a more detailed description of the MVC paradigm.

Figure 4.3 shows the complete class diagram without methods; detailed diagrams containing all important methods are provided for model, MVC-view, and controller separately; see Figures 4.4, 4.5, and 4.6, respectively. The model part respresents the purely combinatorial layer, e. g., compound (di-)graphs with various implementations of graph view maintenance data structures and views. A change-propagation mechanism, realized in the

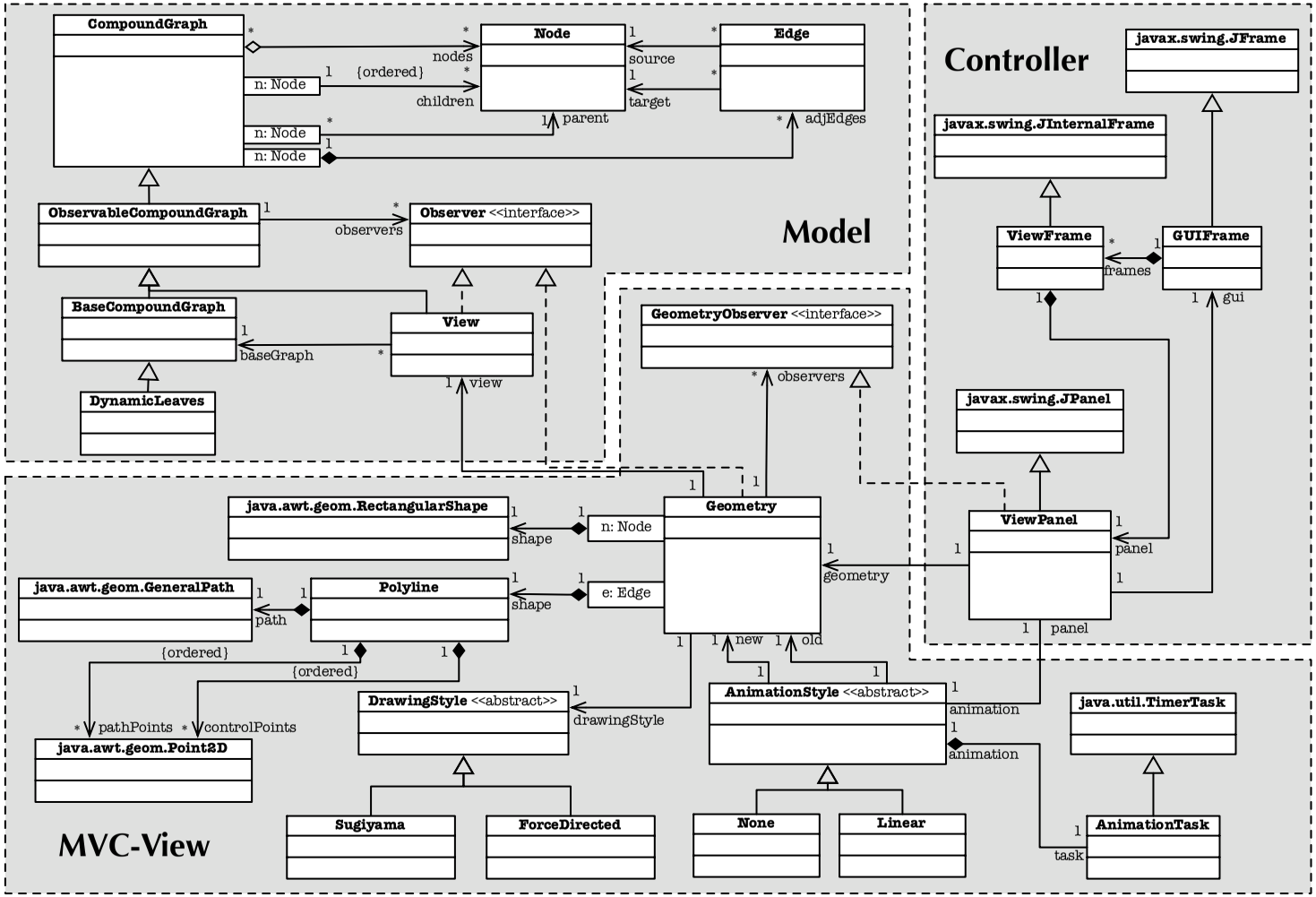

Figure 4.3: Complete class diagram

form of a *Observer* pattern, establishes the dynamic one-to-many relation between a compound (di-)graph and its views. The same mechanism is used to notify the MVC-views; therefore, the views not only are receiving notifications about changes of the underlying compound (di-)graph, they also forward them to their own observers. Note that the model can easily be reused because it does not know anything about the MVC-views and controllers; all updates are propagated through the observer interface.

The MVC-view part of the pattern acts as an observer of a view and adds all geometric information associated with the nodes and edges of a view, e. g., coordinates and dimensions for nodes and positions of bends for edges. Note that the application needs to react upon user input, which usually involves the coordinate where the user clicked; hence, this part also provides methods for finding the node or edge located at a given coordinate. Furthermore, the MVC-view part contains the drawing styles and the animation styles for modifying the geometric information, i. e., for drawing the view and animating the transitions. Acting as an observer, the MVC-view is coupled only loosely with its associated view. Also, there can be arbitrarily many MVC-views for one view, which makes it possible to display the same view in different presentation styles, e. g., in two- or in three-dimensional space. Any changes of a view are forwarded to all attached MVC-views, which are then responsible to reflect the modification appropriately, e. g., to assign coordinates to a new node or to update the drawing after expanding a node. They do this with the help of the current drawing style, e. g., the one described in Chapter 3, and the current animation style.

The controller's task is to transform user input, e. g., clicking a mouse button or pressing a key, into appropriate actions in the model, e. g., expanding a node or inserting a new edge. Not all input triggers changes in the model; sometimes only the MVC-view is affected, e. g., changing the drawing style or redrawing the view entirely. The classes of the controller part depend on the framework used for the graphical user interface, which in our case will be Java Swing. The most central component in the controller, therefore, is a `JPanel` that actually draws the nodes and edges of the view according to the positions specified in the MVC-view.

4.3 Low-Level Architecture

After this high-level description of the architecture, we take a closer look at the most important components in each of the three parts of the MVC pattern. Special emphasis will be laid on how the design goals (DG 1)–

(DG 5) described in Section 4.1 are achieved. Design goal (DG 6), reusability, is already achieved through the MVC pattern, which assures that the model is coupled only loosely with the remainder.

4.3.1 Model

The model essentially consists of classes representing compound (di-)graphs and their views. A view is coupled only loosely with its associated compound (di-)graph: it acts as observer of the compound (di-)graph. This ensures design goal (DG 1), i. e., arbitrarily many, dynamically attachable views per compound (di-)graph. As shown in Figure 4.4, the model comprises the following elements.

`Node`
A node does not store references to its incident edges like, for instance, the nodes in (GTL). This is important because nodes are shared between the graph and its views and their incident edges vary in these compound (di-)graphs. Therefore, the compound (di-)graph is responsible for providing the mapping between nodes and incident edges.

`Edge`
An edge consists of references to its source and target node and methods for accessing them. Deviating deliberately from the definition of a compound (di-)graph, `Edge` is used only for adjacency edges; tree edges are not stored explicitly, but rather as mapping between parent and children and vice versa.

`CompoundGraph`
This is the base class of all compound (di-)graphs. It stores the set of nodes and the mapping between nodes and incident edges. The inclusion hierarchy is modeled as mapping between parent and children and vice versa. The reason for this decision is that tree edges are never used explicitly; they are just traversed to find the parent or the children of a node. Apart from the methods for accessing the nodes, the incident edges of a node, and its parent and children, this base class also provides basic implementations of all methods modifying the compound (di-)graph: adding and removing leaves and adjacency edges as well as splitting and merging of clusters. Since no additional data structure for the graph view maintenance problem is used for this class, these methods focus on the absolutely necessary like, for instance, inserting a new leaf into the mapping between parent and children. More specialized implementations of a compound (di-)graph also have to do this plus some extra work to update their additional data structures.

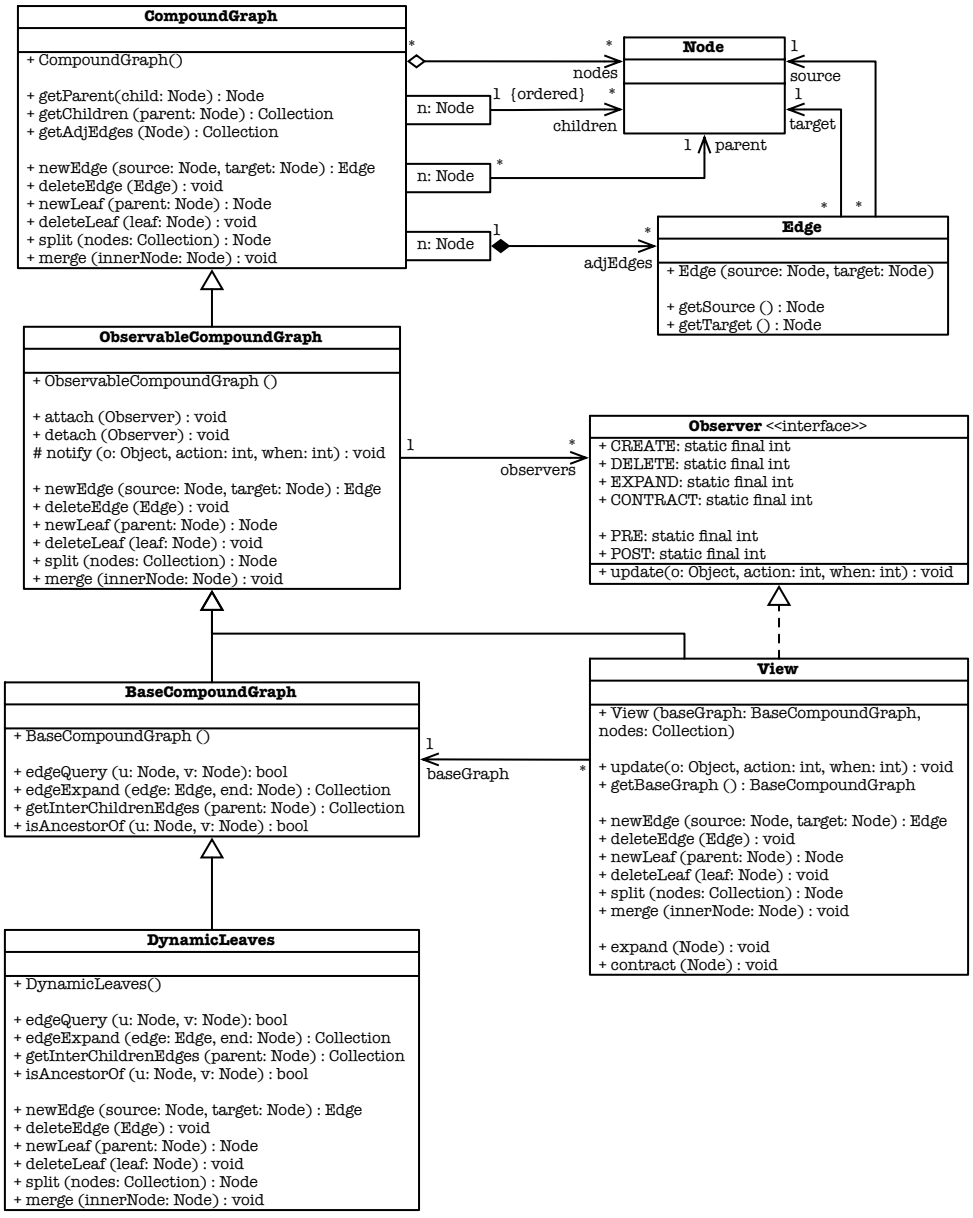

Figure 4.4: Class diagram of the model

ObservableCompoundGraph
Derived from CompoundGraph, this class adds methods to attach, detach, and notify observers. It maintains a set of all currently attached observers and overwrites all methods modifying the graph such that all observers are notified. Every observer is notified twice: before the modification and afterwards. Although in most cases an observer will need to handle only one of these notifications, both are necessary, e. g., for updating the geometric information associated with nodes and edges when a node is expanded; see the description of the class View for details.

The method notify is responsible for forwarding a message to all attached observers. It accepts three parameters: first, the object (Node or Edge) that is affected; second, a message encoding the type of modification defined as symbolic constants in Observer; third, the relative time encoded as a symbolic constant PRE or POST. If, for instance, a new leaf is added, first the message CREATE with relative time PRE is sent; the affected object is null because the new leaf has not yet been inserted. Then the corresponding method in the base class is used to actually add the new leaf which is then sent to all observers together with a CREATE message and relative time POST.

BaseCompoundGraph
BaseCompoundGraph defines some auxiliary methods that an attached view relies on. Determining whether there is a derived edge between two nodes (edgeQuery) or expanding an edge (edgeExpand) fall into this category. Note that this class just defines the interface on which an observing view relies; the concrete implementations go into subclasses like, for instance, DynamicLeaves which uses the data structure described in Chapter 2.

Observer «interface»
This is the interface that all classes observing a compound (di-)graph have to implement. Apart from the symbolic constants for encoding the various action messages and the relative time, it contains only one method: update. Whenever the observed graph is modified, update is called with the appropriate action message, the relative time, and the affected object. Implementing this method, an observer can react appropriately.

View
A view is both an observer of its associated compound (di-)graph and a compound (di-)graph itself. If our focus was only on the model layer, then this would be entirely correct. As the link between the model and the MVC-views, however, the view itself needs to provide some kind of change-propagation mechanism, i. e., it needs to be observable. Hence, the view

is derived from ObservableCompoundGraph instead of CompoundGraph. Since a view is just an abstract representation of its associated compound (di-)graph, it makes no sense to modify it directly through any of the methods it inherits. Hence, the view redefines all methods modifying it such that they are forwarded to its associated compound (di-)graph. Additionally, a view provides methods for expanding and contracting.

The change-propagation mechanism works as follows: whenever the view receives an update from the compound (di-)graph it observes, it first checks whether it actually has to adapt its structure. A new leaf inserted in a subtree rooted at a leaf of the view, for instance, will not be visible in this view. If the view indeed is affected, it notifies its observers in advance, makes the appropriate changes, and notifies its observers again.

Expanding and contracting affect only the view. Its observers are informed with EXPAND or CONTRACT messages. Incidentally, both operations are good examples why notification is needed before *and* after the modification. For the moment, it is sufficient to think of an MVC-view observing a view as an object storing geometric information for every node and edge. The edges incident to the expanded node are no longer part of the view after expanding, which makes it impossible to delete their geometric information afterwards; this has to be done in advance. On the other hand, the new children and their incident edges are not accessible until the node is expanded; their geometric information therefore can be set only afterwards.[1]

DynamicLeaves

The efficient data structures for the problem of graph view maintenance described in Chapter 2 so far have neither been used in CompoundGraph, nor in ObservableCompoundGraph, nor in BaseCompoundGraph. Apart from the basic implementations of all methods for the dynamic graph and hierarchy variant inherited from CompoundGraph, ObservableCompoundGraph adds just the methods related to the Observer pattern. BaseCompoundGraph only defines the interface on which an attached view relies to perform expanding and contracting. Any more efficient data structures and implementations go into subclasses of BaseCompoundGraph, a pattern known as *Strategy* (Gamma et al., 1995, pp. 315–323). Since the view only relies on the interface, the concrete data structures for the graph view maintenance problem are interchangeable as required by (DG 2). In this context, DynamicLeaves implements our solution described in Chapter 2.

[1] This problem could also be solved by defining a more advanced notification protocol. Instead of just sending the affected object, we could also send some action object that encapsulates all objects that have been deleted or added.

4.3.2 MVC-View

The MVC-view part contains all classes that are concerned with the geometric layout of a view. Besides the pure geometric information associated with nodes and edges, this also comprises the different drawing and animation styles. As shown in Figure 4.5, the MVC-view contains the following classes:

Geometry

This class encapsulates the geometric information for the nodes and edges of a view, i. e., a `PolyLine` for each edge and a `RectangularShape` for each node. It is linked to its view through the observer pattern, i. e., `Geometry` is an observer of its associated view and thus gets notified about structural changes to which it reacts by adjusting the geometric information. Note that `Geometry` contains all information for drawing a view, but does not draw it: each `Geometry` object is linked with a `ViewPanel`, a `JPanel`, that fetches the geometric information from it and actually does the drawing. In order to increase the reusability and to reduce the coupling, `Geometry` and `ViewPanel` are connected through a second *Observer* pattern. This makes it possible to replace the `ViewPanel` with, for instance, a class generating images as part of a web-based front-end.

`Geometry` makes use of an associated `DrawingStyle` to create a layout of the view, i. e., to calculate the geometric information and to update it after every expand and contract operation. It only relies on the interface defined in `DrawingStyle`; therefore, the concrete drawing style is dynamically interchangeable as required by (DG 4), again a *Strategy* pattern (Gamma et al., 1995, pp. 315–323).

Note that the type of geometric information stored already determines the presentation style. For `Geometry` we have chosen two-dimensional polylines for edges and rectangular shapes for nodes. Design goal (DG 3) is achieved as follows: other presentation styles, e. g., in three dimensions, need to provide their own geometry classes, drawing styles, animation styles, and controllers; the model, however, can be reused.

Polyline

`PolyLine` represents the geometric information of a single edge. An edge consists of arbitrarily many path points that are connected either with straight lines or with Bézier curves. In the latter case the control points are also stored. For directed edges, an arrowhead is constructed separately. A `PolyLine`, therefore, is drawn in two steps: first the curve itself as `GeneralPath`, which, incidentally, is cached for efficiency reasons, and then the arrowhead as `Shape`.

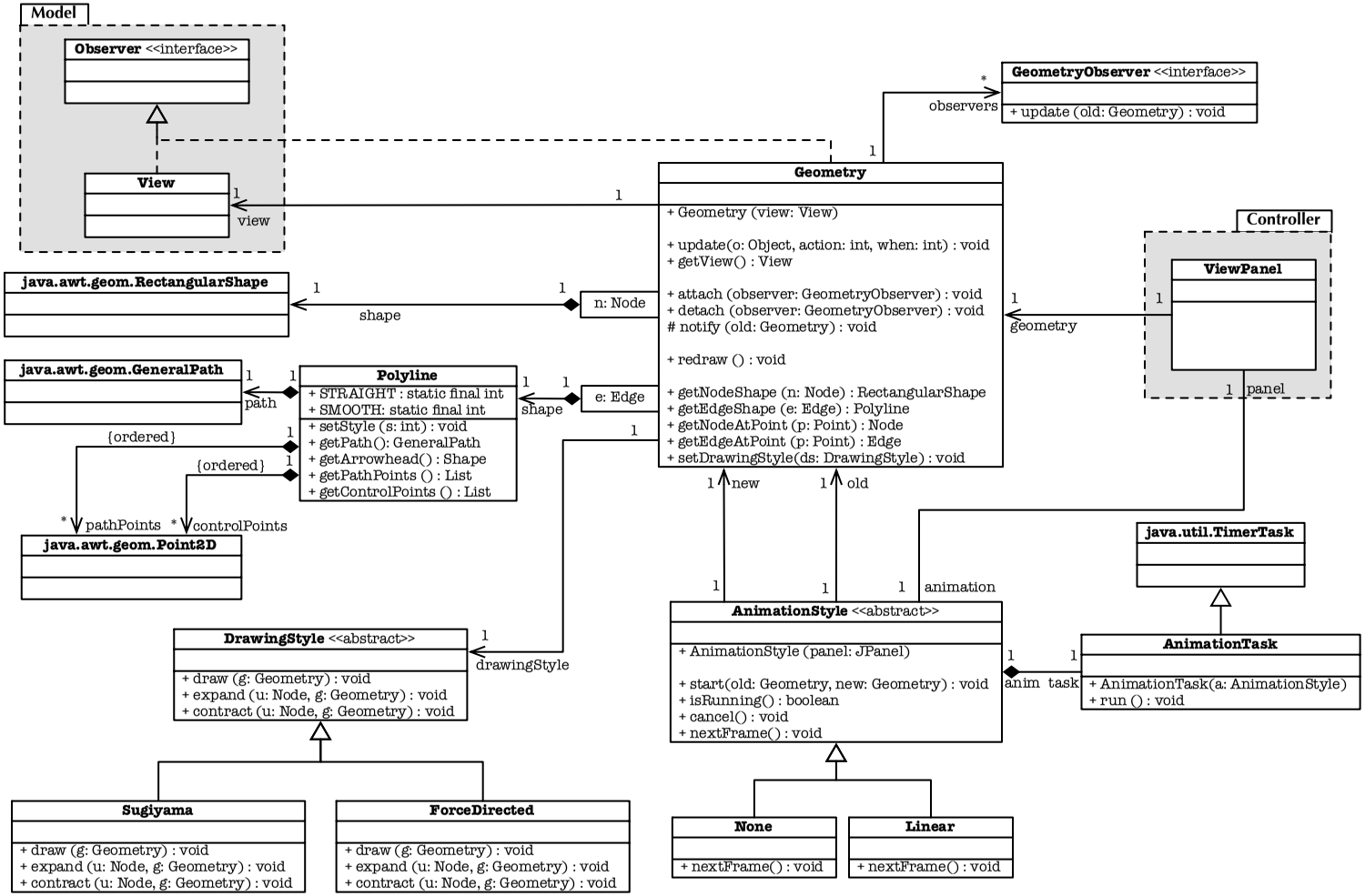

Figure 4.5: Class diagram of the MVC-view

GeometryObserver «interface»
If a class needs to keep track of an associated Geometry object, it has to implement this interface. Whenever the geometric information changes all observers are notified. In order to support animation this notification also includes a copy of the old Geometry object, i. e., the geometric information before the change.

DrawingStyle «abstract»
A DrawingStyle defines the abstract interface of layout algorithms on which Geometry relies. The concrete implementations such as the one described in Chapter 3 are subclasses of it. Geometry contains a reference to its current drawing style. This *Strategy* pattern ensures (DG 4), i. e., switching the different drawing styles on the fly.

Apart from a method for simply drawing a view, a DrawingStyle also contains methods for updating an existing drawing after expanding or contracting a node. To this end, the DrawingStyle may need to store some auxiliary structures of the previous drawing. Although it is beyond the scope of this work, it seems to be natural to stretch the DrawingStyle's field of activity a bit further as regards, for instance, updating the drawing after inserting and deleting leafs.

AnimationStyle «abstract»
Animation consists of two parts: the animation style and a timer that periodically triggers the generation of new drawings, so-called frames. The abstract class AnimationStyle defines the interface on which the ViewPanel relies. The ViewPanel stores a reference to the animation style the user has chosen. Again, this *Strategy* pattern allows us to switch the animation style dynamically as required by (DG 5).

Whenever the ViewPanel is notified that its observed Geometry object has changed, it also receives the old Geometry object. Both the old and the new Geometry are then passed on to the current animation style, which gradually adjusts the old drawing until it finally equals the new. For this purpose, the animation style provides a method nextFrame, which does a single step and then triggers a repaint of the ViewPanel to actually show the step to the user.

AnimationTask
This TimerTask is managed by the AnimationStyle. Its only purpose is to periodically call the nextFrame method of its AnimationStyle.

ViewPanel
The ViewPanel conceptually lies on the borderline between MVC-view and controller. On the one hand, it is responsible for actually drawing a

Geometry object; on the other hand, it handles user inputs like, for instance, highlighting the node a user selects with a mouse click or moving a node by dragging the mouse.

A ViewPanel acts as a GeometryObserver, i. e., it holds a reference to the Geometry object it shows and is informed whenever this object changes. Since the old Geometry is sent along with this notification, the ViewPanel can start the animation by passing both the old and the new Geometry to its AnimationStyle. While the animation is running the ViewPanel does not draw its associated, i. e., the new, Geometry object as usual, but rather the old one, which the AnimationStyle adjusts step by step until it finally is equal to the new one.

Any interaction with nodes or edges is handled in the ViewPanel, which features multiple modes, e. g., one for editing, one for selecting, and one for visual navigation through expanding and contracting. Depending on the action a user requests, the ViewPanel adjusts only the drawing, e. g., for highlighting a selected node, or it changes the Geometry, e. g., for moving a node, or it even modifies the structure of the view or the base graph, e. g., for expanding a node or for adding a leaf.

4.3.3 Controller

The controller comprises all classes handling user events. The graphical user interface of our application will consist of a single window with menu and toolbars, the GUIFrame. Within this window the user can open arbitrarily many internal windows, which are instances of the class ViewFrame, each of which contains a ViewPanel showing the drawing of a view.

GUIFrame

The GUIFrame reacts on user inputs like pressing a button in a toolbar or selecting an entry in a menu. Essentially, the GUIFrame is responsible for all inputs that are not already handled by a ViewPanel. This includes, for instance, loading and saving of compound (di-)graphs, switching editing modes, managing internal windows, or presenting a dialog for modifying the preferences. In most cases, these actions have some influence on the views shown in the internal windows. If the user, for instance, switches the mode the GUIFrame forwards this request to every ViewFrame which in turn sets the mode in its ViewPanel. Conversely, the ViewPanel can affect the GUIFrame, which is used for, e. g., enabling or disabling buttons and menu entries depending on the selected objects.

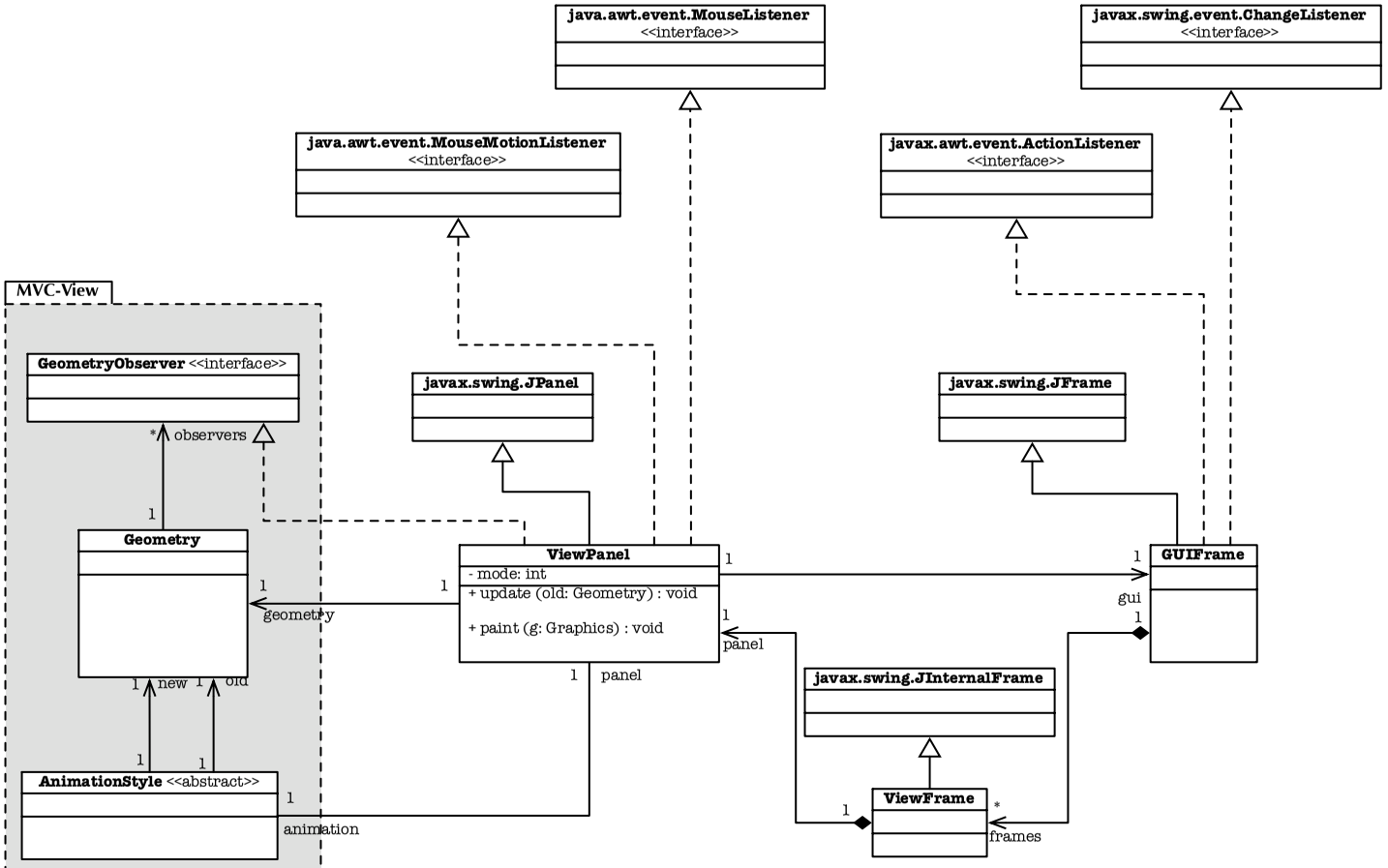

Figure 4.6: Class diagram of the controller

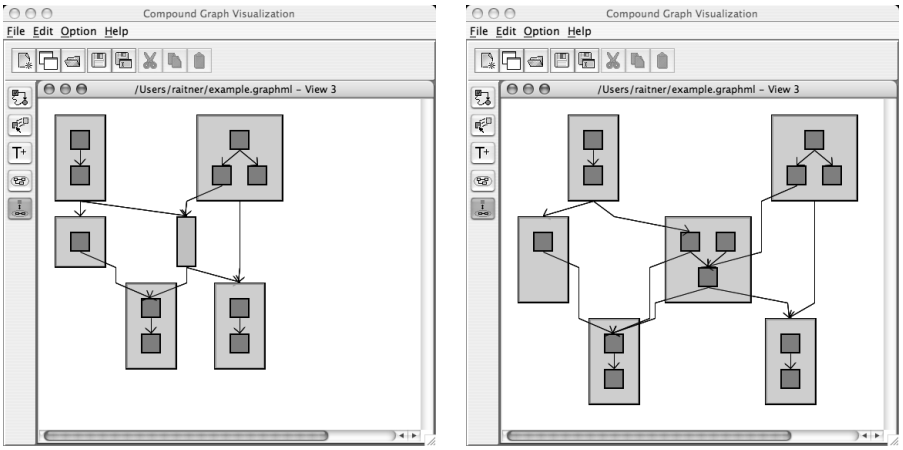

Figure 4.7: Screenshots of our proof-of-concept implementation (Pfeiffer, 2005; Pröpster, 2005) before (left) and after (expanding) the node in the middle.

`ViewFrame`

The class `ViewFrame` is used just as container for the `ViewPanel`. It receives requests from the `GUIFrame` and translates them to one or more requests for its `ViewPanel`.

4.4 Use Cases

So far the proposed architecture has been described only statically. For a deeper understanding, the dynamic interplay of the components needs to be investigated. This will be done by means of sequence diagrams for two significant use cases: expanding a node and adding a new leaf.

4.4.1 Expansion

Figure 4.8 depicts the sequence diagram for expanding a node, which looks in our proof-of-concept implementation (Pfeiffer, 2005; Pröpster, 2005) as shown on the screenshots in Figure 4.7. It starts within `panel`, an instance of the class `ViewPanel`, in a method handling the user input that triggers the expanding, e. g., `mouseClicked`. It is assumed that the position where the user clicked is passed to this method as a parameter `p` of type `Point`.

First, the `panel` asks its associated `Geometry` object which node is drawn at `p` with `getNodeAtPoint(p)`. If there is indeed a node `v` at these coordinates, which we have assumed in Figure 4.8, the `panel` fetches the associated `View`

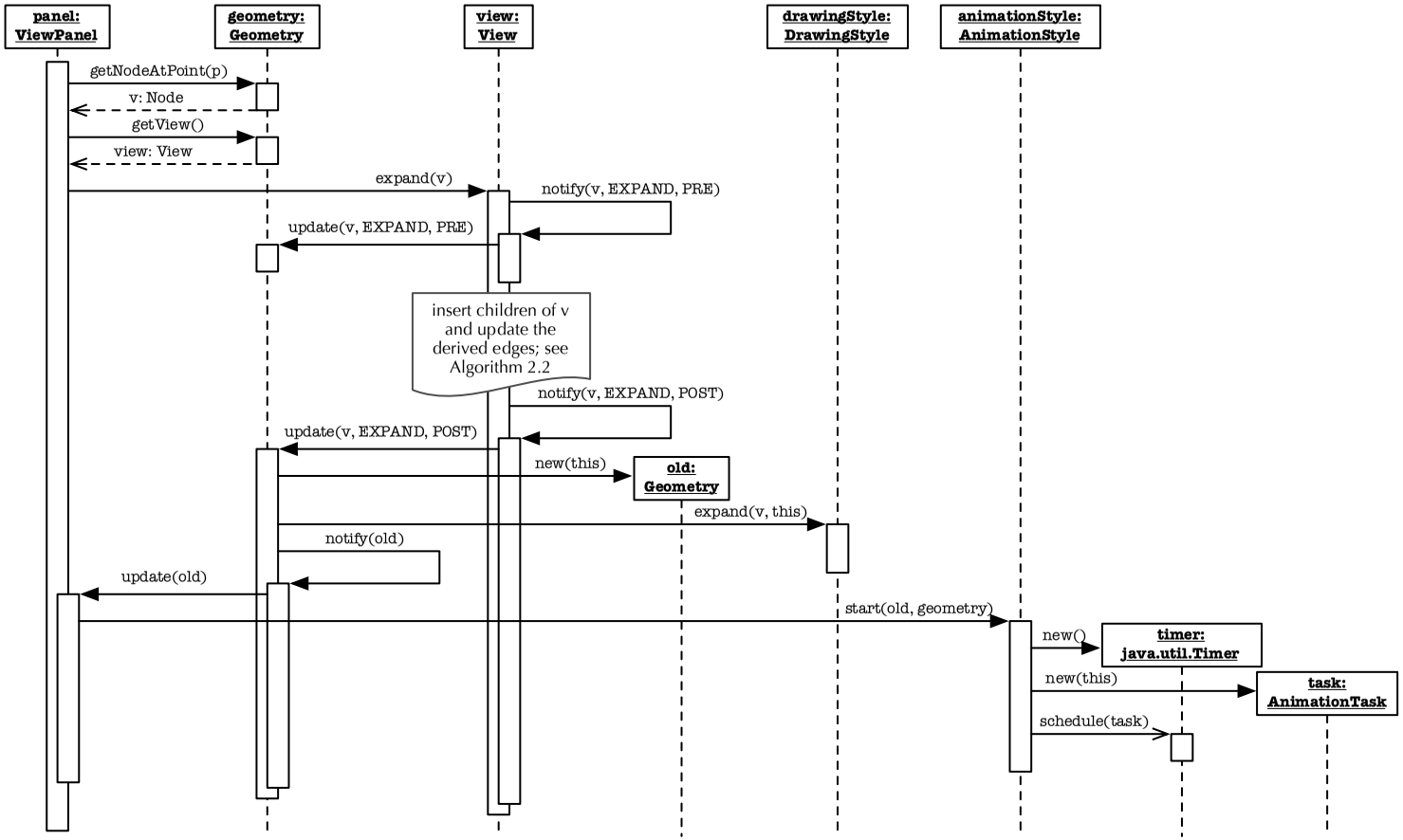

Figure 4.8: Sequence diagram for expanding a node

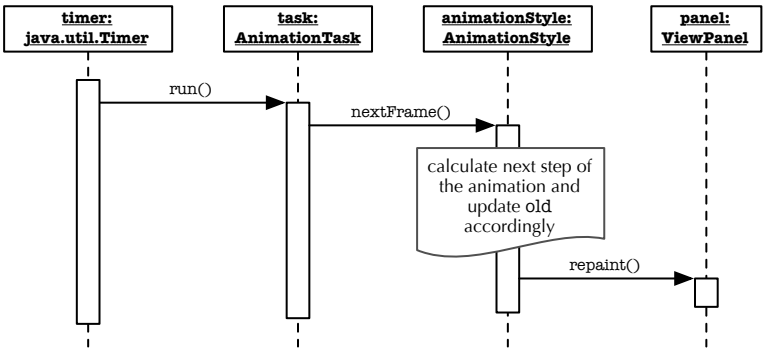

Figure 4.9: Sequence diagram showing one step of the animation

object and calls `expand(v)`. Before actually expanding the node, all attached observers are notified in advance by sending them the affected node `v`, the message `EXPAND`, and the relative time `PRE`. Note that only one of these observers, namely the `Geometry` object associated with `panel`, is shown in Figure 4.8; it uses this notification to delete the geometric information of edges incident to `v`, as they will be replaced with their corresponding expanded edges. Expanding `v`, i. e., inserting the new children and adjusting the derived edges, works as shown in Algorithm 2.2 and thus is abbreviated in Figure 4.8. After the purely combinatorial part of expanding is done, all observers of `view` are notified again, i. e., their method `update` is called with `v`, the message `EXPAND`, and the relative time `POST` as parameters.

The `geometry` object reacts on this update request by adding new geometric information for all new nodes and edges: for the nodes it chooses some default position; the edges are chosen as straight lines. Then it instantiates a new `Geometry` object `old` as copy of itself. Adjusting the geometric information after expanding is done by passing `v` and `geometry` to the method `expand` of the `DrawingStyle` associated with `geometry`; afterwards `geometry` represents the new drawing. Then all observers of `geometry` are notified that it has been modified, i. e., their method `update` is called with `old` as parameter. Usually there is only one observer, namely `panel`, which is why only this call is shown in Figure 4.8. Whenever the `panel`'s `update` method is called, it starts the animation by passing the old and the new `Geometry` object, i. e., `old` and `geometry`, to the `start` method of the associated `AnimationStyle`. The animation is done by instantiating a new `Timer` and scheduling a new `AnimationTask` to be executed periodically. Since `timer` runs the scheduled `task` in its own thread, expanding ends here.

Figure 4.9 shows one step of the animation: each time the `timer` invokes

the task's run method, it performs the next step by calling the nextFrame method of the associated animationStyle. The current state of the animation is always stored in old; nextFrame adjusts old and then triggers a paint request on the panel to actually show this step. Note, that during the animation the panel therefore has to draw old instead of geometry.

4.4.2 Adding a Leaf

Like expanding, adding a new leaf starts in a method of panel handling the user input that triggers this action, e. g., mouseClicked[2]; see Figure 4.10. The parent of the new leaf is determined with getNodeAtPoint(p), where p is the position of the mouse-click. If there is some node pa drawn at position p, which is assumed in Figure 4.10, the new leaf is created by calling newLeaf(pa) on view. As already mentioned in the description of the class View, it makes no sense to modify the view directly; therefore, the newLeaf request is forwarded to its baseGraph. Before the new leaf actually is created and inserted all observers of baseGraph are notified of the upcoming modification. In Figure 4.10 view is the only observer; it completely ignores this notification.

After the new leaf l is created, it is sent to all observers together with a message CREATE and the relative time POST. Now, the view can decide whether the new node l actually is visible. This is the case here if pa is no leaf in view[3], which we have assumed in Figure 4.10. Next all observers of view are informed in advance about the new node; in our example this is only geometry, which ignores this notification. After having added l to the view, it is sent to all observers with the message CREATE and the relative time POST. Now, geometry creates a new RectangularShape and associates it with l. Since no animation is needed, no copy of geometry is created; the observers consequently receive a null value as the old geometry in the following notification.

Whenever the panel receives an update request with a null value, it skips the animation and simply schedules itself for repainting. In our example, however, the panel itself triggered this modification; therefore, it ignores this update. Then the new leaf l is returned to the panel via view. It still

[2] Note that a ViewPanel features multiple modes in order to bind the same user action to different actions in the model, e. g., one for adding nodes and edges and another for expanding and contracting; the user can switch the mode dynamically.

[3] Adding a new leaf with a leaf of view as parent is also possible, yet the new leaf will be hidden, which leads to a somewhat inconsistent user experience. The panel, however, can easily check on this condition and expand the parent just before triggering the repaint.

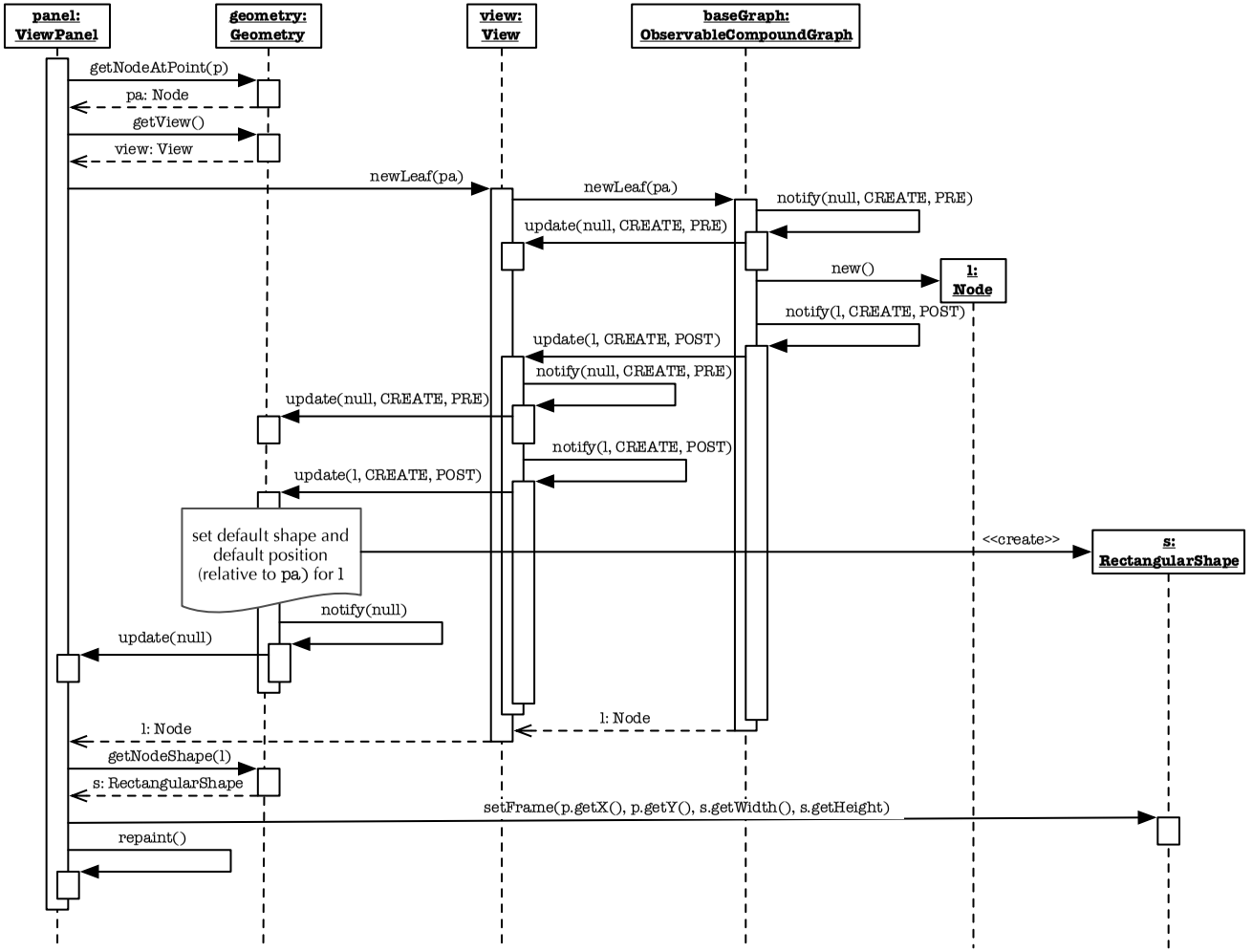

Figure 4.10: Sequence diagram for adding a new leaf.

has its default position, which the `panel` fetches with `getNodeShape` and subsequently moves it to the position where the user clicked. Finally, the `panel` triggers a `repaint`, which actually displays the new drawing to the user.

4.5 Summary

The proposed software architecture for a compound (di-)graph viewer and editor is flexible and extensible as regards drawing styles, animation styles, and data structures for the problem of graph view maintenance. Its parts are coupled only loosely in order to maximize reusability. All of this is achieved through employing well approved design patterns, such as *Observer*, *Strategy*, or *Model-View-Controller*. An implementation based on this architecture (Pfeiffer, 2005) shows not only its suitability and solidity, but also provides a framework for demonstrating the drawing style of Chapter 3, which has been implemented as part of (Pröpster, 2005).

5

Conclusion

Efficient visual navigation of hierarchically structured graphs has several interesting aspects that have been researched in this work. On the one hand, a suitable data structure for the problem of graph view maintenance is required; on the other, the expanding and contracting must be visualized appropriately. Finally, both need to be combined into an extensible and flexible software architecture that demonstrates the feasibility of the approach.

5.1 Results

The problem of graph view maintenance first was generalized from clustered graphs to compound (di-)graphs, which can have adjacency edges between internal nodes and not only between leaves of the inclusion tree. This is especially important for the newly introduced dynamic leaves variant of graph view maintenance, which supports adding and removing leaves in the inclusion tree. So far efficient data structures had been known only for the static and the dynamic graph variant. Both variants, however, do not allow modifications of the node set and thus the corresponding data structures are of limited value for an interactive editor.

We therefore generalized the data structure of Buchsbaum et al. (2000) to the new dynamic leaves variant with a novel technique of superimposing a search tree over an ordered list maintenance structure. The extra cost for this dynamization is roughly a factor of $\mathcal{O}(\log n / \log \log n)$, which seems to be acceptable given that this is the first data structure for the problem of graph view maintenance with a dynamic set of nodes.

Visualizing the expanding and contracting is something that occurs naturally in this context, yet it had not been researched until the last years. We proposed an update scheme for the static drawing algorithm for compound digraphs of Sugiyama and Misue (1991). This algorithm produces layered drawings that have many applications ranging from biochemical pathways to UML diagrams. As our experimental results showed, modifying the intermediate results of the static algorithm only locally makes the update scheme clearly more efficient than re-applying the algorithm after expanding or contracting. Moreover, the performance gain is not at the expense of drawing quality as regards the area and the number of crossings. At the same time, our update scheme preserves the user's mental map well: nodes that are are not affected stay on the same level in the same relative order and expanded edges take the same course as the contracted edge from which they emerged; furthermore, expanding and contracting are visually inverse.

Finally, the data structure and the update scheme for visualizing the expanding and contracting were integrated into an interactive editor and viewer for compound (di-)graphs. The proposed software architecture had been chosen with special emphasis laid on extensibility and reusability: both the data structure and the drawing algorithm are easily exchangeable and the purely combinatorial part, i. e., the compound (di-)graph and its views, is reusable without the editor front-end. Although it is beyond the scope of this work, provision is made for animation, which makes it even easier for the user to follow the transition from one drawing to the next. A proof-of-concept implementation based on the proposed architecture (Pfeiffer, 2005; Pröpster, 2005) shows its feasibility and suitability and provides a framework for demonstrating the proposed update scheme in combination with our data structure.

5.2 Open Problems and Future Work

As already mentioned in Chapter 2, our data structure for the dynamic leaves variant of graph view maintenance partly solves an open problem of Buchsbaum and Westbrook (2000). An efficient data structure for the most dynamic variant, i. e., with splitting and merging of clusters, remains an open problem. Note that both operations are actually straightforward: splitting just creates a new internal node between some node p and a subset of p's children; merging is its inverse. The problems arise only if redundant information like the sets $S(\cdot)$ need to be updated. On the other hand, this information is essential for `edgeQuery` and `edgeExpand`. The real challenge

therefore is to find the right amount of redundant information that on the one hand is enough for the queries and on the other does not complicate splitting and merging too much. To this end, the sets $S(\cdot)$ are not practical, especially not for splitting, where the new node would need to be equipped with such a set, which can have a size of $\mathcal{O}(n)$.

Visualizing the expanding and contracting as discussed in Chapter 3 has been researched only recently: apart from our update scheme for the layered drawing algorithm of Sugiyama and Misue (1991), so far only force-directed methods have been proposed (Huang and Eades, 1998; Dwyer and Eckersley, 2003). We have chosen the algorithm of Sugiyama and Misue (1991) as basis because its local layering naturally supports expanding. Another algorithm for layered drawings of compound digraphs, however, is the one of Sander (1999), which uses a global layering. Considering that it generally produces more compact and more pleasant drawings than the one of Sugiyama and Misue (1991), it appears to be worthwhile to research a similar update scheme for this algorithm. Although the update of the global layering is more complicated, such an update scheme should be achievable in principle (Sander, 2004).

The update scheme deliberatly has been restricted to the two operations expanding and contracting, as the focus of this work lies on visual navigation. In the fully dynamic graph and hierarchy variant, which is what the user expects in an editor, there are, however, more operations that require updating the drawing appropriately, e. g., inserting a new leaf or splitting and merging. For ordinary graphs similar problems have been researched in the field of dynamic graph drawing (Branke, 2001), but for hierarchically structured graphs only few results are known.

The software architecture introduced in Chapter 4 has been chosen deliberately as regards extensibility and flexibility of data structures, drawing styles, and animation styles. Although the proof-of-concept implementation based on this architecture (Pfeiffer, 2005; Pröpster, 2005) is suitable for demonstrating the update scheme and the feasibility of the approach, it still lacks many features of a full-fledged editor for compound (di-)graphs.

Bibliography

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

J. Abello, S. G. Kobourov, and R. Yusufov. Visualizing large graphs with compound-fisheye views and treemaps. In Pach (2004), pages 431–441.

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison Wesley, 1983.

C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: a survey. *Integration, the VLSI Journal*, 19(1):1–81, 1995.

S. Alstrup. Personal communication, 2003.

S. Alstrup and J. Holm. Improved algorithms for finding level ancestors in dynamic trees. In U. Montanari, J. D. P. Rolim, and E. Welzl, editors, *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9–15, 2000, Proceedings*, volume 1853 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2000.

O. Bastert and C. Matuszewski. Layered drawings of digraphs. In Kaufmann and Wagner (2001), chapter 5, pages 87–120.

G. D. Battista, editor. *Graph Drawing, 5th International Symposium, GD '97, Rome, Italy, September 18–20, 1997, Proceedings*, volume 1353 of *Lecture Notes in Computer Science*, 1997. Springer.

M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. In S. Rajsbaum, editor, *LATIN 2002: Theoretical Informatics, 5th Latin American Symposium, Cancun, Mexico, April 3–6, 2002, Proceedings*, volume 2286 of *Lecture Notes in Computer Science*, pages 508–515. Springer, 2002.

M. A. Bender, C. Richard, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In R. H. Möhring and R. Raman, editors, *Algorithms – ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17–21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2002.

O. Berkman and U. Vishkin. Finding level ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214–230, 1994.

F. Bertault and M. Miller. An algorithm for drawing compound graphs. In Kratochvíl (1999), pages 197–204.

K.-F. Böhringer and F. N. Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proc. ACM Conference on Human Factors in Computing Systems (CHI)*, pages 43–51, New York, NY, USA, 1990. ACM Press.

Boost Graph Library. http://www.boost.org/libs/graph/doc/.

F. J. Brandenburg. Layout graph grammars: The placement approach. In *Graph-Grammars and Their Application to Computer Science, 4th International Workshop, Bremen, Germany, March 5–9, 1990, Proceedings*, volume 532 of *Lecture Notes in Computer Science*, pages 144–156. Springer, 1990.

F. J. Brandenburg. Designing graph drawings by layout graph grammars. In Tamassia and Tollis (1995), pages 416–427.

F.-J. Brandenburg, editor. *Graph Drawing, Symposium on Graph Drawing, GD '95, Passau, Germany, September 20–22, 1995, Proceedings*, volume 1027 of *Lecture Notes in Computer Science*, 1996. Springer.

F. J. Brandenburg, M. Forster, A. Pick, M. Raitner, and F. Schreiber. Biopath – exploration and visualization of biochemical pathways. In Jünger and Mutzel (2004), pages 215–236.

U. Brandes. Drawing on physical analogies. In Kaufmann and Wagner (2001), chapter 4, pages 71–86.

U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In Mutzel et al. (2002), pages 31–44.

J. Branke. Dynamic graph drawing. In Kaufmann and Wagner (2001), chapter 9, pages 228–246.

S. Bridgeman and R. Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. In Whitesides (1998), pages 57–71.

R. Brockenauer and S. Cornelsen. Drawing clusters and hierarchies. In Kaufmann and Wagner (2001), chapter 8, pages 193–227.

A. L. Buchsbaum. Personal communication, 2003.

A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, January 9–11, 2000, San Francisco, CA, USA*, pages 566–575, 2000.

A. L. Buchsbaum, M. T. Goodrich, and J. R. Westbrook. Range searching over tree cross products. In M. Paterson, editor, *Algorithms – ESA 2000, 8th Annual European Symposium, Saarbrücken, Germany, September 5–8, 2000, Proceedings*, volume 1879 of *Lecture Notes in Computer Science*, pages 120–131. Springer, 2000.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

R. Castelló, R. Mili, and I. G. Tollis. An algorithmic framework for visualizing statecharts. In Marks (2001), pages 139–149.

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.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.

P. F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, San Francisco, CA, May 5–7, 1982*, pages 122–127, 1982.

P. F. Dietz. Finding level ancestors in dynamic trees. In F. K. H. A. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures, 2nd Workshop WADS '91, Ottawa, Canada, August 14–16, 1991, Proceedings*, volume 519 of *Lecture Notes in Computer Science*, pages 32–40. Springer, 1991.

P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, New York, NY, May 25–27, 1987*, pages 365–372, 1987.

M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

T. Dwyer and P. Eckersley. Wilmascope – a 3d graph visualization system. In Jünger and Mutzel (2004), pages 55–75.

P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

P. Eades and Q.-W. Feng. Multilevel visualization of clustered graphs. In North (1997), pages 101–112.

P. Eades and Q.-W. Feng. Drawing clustered graphs on an orthogonal grid. In Battista (1997), pages 146–157.

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

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

P. Eades and N. C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(1):379–403, 1994.

P. Eades, Q.-W. Feng, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. In North (1997), pages 113–128.

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, 1996b.

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

M. Eiglsperger, S. P. Fekete, and G. W. Klau. Orthogonal graph drawing. In Kaufmann and Wagner (2001), chapter 6, pages 121–171.

J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz – open source graph drawing tools. In Mutzel et al. (2002), pages 483–484.

Q.-W. Feng, R. F. Cohen, and P. Eades. How to draw a planar clustered graph. In D.-Z. Du and M. Li, editors, *Computing and Combinatorics, First Annual International Conference, COCOON '95, Xi'an, China, August 24–26, 1995, Proceedings*, volume 959 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 1995.

R. Fleischer and C. Hirsch. Graph drawing and its applications. In Kaufmann and Wagner (2001), chapter 1, pages 1–22.

A. Formella and J. Keller. Generalized fisheye views of graphs. In Brandenburg (1996), pages 242–253.

M. Forster. *Crossings in Clustered Level Graphs*. PhD thesis, Fakultät für Mathemetik und Informatik, Universität Passau, 2004.

C. Friedrich and P. Eades. Graph drawing in motion. *Journal of Graph Algorithms and Applications*, 6(3):353–370, 2002.

G. W. Furnas. Generalized fisheye views. In *Proc. ACM Conference on Human Factors in Computing Systems (CHI)*, pages 16–23, 1986.

H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, January 22–24, 1990, San Francisco, California*, pages 434–443, 1990.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 3rd edition, 2003.

GTL. http://www.infosun.fmi.uni-passau.de/GTL.

C. Gutwenger, M. Jünger, G. W. Klau, S. Leipert, P. Mutzel, and R. Weiskircher. AGD: A library of algorithms for graph drawing. In Mutzel et al. (2002), pages 473–474.

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

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

D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

D. Hepting. The history of a picture's worth, 1999. http://www2.cs.uregina.ca/~hepting/research/web/words/index.html.

T. Hickl. *Rechtwinkliges Layout von hierarchisch strukturierten Graphen*. PhD thesis, Fakultät für Mathemetik und Informatik, Universität Passau, 1994.

M. L. Huang and P. Eades. A fully animated interactive system for clustering and navigating huge graphs. In Whitesides (1998), pages 374–383.

M. L. Huang, P. Eades, and J. Wang. On-line animated visualization of huge graphs using a modified spring algorithm. *Journal of Visual Languages and Computing*, 9(6):623–645, 1998.

M. Jünger and P. Mutzel, editors. *Graph Drawing Software*. Mathematics and Visualization. Springer, 2004.

T. Kamada and S. Kawai. A simple method for computing general positions in displaying three-dimensional objects. *Computer Vision, Graphics and Image Processing*, 41:43–56, 1988.

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

S. G. Kobourov and M. T. Goodrich, editors. *Graph Drawing, 10th International Symposium, GD 2002, Irvine, CA, USA, August 26–28, 2002, Revised Papers*, volume 2528 of *Lecture Notes in Computer Science*, 2002. Springer.

J. Kratochvíl, editor. *Graph Drawing, 7th International Symposium, GD '99, Stirín Castle, Czech Republic, September 1999, Proceedings*, volume 1731 of *Lecture Notes in Computer Science*, 1999. Springer.

W. Lai and P. Eades. A graph model which supports flexible layout functions. Technical Report 96–15, University of Newcastle, 1996.

J. Lamping and R. Rao. The hyperbolic browser: A focus + context technique for visualizing large hierarchies. *Journal of Visual Languages and Computing*, 7(1):33–55, 1996.

LEDA. http://www.algorithmic-solutions.com/.

T. Lengauer and E. Wanke. Efficient solution of connectivity problems on hierarchically defined graphs. *SIAM Journal on Computing*, 17(6):1063–1080, 1988.

G. Liotta, editor. *Graph Drawing, 11th International Symposium, GD 2003, Perugia, Italy, September 21–24, 2003, Revised Papers*, volume 2912 of *Lecture Notes in Computer Science*, 2004. Springer.

I. A. Lisitsyn and V. N. Kasyanov. Higres – visualization system for clustered graphs and graph algorithms. In Kratochvíl (1999), pages 82–89.

K. A. Lyons, H. Meijer, and D. Rappaport. Algorithms for cluster busting in anchored graph drawing. *Journal of Graph Algorithms and Applications*, 2 (1):1–24, 1998.

J. Marks, editor. *Graph Drawing, 8th International Symposium, GD 2000, Colonial Williamsburg, VA, USA, September 20–23, 2000, Proceedings*, volume 1984 of *Lecture Notes in Computer Science*, 2001. Springer.

C. McCreary, R. Chapman, and F.-S. Shieh. Using graph parsing for automatic graph drawing. *IEEE Transactions on Systems, Man, and Cybernetics*, 28(5):545–561, 1998.

E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2): 257–276, 1985.

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):1–11, 1991.

G. Michal. Biochemical pathways (poster). Boehringer Mannheim, 1993.

K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

M. Müller-Hahnemann. Drawing trees, series-parallel digraphs, and lattices. In Kaufmann and Wagner (2001), chapter 3, pages 46–70.

P. Mutzel, M. Jünger, and S. Leipert, editors. *Graph Drawing, 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001, Revised Papers*, volume 2265 of *Lecture Notes in Computer Science*, 2002. Springer.

H. Nagamochi and K. Kuroya. Convex drawing for c-planar biconnected clustered graphs. In Liotta (2004), pages 369–380.

S. C. North, editor. *Graph Drawing, Symposium on Graph Drawing, GD '96, Berkeley, California, USA, September 18–20, Proceedings*, volume 1190 of *Lecture Notes in Computer Science*, 1997. Springer.

S. C. North and G. Woodhull. Online hierarchical graph drawing. In Mutzel et al. (2002), pages 232–246.

J. Pach, editor. *Graph Drawing, 12th International Symposium, GD 2004, New York, NY, USA, September 29–October 2, 2004, Revised Selected Papers*, volume 3383 of *Lecture Notes in Computer Science*, 2004. Springer.

G. Parker, G. Franck, and C. Ware. Visualization of large nested graphs in 3d: Navigation and interaction. *Journal of Visual Languages and Computing*, 9(3):299–317, 1998.

F. Pfeiffer. Implementation eines Editors für Compound Graphen. Diplomarbeit, University of Passau, 2005.

F. P. Preparata, J. S. Vitter, and M. Yvinec. Output-sensitive generation of the perspective view of isothetic parallelepipeds. *Algorithmica*, 8:257–283, 1992.

M. Pröpster. Visuelle Navigation in Compound Graphen. Diplomarbeit, University of Passau, 2005.

H. C. Purchase. Performance of layout algorithms: Comprehension, not computation. *Journal of Visual Languages and Computing*, 9(6):647–657, 1998.

H. C. Purchase, R.F.Cohen, and M. James. An experimental study of the basis for graph drawing algorithms. *The ACM Journal of Experimental Algorithmics*, 2(4), 1997.

M. Raitner. HGV: A library for hierarchies, graphs, and views. In Kobourov and Goodrich (2002), pages 236–243.

M. Raitner. Maintaining hierarchical graph views for dynamic graphs. Technical Report MIP-0403, Universität Passau, 2004a.

M. Raitner. Visual navigation of compound graphs. In Pach (2004), pages 403–413.

M. Raitner. Dynamic tree cross products. In *Proc. 15th Intl. Symposium on Algorithms and Computation (ISAAC)*, volume 3341 of *Lecture Notes in Computer Science*, pages 793–804. Springer, 2004c.

G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, 1997. World Scientific.

G. Sander. *Visualisierungstechniken für den Compilerbau*. PhD thesis, Universität des Saarlandes, 1996a.

G. Sander. Layout of compound graphs. Technical Report A/03/96, Universität des Saarlandes, FB 14 Informatik, 1996b.

G. Sander. Graph layout for applications in compiler construction. *Theoretical Computer Science*, 217(2):175–214, 1999.

G. Sander. Personal communication, 2004.

M. Sarkar and M. H. Brown. Graphical fisheye views of graphs. In *Proc. ACM Conference on Human Factors in Computing Systems (CHI)*, pages 83–91, 1992.

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

F. Schreiber. *Visualisierung biochemischer Reaktionsnetze*. PhD thesis, Fakultät für Mathemetik und Informatik, Universität Passau, 2001.

F.-S. Shieh and C. McCreary. Directed graphs drawing by clan-based decomposition. In Brandenburg (1996), pages 472–482.

F.-S. Shieh and C. L. McCreary. Clan-based incremental drawing. In Marks (2001), pages 384–395.

SRS. SRS: Sequence Retrieval System. http://srs.ebi.ac.uk.

Statistisches Bundesamt Deutschland. Ausstattung privater Haushalte mit Informations- und Kommunikationstechnik Ergebnis der Einkommens- und Verbrauchsstichprobe 1998 und 2003, 2004. http://www.destatis.de/basis/d/evs/budtab6.php.

K. Sugiyama and K. Misue. An overview of diagram based idea organizer: D-ABDUCTOR. Technical Report IIAS-RR-93-3E, International Institute for Advanced Study of Social Science, Fujitsu Laboratories Ltd., March 1993.

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, 1991.

K. Sugiyama and K. Misue. A generic compound graph visualizer/manipulator: D-ABDUCTOR. In Brandenburg (1996), pages 500–503.

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

R. Tamassia and I. G. Tollis, editors. *Graph Drawing, DIMACS International Workshop, GD '94, Princeton, New Jersey, USA, October 10–12, 1994, Proceedings*, volume 894 of *Lecture Notes in Computer Science*, 1995. Springer.

R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, apr 1985.

A. K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21:101–112, 1984.

P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3), 1977.

P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

R. Weiskirchner. Drawing planar graphs. In Kaufmann and Wagner (2001), chapter 2, pages 23–45.

S. Whitesides, editor. *Graph Drawing, 6th International Symposium, GD '98, Montréal, Canada, August 1998, Proceedings*, volume 1547 of *Lecture Notes in Computer Science*, 1998. Springer.

D. E. Willard. Examining computational geoemtry: van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing*, 29(3):1030–1049, 2000.

D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In C. Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28–30, 1986*, pages 251–260. ACM Press, 1986.

List of Figures

1.1 Clipping of the New York Metro map . . . 2
1.2 Clipping of the biochemical pathway wall chart . . . 3
1.3 Example of a biochemical pathway . . . 6
1.4 Example of a compound digraph . . . 12
1.5 A compound digraph depicted as an inclusion diagram . . . 12
1.6 Upper subtree in a compound digraph . . . 17
1.7 Example of a view . . . 17
1.8 Upper subtree depicted as an inclusion diagram . . . 17
1.9 View depicted as an inclusion diagram . . . 17
1.10 Example for `expand`, before . . . 21
1.11 Example for `expand`, after . . . 21
1.12 Example for `split`, before . . . 21
1.13 Example for `split`, after . . . 21
1.14 Example for `merge`, before . . . 21
1.15 Example for `merge`, after . . . 21
1.16 The update scheme vs. a complete relayout . . . 27
1.17 Screenshot showing three views . . . 32

2.1 Example of a three-dimensional tree cross product . . . 37
2.2 The set $S(\cdot)$ . . . 40
2.3 The three-dimensional set $S_3(\cdot)$ . . . 47
2.4 The two-dimensional projection of the set $S_3(u_1)$ . . . 47
2.5 compound digraph modeled as tree cross product . . . 59
2.6 Initial view . . . 60
2.7 View after contracting the highlighted node v . . . 60
2.8 Using `edgeExpand` in `expand` yields too many edges . . . 60

3.1 Edges in the derived compound digraph . . . 69
3.2 Example of a layer assignment . . . 71
3.3 Improper edge with $\text{parent}(u) = \text{parent}(v)$ . . . 72
3.4 Improper edge with $\text{clev}(\text{parent}(u)) = \text{clev}(\text{parent}(v))$ . . . 72
3.5 Improper edge . . . 73
3.6 Inserting a dummy node complex . . . 73

3.7 Remaining improper edge . . . 73
3.8 Complete replacement of an improper edge . . . 73
3.9 Example of a local hierarchy . . . 74
3.10 Non-uniform heights result in edge-node crossings . . . 75
3.11 Bends . . . 79
3.12 Additional layers in the updated assignment . . . 83
3.13 Updated layer assignment . . . 83
3.14 Layer assignment after re-applying the original algorithm . 83
3.15 Dummy nodes and edges for a contracted edge . . . 87
3.16 Dummy nodes and edges for the expanded edges . . . 87
3.17 Expanding edges up to the first dummy node . . . 87
3.18 Grouping dummy nodes for the expanded edges . . . 87
3.19 Order for an incoming edge lying to the left . . . 90
3.20 Order for an incoming edge lying to the right . . . 90
3.21 Order for an outgoing edge lying to the left . . . 90
3.22 Order for an outgoing edge lying to the right . . . 90
3.23 Reusing of local coordinates . . . 93
3.24 Increase in area as affected by the number of nodes n . . . 97
3.25 Increase in area as affected by the density δ . . . 97
3.26 Increase in area as affected by the number of children γ . . . 97
3.27 Decrease in crossings as affected by the number of nodes n . 99
3.28 Decrease in crossings as affected by the density δ . . . 99
3.29 Decrease in crossings as affected by the number of children γ 99
3.30 Performance gain as affected by the number of nodes n . . . 100
3.31 Performance gain as affected by the density δ . . . 100
3.32 Performance gain as affected by the number of children γ . . 100
3.33 Example of a randomly generated compound digraph . . . 103
3.34 Increase in area as affected by the number of nodes n . . . 104
3.35 Decrease in crossings as affected by the number of nodes n . 104
3.36 Performance gain as affected by the number of nodes n . . . 104

4.1 Screenshot of the proof-of-concept implementation . . . 108
4.2 Schematic overview of the Model-View-Controller paradigm 110
4.3 Complete class diagram . . . 111
4.4 Class diagram of the model . . . 114
4.5 Class diagram of the MVC-view . . . 118
4.6 Class diagram of the controller . . . 121
4.7 Screenshots before and after expanding a node . . . 122
4.8 Sequence diagram for expanding a node . . . 123
4.9 Sequence diagram showing one step of the animation . . . 124

4.10 Sequence diagram for adding a new leaf. 126

List of Tables

1.1 Variants of graph view maintenance 22

2.1 Results for d-dimensional tree cross products 58
2.2 Results for graph view maintenance 66

Index

Symbols

$E(\boldsymbol{u})$ 37
$N(v)$ 59
$S(u_1)$ 40
$S_{d+1}(u_1)$ 46
$\mathcal{D} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ 80
$\mathcal{D}' = (\mathcal{V}', \mathcal{E}', \mathcal{F}')$ 80
$\mathfrak{G} = (\mathfrak{V}, \mathfrak{E})$ 37
$\mathrm{Opt}(U, v)$ 62
$\mathrm{clev} : V \to \bigcup_{i \in \mathbb{N}} \mathbb{N}^i$ 69
$\mathrm{type} : F_\mathrm{D} \to \{\prec, \preceq\}$ 69
$\max(u)$ 40
$\min(u)$ 40
σ 73
$\mathcal{D}_\mathrm{A} = (\mathcal{V}, \mathcal{E}, \mathcal{F}_\mathrm{A}, \mathrm{clev})$ 81
$\mathcal{D}_\mathrm{A}' = (\mathcal{V}', \mathcal{E}', \mathcal{F}_\mathrm{A}', \mathrm{clev}')$ 81
$\mathcal{D}_\mathrm{O} = (\mathcal{V}_\mathrm{P}, \mathcal{E}_\mathrm{P}, \mathcal{F}_\mathrm{P}, \mathrm{clev}, \sigma)$ 86
$\mathcal{D}_\mathrm{O}' = (\mathcal{V}_\mathrm{P}', \mathcal{E}_\mathrm{P}', \mathcal{F}_\mathrm{P}', \mathrm{clev}', \sigma')$ 89
$\mathcal{D}_\mathrm{P} = (\mathcal{V}_\mathrm{P}, \mathcal{E}_\mathrm{P}, \mathcal{F}_\mathrm{P}, \mathrm{clev})$ 84
$\mathcal{D}_\mathrm{P}' = (\mathcal{V}_\mathrm{P}', \mathcal{E}_\mathrm{P}', \mathcal{F}_\mathrm{P}', \mathrm{clev}')$ 85
$D_\mathrm{A} = (V, E, F_\mathrm{A}, \mathrm{clev})$ 69
$D_\mathrm{D} = (V, E, F_\mathrm{D}, \mathrm{type})$ 69
$D_\mathrm{F} = (V, E, F_\mathrm{F}, \mathrm{type})$ 70
$D_\mathrm{O} = (V_\mathrm{P}, E_\mathrm{P}, F_\mathrm{P}, \mathrm{clev}, \sigma)$ 73
$D_\mathrm{P} = (V_\mathrm{P}, E_\mathrm{P}, F_\mathrm{P}, \mathrm{clev})$ 71

A

abridgment 18, 33
absolute coordinate **75**, 92
adjacency edge . . **11**, 12, 69, 70, 81, 84, 96, 101, 113
adjacency list 52
aesthetic criteria 1–2, 8, 23
ancestor 11
  nearest $\sim$ 15
angular resolution 25
animation 10, 29, 32, 34, 124
animation style 107, 109, 112, 119
AVL tree 50

B

barycenter heuristic **74**, 76
BioPath 5, 6
bipartite graph 5, 14, **36**
block graph 77
bottom layer **72**, 80

C

change-propagation 110, 116
child 11
cigraph 13
clan-based decomposition 14
clipping 3
clustered graph . . **13**, 14, 16, 18, 20, 33, 35, 60, 64, 129
clustered planar graph 26
compound digraph . **11**, 12, 17, 33, 59, 64
  assigned $\sim$ **69**, 70, 81–84
  cycle-free $\sim$ 70
  derived $\sim$ **69**, 69, 70, 83, 96
  ordered $\sim$ **73**, 75, 86
  proper assigned $\sim$ **71**, 72, 84
  randomly generated $\sim$ 95–96, 101–103
compound graph **12**, 33, 64
compound layer **69**, 70, 94
compressed tree 22, **44**, 56
  approximation 56
`contract` **19**, 20, 59, 62, 67
contracted edge **20**, 83, 87, 105
contracted stratified tree **41**, 42, 50
controller 110, 112, 121

coordinate
absolute ∼ **75**, 92
local ∼ **75**, 92, 93
coordinate assignment 24, 76–79
crossing minimization
one sided 72, 74
crossing reduction 24, 74, 84, 86
CST . . . . . *see* contracted stratified tree
cycle removal 24, 70, 81, 82

D

DAG . . . . . *see* directed acyclic graph
`deleteEdge` 19, **38**, 39, 44, 47, 55
`deleteLeaf` 19, **38**, 39, 55
depth 11
derived edge 4, **16**, 38, 69, 83, 94
derived hyperedge **37**, 38
derived hypergraph 37
descendant 11
design pattern
Model-View-Controller 10, 33, 34, 108, 110, 115, 127
Observer . . . 10, 107, 112, 117, 120, 127
Strategy . . . 107, 116, 117, 119, 127
digraph 11
compound ∼ **11**, 12, 17, 33, 59, 64
inclusion ∼ 11
directed acyclic graph 14
directed edge 11
directed graph . . . . . *see* digraph
drawing
force-directed ∼ . . 9, 24, 29, 33, 34
layered ∼ 9, 24, 29
multilevel ∼ 25
nested ∼ **25**, 67, 68
orthogonal ∼ 25
polyline ∼ 23
straight-line ∼ 23
drawing convention 23
drawing style . . . 23, 109, 112, 119, 127
dummy node **71**, 72, 73, 87, 90
∼ complex **72**, 73, 79
external ∼ 88
local ∼ 88, 91
dynamic graph drawing 26, 28

E

edge
adjacency ∼ . **11**, 12, 69, 70, 81, 84, 96, 101, 113
derived ∼ 4, **16**, 38, 69, 83, 94
directed ∼ 11
inclusion ∼ 11
incoming ∼ 11
induced ∼ 11
long-span ∼ 24
outgoing ∼ 11
proper ∼ **71**, 72, 73, 86
`edgeExpand` **38**, 39, 43–44, 47, 54, 59–60, 64, 81
`edgeExpand` algorithm 43
`edgeQuery` **38**, 39–41, 46, 53–54
`edgeReport` . . . **38**, 39, 41–43, 46, 53–54
edge routing 78–80
entity relationship diagram 30
`expand` **19**, 20, 59–62, 67
`expand` algorithm 61
expanded edges . **20**, 82, 84, 87, 88, 90, 91, 105
external dummy node 88

F

feedback arc set 70
fisheye view 3, 5, 7, 33
focus and context 3, 34
force-directed drawing 9, 24, 29, 33, 34

G

global layering 30, 131
global order 72
graph
bipartite ∼ 5, 14, **36**
block ∼ 77
clustered ∼ . **13**, 14, 16, 18, 20, 33, 35, 60, 64, 129
clustered planar ∼ 26
compound ∼ **12**, 33, 64
directed ∼ . . . . . *see* digraph
directed acyclic ∼ 14
hierarchical ∼ 12

nearest neighbor ∼ 28
telephone call ∼ 2, 4, 8
undirected ∼ 11
graph drawing 23
dynamic ∼ 26, 28
limits of ∼ 2–3
graph drawing software 23
graph grammar 12
layout ∼ 13
graph view 5, 7, 14, **16**, 16, 17
graph view maintenance 7, **18**, 22, 31, 34, 35, 66, 80, 109, 110, 116, 127, 129
dynamic graph 7, 19, 64
dynamic graph and tree 7, 19
dynamic leaves . . 8, 19, 20, 23, 39, 57, 64, 129, 130
static 7, 19

H

hammock 36
height 11
hierarchical graph 12
higraph 13
horizontal compaction 77
hyperbolic view 3, 7
hyperedge 5, 14, 22, **36**, 37
derived ∼ **37**, 38
hypergraph 5, 14, **36**
derived ∼ 37

I

incident node 11
inclusion diagram 12, 17
inclusion digraph 11
inclusion edge 11
inclusion tree 12, 17
incoming edge 11
induced edge 11
internal node 11

L

layer
bottom ∼ **72**, 80
compound ∼ **69**, 70, 94
local ∼ **70**, 71, 81, 86
top ∼ **72**, 80
layer assignment 24, 69–71, 81, 83, 93–96
layered drawing 9, 24, 29, 67
∼ of compound digraphs . 30, 67, 68, 130, 131
layout graph grammar 13
leaf 11
left expansion set 64
level ancestor **43**, 51–53
list labeling . . . *see* order maintenance
local coordinate **75**, 92, 93
local dummy node 88, 91
local hierarchy **73**, 74, 76, 91
metrical ∼ **76**, 78
local layer **70**, 71, 81, 86
local layering 30
local order 72, 86, 92
long-span edge 24
longest path layering 77

M

mental map 8, 10, 27, 30, 32, 67, 68, 81, 86, 93–95, 105, 130
`merge` 19, 20, 33
metrical local hierarchy **76**, 78
minimum separation constraint . . . 76
model 110, 113, 114
multilevel drawing 25
MVC-view 110, 112, 117, 118

N

nearest ancestor 15
nearest neighbor graph 28
nested drawing **25**, 67, 68
`newEdge` 19, **38**, 39, 44, 47, 54–55
`newLeaf` 19, **38**, 39, 48, 55
node 11
dummy ∼ **71**, 72, 73, 87, 90
internal ∼ 11
node induced subgraph 11

O

order
global ∼ 72
local ∼ 72, 86, 92

order maintenance 48–50, 52, 129
orthogonal drawing 25
orthogonal ordering model 28, 30
outgoing edge 11
overview reaction 5, 6

P

parent 11
path 11
pathway 3, 5–7, 29
PERT chart 9, 30
polyline drawing 23
presentation style 109, 112, 117
priority layout method **76**, 76–78
priority search tree 45
proper edge **71**, 72, 73, 86
proximity relation 28
pruning 15

R

reaction 5, 6
reaction network 2, 8, 9, 14, 18, 29
red-black tree 50
right expansion set 64
root 11
rooted tree 11

S

scaling 7
  non-uniform ∼ 5
scrolling 3
similarity measure 28
software
  graph drawing ∼ 23
software architecture . 31, 32, 107, 108, 127, 130, 131
  key features 32
split 19, 20, 33
splitting method **74**, 88, 91
spring embedder . . . *see* force-directed drawing
statechart 13, 26
straight-line drawing 23
subtree 11
  upper ∼ **15**, 15–17
successor **41**, 54

T

telephone call graph 2, 4, 8
text indexing 35
top layer **72**, 80
tree
  compressed ∼ 22, **44**, 56
  contracted stratified ∼ . **41**, 42, 50
  inclusion ∼ 12, 17
  rooted ∼ 11
  van Emde Boas ∼ 41
tree cross product 22, **36**, 37, 40, 47, 56, 58, 59, 63
tree view 18
2-3 tree 50, 53–55

U

UML diagram 9, 30
undirected graph 11
unrelated nodes 11
updateLevels algorithm 82
updateOrdered algorithm 89
updateProper algorithm 85
update scheme 9, 10, 31, 32, 68, 95, 105, 130
upper subtree **15**, 15–17

V

van Emde Boas tree 41
vertical alignment 77
view . . . . . *see* graph view
  fisheye ∼ 3, 5, 7, 33
  graph ∼ 5, 7, 14, **16**, 16, 17
  hyperbolic ∼ 3, 7
  tree ∼ 18
visualization 1, 67
visually inverse 30, 31, 68, 94, 105, 130

Z

zooming 3