HGV: A Library for Hierarchies, Graphs, and Views

Marcus Raitner

University of Passau, D-94032 Passau, Germany,
raitner@fmi.uni-passau.de

Abstract. We introduce the base architecture of a software library which combines graphs, hierarchies, and views and describes the interactions between them. Each graph may have arbitrarily many hierarchies and each hierarchy may have arbitrarily many views. Both the hierarchies and the views can be added and removed dynamically from the corresponding graph and hierarchy, respectively. The software library shall serve as a platform for algorithms and data structures on hierarchically structured graphs. Such graphs become increasingly important and occur in special applications, e. g., call graphs in software engineering or biochemical pathways, with a particular need to manipulate and draw graphs.

1 Introduction

Graphs are often used to model structured data, e. g., *road maps* with locations connected by roads, the *web-graph* with web-pages connected by links or *biochemical pathways* with substances connected by reactions [3]. Particularly for large graphs it is important to view, manipulate, and automatically draw them using some software tools. Clearly, all these tools rely on an appropriate data structure for graphs. There already exist various libraries implementing data structures for graphs, e. g., LEDA [15], GTL [9], or BOOST GRAPH LIBRARY [2].

There are many applications with very large graphs. Such graphs must be manipulated efficiently by powerful operations acting on subgraphs. Repeated use of subgraphs induces a hierarchical structure. This is a particular means also for the visualization of large graphs, because certain subgraphs may be collapsed and represented by a meta-node. There are various concepts for extending graphs with such a hierarchical structure [7,10,12,18]. Moreover there are layout algorithms [7,18] and interactive systems [17,13] working with these concepts.

The inclusion hierarchy on top of a graph can be used to define abstract representations of the graph, so called *views* [5,6,7] or *abridgements* [13]. Instead of displaying every single node of the graph, in a view a subset of the nodes is represented as one (abstract) node. These nodes are connected by an (abstract) edge if there is an edge between nodes in the corresponding subsets. In a graph editor views are very convenient since they simultaneously provide an overview of the whole graph and some details from a special portion. In some respects a view can be compared to the well known tree views of file systems where initially only the topmost layer of folders is shown and the folders of interest can be expanded *within* the view.

Up to now there is no thorough description of the software architecture for a library providing graphs with hierarchies and views. This contribution and the prototype implementation [11] are a first step.

1.1 Related Work

There are several notions of hierarchically structured graphs in the literature. Depending on their purpose these definitions come in different flavors ranging from *succinct representations* of large graphs [16] to *graph drawing* [7, 18]. The latter is surveyed in [4].

Structuring a graph hierarchically was first employed on *statecharts* [10]. There the term *higraph* is defined as an ordinary graph with an acyclic inclusion relation on its nodes. A higraph can be seen as a directed acyclic graph (DAG) describing an inclusion hierarchy with additional graph edges connecting arbitrary nodes.

The *compound graphs* [18] consist of a set of nodes together with *inclusion edges* and *adjacency edges* such that the inclusion edges induce a directed graph (mostly a tree) and the adjacency edges induce a directed graph. Like a higraph a compound graph can be seen as a directed graph describing the hierarchy with additional (adjacency) edges connecting arbitrary nodes.

Closely related to compound graphs and higraphs are the *cigraphs* [14]. A cigraph is a root node together with a possibly empty set of sub-cigraphs and a set of edges between nodes in *different* sub-cigraphs. This means that an edge is always stored at the least common ancestor of the cigraphs it connects.

Neither higraphs nor compound graphs nor cigraphs distinguish properly between the nodes of the underlying graph and the nodes of the hierarchy. On the other hand the *clustered graphs* [7] consist of an ordinary graph *and* a tree with the leaves of the tree being exactly the nodes of the graph. This can be seen as a tree describing an inclusion hierarchy with additional graph edges between its leaves only. Each node in the tree represents a *cluster* of nodes of the graph that are leaves of the subtree rooted at the node. With the additional restriction that there are no long tree edges, i. e., tree edges connecting nodes whose heights in the tree differ by more than one, a *view at level i* of a clustered graph is a graph consisting of all nodes of height i in the tree. Two nodes in the view are connected by an edge if there is at least one graph edge connecting the respective clusters.

In [6, 5] the notion of a view of a clustered graph is generalized. There a view is a subset of the nodes of the tree such that the corresponding clusters partition the set of nodes of the graph. Hence the view can be detailed and coarsened as needed. In [6] it is shown how to maintain a view on a clustered graph efficiently while navigating up and down the hierarchy using the methods `collapse` and `expand`.

Some interactive systems [13, 17] actually use these two operations to explore clustered graphs. However, they do not describe thoroughly the coherence of graph, hierarchies and views from a software-engineering perspective. Other systems [1] model clustered graphs as an extension of ordinary graphs and do not provide views at all.

1.2 Our Results

We propose a software architecture for the interaction of graphs, hierarchies, and views. Employing the *Observer* design pattern [8] twice, our model features an *arbitrary* number of hierarchies for each graph as well as an *arbitrary* number of views for each hierarchy. Our model prepares the ground for a graph library with hierarchies and views. Such a library, integrated in a graph editor or a graph layout tool, provides an additional dimension for the structuring of graphs and makes working with large graphs more convenient. A first implementation of our model is available [11].

Since we allow more than one hierarchy per graph we need a rigid distinction between graph and hierarchy. Therefore we extend the clustered graphs by *cross edges* between tree nodes. Cross edges can be seen as edges on a higher level of abstraction, They describe a relation between clusters of nodes while edges in the graph describe a relation between the nodes of the graph. For our model we generalize the *views* in [6, 5], which are defined only on clustered graphs. Moreover, our views need not cover the whole graph and can therefore be used to model subgraphs and views of subgraphs.

2 Basic Notions

Definition 1. *Let G be a graph with nodes $V(G)$ and edges $E(G)$. A* hierarchy *H over G consists of nodes $V(H) = V(G)\dot{\cup}V_{\mathrm{i}}(H)$,* cross edges *$E_{\mathrm{c}}(H)$ and a grouping function $c_H : V(H) \to \mathfrak{P}(V(G))$ such that*

(i) $\forall\, v \in V(G) : c_H(v) = \{v\}$
(ii) $\exists\, v \in V_{\mathrm{i}}(H) : c_H(v) = V(G)$
(iii) $\forall\, u, v \in V(H) : c_H(u) \cap c_H(v) \neq \emptyset \Rightarrow (c_H(v) \subseteq c_H(u) \;\vee\; c_H(u) \subseteq c_H(v))$
(iv) $\forall\, e = (u, v) \in E_{\mathrm{c}}(H) : c_H(u) \cap c_H(v) = \emptyset$

The nodes $V_{\mathrm{i}}(H)$ are the called inner nodes *and $c_H(v)$ is called the* cluster *of v.*

A hierarchy over a graph can be seen as a rooted tree with its leaves corresponding exactly to the nodes of the underlying graph and with cross edges connecting tree nodes which are not predecessors of one another. In Fig. 1 the hierarchy over a graph is depicted as an inclusion diagram. The solid lines are edges of the graph and the dashed ones are cross edges. The boxes represent the clusters $c_H(v)$ for $v \in V(H)$. Figure 2 shows the same graph and the same hierarchy as a tree with graph edges and cross edges.

Because of the cross edges this notion of a hierarchy is more general than the clustered graphs of [7]. Our definition differs from compound graphs of [18] and the higraphs of [10] since the hierarchy may not be a DAG and only cross

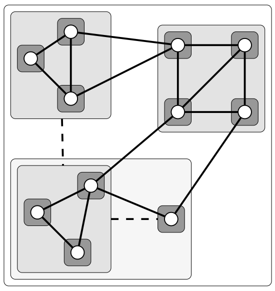

Fig. 1. Inclusion Diagram

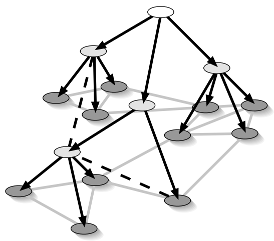

Fig. 2. Tree

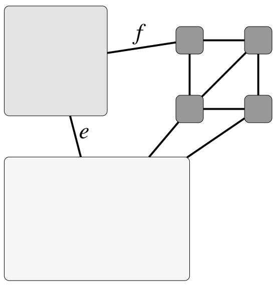

Fig. 3. View

edges are allowed. The cigraphs of [14] are equivalent to our model, but do not provide a rigid distinction between graph and hierarchy which is necessary for more than one hierarchy per graph.

Definition 2. *A view S of a hierarchy H over a graph G is a graph with nodes $V(S) \subset V(H)$ such that $\forall\, u, v \in V(S) : u \neq v \Rightarrow c_H(u) \cap c_H(v) = \emptyset$. Two nodes $u, v \in V(S)$ are connected by an edge if and only if*

(i) $\exists\, u', v' \in V(H) : c_H(u') \subseteq c_H(u) \;\wedge\; c_H(v') \subseteq c_H(v) \;\wedge\; u'$ *is connected to* v' *by a cross edge or*
(ii) $\exists\, u' \in c_H(u), v' \in c_H(v) : u'$ *and* v' *are connected by an edge in* G.

Figure 3 shows a view of the hierarchy of Fig. 1. The edges e and f are examples for (i) and (ii), respectively. There are many other views.

Our definition generalizes the one given in [6, 5] in various respects. The clusters of nodes in the view need not cover the whole graph. Therefore we can model subgraphs and views of subgraphs. Using cross edges, our views can have edges connecting clusters which are not connected in the underlying graph.

3 Core Architecture

The three main classes in the class diagram in Fig. 4 are `graph`, `hierarchy`, and `view`. The other two classes, `observable_graph` and `observer`, are primarily abstract interfaces modeling the observer design pattern [8].

3.1 Assumptions

The following assumptions guided the design of our library.

1. *Fully dynamic graph, hierarchy and view:* Nodes and edges can be added or removed from the underlying graph. The inner nodes of the hierarchy can be inserted or deleted and nodes in the view can be expanded or collapsed.

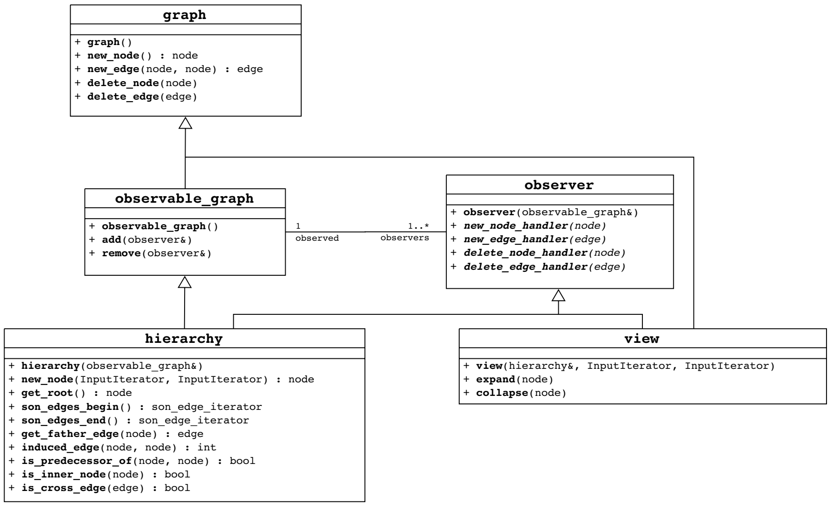
graph
+ graph()
+ new_node() : node
+ new_edge(node, node) : edge
+ delete_node(node)
+ delete_edge(edge)
observable_graph
+ observable_graph()
+ add(observer&)
+ remove(observer&)
1
observed
1..*
observers
observer
+ observer(observable_graph&)
+ new_node_handler(node)
+ new_edge_handler(edge)
+ delete_node_handler(node)
+ delete_edge_handler(edge)
hierarchy
+ hierarchy(observable_graph&)
+ new_node(InputIterator, InputIterator) : node
+ get_root() : node
+ son_edges_begin() : son_edge_iterator
+ son_edges_end() : son_edge_iterator
+ get_father_edge(node) : edge
+ induced_edge(node, node) : int
+ is_predecessor_of(node, node) : bool
+ is_inner_node(node) : bool
+ is_cross_edge(edge) : bool
view
+ view(hierarchy&, InputIterator, InputIterator)
+ expand(node)
+ collapse(node)

Fig. 4. Core Architecture

2. *Arbitrarily many hierarchies per graph:* A hierarchy always describes one dimension of abstraction, e. g., grouping locations in a road map by their geographical proximity. Sometimes it may be necessary or convenient to use more than one dimension of abstraction for the same graph, i. e., more than one hierarchy. Consider for instance the web-graph, which can be grouped either by domain or by topic.
3. *Arbitrarily many views per hierarchy:* A view defines an abstract version of a graph in terms of an associated abstraction hierarchy (cf. Definition 2). Clearly, it is very convenient to have more than one view, especially in a viewer or editor, where the user can see one abstract overview and work in another more detailed view.
4. *Minimize redundant information:* By Definition 1 the graph is part of each associated hierarchy and each view is part of a hierarchy (cf. Definition 2). Since all objects are subject to change it is very important to minimize redundancy.
5. *Reusable algorithms:* Hierarchies and views can be seen as graphs and thus many graph algorithms are applicable to them. Clearly, these algorithms should be implemented only once.

3.2 Graphs, Nodes and Edges

The class `graph` is the common base class for `hierarchy` and `view`. It consists of nodes and edges and the basic methods for adding and removing these objects. It also provides methods for traversing and accessing nodes and edges. This includes methods for traversing the adjacency of a node in particular. In other words, the graph has full control and the nodes and edges are only handles without state or functionality of their own. Examples for this model are the graphs in LEDA [15] and in BOOST GRAPH LIBRARY [2].

This is more appropriate here than the alternative, where nodes and edges are objects of their own controlling their adjacency lists themselves as in GTL [9], because then there can be at most one adjacency list for *each* node. On the other hand there can be at most one adjacency list *per graph and node* if the graph manages the lists. In other words nodes can be shared by several graphs, which is important to avoid redundancy.

3.3 The Observer Pattern

The partly abstract classes `observer` and `observable_graph` form the observer design pattern [8]. `observable_graph` extends `graph` with methods for adding and removing observers dynamically. Any object used as such an observer has to be derived from `observer` and thus has to implement the callback methods like `new_node_handler`. All the methods that modify the graph are redefined in `observable_graph` in order to trigger the appropriate callback method in all its observers with the modified object as argument. Notification of a new node or edge occurs *after* the change whereas deleting a node or edge is announced *in advance.*

3.4 Hierarchies

As shown in Fig. 4 a hierarchy is both `observer` and `observable_graph`. The `observer` part keeps track of modifications of the underlying graph whereas the `observable_graph` part informs the attached views about changes of either the graph or the hierarchy.

By definition a hierarchy cannot exist without an underlying graph. Therefore its constructor takes an `observable_graph` as argument. Initially a hierarchy consists of one root with all the nodes of the associated graph as sons and no cross edges.

A hierarchy is a graph and thus it can be modified using the methods already defined in `graph`. On the other hand, a hierarchy is a special graph, namely a tree with some cross-edges, and imprudent use of such a method could violate this invariant. Therefore these methods are redefined as follows:

– `new_node()`: Creates a new leaf, i. e., a new node in the underlying graph, and attaches it to the root.
– `new_edge(s,t)`: If both `s` and `t` are leaves then a new edge in the underlying graph is created. If at least one is an inner node and neither is a predecessor of the other a new cross-edge is inserted.
– `delete_node(n)`: If `n` is a leaf it is deleted in the underlying graph. If it is an inner node and is not the root all its sons are attached directly to its father before it is deleted.
– `delete_edge(e)`: Depending on whether `e` is a cross-edge or an edge between leaves it is deleted in the hierarchy or in the underlying graph, respectively.

For inserting a new *inner* node the method `new_node(it,end)` is provided. Its arguments specify a set of nodes as the range `[it,end)`, where `it` and `end` are iterators over some collection of nodes. It is required that all the nodes in this range have the same father in the hierarchy. The new inner node is inserted between these nodes and their father, i. e., the new node becomes the new father of these nodes and is inserted as son of the old father.

Apart from these explicit changes a hierarchy must be adapted whenever the underlying graph is modified. This is achieved by implementing the callback methods of the observer interface accordingly. A new node in the underlying graph becomes a leaf attached to the root node and deleting a node in the graph results in removing the corresponding leaf from the hierarchy. Adding or removing an edge from the graph does not result in a change in the hierarchy. However, all attached views are notified about changes of either the graph or the hierarchy.

The other new methods in `hierarchy` as shown in Fig. 4 are either for navigation in the hierarchy, e. g., `son_edges_begin` or `get_father_edge`, or provide information used by the views, e. g., `induced_edge` or `is_predecessor_of`.

3.5 Views

A view is both a `graph` and an `observer`. The `graph` part is the abstract version of the underlying graph in terms of the associated hierarchy. The `observer` part listens to changes of this hierarchy and updates the view accordingly.

At any time a view consists of a subset of the nodes of its hierarchy and thus the constructor of `view` takes two arguments: the hierarchy and the initial subset for this view. The subset is given as a range `[it,end)` in a collection of nodes, where `it` and `end` are iterators.

Since a view is a graph it can be modified through the standard methods defined in `graph`. On the other hand a view consists of nodes of the associated hierarchy and induced edges and thus all modifications must be forwarded to the hierarchy. Hence those methods are redefined in `view` in order to call the respective methods in `hierarchy`.

Whenever a view is notified of a change of the hierarchy (or the graph) it has to check whether it is affected. For instance, if a node in the hierarchy is deleted the view must be adapted if and only if this node was part of the view. In order to perform these updates efficiently the view makes use of the query methods provided in `hierarchy`, e. g., `induced_edge` or `is_predecessor_of`.

The method `view::expand` replaces a node in the view by its sons, and conversely `view::collapse` replaces all the sons of a node with the node itself. For both methods the view removes some nodes with all their incident edges and inserts one or more others. After inserting the new nodes the view uses query methods like `hierarchy::induced_edge` to determine their adjacency. Thus a view can not only be used to represent an abstract version of a graph but also to navigate through the hierarchy in either direction.

4 Conclusion

We have presented an architecture model for a library featuring graphs with an arbitrary number of hierarchies and views. Such a library shall help in handling large graphs in a convenient manner. It can be used in graph editors for drawing and exploring large graphs interactively.

Our model prepares the ground for data structures for the efficient implementation of hierarchies and views and algorithms taking advantage of the additional hierarchical structure.

References

1. Agd. `http://www.ads.tuwien.ac.at/AGD/index.html`.
2. Boost Graph Library. `http://www.boost.org/libs/graph/doc/`.
3. F. J. Brandenburg, M. Forster, A. Pick, M. Raitner, and F. Schreiber. Biopath. In *Proc. GD 2001*, LNCS 2265, pp. 451–456, 2001.
4. R. Brockenauer and S. Cornelsen. Drawing clusters and hierarchies. In *Drawing Graphs – Methods and Models*, LNCS 2025, pp. 193–227, 2001.
5. A. L. Buchsbaum, M. T. Goodrich, and J. R. Westbrook. Range searching over tree cross products. In *8th ESA*, 2000.
6. A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *11th ACM-SIAM Symposium on Discrete Algorithms*, 2000.
7. P. Eades and Q.-W. Feng. Multilevel visualization of clustered graphs. In *Proc. GD 1996*, LNCS 1190, pp. 101–112, 1996.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software.* AW Professional Computing Series, 1995.
9. Gtl. `http://www.infosun.fmi.uni-passau.de/GTL`.
10. D. Harel. On visual formalisms. *Comm. of the ACM*, 31(5):588–600, 1988.
11. Hgv. `http://www.infosun.fmi.uni-passau.de/~raitner/HGV/`.
12. M. Himsolt. *Konzeption und Implementierung von Grapheneditoren.* PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1993.
13. M. L. Huang and P. Eades. A fully animated interactive system for clustering and navigating huge graphs. In *Proc. GD 1998*, LNCS 1547, pp. 374–383, 1998.
14. W. Lai and P. Eades. A graph model which supports flexible layout functions. Technical Report 96–15, 1996.
15. Leda. `http://www.algorithmic-solutions.com/`.
16. T. Lengauer and E. Wanke. Efficient solution of connectivity problems on hierarchically defined graphs. *SIAM Journal on Computing*, 17(6):1063–1080, 1988.
17. I. A. Lisitsyn and V. N. Kasyanov. Higres - visualization system for clustered graphs and graph algorithms. In *Proc. GD 1999*, LNCS 1731, pp. 82–89, 1999.
18. K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Trans. Systems, Man and Cybernetics*, 21(4):876–892, 1991.