VCG
Visualization of Compiler Graphs

User Documentation V.1.30

Georg Sander
sander@cs.uni-sb.de
Universität des Saarlandes
66041 Saarbrücken
Germany

Feb. 9, 1995

This work is supported by the ESPRIT project 5399 COMPARE. We thank the COMPARE Consortium for the permission to distribute this software and this documentation freely. The members of the COMPARE Consortium are ACE Associated Computer Experts bv, GMD Forschungsstelle an der Universität Karlsruhe, Harlequin Limited, INRIA, STERIA, Stichting Mathematisch Centrum (CWI), and Universität des Saarlandes.

Contents

List of Tables

List of Figures

1 Introduction

Visualization allows better understanding of the intermediate representations (IRs) used in compilers. Many parts of the IR are trees or graphs, e.g., the syntax tree, the control flow graph, the call graph or the data dependence graph [WiMa92]. A simple textual visualization of trees and graphs is often too confusing or unreadable. A special visualization tool that shows trees and graphs in a natural way is more helpful. It allows powerful debugging of internals of the compiler and the examination of the effect of engines on the IR.

In the early phases of the project COMPARE, the **Edge** tool [PaTi90] was used for this purpose. This tool has the advantage to be easily adaptable by reading a graph specification from a file that can be used as interface format between engines (compiler phases) and the tool. Further, the tool can be used for postmortem debugging, which is important if the compiler graphs are such large that the online debugging would slow down the compiler to an unreasonable degree.

However, the **Edge** tool is very slow on typical IR graphs (e.g., visualization of a syntax tree of a CLaX program of 200 lines needed more than 20 minutes on a Sun Sparc ELC), and the layout is sometimes a little bit strange for the graphs used in compilers. Furthermore, it is only possible in a limited way to show condensations of graphs, that often help to understand algorithms in compiler construction. To overcome these deficiencies, a new visualization tool is implemented, the **VCG** tool. It combines the advantages of the **Edge** tool (easy adaptability) with reasonable speed (a few seconds for the same example as above) and new concepts of graph approximations. Therefore, we define a language that describes graphs and the layout of their nodes and edges (**GDL** - graph description language). The core of this language is compatible to the input specification language of the **Edge** tool, to allow interchangeability. Additionally, some extensions are implemented (colors, edge classes and priorities, splines, graph folding for path approximations), but also some layout description limits are introduced to speed up runtime. If no layout is given in the graph specification, the tool computes a appropriate one. With respect to 'readability' of the graph the following criteria are used:

1. Place the vertices (nodes) in a hierarchy of layers
2. Place the nodes without overlapping
3. Avoid crossings of lines (edges)
4. Keep edges short and straight
5. Favor a balanced placement
6. Position related nodes close together

In the following sections, we first give an overview of the phases during the calculation of the layout. Next, we define the graph description language and show some examples. Then, we explain the usage of the **VCG** tool. The remaining sections describe some experiences, and statistics concerning speed and applicability. The details of the algorithms used in the **VCG** tool are not described here. There exists a technical report that explains these algorithms (see [Sa94] [Sa95]).

2 Overview of the Layout Phases

The task of the **VCG** tool is to parse a graph specification, to assign horizontal and vertical positions to each node, if necessary, and to find polygon segments or splines for the edges such that they do not overlap with nodes, and finally, to draw the resulting picture in a window. The specification that is given as input to the **VCG** tool is a readable ASCII text. The output window can be used to browse through the graph, to shrink or enlarge the graph, to fold parts of the graph and to export a bitmap of the graph or a PostScript file. Graph folding results potentially in a relayout of the graph. The layout of the graph can be influenced in a wide range by attributes of edges, nodes and graphs, or by different variations of the layout algorithm. Because graph layout and drawing is a rather complex process, the tool gives messages in form of a single character to indicate its state.

2.1 Parsing

The first phase of the tool is to parse the specification and to construct internal data structures representing the graph. The specification may contain attributes denoting initially folded parts of the graph. This phase is indicated by the message character ‘a’.

2.2 Folding

This phase is executed as start of each relayout, i.e. after the start of the tool, or whenever a folding operation was selected by the user. It is indicated by the message character ‘f’. Folding of a graph allows to inspect the graph in a more compact way: Unimportant parts are hidden while important parts are shown in detail. To fold parts of the graph also improves the performance of the tool because the folded parts need not to be laid out. Examples are: fold the procedure parts of a syntax tree, hide annotations of a graph (e.g., syntax tree attributes), display approximations of paths in a graph, show the condensation to strongly connected regions, etc. There are 3 general methods to fold the visualized graph:

- **folding of complete subgraphs:** The GDL specification allows to partition the graph statically into nested subgraphs. (Statically means: there is no way to change this partitioning interactively). See section 3.1, attribute `folding`. Subgraphs can be visualized explicitly, i.e all nodes are drawn, or in a compressed manner by displaying only one summary node for the whole graph.

- **hiding of edges:** The edges of the graph can be statically partitioned into classes. Edge classes are specified by numbers (1, 2, . . .). Every class can separately be hidden. In this case, all edges of this class are not laid out and not drawn. (Note that edges with the linestyle `invisible` are laid out, even if they are not drawn. See section 3.3.) Additionally, all nodes that are only reachable by edges currently hidden are not drawn, i.e. all nodes whose incoming and outgoing edges are hidden become invisible, too. See section 3.1, attribute `hidden`. However, nodes without any incoming or outgoing edge *are drawn* (but see also section 3.1, attribute `ignore_singles`). This method allows to hide regions of the graph that are only connected by edges of a certain class. See the

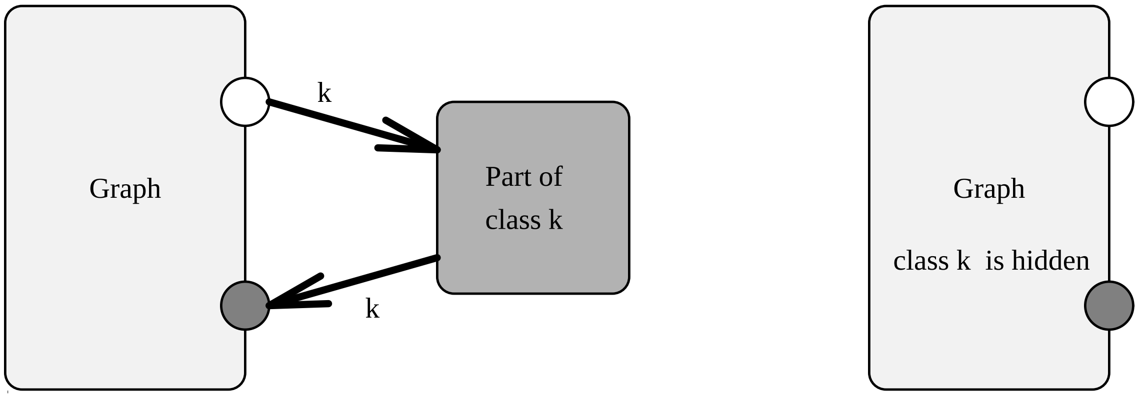


Figure 1: Hiding of Edges and their Region

The annotation (dark grey box) is a graph where all edges are in class k while the main graph is connected via class j ($j \neq k$). The bold edges of class k connect the annotation with the main graph, thus after hiding with respect to class k, the annotation is invisible. Other nodes (e.g., the grey node) in the main graph are unchanged even if there are edges from the invisible annotations to these nodes.

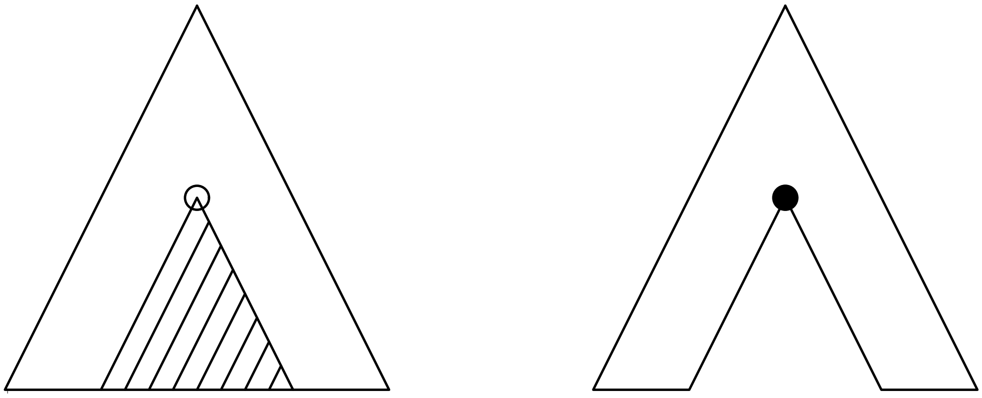


Figure 2: Folding a Subtree

The striped subtree is folded to the black summary node.

sketch in figure 1. Note that the hiding of edges may change the layout of a graph very much, if certain variations of the layout algorithms are selected (variation `maxdegree` ...`minoutdegree`).

- **folding of connected regions:** While subgraphs allow statically to specify regions that can be folded to one node, we can also use the class concept of edges to fold

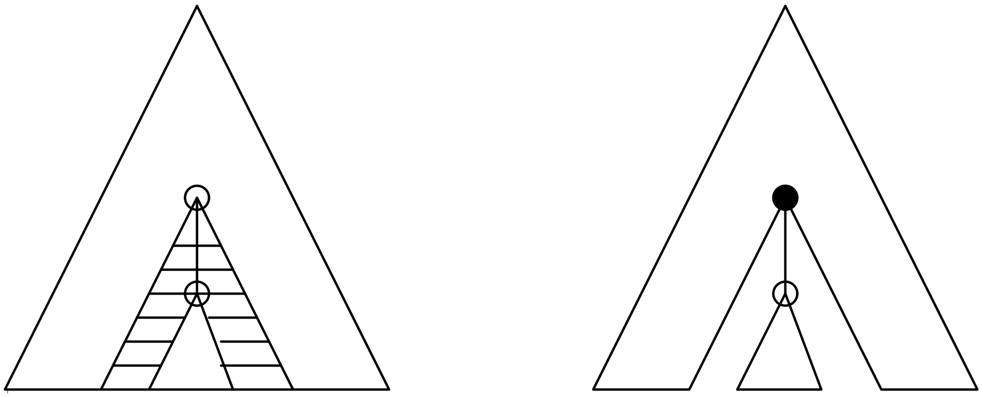


Figure 3: Folding a Subtree until a Node

The striped region is folded to the black summary node.

dynamically specified regions (dynamically = interactively specified). The connected region of a start node n with respect to an edge class k is the set of nodes which are reachable from n by edges of classes less or equal than k. It is possible to fold a connected region of n into one node. It is also possible to select a node m inside the region of n where the folding stops: in this case, the connected region of n without the connected region of m is folded, or, with other words, the folding of the region of n stops at the predecessors of m. This method can be used to visualize approximated path.

A simple example is a tree where all edges have the same class. Folding a node n is folding the whole subtree starting from n into one summary node (see figure 2). Folding n until m (where m is in the subtree of n) is folding the path from n to m and all subtrees along this path except the subtree that starts from m (see figure 3).

Note that nested foldings are possible. Folding regions or subgraphs may interfere with hiding of edges. In this case, first the summary node of the folded region or subgraph is calculated, and then the hiding of edges is performed.

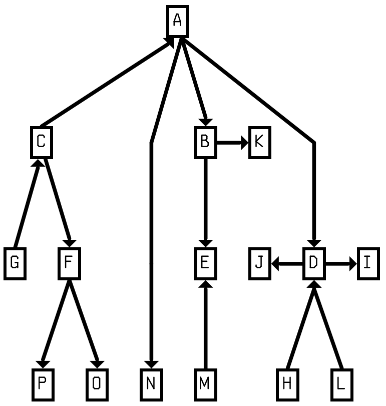
2.3 Assignment of Ranks

After folding, all visible nodes are determined. If all visible nodes are specified by the user with valid coordinates, the graph is drawn immediately. However, if the coordinates of at least one node is missing, an appropriate layout must be calculated. The first pass places the nodes into discrete ranks. All nodes of the same rank will appear at the same vertical position. The partitioning of the graph into levels of nodes of the same rank is indicated by the message character 'p'.

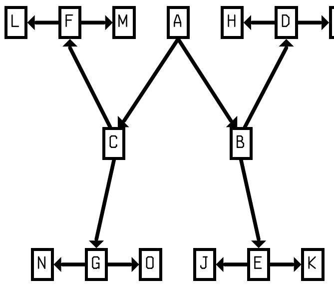
There are many possibilities to assign the rank. The normal method is to calculate a spanning tree by determing the strongly connected components of the graph. All edges should be oriented top down. A heuristics tries to find a minimal set of edges which cannot be oriented top down. This is necessary in cycles of the graph. A faster method is to calculate the spanning tree of a graph by depth first search (DFS). However, the order the nodes are visited in influences heavily the layout. The initial order of the nodes is the order given by the specification of the graph. Thus we have implemented various versions of such methods:

- `dfs`: Calculate the spanning tree by one single DFS traversal. This is the fastest method, but the quality of the result depends heavily on the initial order of the nodes in the specification, and might be poor for some graphs.
- `maxdepth`: Calculate the spanning tree by DFS with the initial order and with the reverted initial order, and take the deeper spanning tree. This results in more levels, i.e. the graph is larger in y-direction.
- `mindepth`: Take the flatter spanning tree of both DFS. This results in less levels, i.e. more nodes at same levels, and the graph is larger in x-direction.
- `maxdepthslow, mindepthslow`: While the previous algorithms are fast heuristics to increase or decrease the depth of the layout, these algorithms really calculate a good order to get a maximal or minimal spanning tree. The disadvantage is, that they are rather slow. Warning: a minimal spanning tree does not necessarily mean that the depth of the layout is minimal. However, it is a good hint to get a flat layout. See the examples in section 4.3.
- `maxdegree, mindegree,` etc.: We can also presort the nodes by different criteria before DFS such that the nodes are scheduled in a different order. Possible criterias are the number of incoming edges, the number of outgoing edges, and the number of edges at all on a node. The sorting of nodes may have various effects and can sometimes be used as a fast replacement of `maxdepthslow` or `mindepthslow`.
- `minbackward`: Instead of calculating strongly connected components, we can also perform topological sorting to assign ranks to the nodes. This is much faster, if the graph is already known to be acyclic.
- `tree`: This method works only, if the graph is a forest of *downward laid out trees*, i.e. each node at rank l has at most one adjacent edge coming from a node of an upper rank $k < l$ to it. A node may have edges pointing to nodes at the same level, and many adjacent edges coming from nodes of lower ranks $k > l$, and the direction of the edges can be arbitrary, but the picture of the layout (if the arrow heads are ignored) must be a tree (see figure 4). The assignment of ranks is done by DFS. Then, the graph is checked whether it is a forest of downward laid out trees. If this is not the case, the standard layout is used as fallback solution. As advantage of this method, crossing reduction (see next section) is not necessary for downward laid out trees, and a very fast positioning algorithm can be used.

A further possibility to influence the layout are the priorities of edges. During the calculation of the spanning tree, edges of higher priority are preferred. After the partitioning, a



Structurally, this is not a tree (e.g., many edges point to the node "D"). But the layout has the shape of a tree, thus it is a *downward laid out tree*.



Structurally, this is a tree. But the layout is *not a downward laid out tree* because of the edges at the nodes "B" and "C".

Figure 4: Downward Laid Out Trees and Structural Trees

fine tuning phase tries to improve the ranks in order to avoid very long edges. Remaining too long edges are split into small edges and dummy nodes.

2.4 Reduction of Crossings

The next pass calculates a good order of the nodes within levels to avoid edge crossings. This pass is not necessary, if the method for downward laid out trees is used. The first step is to unmerge connected components of the graph and to handle each component separately. The message character 'u' indicates this.

The crossing reduction algorithm calculates the weights of the nodes dependent on the possible crossings, and reorders the nodes of a level according to these weights. Because the ordering of nodes within one level influences the weights of the adjacent levels, this is performed iteratively until no improvement is anymore recognized. This is the phase 1 of the crossing reduction, indicated by the message character 'b'.

What happens, if the weights of some nodes are equal ? Then, the selected order of these nodes is arbitrary. In order to improve the layout further, a permutation of these nodes is tried. Sometimes, a permutation allows further to reduce the crossings. This is the phase 2 of the crossing reduction, indicated by the message character 'B'.

However, the final result need not to be optimal. The crossing reduction is only a heuristics. A local optimization phase follows (message character 'l').

There are four possibilities to calculate weights for crossing heuristics. The default weights are the `barycenter` weights [STM81], while the `mediancenter` weights [GNV88] are sometimes more appropriate, especially if the average degree (number of edges) at the nodes is small. The `barymedian` weights are the combination of `barycenter` and `mediancenter`, where `barycenter` has the first priority and `mediancenter` is only used for those nodes where the `barycenter` weights are equal. Conversely, the `medianbary` weights are the combination of `barycenter` and `mediancenter`, where `mediancenter` has the first priority.

2.5 Calculation of Coordinates

After partitioning of nodes into levels and ordering of the nodes within the levels, we can assign coordinates to the nodes. Here, the nodes can be aligned at the bottom or at the top of a level or centered at a level, and there is a minimal distance between the levels (`yspace`). This influences the y-coordinates. The x-coordinate must be calculated such that there is a minimum distance between the nodes (`xspace`) and a minimal distance between the bend points of edges (`xlspace`). Further, the layout must be balanced, such that the edges are short and straight.

To achieve this, either a special method for downward laid out trees is used (message character 'T'), or, two general iteration phases are performed: The first phase simulates a physical pendulum. The nodes are the balls and the edges are the strings. The balls hanging on the strings pendel, i.e. the nodes move inside their level and influence the neighbored nodes, until the layout is sparse enough such that each node has space to be placed on a good position. This phase is indicated by the message character 'm'.

Next, the nodes are centered with respect to their edges. This phase simulates a rubber-band network: The edges pull on a node with a power proportional to their length. As effect, the node moves to a position such that the sum of the forces of its edges is zero. Then, the length of the edges is balanced. The message character 'c' indicates this phase.

An optional fine tuning phase tries to recognize long edges and tries to position these edges by long line segments with gradient 90 degree. This is useful for the orthogonal layout methods. The message character 'S' indicates this phase.

Unfortunately, both physical simulations need not to be convergent, i.e. it may happen that they are iterated infinitely often without resulting in a stable layout. These cases are seldom. To prevent infinite execution, the number of iterations is artificially restricted. The running in such a 'timeout' situation is indicated by the message character 't'.

2.6 Bending of Edges

If a graph contains nodes with different sizes, it might happen that an edge starting at a very small node is drawn through a neighbored large node. To avoid this situation, we allow that edges are bend at certain points. Also, if an orthogonal layout method is selected, the edges are bend such that only orthogonal line segments exist. These bendings are calculated in an iterative phase indicated by the message character 'e'.

2.7 Drawing

Finally, the graph is drawn in a window, or it is exported into a file. Edges can be drawn as polygon segments or optionally as splines. The drawing of splines is very slow, thus it is indicated by the character 'd'.

The exporting into PostScript or into bitmap formats (PBM and PPM) is also possible and is indicated by further message characters.

3 Graph Description Language

The graph description language is very similar to GRL defined in [MaPa91]. A specification describes a graph that consists of subgraphs, nodes, and edges. Subgraphs are parts of a graph that may be folded to one node during visualization or may be drawn explicitly. These may have attributes that specify details of the graph's appearance on the screen like labels, colors, sizes etc. The following shows an overview of the format of a GDL description:

```
graph: {
        < general graph attributes >
        < list of nodes or subgraphs >
        < list of edges >
}
```

where nodes and edges are specified by

```
node: {
        title: < node title >
        < node attributes >
}
```

and

```
edge: {
        sourcename: < title of source node >
        targetname: < title of target node >
        < edge attributes >
}
```

There are some special kinds of edges:

back edges These edges are not laid out in the normal orientation, but are reverted. For instance, if the layout algorithm tries to give all normal edges a top down orientation, it tries to give the back edges a bottom up orientation. If a graph contains a cycle, not all edges can have the same orientation: Some edges must be reverted. In this case, the layout algorithm prefers back edges before selecting any other edge to be reverted.

near edges These special edges are laid out such that their source and target node are directly neighbored at the same level. Near edges are drawn as short horizontal lines which are not crossed by other edges or nodes. Invisible near edges can be used to group nodes at a level together. A node can have maximal two near edges, whose one is positioned to the left and the other is positioned to the right. (Other restrictions, originated by the use of *anchor points*, are explained later).

bent near edges These special edges consist of a horizontal part, a bend point and a vertical part. If an edge label is drawn, it is placed just at the bend point. Implemented is such an edge by the combination of a near edge and a normal edge.

Beside that, back edges, near edges and bent near edges are normal edges. Back edges are specified by

```
backedge: {
      sourcename: < title of source node >
      targetname: < title of target node >
      < edge attributes >
}
```

Near edges are specified by

```
nearedge: {
      sourcename: < title of source node >
      targetname: < title of target node >
      < edge attributes >
}
```

Bent near edges are specified by

```
bentnearedge: {
      sourcename: < title of source node >
      targetname: < title of target node >
      < edge attributes >
}
```

Attributes are specified in the form

```
< attribute keyword > : < attribute value >
```

The order of attributes is irrelevant. Most attributes are optional. It is possible to specify default values of all nodes or edges in the attribute section of a graph by

```
node.< attribute keyword > : < attribute value >
```

or

```
edge.< attribute keyword > : < attribute value >
```

These default attribute values are valid for all nodes and edges (even back edges, near edges and bent near edges) where the corresponding attribute is not set explicitly. Regions of nodes and edges can be folded (see section 2.2). As result, a summary node is displayed for all nodes of a region, and edges to this summary node are displayed for sets of edges to nodes of the region. It is possible to specify the attributes for such nodes and edges that are originated by a folding operation. This allows to give the folded regions a different appearance than the normal nodes and edges. The attributes for such summary nodes or replacement edges are specified by

`foldnode.`< *attribute keyword* > : < *attribute value* >

or

`foldedge.`< *attribute keyword* > : < *attribute value* >

The order of subgraphs, nodes, and edges may influence the final layout, because the first node is scheduled first. Strings must be enclosed in quote marks and may contain the normal C escapes (e.g., `\"`, `\n`, `\f`, ...). Integers are sequences of digits. Floating point numbers consist of a sequence of digits, followed by a dot '.' and a sequence of digits. C style comments (`/* */`) and C++ style comments (`//`) are allowed.

3.1 Graph Attributes

The graph information is delimited by the keywords `graph: {` and `}`. The complete list of attributes with their types and default values is shown in the tables 1 and 2.

title specifies the name (a string) associated with the graph. The default name of a subgraph is the name of the outer graph, and the name of the outmost graph is the name of the specification input file. The name of a graph is used to identify this graph, e.g., if we want to express that an edge points to a subgraph. Such edges point to the root of the graph, i.e. the first node of the graph or the root of the first subgraph in the graph, if the subgraph is visualized explicitly.

label the text displayed inside the node, when the graph is folded to a node. If no label is specified then the title of the graph will be used. Note that this text may contain control characters like `NEWLINE` that influences the size of the node. See section 3.6 for more details.

info1, info2, info3 combines additional text labels with a node or a folded graph. info1, info2, info3 can be selected from the menu interactively. The corresponding text labels can be shown by mouse clicks on nodes.

color specifies the background color for the outermost graph, or the color of the summary node for subgraphs. Colors are `black`, `blue`, `red`, `green`, `yellow`, `magenta`, `cyan`, `white`, `darkgrey`, `darkblue`, `darkred`, `darkgreen`, `darkyellow`, `darkmagenta`, `darkcyan`, `gold`, `lightgrey`, `lightblue`, `lightred`, `lightgreen`, `lightyellow`, `lightmagenta`, `lightcyan`, `lilac`, `turquoise`, `aquamarine`, `khaki`, `purple`, `yellowgreen`, `pink`, `orange` and `orchid`. If more than these default colors are needed, a color map with maximal 256 entries can be used. The first 32 entries correspond to the colors just listed. A color of the color map can selected by the color map index, an integer, for instance red has index 2, green has index 3, etc. See section 3.5 for more details.

textcolor is the color for the label text of summary nodes.

attribute name	*attribute type*	*default value*
title	string	*file name / name of outer graph*
label	string	*string of the title*
info1	string	*empty string*
info2	string	*empty string*
info3	string	*empty string*
color	black, white, red, ...	white *for background* white *or transparent for summary nodes*
textcolor	black, white, red, ...	black *for summary nodes,*
bordercolor	black, white, red, ...	*textcolor for summary nodes,*
width	int	*width of root screen* - 100 *width of the label for subgraphs*
height	int	*height of root screen* - 100 *height of the label for subgraphs*
borderwidth	int	2
x	int	0 / *unspecified for subgraphs*
y	int	0 / *unspecified for subgraphs*
folding	int	0
shrink	int	1
stretch	int	1
textmode	center, ...	center
shape	box, rhomb, ...	box
vertical_order	int	*unspecified for subgraphs*
horizontal_order	int	*unspecified for subgraphs*
xmax	int	*width of root screen* - 90
ymax	int	*height of root screen* - 90
xbase	int	5
ybase	int	5
xspace	int	20
xlspace	int	$\frac{1}{2}$*xspace if polygons are used* $\frac{4}{5}$*xspace if splines are used*
yspace	int	70
xraster	int	1
xlraster	int	1
yraster	int	1
hidden	int	*none*
classname	int : string	*1, 2, ...*
infoname	int : string	*1, 2, or 3*
colorentry	int : int triple	*default color map*

Table 1: Graph Attributes, Part 1

attribute name	*attribute type*	*default value*
layoutalgorithm	maxdepth, mindepth, minbackward, ...	*normal*
layout_downfactor	int	1
layout_upfactor	int	1
layout_nearfactor	int	1
layout_splinefactor	int	70
late_edge_labels	yes, no	no
display_edge_labels	yes, no	no
dirty_edge_labels	yes, no	no
finetuning	yes, no	yes
ignore_singles	yes, no	no
straight_phase	yes, no	no
priority_phase	yes, no	no
manhattan_edges	yes, no	no
smanhattan_edges	yes, no	no
near_edges	yes, no	yes
orientation	top_to_bottom, ...	top_to_bottom
node_alignment	top, bottom, center	center
port_sharing	yes, no	yes
arrow_mode	fixed, free	fixed
spreadlevel	int	1
treefactor	float	0.5
crossing_phase2	yes, no	yes
crossing_optimization	yes, no	yes
crossing_weight	bary, median barymedian, medianbary	bary
view	cfish, pfish, fcfish, fpfish	*normal view*
edges	yes, no	yes
nodes	yes, no	yes
splines	yes, no	no
bmax	int	100
cmax	int	*infinite*
cmin	int	0
pmax	int	100
pmin	int	0
rmax	int	100
rmin	int	0
smax	int	100

Table 2: Graph Attributes, Part 2

bordercolor is the color of the summary node's border. Default color is the *textcolor*.

width, height are width and height of the displayed part of the window of the outermost graph in pixels, or width and height of the summary node of inner subgraphs.

borderwidth specifies the thickness of the summary node's border in pixels.

x, y are the x-position and y-position of the graph's window in pixels, relatively to the root screen, if it is the outermost graph. The origin of the window is upper, left hand. For inner subgraphs, it is the position of the folded summary node. The position can also be specified in the form `loc: {x:`*int* `y:`*int*`}`.

folding of a subgraph is 1, if the subgraph is fused, and 0, if the subgraph is visualized explicitly. There are commands to unfold such summary nodes, see section 5.

shrink, stretch gives the shrinking and stretching factor for the graph's representation (default is 1, 1). $((stretch/shrink) * 100)$ is the scaling of the graph in percentage, e.g., (`stretch`,`shrink`) = (1,1) or (2,2) or (3,3) ... is normal size, (`stretch`,`shrink`) = (1,2) is half size, (`stretch`,`shrink`) = (2,1) is double size. For subgraphs, it is also the scaling factor of the summary node. The scaling factor can also be specified by `scaling:` *float* (here, scaling 1.0 means normal size).

textmode specifies the adjustment of the text within the border of a summary node. The possibilities are `center`, `left_justify` and `right_justify`.

shape can be specified for subgraphs only. It is the shape of the subgraph summary node that appears if the subgraph is folded: `box`, `rhomb`, `ellipse`, and `triangle`.

vertical_order is the level position (rank) of the summary node of an inner subgraph, if this subgraph is folded. We can also specify `level:` *int*. The level is only recognized, if an automatical layout is calculated. See sections 2.3 and 3.6 for more details.

horizontal_order is the horizontal position of the summary node within a level. The nodes which are specified with horizontal positions are ordered according to these positions within the levels. The nodes which do not have this attribute are inserted into this ordering by the crossing reduction mechanism (see section 2.4). Note that connected components are handled separately, thus it is not possible to intermix such components by specifying a horizontal order.
If the algorithm for downward laid out trees is used, the horizontal order influences only the order of the child nodes at a node, but not the order of the whole level.

xmax, ymax specify the maximal size of the virtual window that is used to display the graph (see figure 5). This is usually larger than the displayed part, thus the width and height of the displayed part cannot be greater than `xmax` and `ymax`. Only those parts of the graph are drawn that are inside the virtual window. The virtual window can be moved over the potential infinite system of coordinates by special positioning commands.

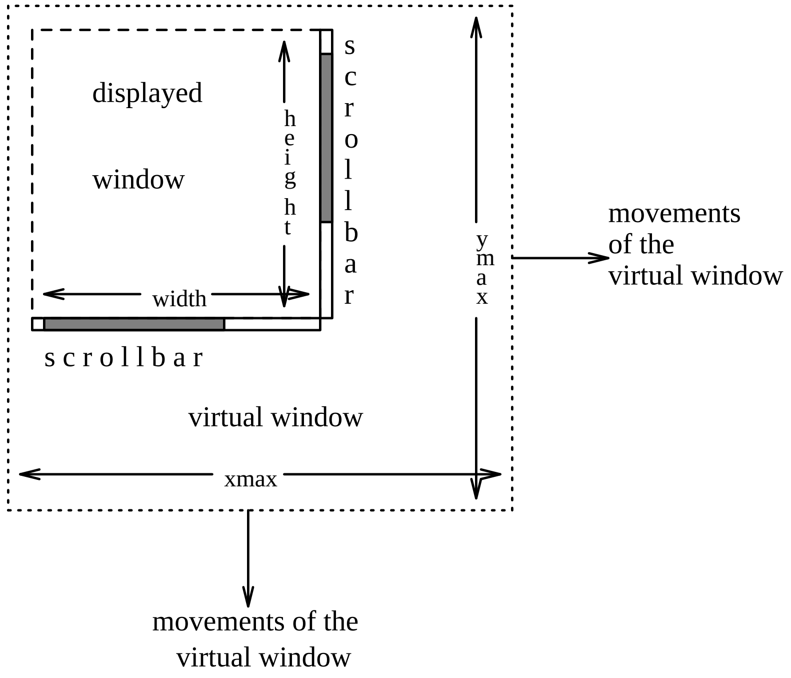


Figure 5: Displayed Window and Virtual Window

Therefore the graph may be larger than the virtual window. It is recommended to set `xmax`, `ymax` not larger than the root screen to get a good performance.

xbase, ybase specify the horizontal and vertical offset between the graph's window and the upper, left hand corner of the graph, i.e. the position of the origin of the system of coordinates relatively to the upper, left hand corner of the virtual window.

xspace, yspace the minimum horizontal and vertical distance between nodes.

xlspace is the horizontal distance between lines at the points where they cross the levels. (At these points, dummy nodes are used. In fact, this is the horizontal distance between dummy nodes.) It is recommended to set `xlspace` to a larger value, if splines are used to draw edges, to prevent sharp bendings.

xraster, yraster specifies the raster distance for the position of the nodes. The center of a node is aligned to this raster.

xlraster is the horizontal raster for the positions of the line control points (the dummy nodes). It should be a divisor of `xraster`.

hidden specifies the classes of edges that are hidden. See section 2.2 and section 3.2. Edges that are within such a class are not laid out nor drawn. Nodes that are only reachable (forward or backward) by edges of an hidden class are not drawn. However, nodes that are not reachable at all *are drawn.* (But see attribute `ignore_singles`.) Specification of classes of hidden edges allows to hide parts of a graph, e.g., annotations of a syntax tree. This attribute is only allowed at the outermost level. More than one settings are possible to specify exactly the set of classes that are hidden. Note the important difference between hiding of edges and the edge line style `invisible` (see section 3.3). Hidden edges are not existent in the layout. Edges with line style `invisible` are existent in the layout; they need space and may produce crossings and influence the layout, but you cannot see them.

classname allows to introduce names for the edge classes. The names are used in the menus.

infoname allows to introduce names for the additional text labels. The names are used in the menus.

colorentry allows to fill the color map. A color is a triplet of integer values for the red/green/blue-part. Each integer is between 0 (off) and 255 (on), e.g., 0 0 0 is black and 255 255 255 is white. For instance `colorentry 75 : 70 130 180` sets the map entry 75 to steel blue. This color can be used by specifying just the number 75. See section 3.5 for more details.

layoutalgorithm chooses different graph layout algorithms Possibilities are `maxdepth`, `mindepth`, `maxdepthslow`, `mindepthslow`, `maxdegree`, `mindegree`, `maxindegree`, `minindegree`, `maxoutdegree`, `minoutdegree`, `minbackward`, `dfs` and `tree`. The default algorithm tries to give all edges the same orientation and is based on the calculation of strongly connected components. The algorithms that are based on depth first search are faster. While the simple `dfs` does not enforce additionally constraints, the algorithm `maxdepth` tries to increase the depth of the layout and the algorithm `mindepth` tries to increase the wide of the layout. These algorithms are fast heuristics. If they are not appropriate, the algorithms `maxdepthslow` or `mindepthslow` also increase the depth or wide, but they are very slow.

The algorithm `maxindegree` lays out the nodes by scheduling the nodes with the maximum of incoming edges first, and `minindegree` lays out the nodes by scheduling the nodes with the minimum of incoming edges first. In the same manner work the algorithms `maxoutdegree` and `minoutdegree` for outgoing edges, and `maxdegree` and `mindegree` for the sum of incoming and outgoing edges. These algorithms may have various effects, and can sometimes be used as replacements of `maxdepthslow` or `mindepthslow`.

The algorithm `minbackward` can be used if the graph is acyclic. See section 2.3 for details.

The algorithm `tree` is a specialized method for downward laid out trees (see section 2.3). It is much faster on such tree-like graphs and results in a balanced layout.

layout_downfactor, layout_upfactor, layout_nearfactor The layout algorithm partitions the set of edges into edges pointing upward, edges pointing downward, and edges pointing sidewards. The last type of edges is also called near edges.

If the `layout_downfactor` is large compared to the `layout_upfactor` and the `layout_nearfactor`, then the positions of the nodes is mainly determined by the edges pointing downwards. If the `layout_upfactor` is large compared to the `layout_downfactor` and the `layout_nearfactor`, then the positions of the nodes is mainly determined by the edges pointing upwards. If the `layout_nearfactor` is large, then the positions of the nodes is mainly determined by the edges pointing sidewards. These attributes have no effect, if the method for downward laid out trees is used.

late_edge_labels `yes` means that the graph is first partitioned and then, labels are introduced. The default algorithm first creates labels and then partitions the graph (see section 2.3), which yield a more compact layout, but may have more crossings.

display_edge_labels `yes` means display labels and `no` means don't display edge labels.

dirty_edge_labels `yes` enforces a fast layout of edge labels, which may very ugly because several labels may be drawn at the same place. Dirty edge labels cannot be used if splines are used.

finetuning `no` switches the fine tuning phase of the graph layout algorithm off, while it is `on` as default (see section 2.3). The fine tuning phase tries to give all edges the same length.

ignore_singles `yes` hides all nodes which would appear single and unconnected from the remaining graph. Such nodes have no edge at all and are sometimes very ugly. Default is to show all nodes.

straight_phase `yes` initiates an additional phase that tries to avoid bendings in long edges. Long edges are laid out by long straight vertical lines with gradient 90 degree. Thus, this phase is not very appropriate for normal layout, but it is recommended, if an orthogonal layout is selected (see `manhattan_edges`).

priority_phase `yes` replaces the normal pendulum method by a specialized method: It forces straight long edges with 90 degree, just as the straight phase. In fact, the straight phase is a fine tune phase of the priority method. This phase is also recommended, if an orthogonal layout is selected (see `manhattan_edges`).

manhattan_edges `yes` switches the orthogonal layout on. Orthogonal layout (or manhattan layout) means that all edges consist of line segments with gradient 0 or 90 degree. Vertical edge segments might by shared by several edges, while horizontal edge segments are never shared. This results in very aesthetical layouts just for flowcharts. If the orthogonal layout is used, then the priority phase and straight phase should be used. Thus, these both phases are switched on, too, unless priority layout and straight line tuning are switched off explicitly.

smanhattan_edges `yes` switches a specialized orthogonal layout on: Here, all horizontal edge segments between two levels share the same horizontal line, i.e. not only vertical edge segments are shared, but horizontal edge segments are shared by several edges, too.

This looks nice for trees but might be too confusing in general, because the location of an edge might be ambiguously.

near_edges `no` suppresses near edges and bent near edges in the graph layout.

orientation specifies the orientation of the graph: `top_to_bottom`, `bottom_to_top`, `left_to_right` or `right_to_left`. Note: the normal orientation is `top_to_bottom`. All explanations here are given relatively to the normal orientation, i.e., e.g., if the orientation is left to right, the attribute `xlspace` is not the horizontal but the vertical distance between lines, etc.

node_alignment specified the vertical alignment of nodes at the horizontal reference line of the levels. If `top` is specified, the tops of all nodes of a level have the same y-coordinate; on `bottom`, the bottoms have the same y-coordinate, on `center` the nodes are centered at the levels.

port_sharing `no` suppresses the sharing of ports of edges at the nodes. Normally, if multiple edges are adjacent to the same node, and the arrow head of all these edges has the same visual appearance (color, size, etc.), then these edges may share a port at a node, i.e. only one arrow head is draw, and all edges are incoming into this arrow head. This allows to have many edges adjacent to one node without getting confused by too many arrow heads. If no port sharing is used, each edge has its own port, i.e. its own place where it is adjacent to the node.

arrow_mode `fixed` (default) should be used, if port sharing is used, because then, only a fixed set of rotations for the arrow heads are used. If the arrow mode is `free`, then each arrow head is rotated individually to each edge. But this can yield to a black spot, where nothing is recognizable, if port sharing is used, since all these differently rotated arrow heads are drawn at the same place. If the arrow mode is `fixed`, then the arrow head is rotated only in steps of 45 degree, and only one arrow head occurs at each port.

treefactor The algorithm `tree` for downward laid out trees tries to produce a medium dense, balanced tree-like layout. If the tree factor is greater than 0.5, the tree edges are spread, i.e. they get a larger gradient. This may improve the readability of the tree.
Note: it is not obvious whether spreading results in a more dense or wide layout. For a tree, there is a tree factor such that the whole tree is minimal wide.

spreadlevel This parameter only influences the algorithm `tree`, too. For large, balanced trees, spreading of the uppermost nodes would enlarge the width of the tree too much, such that the tree does not fit anymore in a window. Thus, the spreadlevel specifies the minimal level (rank) where nodes are spread. Nodes of levels upper than spreadlevel are not spread.

crossing_weight specifies the weight that is used for the crossing reduction: `bary` (default), `median`, `barymedian` or `medianbary`. See section 2.4. We cannot give a general recommendation, which is the best method. For graphs with very large average degree of edges (number of incoming and outgoing edges at a node), the weight `bary` is the fastest method. With the weights `barymedian` and `medianbary`, equal weights of different nodes are not very probable, thus the crossing reduction phase 2 might be very fast.

crossing_phase2 is the most time consuming phase of the crossing reduction (see section 2.4). In this phase, the nodes that happen to have equal crossing weights are permuted. By specifying `no`, this phase is suppressed.

crossing_optimization is a postprocessing phase after the normal crossing reduction: we try to optimize locally, by exchanging pairs of nodes to reduce the crossings. Although this phase is not very time consuming, it can be suppressed by specifying `no`.

view allows to select the fisheye views (see section 5.8). Because of the fixed size of the window that shows the graph, we normally can only see a small amount of a large graph. If we shrink the graph such that it fits into the window, we cannot recognize any detail anymore. Fisheye views are coordinate transformations: the view onto the graph is distort, to overcome this usage deficiency. The polar fisheye is easy to explain: assume a projection of the plane that contains the graph picture onto a spheric ball. If we now look onto this ball in 3 D, we have a polar fisheye view. There is a focus point which is magnified such that we see all details. Parts of the plane that are far away from the focus point are demagnified very much. Cartesian fisheye have a similar effect; only the formula for the coordinate transformation is different. Selecting `cfish` means the cartesian fisheye is used which demagnifies such that the whole graph is visible (self adaptable cartesian fisheye). With `fcfish`, the cartesian fisheye shows the region of a fixed radius around the focus point (fixed radius cartesian fisheye). This region might be smaller than the whole graph, but the demagnification needed to show this region in the window is also not so large, thus more details are recognizable. With `pfish` the self adaptable polar fisheye is selected that shows the whole graph, and with `fpfish` the fixed radius polar fisheye is selected.

edges `no` suppresses the drawing of edges.

nodes `no` suppresses the drawing of nodes.

splines specifies whether splines are used to draw edges (`yes` or `no`). As default, polygon segments are used to draw edges, because this is much faster. Note that the spline drawing routine is not fully validated, and is very slow. Its use is mainly to prepare high quality PostScript output for very small graphs.

layout_splinefactor determines the bending at splines. The factor 100 indicates a very sharp bending, a factor 1 indicates a very flat bending. Useful values are 30 . . . 80.

cmin set the minimal number of iterations that are done for the crossing reduction with the crossing weights. The normal method stops if two consecutive checks does not reduce the number of crossings anymore. However, this increasing of the number of crossings might be locally, such that after some more iterations, the crossing number might decrease much more.

cmax set the maximal number of interactions for crossing reduction. This is helpful for speedup the layout process. See section 5.13.

pmin set the minimal number of iterations that is done with the pendulum method. Similar to the crossing reduction, this method stops if the 'imbalancement weight' does not decreases anymore. However, the increasing of the imbalancement weight might be locally, such that after some more iterations, the imbalancement weight might decrease much more.

pmax set the maximal number of iterations of the pendulum method. This is helpful for speedup the layout process.

rmin set the minimal number of iterations that is done with the rubberband method. This is similar as for the pendulum method.

rmax set the maximal number of iterations of the rubberband method. This is helpful for speedup the layout process.

smax set the maximal number of iterations of the straight line recognition phase (useful only, if the straight line recognition phase is switched on, see attribute `straight_phase`).

bmax set the maximal number of iterations that are done for the reduction of edge bendings.

Note: the attributes xmax, ymax, xbase, ybase, xspace, yspace, xlspace, xraster, yraster, xlraster, hidden, classname, infoname, colorentry, layoutalgorithm, layout_downfactor, layout_upfactor, layout_nearfactor, late_edge_labels, display_edge_labels, dirty_edge_labels, finetuning, ignore_singles, straight_phase, priority_phase, manhattan_edges, smanhattan_edges, near_edges, orientation, node_alignment, port_sharing, arrow_mode, treefactor, spreadlevel, crossing_weight, crossing_phase2, crossing_optimization, view, edges, nodes, splines, layout_splinefactor, cmin, cmax, pmin, pmax, rmin, rmax, and smax can only be specified for the outermost graph. They influence the whole layout of all subgraphs, or change the general usage mode of the **VCG** tool.

3.2 Node Attributes

Node specifications occur as parts of graph specifications. The node information is delimited by the keywords `node: {` and `}`. At least, every node has to contain a title, other attributes are optional. It is possible to specify default attribute values of nodes for every subgraph by `node.`< *attribute keyword* > : < *attribute value* > . The complete list of attributes with their types and default values is shown in table 3.

title the unique string identifying the node. This attribute is mandatory.

label the text displayed inside the node. If no label is specified then the title of the node will be used. Note that this text may contain control characters like `NEWLINE` that influences the size of the node.

attribute name	*attribute type*	*default value*
title	string	*mandatory*
label	string	*string of the title*
loc x	int	*none*
loc y	int	*none*
vertical_order	int	*unspecified*
horizontal_order	int	*unspecified*
width	int	*width of the label*
height	int	*height of the label*
shrink	int	1
stretch	int	1
folding	int	*none*
shape	box, rhomb, ...	box
textmode	center, left_justify, right_justify	center
borderwidth	int	2
color	black, white, red, ...	white or *transparent*
textcolor	black, white, red, ...	black
bordercolor	black, white, red, ...	*textcolor*
info1	string	*empty string*
info2	string	*empty string*
info3	string	*empty string*

Table 3: Node Attributes

loc is the location as x, y position relatively to the system of coordinates of the graph. Locations are specified in the form `loc: { x:` *xpos* `y:` *ypos* `}`. The locations of nodes are only valid, if the whole graph is fully specified with locations and no part is folded. The layout algorithm of the tool calculates appropriate x, y positions, if at least one node that must be drawn (i.e., is not hidden by folding or edge classes) does not have fixed specified locations.

vertical_order is the level position (rank) of the node. We can also specify `level:` *int*. Level specifications are only valid, if the layout is calculated, i.e. if at least one node does not have a fixed location specification. The layout algorithm partitioned all nodes into levels 0 ... *maxlevel*. Nodes at the level 0 are on the upper corner. The algorithm is able to calculate appropriate levels for the nodes automatically, if no fixed levels are given (see sections 2.3). Specifications of levels are additional constraints, that may be ignored, if they are in conflict with *near edge* specifications. See section 3.6 for more details.

horizontal_order is the horizontal position of the node within a level. The nodes which are specified with horizontal positions are ordered according to these positions within the levels. The nodes which do not have this attribute are inserted into this ordering by the crossing reduction mechanism (see section 2.4). Note that connected components are handled separately, thus it is not possible to intermix such components by specifying a

horizontal order.
If the algorithm for downward laid out trees is used, the horizontal order influences only the order of the child nodes at a node, but not the order of the whole level.

width, height is the width and height of a node including the border. If no value (in pixels) is given then width and height are calculated from the size of the label.

shrink, stretch gives the shrinking and stretching factor of the node. The values of the attributes `width`, `height`, `borderwidth` and the size of the label text is scaled by $((stretch/shrink) * 100)$ percent. Note that the actual scale value is determined by the scale value of a node relatively to a scale value of the graph, i.e. if (`stretch`,`shrink`) = (2,1) for the graph and (`stretch`,`shrink`) = (2,1) for the node of the graph, then the node is scaled by the factor 4 compared to the normal size. The scale value can also be specified by `scaling:` *float*.

folding specifies the default folding of the nodes. The folding k (with $k > 0$) means that the graph part that is reachable via edges of a class less or equal to k is folded and displayed as one node. There are commands to unfold such summary nodes, see section 5. If no folding is specified for a node, then the node may be folded if it is in the region of another node that starts the folding. If folding 0 is specified, then the node is never folded. In this case the folding stops at the predecessors of this node, if it is reachable from another folding node. The summary node inherits some attributes from the original node which starts the folding (all color attributes, textmode and label, but not the location). A folded region may contain folded regions with smaller folding class values (nested foldings). If there is more than one node that start the folding of the same region (this implies that the folding class values are equal) then the attributes are inherited by one of these nodes nondeterministically (see section 2.2). If *foldnode* attributes are specified, then the summary node attributes are inherited from these attributes.

shape specifies the visual appearance of a node: `box`, `rhomb`, `ellipse`, and `triangle`. The drawing of ellipses is much slower than the drawing of the other shapes.

textmode specifies the adjustment of the text within the border of a node. The possibilities are `center`, `left_justify` and `right_justify`.

borderwidth specifies the thickness of the node's border in pixels.

color is the background color of the node. If none is given, the node is white. For the possibilities, see the attribute `color` for graphs (section 3.1).

textcolor is the color for the label text.

bordercolor is the color of the border. Default color is the *textcolor*.

info1, info2, info3 combines additional text labels with a node or a folded graph. info1, info2, info3 can be selected from the menu. The corresponding text labels can be shown by mouse clicks on nodes.

3.3 Edge Attributes

Edge specifications also occur as parts of graph specifications. The edge information is delimited by the keywords `edge: {` and `}`. The attributes `sourcename` and `targetname` are mandatory. They specify the source and target node of the edge. It is possible to specify default attribute values of edges for every subgraph by `edge.`< *attribute keyword* > : < *attribute value* > . The position of the edge is determined by the position of its nodes. Thus, there is no way to specify (x, y) positions of the edge. The complete list of attributes with their types and default values is shown in table 4.

attribute name	*attribute type*	*default value*
sourcename	string	*mandatory*
targetname	string	*mandatory*
label	string	*no label*
linestyle	continuous, dashed, dotted, invisible	continuous
thickness	int	2
class	int	1
color	black, white, red, ...	black
textcolor	black, white, red, ...	*color*
arrowcolor	black, white, red, ...	*color*
backarrowcolor	black, white, red, ...	*color*
arrowsize	int	10
backarrowsize	int	0
arrowsstyle	solid, line, none	solid
backarrowsstyle	solid, line, none	none
priority	int	1
anchor	int	*none*
horizontal_order	int	*unspecified*

Table 4: Edge Attributes

sourcename is the title of the source node of the edge.

targetname is the title of the target node of the edge.

label specifies the label of the edge. It is drawn if `display_edge_labels` is set to `yes`.

linestyle specifies the style the edge is drawn. Possibilities are:

- **continuous** a solid line is drawn (—)
- **dashed** the edge consists of single dashes (- - -)
- **dotted** the edge is made of single dots ($\cdots$)
- **invisible** the edge is not drawn. The attributes of its shape (color, thickness) are ignored.

To draw a dashed or dotted line needs more time than solid lines.

thickness is the thickness of an edge.

class specifies the folding class of the edge. Nodes reachable by edges of a class less or equal to a constant k specify folding regions of k. See the node attribute `folding` (section 3.2) and the folding commands (section 5).

arrowstyle, backarrowstyle Each edge has two arrow heads: the one appears at the target node (the normal arrow head), the other appears at the source node (the backarrow head). Normal edges only have the normal `solid` arrow head, while the backarrow head is not drawn, i.e. it is `none`. Arrowstyle is the style of the normal arrow head, and backarrowstyle is the style of the backarrow head. Styles are `none`, i.e. no arrow head, `solid`, and `line`.

arrowsize, backarrowsize The arrow head is a right-angled, isosceles triangle and the cathetuses have length arrowsize.

color is the color of the edge. For the possibilities, see the attribute `color` for graphs (section 3.1)

textcolor is the color of the label of the edge.

arrowcolor, backarrowcolor is the color of the arrow head and of the backarrow head.

priority The positions of the nodes are mainly determined by the incoming and outgoing edges. One can think of rubberbands instead of edges that pull a node into its position. The priority of an edges corresponds to the strength of the rubberband.

anchor An anchor point describes the vertical position in a node where an edge goes out. This is useful, if node labels are several lines long, and outgoing edges are related to label lines. (E.g., this allows a nice visualization of structs containing pointers as fields.)

horizontal_order is the horizontal position the edge. This is of interest only if the edge crosses several levels because it specifies the point where the edge crosses the level. within a level. The nodes which are specified with horizontal positions are ordered according to these positions within a level. The horizontal position of a long edge that crosses the level specifies between which two node of that level the edge has to be drawn. Other edges which do not have this attribute are inserted into this ordering by the crossing reduction mechanism (see section 2.4). Note that connected components are handled separately, thus it is not possible to intermix such components by specifying a horizontal order.

3.4 Grammar of GDL

Now we give the grammar of GDL in EBNF (Extended Bacchus Naur Form). Terminals are enclosed in “double quotes”, nonterminals are written *italic*, finite iteration is specified by (...)*. Note that C style comments (`/* */`) and C++ style comments (`//`) are allowed.

graph	:	"**graph:** {" (*graph_entry*)* "}"
graph_entry	:	*graph_attribute*
	|	*node_defaults*
	|	*edge_defaults*
	|	*foldnode_defaults*
	|	*foldedge_defaults*
	|	*graph*
	|	*node*
	|	*edge*
	|	*backedge*
	|	*nearedge*
	|	*bentnearedge*
graph_attribute	:	*graph_attribute_name* ":" *attribute_value*
graph_attribute_name	:	any attribute shown in table 1 and 2
node_defaults	:	"**node.**" *node_attribute*
edge_defaults	:	"**edge.**" *edge_attribute*
foldnode_defaults	:	"**foldnode.**" *node_attribute*
foldedge_defaults	:	"**foldedge.**" *edge_attribute*
node	:	"**node:** {" *(node_attribute)** "}"
edge	:	"**edge:** {" *(edge_attribute)** "}"
backedge	:	"**backedge:** {" *(edge_attribute)** "}"
nearedge	:	"**nearedge:** {" *(edge_attribute)** "}"
bentnearedge	:	"**bentnearedge:** {" *(edge_attribute)** "}"
node_attribute	:	*node_attribute_name* "**:**" *attribute_value*
edge_attribute	:	*edge_attribute_name* "**:**" *attribute_value*
node_attribute_name	:	any attribute shown in table 3
edge_attribute_name	:	any attribute shown in table 4
attribute_value	:	*integer_value*
	|	*float_value*
	|	*string_value*
	|	*enum_value*
integer_value	:	any integer constant in **C** style
float_value	:	any float constant in **C** style
string_value	:	""" *(character)** """
enum_value	:	any possible constant value shown in tables 1 , 2, 3 , 4
character	:	any printable **ASCII** character

3.5 Colors

The **VCG** tool has a color map of 256 colors, where 254 of these are free available. The first 32 colors (index 0 – 31) of the color map are the default colors. These colors can be specified by name, all other colors are specified by their color map index number. The color map is changed by specifying a sequence of colorentry attributes, for instance

```
colorentry 32 : 205 198 115  /* khaki      */
colorentry 33 : 210 218 255  /* AliceBlue  */
colorentry 34 : 205 92 92    /* indian red */
```

introduce the colors 'khaki', 'AliceBlue' and 'indian red' with the color index 32, 33 and 34. If we want to use blue, which is a default color, we can specify `color: blue` or `color: 1`. If we want to use khaki as a color of a node, we cannot specify `color: khaki` since this name 'khaki' is unknown for the **VCG** tool. Instead, we specify `color: 32`.
More tricky, we can even overwrite the default colors. If we specify

```
colorentry 1 : 205 198 115  /* khaki      */
colorentry 2 : 210 218 255  /* AliceBlue  */
colorentry 3 : 205 92 92    /* indian red */
```

then the default colors blue, red and green are overwritten by khaki, AliceBlue and indian red. If we now specify `color: blue`, then the color khaki will appear. Table 5 shows the default color map.

white	00	blue	01	red	02	green	03
yellow	04	magenta	05	cyan	06	darkgrey	07
darkblue	08	darkred	09	darkgreen	10	darkyellow	11
darkmagenta	12	darkcyan	13	gold	14	lightgrey	15
lightblue	16	lightred	17	lightgreen	18	lightyellow	19
lightmagenta	20	lightcyan	21	lilac	22	turquoise	23
aquamarine	24	khaki	25	purple	26	yellowgreen	27
pink	28	orange	29	orchid	30	black	31

Table 5: Color Codes of the Default Color Map

3.6 Further Remarks

A few important restrictions should be considered. All titles of graphs and nodes must be unique. In order to decide which are the source and the target node of an edge, this restriction is very important.

A node can only be touched by 2 near edges. If more than 2 near edges are specified to touch the node, the remaining near edges are converted into normal edges. A node that has anchored edges can have only maximal 1 near edge. Further, if anchored edges occur, the orientation is always `top_to_bottom`.

It is possible to change the colors or underline during the output of text, e.g., drawing of labels or info fields. This is controlled by special characters in the corresponding string

\fi000	\fi001	\fi002	\fi003	\fi004	\fi005	\fi006	\fi007
\fi008	\fi009	\fi010	\fi011	\fi012	\fi013	\fi014	\fi015
\fi016	\fi017	\fi018	\fi019	\fi020	\fi021	\fi022	\fi023
\fi024	\fi025	\fi026	\fi027	\fi028	\fi029	\fi030	\fi031
\fi032	\fi033 !	\fi034 "	\fi035 #	\fi036 $	\fi037 %	\fi038 &	\fi039 '
\fi040 (	\fi041)	\fi042 *	\fi043 +	\fi044 ,	\fi045 -	\fi046 .	\fi047 /
\fi048 0	\fi049 1	\fi050 2	\fi051 3	\fi052 4	\fi053 5	\fi054 6	\fi055 7
\fi056 8	\fi057 9	\fi058 :	\fi059 ;	\fi060 <	\fi061 =	\fi062 >	\fi063 ?
\fi064 @	\fi065 A	\fi066 B	\fi067 C	\fi068 D	\fi069 E	\fi070 F	\fi071 G
\fi072 H	\fi073 I	\fi074 J	\fi075 K	\fi076 L	\fi077 M	\fi078 N	\fi079 O
\fi080 P	\fi081 Q	\fi082 R	\fi083 S	\fi084 T	\fi085 U	\fi086 V	\fi087 W
\fi088 X	\fi089 Y	\fi090 Z	\fi091 [	\fi092 \	\fi093]	\fi094 ^	\fi095 _
\fi096 `	\fi097 a	\fi098 b	\fi099 c	\fi100 d	\fi101 e	\fi102 f	\fi103 g
\fi104 h	\fi105 i	\fi106 j	\fi107 k	\fi108 l	\fi109 m	\fi110 n	\fi111 o
\fi112 p	\fi113 q	\fi114 r	\fi115 s	\fi116 t	\fi117 u	\fi118 v	\fi119 w
\fi120 x	\fi121 y	\fi122 z	\fi123 {	\fi124 |	\fi125 }	\fi126 ~	\fi127
\fi128	\fi129	\fi130	\fi131	\fi132	\fi133	\fi134	\fi135
\fi136	\fi137	\fi138	\fi139	\fi140	\fi141	\fi142	\fi143
\fi144 ı	\fi145 `	\fi146 ´	\fi147 ˆ	\fi148 ˜	\fi149 ¯	\fi150 ˘	\fi151 ˙
\fi152 ¨	\fi153	\fi154 ˚	\fi155 ¸	\fi156	\fi157 ˝	\fi158 ˛	\fi159 ˇ
\fi160	\fi161 ¡	\fi162 ¢	\fi163 £	\fi164 ¤	\fi165 ¥	\fi166 ¦	\fi167 §
\fi168 ¨	\fi169 ©	\fi170 ª	\fi171 «	\fi172 ¬	\fi173 -	\fi174 ®	\fi175 ¯
\fi176 °	\fi177 ±	\fi178 ²	\fi179 ³	\fi180 ´	\fi181 µ	\fi182 ¶	\fi183 ·
\fi184 ¸	\fi185 ¹	\fi186 º	\fi187 »	\fi188 ¼	\fi189 ½	\fi190 ¾	\fi191 ¿
\fi192 À	\fi193 Á	\fi194 Â	\fi195 Ã	\fi196 Ä	\fi197 Å	\fi198 Æ	\fi199 Ç
\fi200 È	\fi201 É	\fi202 Ê	\fi203 Ë	\fi204 Ì	\fi205 Í	\fi206 Î	\fi207 Ï
\fi208 Ð	\fi209 Ñ	\fi210 Ò	\fi211 Ó	\fi212 Ô	\fi213 Õ	\fi214 Ö	\fi215 ×
\fi216 Ø	\fi217 Ù	\fi218 Ú	\fi219 Û	\fi220 Ü	\fi221 Ý	\fi222 Þ	\fi223 ß
\fi224 à	\fi225 á	\fi226 â	\fi227 ã	\fi228 ä	\fi229 å	\fi230 æ	\fi231 ç
\fi232 è	\fi233 é	\fi234 ê	\fi235 ë	\fi236 ì	\fi237 í	\fi238 î	\fi239 ï
\fi240 ð	\fi241 ñ	\fi242 ò	\fi243 ó	\fi244 ô	\fi245 õ	\fi246 ö	\fi247 ÷
\fi248 ø	\fi249 ù	\fi250 ú	\fi251 û	\fi252 ü	\fi253 ý	\fi254 þ	\fi255 ÿ

Table 6: The ISO Latin 1 Character Set

values. Note: the ASCII value of the control characters depends on the operating system and the C compiler. The following control characters are allowed:

Newline (corresponds to the C sequence `"\n"`, mostly implemented by ASCII code 10) continue drawing text at the beginning of the next line.

Tabular (C sequence `"\t"`, ASCII code 9) draw 8 space characters.

Beep (C sequence `"\a"`, ASCII code 7) produce an audible or visible alert (equivalent to `printf("\a");`). The position for the next character to be drawn is not changed.

Backspace (C sequence `"\b"`, ASCII code 8) go one character back and continue drawing at that place.

Formfeed (C sequence `"\f"`, ASCII code 12) This occurs with an additional parameter (the next few characters) and changes the actual form of output.

`\fu` (ASCII codes 12 117) starts underlining.
`\fb` (ASCII codes 12 98) starts bold typeface.
`\fB` (ASCII codes 12 66) starts very bold typeface.
`\fn` (ASCII codes 12 110) stops underlining and bold typefaces, i.e. set to normal typeface.
`\fi000` (ASCII codes 12 105 48 48 48) prints the ISO character 0.
`\fi223` (ASCII codes 12 105 50 50 51) prints the ISO character 223 (the German ß).
`\fi252` (ASCII codes 12 105 50 53 50) prints the ISO character 252 (the German ü). See table 6 for the ISO Latin 1 character set.
`\f00` (ASCII codes 12 48 48) sets the color to white (or, to the color that currently has index 0 in the color table).
`\f31` (ASCII codes 12 51 49) sets the color to black (or, to the color that currently has index 31 in the color table). By this way, it is possible to access to the first 99 colors of the map. See table 5 for the default color map.

The level of nodes (also: summary nodes of subgraphs) is only recognized, if the whole graph is laid out automatically, i.e. if at least one node has no specified location. Normally, all nodes of level 0 form the uppermost layer, nodes of other levels form the next layer top down. The level specification may be in conflict with a *near edge* specification, because the source and target node of a near edge must have the same level. In this case, the level specification of source or target node of the near edge is ignored.

4 Examples of GDL Specifications

Here we give some GDL specifications with the displayed graphs.

4.1 A Cyclic Graph

Example 1 is a small cyclic graph without labels. The title is displayed in the nodes.

Example 1:

```
graph: {
        /* list of nodes */
        node: { title: "A" }   node: { title: "B" }   node: { title: "C" }
        node: { title: "D" }   node: { title: "E" }
        /* list of edges */
        edge: { thickness: 3 sourcename: "A" targetname: "B" }
        edge: { thickness: 3 sourcename: "A" targetname: "C" }
        edge: { thickness: 3 sourcename: "C" targetname: "D" }
        edge: { thickness: 3 sourcename: "D" targetname: "E" }
        edge: { thickness: 3 sourcename: "D" targetname: "A" }
}
```

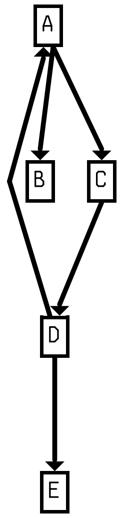


Figure 6: Example 1

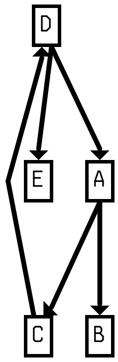


Figure 7: Example 2

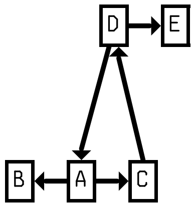


Figure 8: Example 3

The **VCG** tool tries to give all edges the same orientation. But since the graph is cyclic, one edge must be reverted (edge D→A). We can also select, which edge should be reverted, by specifying a back edge (edge C→D in example 2). .

Example 2:

```
graph: {
        /* list of nodes */
        node: { title: "A" }   node: { title: "B" }   node: { title: "C" }
        node: { title: "D" }   node: { title: "E" }
        /* list of edges */
        edge:      { thickness: 3 sourcename: "A" targetname: "B" }
        edge:      { thickness: 3 sourcename: "A" targetname: "C" }
        backedge:{ thickness: 3 sourcename: "C" targetname: "D" }
        edge:      { thickness: 3 sourcename: "D" targetname: "E" }
        edge:      { thickness: 3 sourcename: "D" targetname: "A" }
}
```

Again, the most of the edges have the same orientation. The tool selects the node D as topmost node now. The same cyclic graph looks completely different, if we add some near edges. The nodes connected by near edges are drawn at the same level (example 3).

Example 3:

```
graph: {
        /* list of nodes */
        node: { title: "A" }   node: { title: "B" }   node: { title: "C" }
        node: { title: "D" }   node: { title: "E" }
        /* list of edges */
        nearedge: { thickness: 3 sourcename: "A" targetname: "B" }
        nearedge: { thickness: 3 sourcename: "A" targetname: "C" }
        backedge: { thickness: 3 sourcename: "C" targetname: "D" }
        nearedge: { thickness: 3 sourcename: "D" targetname: "E" }
        edge:       { thickness: 3 sourcename: "D" targetname: "A" }
}
```

In some situations, we want to have edges that are horizontally anchored, but the target nodes should not be at the same level. Such edges must have a bend point. Here, we can use bent near edges (A→B and D→E in example 4).

Example 4:

```
graph: {
        /* list of nodes */
        node: { title: "A" }   node: { title: "B" }   node: { title: "C" }
        node: { title: "D" }   node: { title: "E" }
        /* list of edges */
        bentnearedge: { thickness: 3 sourcename: "A" targetname: "B" }
        nearedge:     { thickness: 3 sourcename: "A" targetname: "C" }
        backedge:     { thickness: 3 sourcename: "C" targetname: "D" }
        bentnearedge: { thickness: 3 sourcename: "D" targetname: "E" }
        edge:         { thickness: 3 sourcename: "D" targetname: "A" }
}
```

In order to indicate that node "D" represents a struct with two fields, whose first points to "E" and second points to "A", we can use the attribute `anchor` for the specification of the edges (example 5).

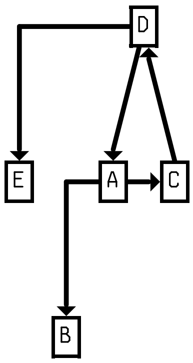


Figure 9: Example 4

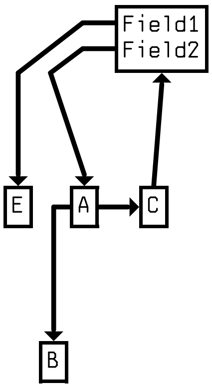


Figure 10: Example 5

Example 5:

```
graph: {
        /* list of nodes */
        node: { title: "A" }   node: { title: "B" }   node: { title: "C" }
        node: { title: "D" label: "Field1\nField2:" }   node: { title: "E" }
        /* list of edges */
        bentnearedge: { thickness: 3 sourcename: "A" targetname: "B" }
        nearedge:     { thickness: 3 sourcename: "A" targetname: "C" }
        backedge:     { thickness: 3 sourcename: "C" targetname: "D" }
        edge:         { thickness: 3 sourcename: "D" targetname: "E" anchor: 1 }
        edge:         { thickness: 3 sourcename: "D" targetname: "A" anchor: 2 }
}
```

4.2 A Control Flow Graph

Example 6 is a control flow graph of a procedural program. The nodes contain the text of statements as labels. Not all edges have labels. The displayed program (in the pseudo language CLaX) consists of a procedure `test` and a main routine:

```
PROCEDURE test( VAR b : INTEGER; c : INTEGER);
BEGIN
  b := c + 5;
END

BEGIN /* main routine of a nonsense program */
  x := 1;
  WHILE (x = 1) DO
    x := 2;
    test ( x, 1 );
    x := 3;
  OD;
  WHILE (x = 1) DO
    x := 4;
    x := 5;
    test ( x, 2 );
  OD;
  WHILE (x = 1) DO
    x := 6;
    IF (x = 7) THEN x := 8; ELSE test ( x, 3 );
    FI;
  OD;
END.
```

Example 6:

```
graph: { title: "CFG_GRAPH"
         splines: yes
         layoutalgorithm: dfs finetuning: no
         display_edge_labels: yes
         yspace: 55
         node: { title:"18" label: "test_b := test_c + 5" }
         node: { title:"17" label: "Exit" }
         node: { title:"16" label: "test(x,3)" }
         node: { title:"15" label: "x := 8" }
         node: { title:"14" label: "x = 7" }
         node: { title:"13" label: "x := 6" }
         node: { title:"12" label: "x = 1" }
         node: { title:"11" label: "test(x,2)" }
         node: { title:"10" label: "x := 5" }
         node: { title:"9" label: "x := 4" }
         node: { title:"8" label: "x = 1" }
         node: { title:"7" label: "x := 3" }
         node: { title:"6" label: "test(x,1)" }
         node: { title:"5" label: "x := 2" }
         node: { title:"4" label: "x = 1" }
         node: { title:"3" label: "x := 1" }
         node: { title:"2" label: "Start" }
         node: { title:"1" label: "Exit_point\ntest" }
         node: { title:"0" label: "Entry_point\ntest" }
         edge: { thickness: 4 sourcename:"18" targetname:"1" }
         edge: { thickness: 4 sourcename:"0" targetname:"18" }
         edge: { thickness: 4 sourcename:"12" targetname:"17" label: "false" }
```

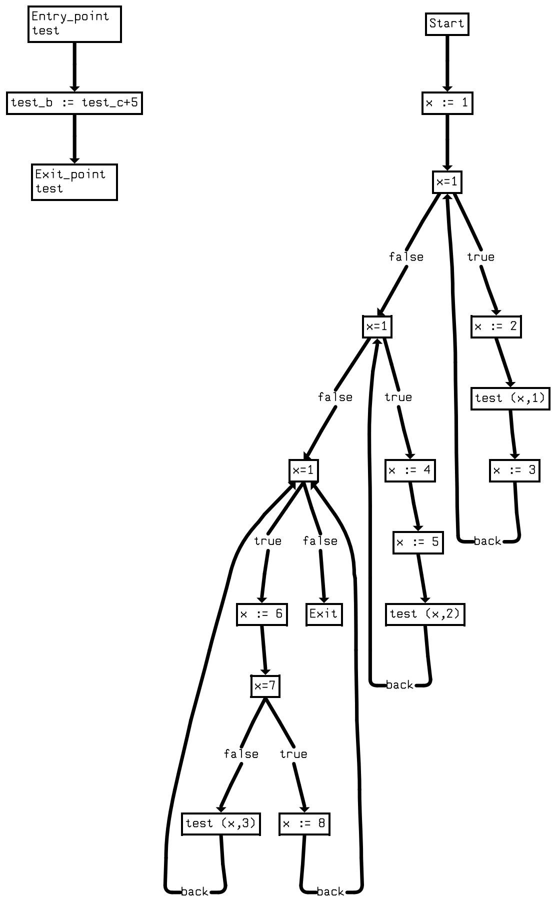


Figure 11: Example 6

```
    edge: { thickness: 4 sourcename:"8" targetname:"12" label: "false" }
    edge: { thickness: 4 sourcename:"16" targetname:"12" label: "back" }
    edge: { thickness: 4 sourcename:"15" targetname:"12" label: "back" }
    edge: { thickness: 4 sourcename:"13" targetname:"14" }
    edge: { thickness: 4 sourcename:"14" targetname:"16" label: "false" }
    edge: { thickness: 4 sourcename:"14" targetname:"15" label: "true" }
    edge: { thickness: 4 sourcename:"12" targetname:"13" label: "true" }
    edge: { thickness: 4 sourcename:"4" targetname:"8" label: "false" }
    edge: { thickness: 4 sourcename:"11" targetname:"8" label: "back" }
    edge: { thickness: 4 sourcename:"10" targetname:"11" }
    edge: { thickness: 4 sourcename:"9" targetname:"10" }
    edge: { thickness: 4 sourcename:"8" targetname:"9" label: "true" }
    edge: { thickness: 4 sourcename:"3" targetname:"4" }
    edge: { thickness: 4 sourcename:"7" targetname:"4" label: "back" }
    edge: { thickness: 4 sourcename:"6" targetname:"7" }
    edge: { thickness: 4 sourcename:"5" targetname:"6" }
    edge: { thickness: 4 sourcename:"4" targetname:"5" label: "true" }
    edge: { thickness: 4 sourcename:"2" targetname:"3" }
}
```

This previous example was a very simple translation into a control flow graph. The start, exit and branch nodes can be better recognized if we use different shapes for them. The edges that close a cycle can be specified as back edges, in order to see the uniform flow of the other edges. The decision edges should be anchored left and right to the branch nodes, thus, we use bent near edges. The result is example 7.

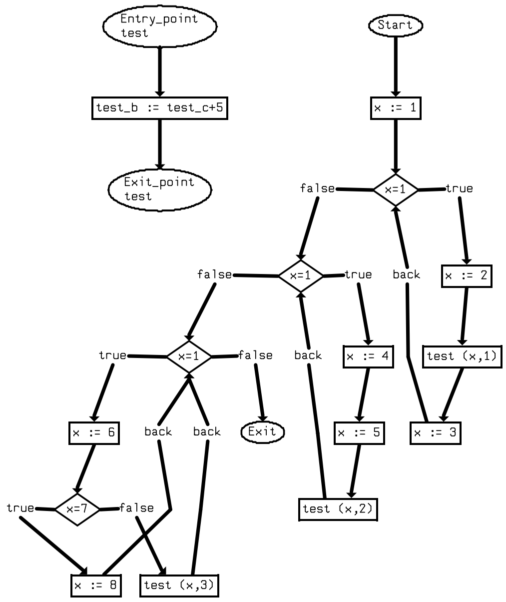


Figure 12: Example 7

Example 7:

```
graph: { title: "CFG_GRAPH"
          layoutalgorithm: dfs
          finetuning: no
          display_edge_labels: yes
          yspace: 55
          node: { title:"18" label: "test_b := test_c + 5" }
          node: { title:"17" label: "Exit" shape: ellipse }
          node: { title:"16" label: "test(x,3)" }
          node: { title:"15" label: "x := 8" }
          node: { title:"14" label: "x = 7" shape: rhomb }
          node: { title:"13" label: "x := 6" }
          node: { title:"12" label: "x = 1" shape: rhomb }
          node: { title:"11" label: "test(x,2)" }
          node: { title:"10" label: "x := 5" }
          node: { title:"9" label: "x := 4" }
          node: { title:"8" label: "x = 1" shape: rhomb }
          node: { title:"7" label: "x := 3" }
          node: { title:"6" label: "test(x,1)" }
          node: { title:"5" label: "x := 2" }
          node: { title:"4" label: "x = 1" shape: rhomb }
```

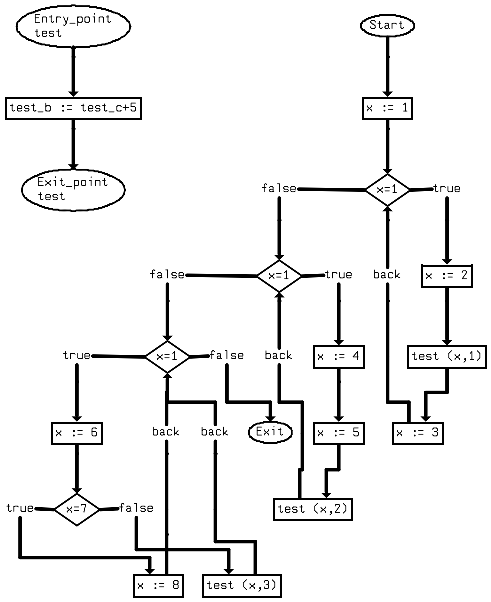


Figure 13: Example 8

```
    node: { title:"3" label: "x := 1" }
    node: { title:"2" label: "Start" shape: ellipse }
    node: { title:"1" label: "Exit_point\ntest" shape: ellipse }
    node: { title:"0" label: "Entry_point\ntest" shape: ellipse }
    edge: { thickness: 4 sourcename:"18" targetname:"1" }
    edge: { thickness: 4 sourcename:"0" targetname:"18" }
    bentnearedge: { thickness: 4 sourcename:"12" targetname:"17" label: "false" }
    bentnearedge: { thickness: 4 sourcename:"8" targetname:"12" label: "false" }
    backedge: { thickness: 4 sourcename:"16" targetname:"12" label: "back" }
    backedge: { thickness: 4 sourcename:"15" targetname:"12" label: "back" }
    edge: { thickness: 4 sourcename:"13" targetname:"14" }
    bentnearedge: { thickness: 4 sourcename:"14" targetname:"16" label: "false" }
    bentnearedge: { thickness: 4 sourcename:"14" targetname:"15" label: "true" }
    bentnearedge: { thickness: 4 sourcename:"12" targetname:"13" label: "true" }
    bentnearedge: { thickness: 4 sourcename:"4" targetname:"8" label: "false" }
    backedge: { thickness: 4 sourcename:"11" targetname:"8" label: "back" }
    edge: { thickness: 4 sourcename:"10" targetname:"11" }
    edge: { thickness: 4 sourcename:"9" targetname:"10" }
    bentnearedge: { thickness: 4 sourcename:"8" targetname:"9" label: "true" }
    edge: { thickness: 4 sourcename:"3" targetname:"4" }
    backedge: { thickness: 4 sourcename:"7" targetname:"4" label: "back" }
    edge: { thickness: 4 sourcename:"6" targetname:"7" }
    edge: { thickness: 4 sourcename:"5" targetname:"6" }
    bentnearedge: { thickness: 4 sourcename:"4" targetname:"5" label: "true" }
    edge: { thickness: 4 sourcename:"2" targetname:"3" }
}
```

If we use the orthogonal layout, the graph looks like a typical flowchart. Here, the downfactor should be large while the nearfactor and the upfactor must be zero. The result is example 8.

Example 8:

```
graph: { title: "CFG_GRAPH"
         manhattan_edges: yes
         layoutalgorithm: dfs
         finetuning: no
         display_edge_labels: yes
         layout_downfactor: 100
         layout_upfactor: 0
         layout_nearfactor: 0
         xlspace: 12
         yspace: 55
         ... nodes and edges as in example 7 ...
}
```

4.3 The Effect of the Layout Algorithms

The following sequence of pictures shows several times the same graph visualized by different layout algorithms. The graph is cyclic, thus it depends on the personal taste which layout is the best. Of course, the algorithm `tree` is not applicable. If the graph is acyclic, the default layout algorithm or the layout algorithm `minbackward` are the most appropriate in nearly all cases. Very often, the main problem is to select the nodes that appear at the top level of the graph. The layout algorithm looks for candidates that have no incoming edges but at least one outgoing edge. If such a node does not exist – as in example 9 – the algorithms `mindegree`, ..., `maxoutdegree` are helpful.

The fine tuning phase eliminates long edges. The tuned graph is more compact. The tuned graph created by `maxdepthslow` need not to be maximal deep because the fine tuning may have reduced the deep better with another variant of the layout algorithm. The tuned graph created by `mindepthslow` need not to be minimal deep, too. All these partitioning algorithms are only heuristics.

Example 9:

```
graph: {
        xspace: 25
        node: { title: "A" label: "Start of all" }
        node: { title: "B" }   node: { title: "C" }   node: { title: "D" }
        node: { title: "E" }
        node: { title: "F" }   node: { title: "G" }   node: { title: "H" }
        node: { title: "I" }   node: { title: "J" }   node: { title: "K" }
        edge: { thickness: 3 sourcename: "A" targetname: "B" }
        edge: { thickness: 3 sourcename: "A" targetname: "C" }
        edge: { thickness: 3 sourcename: "A" targetname: "D" }
        edge: { thickness: 3 sourcename: "A" targetname: "E" }
        edge: { thickness: 3 sourcename: "A" targetname: "F" }
        edge: { thickness: 3 sourcename: "A" targetname: "J" }
        edge: { thickness: 3 sourcename: "B" targetname: "D" }
        edge: { thickness: 3 sourcename: "C" targetname: "E" }
        edge: { thickness: 3 sourcename: "D" targetname: "F" }
        edge: { thickness: 3 sourcename: "F" targetname: "K" }
        edge: { thickness: 3 sourcename: "J" targetname: "K" }
        edge: { thickness: 3 sourcename: "A" targetname: "G" }
        edge: { thickness: 3 sourcename: "G" targetname: "H" }
        edge: { thickness: 3 sourcename: "H" targetname: "I" }
        edge: { thickness: 3 sourcename: "I" targetname: "A" }
}
```

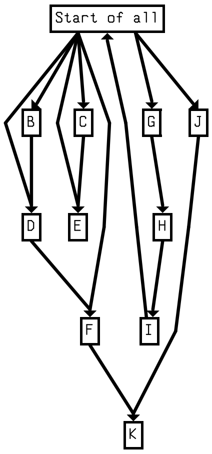


Figure 14: `normal` without fine tuning

The normal layout algorithm breaks the cycle such that only one reverted edge is necessary.

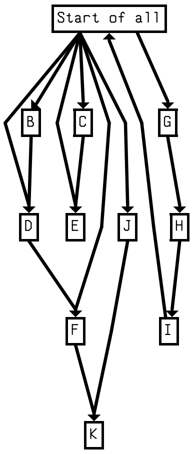


Figure 15: `normal` with fine tuning

Compared to the previous layout, the fine tuning phase has balanced the position of the node `J`. The long edge `I->Start` will not be balanced since this would create additional reverted edges.

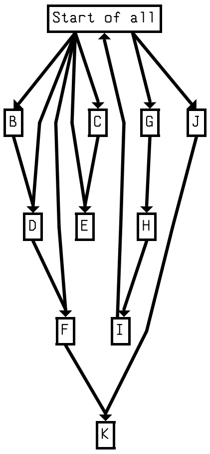


Figure 16: `minbackward` without fine tuning

This is nearly the same picture as for `normal`. Again, only one reverted edge is necessary. The layout algorithm `maxdepth` without fine tuning results in the same picture.

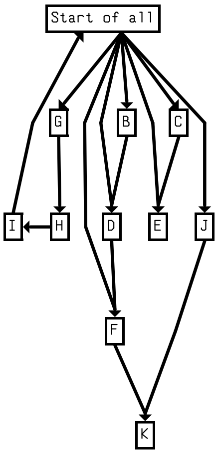


Figure 17: `minbackward` with fine tuning

Compared to the previous layout, the fine tuning phase has partially eliminated the long edge `I->Start` and has again balanced the position of node `J`.

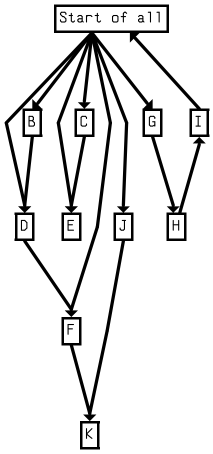


Figure 18: `maxdepth` with fine tuning

The long edge `I->Start` is now fully eliminated. Here, the fine tuning phase is allowed to revert additional edges.

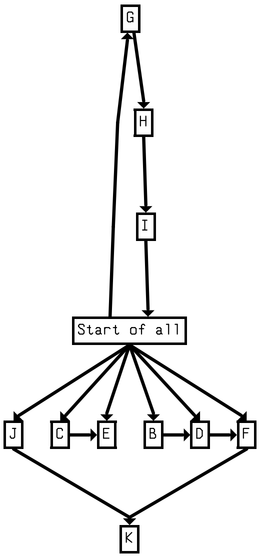


Figure 19: `maxdepthslow` without fine tuning

This layout with depth 6 is in fact maximal deep, compared to all other variants.

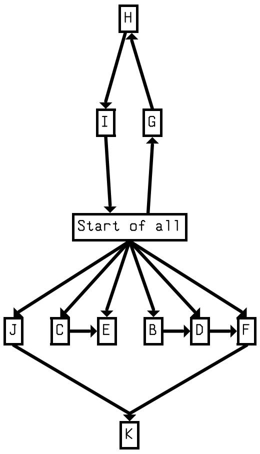


Figure 20: `maxdepthslow` with fine tuning

The fine tuning phase eliminates the long edge `Start->G`. Thus, the layout is not anymore maximal deep: Fine tuning destroys the property to be maximal deep.

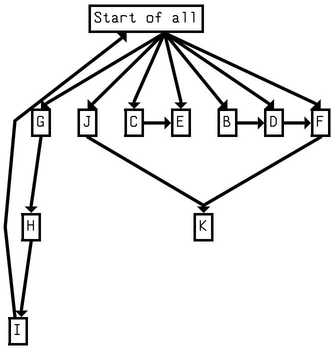


Figure 21: `mindepth` without fine tuning

The layout algorithms `dfs` and `minindegree` happen to result in the same picture.

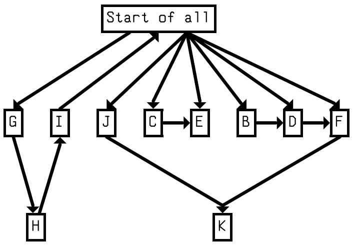


Figure 22: `mindepth` with fine tuning

Compared to the previous layout, the long edges `I->Start` is eliminated. In fact, this is the layout with the minimal depth.

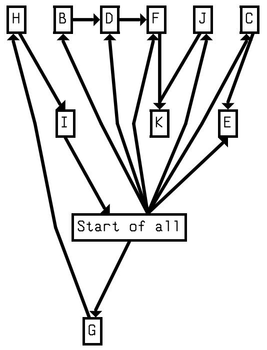


Figure 23: `mindepthslow` without fine tuning

Graphs that are minimal deep tend to have many nodes at the top level. Compared to all untuned graphs, this layout is minimal deep. However note, that the algorithm `mindepth` *with fine tuning* is able to produce a flatter layout.

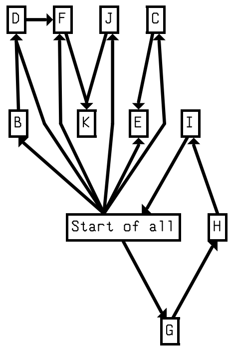


Figure 24: `mindepthslow` with fine tuning

The long edges `G->H` and `Start->B` are eliminated. Note that the fine tuning phase of algorithm `mindepth` happens to reduce the deep while here, this is not possible. Thus, compared to all fine tuned graphs, `mindepthslow` does not produce the flattest layout.

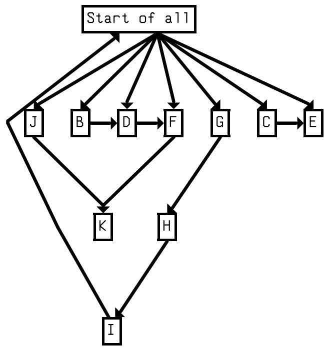


Figure 25: `maxdegree` without fine tuning

The node `Start` has the most adjacent edges. Thus it is selected as start node of the spanning tree, i.e. it appears at the topmost level.

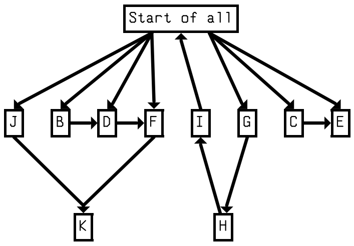


Figure 26: `maxdegree` with fine tuning

Compared to the previous picture, the long edge `I->Start` is eliminated. This is also a layout with minimal depth.

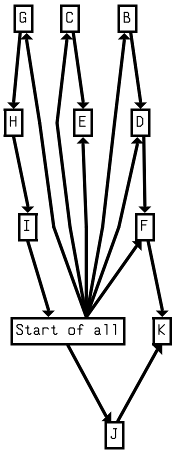


Figure 27: `mindegree` without fine tuning

The candidates for start nodes of the spanning tree are the nodes `B, C, G, H, I` and `J` because they have the minimal degree 2. From these nodes, `B`, `C` and `G` happened to be selected. Note: the nodes `E` and `K` (also degree 2) are no candidates of start nodes because they do not have outgoing edges.

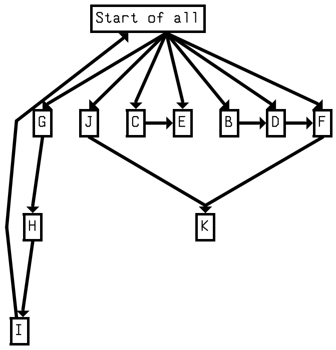


Figure 29: `minindegree` without fine tuning

The candidates for start nodes are `Start, B, C, G, H, I` and `J`, from which `Start` was selected. The algorithm `maxoutdegree` results in the same picture.

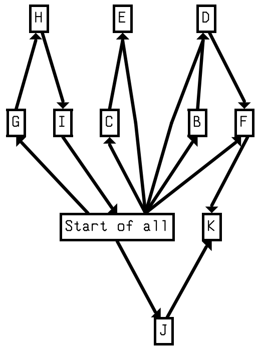


Figure 28: `mindegree` with fine tuning

The long edges `Start->B` and `Start C` are eliminated. This changes the structure of the layout completely.

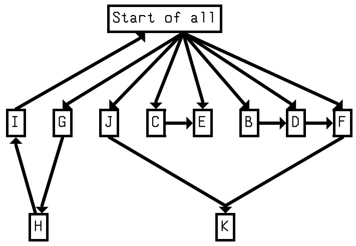


Figure 30: `minindegree` with fine tuning

The long edge `I->Start` is eliminated. This is again a layout with minimal depth.

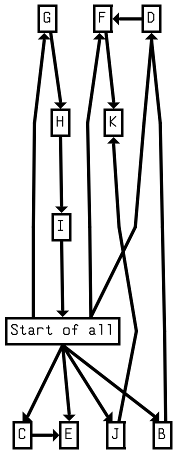


Figure 31: `maxindegree` without fine tuning

This time, the candidates are `D` and `F`, from which `F` was selected as start node resulting in the spanning tree `F->K`. Because `K` has no outgoing edges, this component of the spanning tree cannot be larger. Thus, a second component of the spanning tree is needed, which starts at `D` and is `D` since it has no outgoing edges to not yet scheduled nodes. A third component starts at `G` which is one of the not yet scheduled nodes with maximal indegree.

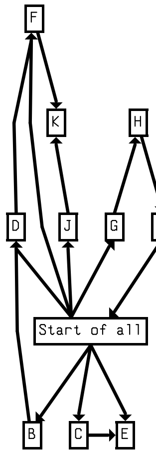


Figure 33: `maxindegree` with fine tuning

`F` is again start node of one component of the spanning tree. Compared to the previous example, the long edges `Start->G`, `B->D` and `Start D` are eliminated.

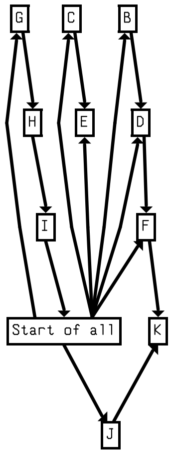


Figure 32: `minoutdegree` without fine tuning

The nodes `E` and `K` with minimal outdegree 0 cannot be start nodes, because start nodes must have at least one successor. Otherwise, they would create one-node components of the spanning tree. The useful candidates are all other nodes except `Start`, from which `B, C` and `G` happened to be selected.

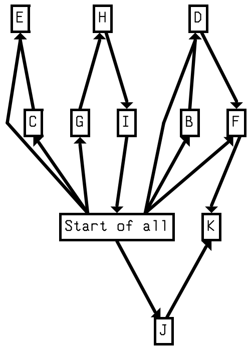
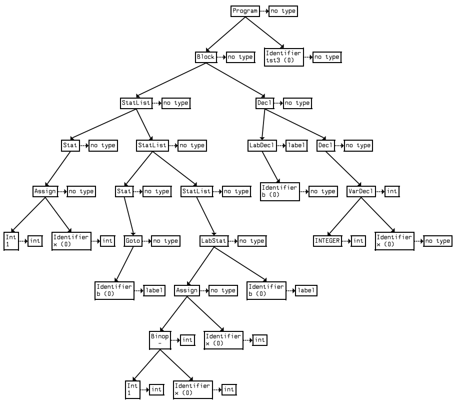


Figure 34: `minoutdegree` with fine tuning

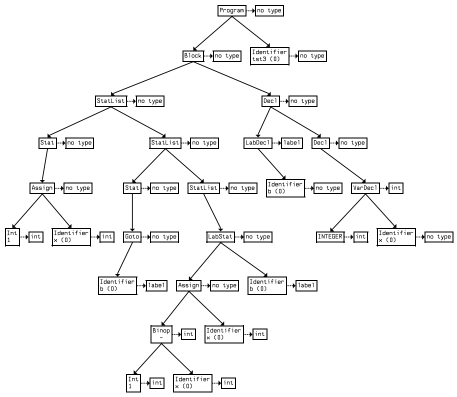
The long edges `Start->G`, `Start->B` and `Start->C` are eliminated.

4.4 Tree Layouts

The following example shows a typed syntax tree. This tree can either be laid out by the specialized algorithm for downward laid out trees, or by the normal algorithms using crossing reduction and rubberband methods. The layout of a tree is quite strange, if the layout_downfactor is not used. The incoming edges draw the nodes too much into the direction of of the parent node. The nicest layout is produced by the specialized tree algorithm with a tree factor of 0.9. If an orthogonal layout is needed, the attribute `smanhattan_edges` can be used. For trees, it is more appropriate than the normal manhattan layout with `manhattan_edges`.



Positioning by the rubberband method. The layout_downfactor, layout_upfactor and layout_nearfactor are 1. The nodes are pulled in direction of their parent nodes.



Positioning by the rubberband method. The layout_downfactor is 10. The layout_upfactor and layout_nearfactor are 1. The nodes are not anymore pulled in direction of their parent nodes.

Figure 35: Example 10: Layout algorithm `maxdepth`

Example 10:

```
graph: {
        title: "typed syntax tree"
        node:      { title: "503160" label: "Identifier\ntst3 (0)" }
        node:      { title: "503240" label: "Identifier\nx (0)"    }
        node:      { title: "502952" label: "INTEGER"              }
        node:      { title: "503304" label: "VarDecl"              }
        ...
        node:      { title: "T0"     label: "no type"              }
        node:      { title: "T1"     label: "no type"              }
        node:      { title: "T2"     label: "int"                  }
        ...
        edge:      { sourcename: "503304" targetname: "503240" }
        edge:      { sourcename: "503304" targetname: "502952" }
        ...
        nearedge:  { sourcename: "503160" targetname: "T0"     linestyle: dotted }
        nearedge:  { sourcename: "503240" targetname: "T1"     linestyle: dotted }
```

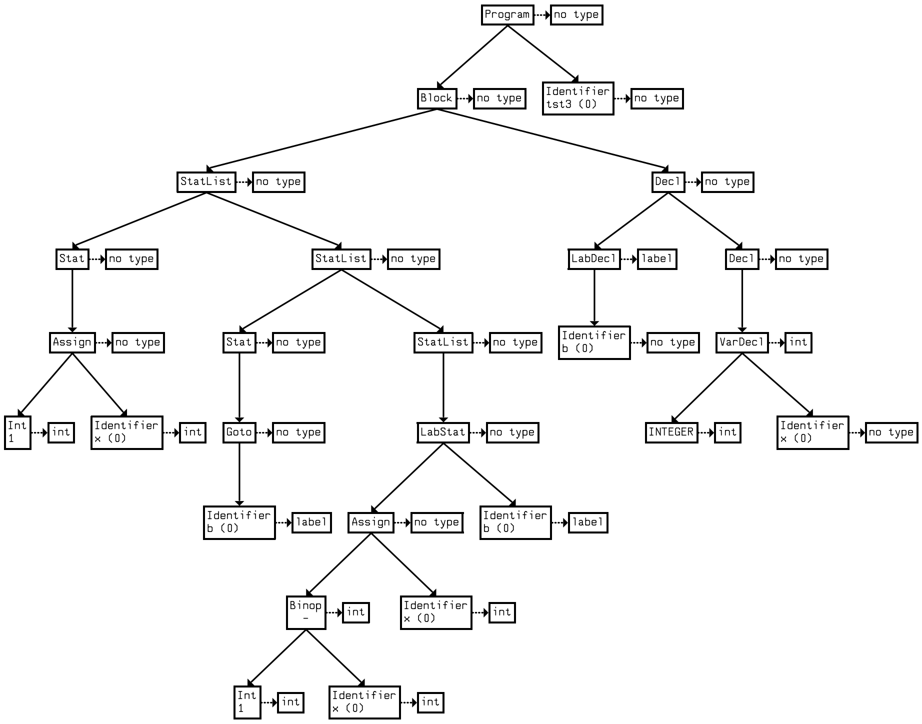


Figure 36: Example 10: Layout algorithm `tree, treefactor:0.9`

```
        nearedge: { sourcename: "502952" targetname: "T2"     linestyle: dotted }
        ...
}
```

4.5 The Combination of Features

The following example is taken from [GKNV93] and shows the dependencies of different shell programs. To visualize it, a combination of features of the **VCG** tool is used. There is a time scale that should indicate the origin of the programs. The shells themselves are nodes that must be placed at the same rank as their birth dates. We use the attribute `vertical_order` to set the nodes to these positions. Furthermore, we want to have the time axis at the left side of the shell dependence graph. This is achieved by the attribute `horizontal_order` at some of the nodes. However, this attribute only works if the graph is connected. Thus, we create three invisible edges to make the graph connected.

Invisible edges, as all other edges, influence the positions of the nodes as they would pull their adjacent nodes together. To avoid this effect for the invisible edges, we set the priority of the invisible edges to zero and the priority of the visible edges to 100. There are many possibilities to change the priority: we can set the attribute `priority`, but we can also set the layout factors `downfactor`, `upfactor` and `nearfactor`. The real priority of a downward

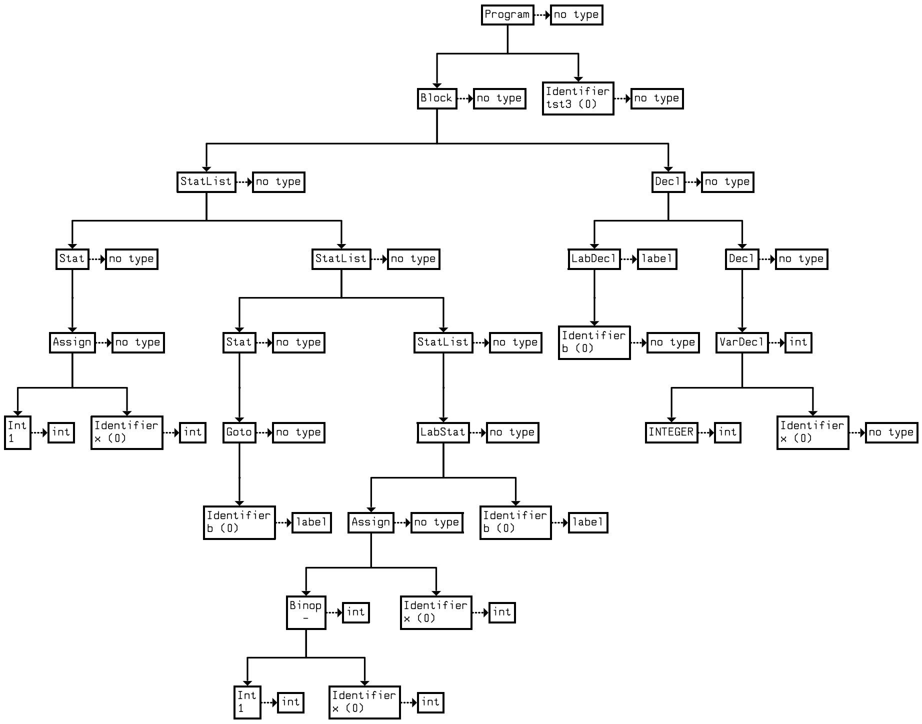


Figure 37: Example 10: Layout algorithm `tree, smanhattan_edges`

edge is the product *downfactor* $\times$ *priority*.

We want to have the shell `Bourne` left to the shell `Mashey` and `csh` right to `Mashey`. Thus we also give the nodes at level 2 a horizontal order. However, `csh` is on level 3, and only its edge crosses level 2. Thus we set the attribute `horizontal_order` for this edge, too, and now this edge is drawn to the right of `Mashey`.

To reduce the amount of specification, we use default attribute specifications for the height, width and borderwidth of nodes and for the style of edges. To differentiate, we use ellipses for the different variations of the KornShell, triangles for C-Shells and a rhomb for `tcl`. The graph is acyclic, thus the layout algorithm `minbackward` is used. Edges are drawn by splines.

Example 11:

```
graph: {
        title: "shells"
        splines: yes
        layoutalgorithm: minbackward
        layout_nearfactor: 0
        layout_downfactor: 100
        layout_upfactor: 100

        // First the time scale
```

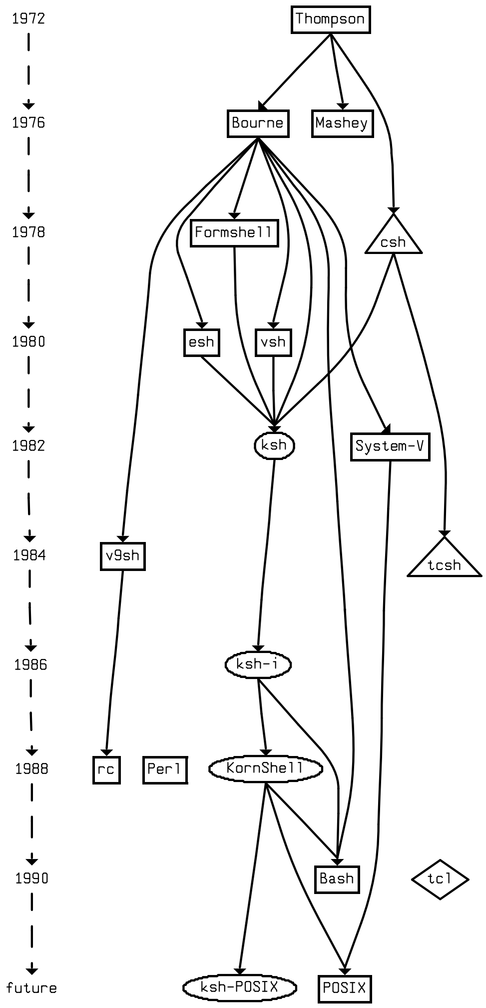


Figure 38: Example 11

node.height: 26
node.width: 60
node.borderwidth: 0
edge.linestyle: dashed

node: { title: "1972" vertical_order: 1 horizontal_order: 1 }
node: { title: "1976" vertical_order: 2 horizontal_order: 1 }
node: { title: "1978" vertical_order: 3 }
node: { title: "1980" vertical_order: 4 }

```
node:     { title: "1982"   vertical_order: 5  horizontal_order: 1 }
node:     { title: "1984"   vertical_order: 6  }
node:     { title: "1986"   vertical_order: 7  }
node:     { title: "1988"   vertical_order: 8  }
node:     { title: "1990"   vertical_order: 9  }
node:     { title: "future" vertical_order: 10 horizontal_order: 1 }

edge:     { sourcename: "1972" targetname: "1976"  }
edge:     { sourcename: "1976" targetname: "1978"  }
edge:     { sourcename: "1978" targetname: "1980"  }
edge:     { sourcename: "1980" targetname: "1982"  }
edge:     { sourcename: "1982" targetname: "1984"  }
edge:     { sourcename: "1984" targetname: "1986"  }
edge:     { sourcename: "1986" targetname: "1988"  }
edge:     { sourcename: "1988" targetname: "1990"  }
edge:     { sourcename: "1990" targetname: "future"}

// We need some invisible edge to make the graph fully connected.
// Otherwise, the horizontal_order attribute would not work.

edge:     { sourcename: "ksh-i" targetname: "Perl"  linestyle: invisible priority: 0 }
edge:     { sourcename: "tcsh" targetname: "tcl"    linestyle: invisible priority: 0 }
nearedge: { sourcename: "1988" targetname: "rc"     linestyle: invisible }
nearedge: { sourcename: "rc" targetname: "Perl"     linestyle: invisible }

// Now the shells themselves
// Note: the default value -1 means: no default

node.height: -1
node.width: -1
node.borderwidth: 2
edge.linestyle: solid
node:     { title: "Thompson"   vertical_order: 1   horizontal_order: 2 }
node:     { title: "Mashey"     vertical_order: 2   horizontal_order: 3 }
node:     { title: "Bourne"     vertical_order: 2   horizontal_order: 2 }
node:     { title: "Formshell"  vertical_order: 3   }
node:     { title: "csh"        vertical_order: 3   shape: triangle }
node:     { title: "esh"        vertical_order: 4   horizontal_order: 2 }
node:     { title: "vsh"        vertical_order: 4   }
node:     { title: "ksh"        vertical_order: 5   horizontal_order: 3 shape: ellipse }
node:     { title: "System-V"   vertical_order: 5   horizontal_order: 5 }
node:     { title: "v9sh"       vertical_order: 6   }
node:     { title: "tcsh"       vertical_order: 6   shape: triangle }
node:     { title: "ksh-i"      vertical_order: 7   shape: ellipse   }
node:     { title: "KornShell"  vertical_order: 8   shape: ellipse   }
node:     { title: "Perl"       vertical_order: 8   }
node:     { title: "rc"         vertical_order: 8   }
node:     { title: "tcl"        vertical_order: 9   shape: rhomb     }
node:     { title: "Bash"       vertical_order: 9   }
node:     { title: "POSIX"      vertical_order: 10  horizontal_order: 3 }
node:     { title: "ksh-POSIX"  vertical_order: 10  horizontal_order: 2 shape: ellipse }

edge:     { sourcename: "Thompson"  targetname: "Mashey"     }
edge:     { sourcename: "Thompson"  targetname: "Bourne"     }
edge:     { sourcename: "Thompson"  targetname: "csh"        horizontal_order: 4 }
edge:     { sourcename: "Bourne"    targetname: "ksh"        }
edge:     { sourcename: "Bourne"    targetname: "esh"        }
edge:     { sourcename: "Bourne"    targetname: "vsh"        }
edge:     { sourcename: "Bourne"    targetname: "System-V"   }
edge:     { sourcename: "Bourne"    targetname: "v9sh"       }
edge:     { sourcename: "Bourne"    targetname: "Formshell"  }
edge:     { sourcename: "Bourne"    targetname: "Bash"       }
```

```
        edge:   { sourcename: "csh"          targetname: "tcsh"       }
        edge:   { sourcename: "csh"          targetname: "ksh"        }
        edge:   { sourcename: "Formshell"    targetname: "ksh"        horizontal_order: 4 }
        edge:   { sourcename: "esh"          targetname: "ksh"        }
        edge:   { sourcename: "vsh"          targetname: "ksh"        }
        edge:   { sourcename: "ksh"          targetname: "ksh-i"      }
        edge:   { sourcename: "System-V"     targetname: "POSIX"      }
        edge:   { sourcename: "v9sh"         targetname: "rc"         }
        edge:   { sourcename: "ksh-i"        targetname: "KornShell"  }
        edge:   { sourcename: "ksh-i"        targetname: "Bash"       }
        edge:   { sourcename: "KornShell"    targetname: "Bash"       }
        edge:   { sourcename: "KornShell"    targetname: "POSIX"      }
        edge:   { sourcename: "KornShell"    targetname: "ksh-POSIX"  }
}
```

5 Usage of the VCG tool

The usage of the **VCG** tool is very simple. It is designed as an auxiliary tool that works in combination with programs that provide automatically the input of the tool. Thus, the possibilities to change the visualized graph interactively are very limited. The interactive commands are concentrated to improve the readability of existing graphs, i.e. to show important parts and hide other parts.

5.1 Starting the Tool

The invocation of the **VCG** tool is:

xvcg [*filename*]

If the optional parameter *filename* is set to "–", the input file is `<stdin>`. If *filename* is not specified, the tool asks for the filename containing the graph description in GDL. If multiple graph specifications should be visualized sequentially, the tool is invoked by

xvcg -multi $filename_1$ $filename_2$ $filename_3$ ...

Instead of terminating the tool after the visualization of $filename_1$, the tool is automatically reinvoked to visualize $filename_2$, $filename_3$, etc. The command "xvcg -h" prints an explanation of the usage on the screen. Other options of the tool are explained in the manual page. After reading the input, the visualization layout is calculated with the parameters given in the GDL file. The graph is drawn in a X11 window [Pet91]. Interactive commands are entered by a mouse menu (pull down menu using the left or right mouse button). A summary of commands is shown in table 7.

5.2 The Graph Window

The graph window consists of a drawing area where the graph appears, a text area below where the messages are printed, 5 scrollbars and a small button (right, below the right scrollbar). The menu becomes visible on a mouse click into the drawing area, as shown in figure 39.

Item	*Description*
Fold Subgraph	fold a subgraph to a summary node
Unfold Subgraph	unfold a subgraph
Expose/Hide Edges	hide or expose edges and their regions
Fold Region	fold a region of class k
Unfold Region	unfold a region
Scroll	scroll the virtual window
Node Information	show the info1, info2, or info3 text, the label or layout attributes of a node.
Position	position the virtual window absolutely
Pick Position	position the virtual window accordingly to mouse position
Center Node	position the virtual window to center a node
Follow Edge	center the node at the end of an edge in the window
Ruler	switch position rulers on or off
Layout	change the layout parameters of the graph
View	change the view parameters
Scale	set (shrink/stretch) factor for magnification
File	store the graph in VCG, PBM, PPM or PostScript format, load a new file, or reload the actual file again
Quit	exit the tool

Table 7: Menu Items

As described in section 3.1, the displayed window shows a part of the virtual window that contains the actually drawn part of the graph (see figure 5). Parts that are not inside the virtual window are not drawn because of performance reasons. The displayed window can be closed or opened, but cannot be larger than the virtual window. The first left scrollbar is used to position the virtual window to a y-coordinate, i.e. to move the window vertically through the system of coordinates. The second left scrollbar is used to scroll the displayed window vertically through the virtual window, i.e. to fine tune the vertical position of the visible part of the graph.

The first lower scrollbar is used to position the virtual window to a x-coordinate, i.e. to move the window horizontally through the system of coordinates. The second lower scrollbar is used to scroll the displayed window horizontally through the virtual window, i.e. to fine tune the horizontal position of the visible part of the graph.

Note: if the virtual window is positioned, this causes a redrawing of a part of the graph. If the visible window is scrolled through the virtual window, this causes refresh of the visible window, which is usually a much faster operation, because of a graphic buffer.

The right scrollbar is used to set the scaling of the graph. If the scrollthumb is in the middle of the scrollbar, the scaling is 100%, i.e. it is normal size. The small buttons at the beginning and end of each scrollbar can be used to increment or decrement the scrollbar value in fine grained steps. The amount of increment or decrement depends on the selected mouse button used to push onto the scrollbar buttons. The small button below the right scrollbar is used to set an appropriate scaling such that the whole graph is completely visible. For large

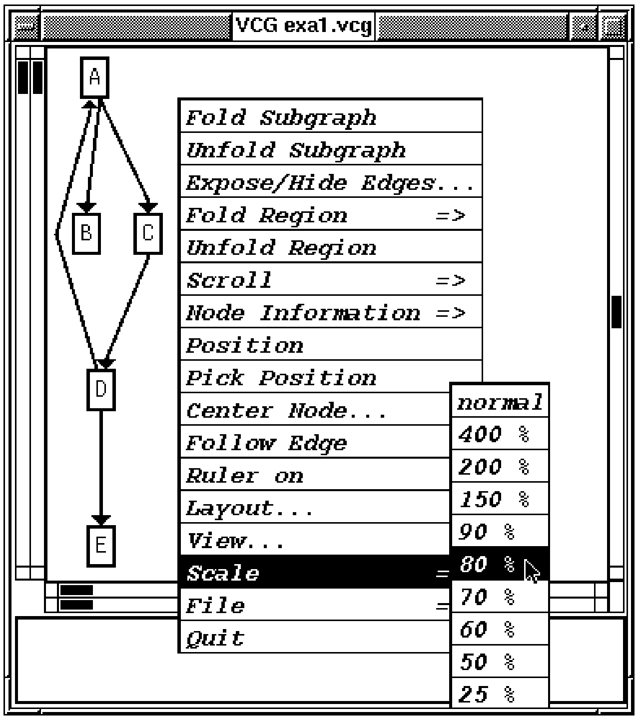


Figure 39: The Graph Window

graphs, this will set a large demagnification such that no details are anymore visible.

5.3 Folding

As already mentioned, the graph can be partitioned into nested subgraphs, that can be folded by selecting one of their nodes, and unfolded by selecting the summary node of a subgraph. Further, a class of edges can be hidden, which also hides the region of nodes only reachable by edges of this class. Finally, a connected region can be folded dynamically by selecting the start nodes and the end node of a region and an edge class k. All nodes reachable from the start nodes by edges of classes less or equal then k up to (and unless) the end nodes are condensed into one summary node. Nested foldings are possible. To activate the different folding methods, the following items are in the mouse menu:

- **Fold Subgraph**: After selection of this item, an arbitrary node of the subgraph to be folded must be selected. The corresponding subgraph is folded.
- **Unfold Subgraph**: The summary node of a subgraph must be selected to show this subgraph explicitly.
- **Expose/Hide Edges**: A dialog box of all edge classes appears (see figure 40 for an example). There is at least the default edge class “1”. The edge classes are shown by

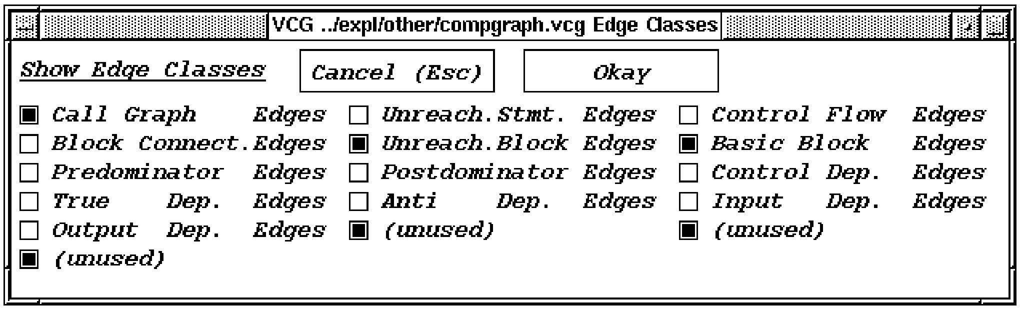


Figure 40: The Edge Class Menu of an Example

The text of the edge classes is specified in this example graph and changes with each new graph (see attribute `classname`).

numbers, or by the names that are assigned to the classes in the specification (see attribute `classname`). The classes currently exposed are highlighted. Here, the classes to hide and the classes to expose can be selected. As at all dialog boxes, the selection of the "Okay" button causes the relayout, while the selection of the "Cancel" button cancels this operation.

- **Fold Region**: An edge class must be selected from the submenu. First, nodes are selected by the left mouse button where the following "Fold Region" operation stops. This corresponds to the folding attribute value 0 of nodes. After pressing the right mouse button, nodes can be marked where the folding process starts. The connected region of this class is folded until the foldstops are reached (if there are any).
- **Unfold Region**: After selection of a summary node, the corresponding connected region is unfolded.

5.4 Positioning

The displayed window can be scrolled through the virtual window by scrollbars. The virtual window can be positioned over the potential infinite system of coordinates of the graph by scrollbars, too. If the fisheye view is selected (see section 5.8), the positioning moves the focus point instead of the virtual window. Additionally, there is the item **Scroll** in the mouse menu, which opens a submenu with

- **left**, **right**, **up**, **down**: move the virtual window (or focus point) 32 pixels to the corresponding direction.
- **lleft**, **rright**, **uup**, **ddown**: move the virtual window (or focus point) 256 pixels to the corresponding direction.

- **llleft**, **rrright**, **uuup**, **dddown**: move the virtual window (or focus point) one screensize to the corresponding direction. The screensize is given by the attributes `width` and `height` of the outermost graph (see section 3.1).
- **origin** move the virtual window to the position (0,0), or move the focus point to the center of the graph.

Additionally, there is the item **Position** in the mouse menu that allows to change the absolute position of the virtual window, and the item **Pick Position** which allows to select the new origin of the coordinate system by mouse picks, the item **Center Node** which centers a node whose title was entered in the virtual window, and the item **Follow Edge** that allows to follow an edge to its start or end point.

The operation **Pick Position** has two modes: New origins (focus points for fisheye views, see 5.8) can be selected continuously by short left mouse clicks. This operation continues until a right mouse button is selected. This is helpful to browse through the graph with fisheye view, e.g., to move the focus point over all points of interest. However, if the left mouse button is pressed, hold and drawn over the window, a rubberband appears. In this case, not only the origin is set but also a scaling is calculated such that just the region of the rubberband is magnified to fit into the window. Selecting regions by this way does not continue as for the short mouse clicks. It stops directly.

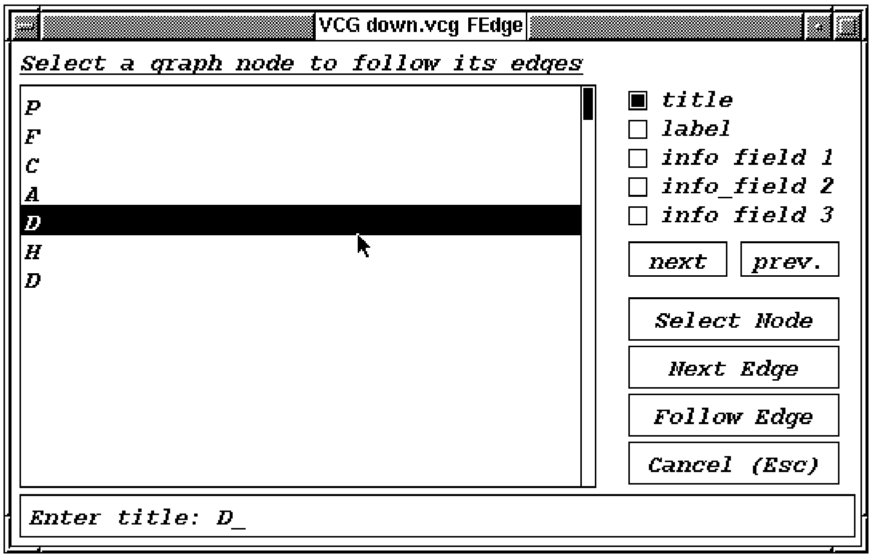


Figure 41: The Follow Edge History Box

The operation **Follow edge** works as follows: first one node must be selected by the mouse, then one edge that starts or ends at this node is chosen by mouse button clicks. The other end point of the edge is centered. Now, an edge of the end point can be selected by button clicks to center a new end point, etc. The operation stops if the right mouse button is selected at the end point. This method is also helpful to browse through the graph. However, how the find the way back to a node where the operation has been started ? To support this, a history is implemented. By pressing the key 'h' during the **Follow edge** operation, the history dialog box appears (see fig. 41) and shows all nodes that have been centered during the **Follow edge** operation. Selecting a node using the button "Select Node" browses back and centers this node (or sets the focus point to it), touching the button "Next Edge" allows to select the next edge to follow, and touching the button "Follow Edge" allows to follow the edge.

The selection of the menu item **Ruler** gives a hint of the current position of the virtual window: Horizontal and vertical rulers are switched on or off at the margins of the displayed window to display the coordinates. Of course, this does not work for fisheye views, since the coordinate system is distort.

5.5 Node Information

This submenu contains several points that allow to see more information about the nodes and the graph.

- **Info 1, Info 2, Info 3**: The name of these items can be selected as attribute `infoname` in the specification, otherwise the item numbers "1", "2", and "3" appear. After the selection of nodes, their info fields are displayed. The info fields can be used to provide the nodes with additional textual information that would be too large as labels of the nodes.

- **Layout Attributes**: After the selection of nodes, their layout attributes are shown. This includes the attributes of the specification, but also the calculated position. If a horizontal or vertical order was specified, it may happen that this order was corrected, because a level was not possible for a node or the layout algorithm failed to validate the specified horizontal orders. In this case, we see for instance the entry `1 -> 5` which means that the order was specified to be 1, but was corrected by the layout algorithm to the value 5.

- **Label of Node** displays the label of a node in normal size. This is useful, if the graph is shrunken very much, such that the label text is not readable because it is too small.

- **Statistics** shows the statistics of the graph, which includes the size of the graph, the number of nodes and edges, the number of crossings etc.

5.6 Scaling

Additionally to the right scrollbar, the submenu **Scale** is used to scale the current visualization up and down. It sets the global values of *stretch* and *shrink*:

Item	*New stretch*	*New shrink*
normal	1	1
400 %	stretch * 4	shrink
200 %	stretch * 2	shrink
150 %	stretch * 3	shrink * 2
90 %	stretch * 9	shrink * 10
80 %	stretch * 8	shrink * 10
70 %	stretch * 7	shrink * 10
60 %	stretch * 6	shrink * 10
50 %	stretch	shrink * 2
25 %	stretch	shrink * 4

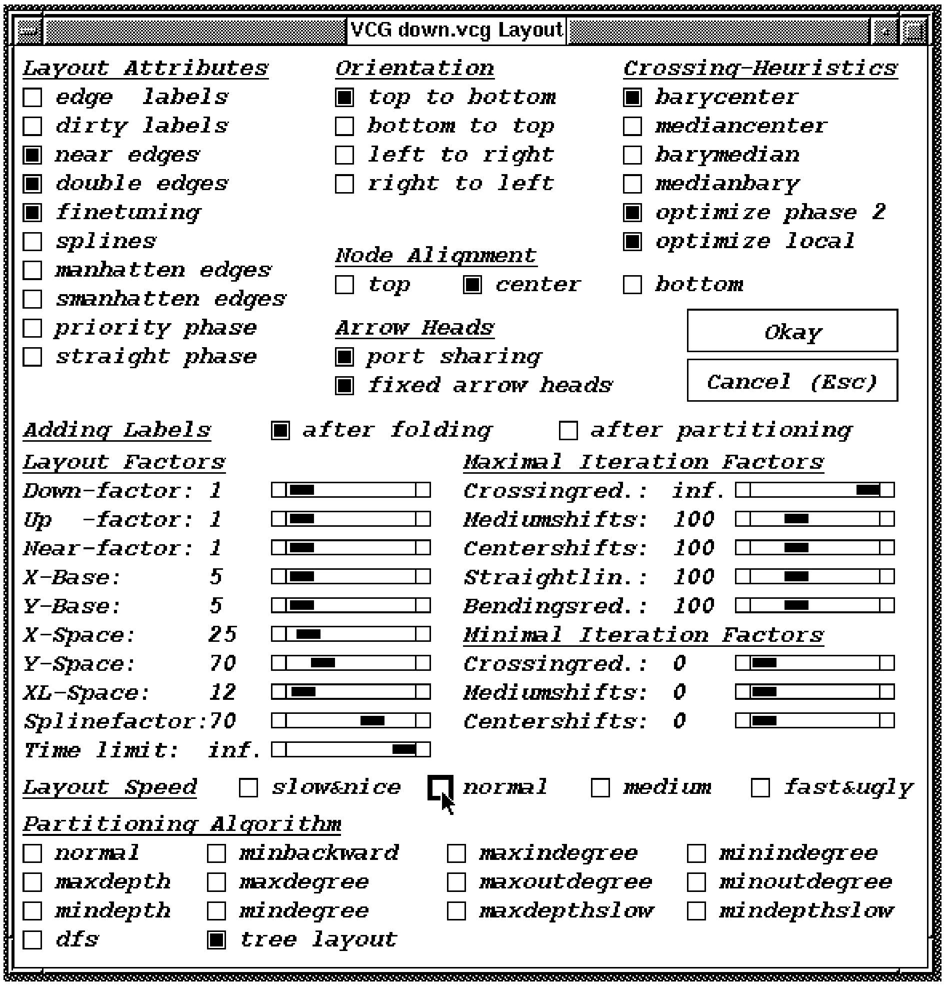


Figure 42: The Layout Parameter Box

5.7 Layout Parameters

After the selection of the menu item **Layout**, a dialog box appears (see fig. 42). Here, we can select the way and whether edge labels are drawn, the orientation of the graph (see attribute `orientation`), the crossing reduction method (`barycenter` weights if the degree of the nodes is high, `mediancenter` weights if the degree is small, or one of the hybrid methods `barymedian` or `medianbary`; the crossing reduction phase 2 or the local optimization phase can also be switched on or off), the node alignment (see attribute `node_alignment`), the mode for the arrow heads (see attribute `port_sharing` and `arrow_mode`) and the layout algorithm. The attribute `late_edge_labels` corresponds to the selection of the point "Adding labels ... after partitioning". Further, we have access to all layout factors by some scrollbars: for instance, if the graph is too dense, we set `xspace` and `yspace`, if splines are too sharp, we reduce the spline factor and increase `xlspace`, if the layout iteration phases run into timeouts, we increase the maximal number of iterations, etc.

The layout parameters become valid, if the dialog box is closed by selecting the "Okay" button. This yield a relayout of the graph. On button "Cancel", the old parameters remain valid.

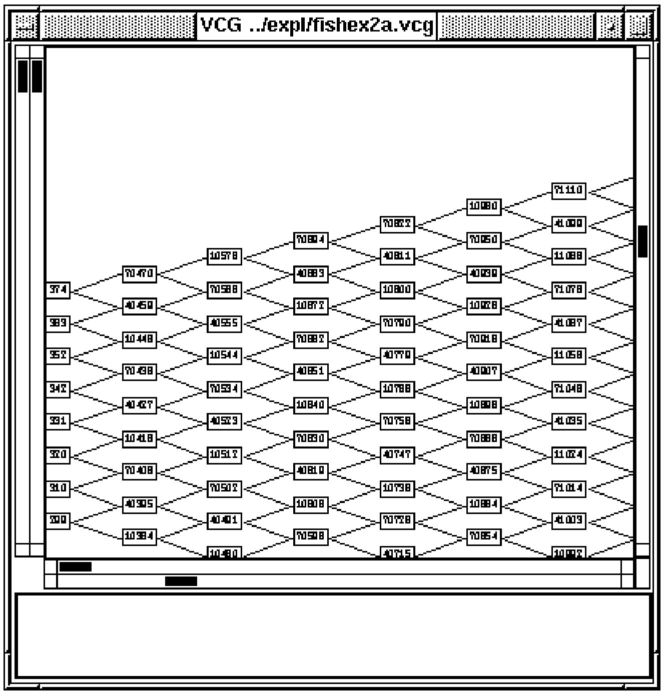


Figure 43: Normal Flat View

5.8 View Parameters

After the layout, we have a view onto the graph. The "view" is the way how the graph appears: Normally, it appears in the window that realizes a flat coordinate system with linear scale (see fig. 43). Unfortunately, large graphs do not fit well into a small window such that the normal view either shows the full graph demagnified such that no details are visible, or it shows a small region in an appropriate magnification such that the node labels are readable. But then, only a part of the graph is visible and the structure of the whole graph and the relations between this part and the remaining graph is not recognizable.

The idea of a solution of this conflict is to distort the coordinate system. The main point of interest is the focus point. It is magnified such that its label is readable. Parts far away from the focus point are demagnified. Thus, the whole graph or at least a very large part is visible.

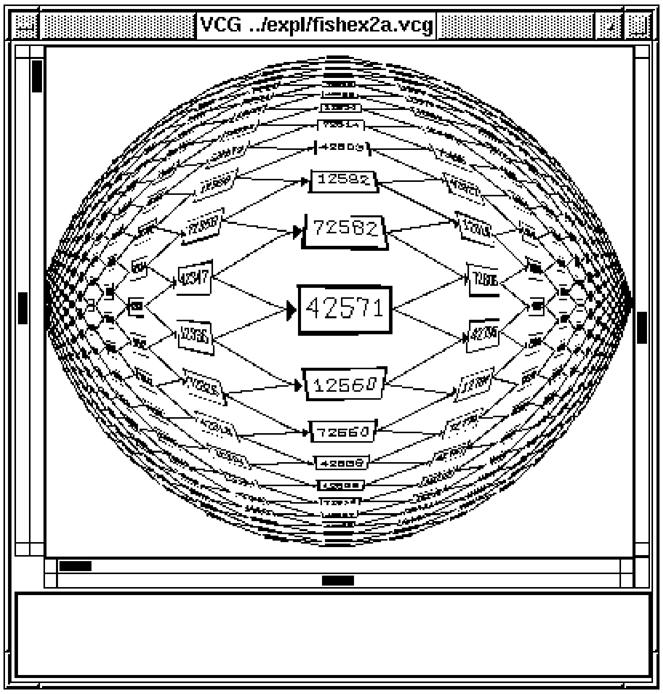


Figure 44: Polar Fisheye View

This mechanism has similarities with the fisheye camera lenses in the photography. The polar fisheye view (fig. 44) is a coordinate transformation where the plane of the normal coordinate system is projected onto a spheric ball. If we look onto this ball in 3 D, we have a polar fisheye view. The point most near to us looks very large; it is the focus point. It appears in the magnification that is currently valid due to the right scrollbar setting or one of the scaling operations. Points near the border of the visible half of the ball are shown very small. A polar fisheye view has also disadvantages: it distorts the graph, thus distances between the nodes, angles between the edge segments, and even straightness of lines are not anymore recognizable. Since the drawing of lines is optimized, it may even happen that we see a crossing of lines when there is no crossing in the plane view. But these cases are very seldom.

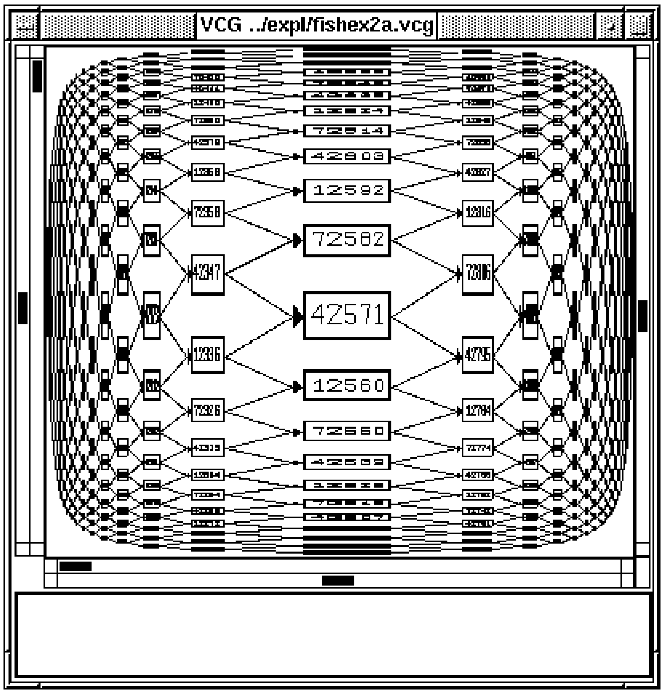


Figure 45: Cartesian Fisheye View

The cartesian fisheye view (fig. 45) is a similar projection. The polar fisheye is a transformation of the polar coordinate system, while the cartesian fisheye is a transformation of the cartesian coordinate system. The advantage is: in a polar view, horizontal and vertical lines do not appear orthogonal, they seem to be bend. In a cartesian view, they are still drawn as parallel horizontal and vertical lines. Since important forms of nodes and also the orthogonal layout (see attribute `manhattan_edges`) contain many orthogonal lines, this improves the readability.

The browsing through a fisheye view is the moving of the focus point. This can be done by the command **Pick Position** and the various other positioning operations that allow to set the origin in the flat view. We have two different modes for fisheye views:

Self adaptable fisheyes The whole graph is visible The distortion scale of the fisheye is automatically adapted to the actual fisheye, such that the graph just fits into the window. The position where the focus point appears in the window is calculated from the position of the focus point in the graph, i.e., e.g., if the focus is set to the upper left

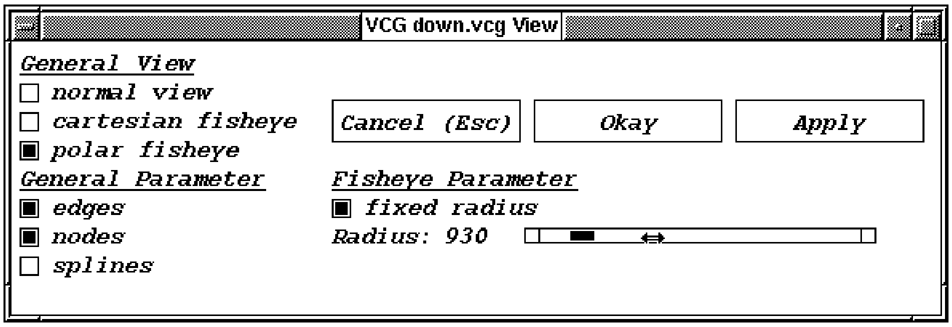


Figure 46: The View Parameter Box

corner of the graph, it will also appear in the upper left corner of the window. This helps to keep the orientation when browsing through the graph.

Fisheyes with a fixed radius If the graph is even too large for a self adaptable fisheye, this mode may be useful. Here, not the whole graph is visible but only a region of a fixed radius around the focus point. In this case, the focus point is always centered in the graph window.

The view parameters can be selected by a dialog box (fig. 46). Here, not only the mode of the fisheye can be chosen, but also whether edges or nodes generally should appear, or whether splines should be used to draw edges.

5.9 File Operations

There is a submenu with the following items:

- **Save to File** writes the graph with all calculated layout parameters into a file. The result is a valid GDL specification that can be read by the **VCG** tool.
- **Export Part**: after selecting a region to be exported, an image is saved in monochromatic PBM-P4 format, colored PPM-P6 format, or PostScript. For PostScript, multiple page output up to 25 pages is possible, too. Note: if we use splines, it is only possible to export the whole graph.
- **Export Graph**: The whole graph is exported, just similar as above.
- **Load**: a new GDL-file is read and the described graph is displayed.
- **Reload**: the actual GDL-file is read again and the described graph is displayed. This does not work if the actual file is `<stdin>`.

To export an image, it is useful first to shrink the graph to a size that the part to export is completely visible. With a rectangular rubberband, this part is selected. Hint: If a corner of the part is too close to the corner of the window, it is more comfortable to open the rubberband at the opposite corner and to draw it over to corner of the window. The selection by rubberband is not necessary, if the whole graph is exported.

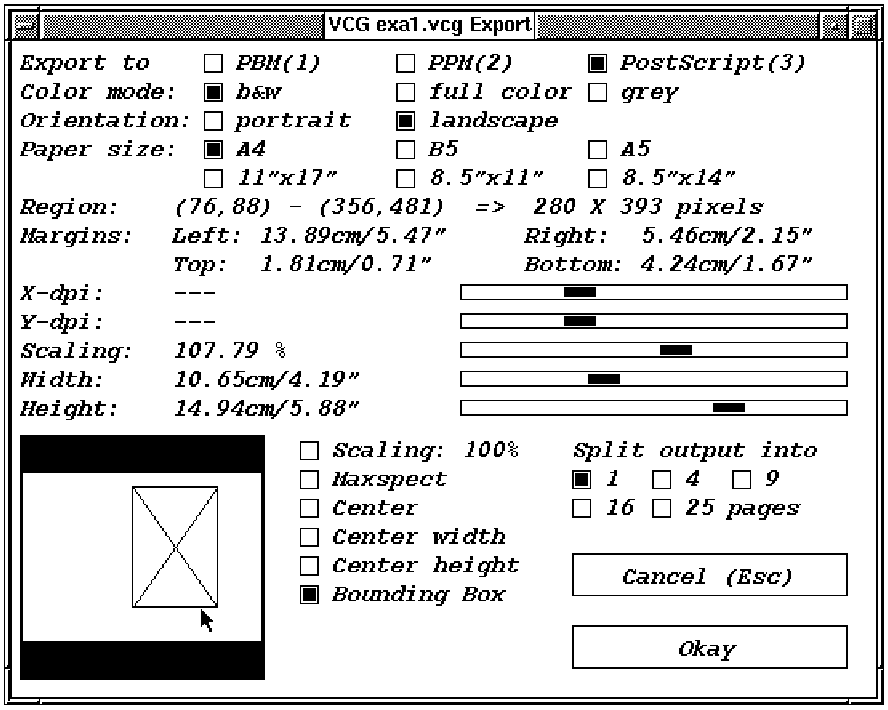


Figure 47: The Export Box

Now, a dialog box is opened that allows to select the format, scaling, size and position of the image (see figure 47). Basically, this export mechanism is designed to create files that can be printed. PBM and PPM are bitmap formats that often create rather large files. (Example: A din A4 page at 300 dpi needs in PBM format nearly 1 MB, and in PPM about 24 MB.) However, there are many printer drivers for PBM and PPM format in the world. PBM is a monochromatic format (`b&w`) and PPM is a color format. PostScript is an image description language that can be used to create colored images, grey scaled images and monochromatic images. PostScript images can be split into many pages, such that it is possible to dispatch a very large graph onto several pages, in order to avoid that the labels of nodes becomes unreadable small.

After selecting the format, the paper size and orientation, and perhaps the number of PostScript pages, the size and position of the images must be selected. For the bitmap formats, the dpi-factor of the printer must be selected first, because the size depend on this factor. The factors x-dpi and y-dpi are independent of each other, such that it is possible to distort the images with these factors. This is necessary for printing with the usual 9-dot printers, which often have different horizontal and vertical resolutions. Next, we prefer to select the buttons “Scaling: 100%” if the image should be normal size, or “Maxspect”, if the image should be maximal large. The scrollbars “Scaling”, “Width” and “Height” are combined scrollbars. Changing one of them influences the others, to preserve the aspect ratio. The image is maximal wide, if the scrollthumb is at the right side of the bar labeled with “Width”, and maximal heigh, if the scrollthumb is at the right side of the bar labeled with “Height”.

To position the image on the paper, we move the small rectangular that has diagonals within the panner, or select one of the buttons “Center”, “Center width” and “Center height”. On PostScript multipage output, the position cannot be changed.

5.10 The File Selector Box

On all file operation, a file selector box appears to help the selection of a file name (see figure 48). Additionally to the file name, a file info is shown that may be the size of the file, the access mode, the creation date, the owner or the group. The file entries in the box can be sorted by names, or by this file info, and can be preselected by different name extensions like `*.vcg`, `*.ps` etc. The directories are always shown as file entries, where ‘`.`’ indicates the actual directory and ‘`..`’ indicates the parent directory. To switch into a directory, a double mouse click on the corresponding entry is necessary. Alternately, a path can be specified, which becomes valid if the button “Rescan” is selected to reread the file name entries.

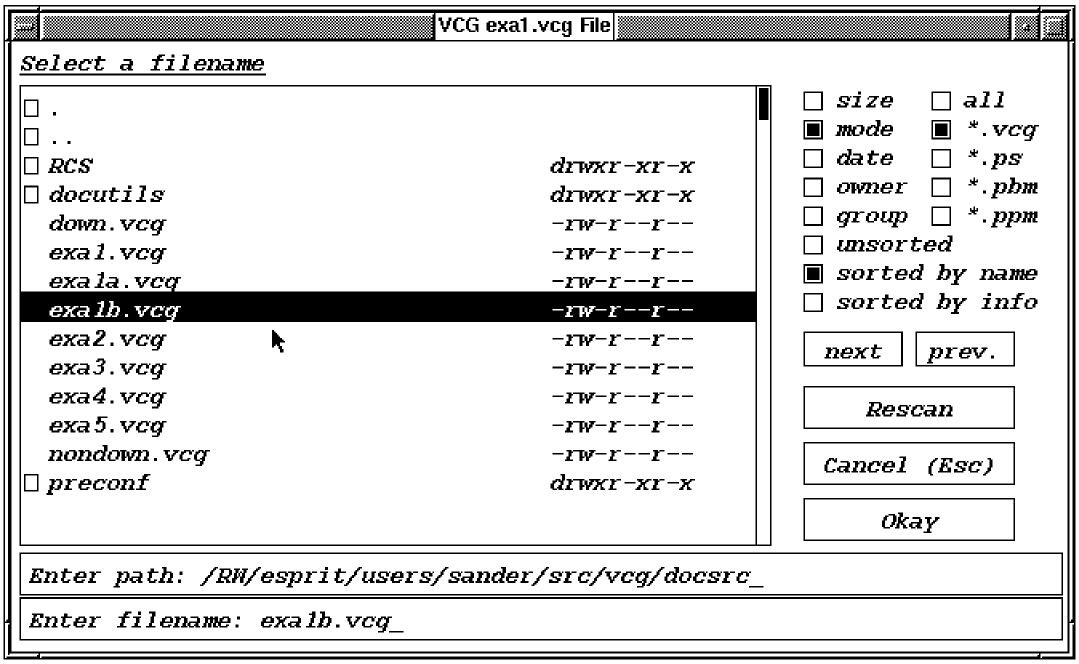


Figure 48: The File Selector Box

To scroll through the list of file name entries, a scrollbar and the buttons “next” and “prev” are available. An entry is selected as file name on a double mouse click on it. The file name becomes valid if the dialog box is closed by the button “Okay”. Note that the file selector box is immediately reopened, if the file name was not appropriate, e.g., if we try to read a nonexisting file or to write to an existing file.

normal commands	
q	quit the tool
r	show or hide the ruler
f	load another file
g	reload the same file
l	change layout
v	change view
1 . . . 9	hide/expose the corresponding edge class
i	show the info field 1 of nodes
I	show the info field 2 of nodes
j	show the info field 3 of nodes
position commands	
a d (arrow keys) c b	scroll to the left/right/up/down
o	go to the origin (0,0)
P	enter a position by coordinates
p	pick a position by the mouse
n	position such that a node is centered
e	follow an edge
scaling commands	
+ or =	stretch
- or _	shrink
0 *(null)*	set the scale factor to normal

Table 8: Key Commands in the Graph Window

5.11 Animations

On some computer systems, there is a simple possibility to implement animations: The signal USRSIG1 (on SunOs: UNIX software signal 30, e.g., `kill -30`) causes the tool to reload the actual GDL-file. An engine (or some other program) can continuously produce GDL-specifications into a file while **VCG** visualizes in parallel according to this file. When the engine has produced one instance of output, it sends the signal USRSIG1 to the tool. The tool then displays the new instance of the graph. Depending on the option used to start **VCG**, the tool indicates the completion of the visualization of a reload by touching its input

l	switch edge labels on or off
d	switch dirty edge labels on or off
s	set slow and nice layout
n	set normal layout
m	set medium layout
f	set fast and ugly layout
o	optimze crossing phase 2
1	set top to bottom orientation
2	set bottom to top orientation
3	set left to right orientation
4	set right to left orientation
7	set node alignment to top
8	set node alignment to center
9	set node alignment to bottom
`RETURN`	quit the dialog box
`ESC`	cancel the dialog box

Table 9: Key Commands in the Layout Dialog Box

v	select normal view
c	select cartesian view
p	select polar view
e	switch edges on or off
n	switch nodes on or off
s	switch splines on or off
f	switch fixed radius on of off
`RETURN`	quit the dialog box
`ESC`	cancel the dialog box

Table 10: Key Commands in the View Dialog Box

file to create a new time stamp, or by sending signal USRSIG1 to the caller. The signal USRSIG2 (on SunOs: UNIX software signal 31) causes the tool to close its main window. It is recommended to use this simple animation mechanism only if the engine produces a GDL-description with fixed layout (i.e. all nodes have attributes `loc`).

5.12 Keyboard Commands

The most used commands are available on key press. Which commands are available depends on the window/dialog box that is currently open. If it is not ambiguous, the uppercase and lowercase keys have the same functionality. Only the pairs I/i and P/p must be distinguished

1	switch edge class 1 on or off
2	switch edge class 2 on or off
3	switch edge class 3 on or off
4	switch edge class 4 on or off
5	switch edge class 5 on or off
6	switch edge class 6 on or off
7	switch edge class 7 on or off
8	switch edge class 8 on or off
9	switch edge class 9 on or off
`RETURN`	quit the dialog box
`ESC`	cancel the dialog box

Table 11: Key Commands in the Edge Class Selection Dialog Box

1	PBM output format
2	PPM output format
3	PostScript output format
f	full color
g	greyscale
b	black and white
l	orientation: landscape
p	orientation: portrait
s	scaling: 100 %
m	scaling: maxspect
c	center the position
q	quit the dialog box
`RETURN`	quit the dialog box
`ESC`	cancel the dialog box

Table 12: Key Commands in the Export Dialog Box

on the graph window. During the selection of nodes, all key commands are switched off except q, a, b, c, d (arrows), o, +, - and 0. See the tables 8, 9, 10, 11, 12, 13, and 14.

5.13 Speedup the Layout

The **VCG** tool was designed to explore large graphs. However, the layout of large graphs needs a lot of time. Thus, there are many possibilities to speedup the layout algorithm: the graph can be folded, iterations can be limited, and time limits can be specified.

The first step to visualize a large graph is to select the parts of the graph that are currently not of interest. We specify these parts as initially folded. Folding makes the remaining visible

s	additional info: size
m	additional info: mode
d	additional info: date
o	additional info: owner
g	additional info: group
u	order of entries: unsorted
b	order of entries: sorted by name
i	order of entries: sorted by info
a	entry selection: all
v	entry selection: `*.vcg`
- or p	scroll entry list up
+ or n	scroll entry list down
r	rescan entry list
q	quit the dialog box
`RETURN`	quit the dialog box
`ESC`	cancel the dialog box

Table 13: Key Commands in the File Selector Box

t	show titles
l	show labels
1	show info fields 1
2	show info fields 2
3	show info fields 3
c	show coordinates
- or p	scroll entry list up
+ or n	scroll entry list down
a	apply current selection
q	quit the dialog box
`RETURN`	quit the dialog box
`ESC`	cancel the dialog box

Table 14: Key Commands in the Title Selector Box and Follow Edge History Box

graph smaller, thus the layout can be calculated faster and the quality of the layout is better. It is of course useful first to try the fast algorithms (`dfs`, `minbackward`, `tree`), then the medium fast methods (`normal`, `mindepth`, `maxdepth`, ...) before the slow methods (`mindepthslow`, `maxdepthslow`).

If the **VCG** tool is still too slow, we must omit some phases or limit the iteration factors. This decreases the quality of the layout: the picture will be more ugly. First, we should try to skip the crossing reduction phase 2 (option `-nocopt2`, attribute `crossing_phase2`). It probably takes the most time, which we can recognize if the message character 'B' appears a long time. Next, we should try to limit the iterations for the crossing reduction (option `-cmax`, attribute `cmax`) or try to select another crossing weight (option `-bary`, etc., attribute `crossing_weight`). Normally, it is not necessary to switch off the local crossing optimization, because this step is very fast and effective.

If the graph is very unbalanced, then the pendulum method probably needs a lot of time. We can recognize this if the message character 'm' appears a long time. In this case, we limit the iterations for the crossing reduction (option `-pmax`, attribute `pmax`). If the message character 'S' does not disappear immediately after the pendulum method, then we limit the straight line fine tuning phase instead (option `-smax`, attribute `smax`).

The other parameters normally need not to be changed, because the corresponding phases are very fast. In particular, bending reduction improves the layout quality much and is so fast, such that the option `-bmax` is needed very seldom. Further, the fast mode (option `-f`) should be avoided, because it reduces the iteration limits so much that the result is very ugly. The drawing of splines is very slow. Thus it should be avoided on large graphs.

If we don't want to deal with the exact iteration limits, we can set a time limit (option `-timelimit`). If the time limit is exceeded, the fastest possible mode for the actual iteration phase is switched on. The time limit does not mean that the layout really needs so much time: The layout may be faster, because the graph structure is very simple, but more often, it will be slower, because even the fastest possible methods already exceed the time limit. The time limit is only a hint for the **VCG** tool. Another problem: the time limit is real time, thus the result of the layout with time limit depends on the load of the computer. Thus, given a time limit, two identical trys need not to give identical results.

6 Experiences

Compiler graphs are usually a rather large network of interwoven graphs. Often, there is one base graph and a lot of subgraphs that are annotations of the base graph. Not all aspects of a compiler graph are of interest at the same time: either an overview of the graph is needed, or some details are inspected. In the first case, the graph must be laid out completely and nice, i.e. edges must be straight, nodes must be centered, etc. The user normally has shrunken the graph very much. But in the first case, not all nodes must be drawn, large parts like annotations can be folded, because only the main structure is of interest. In the second case, the readability of the complete layout is not important. The layout may be ugly, but the user only looks at small regions, and has stretched the graph to see the details. The **VCG** tool provides reasonable facilities to support both situations, it can even combine both situations using fisheye views. The layout algorithm can be controlled to be fast and ugly, or slow and nice. Folding allows to reduce the number of information seen at the same time. If the user looks at details, there are possibilities to find nodes and edges that are currently not in the window. The user can influence the structure of the interwoven graphs by near edges, anchor points and priorities. Nevertheless, visualization of large graphs is a difficult task and needs a lot of time.

Table 15 shows the performance of the **VCG** tool on a Sun Sparc ELC. The "Time for Loading" includes the start of the tool, loading of the specification, automatical layout and drawing. The measurements are done by hand. The speed is reasonable.

The examples are:

- Graph 1 shows a LR deterministic automaton produced by the TrafoLa parser generator (see [HeSa93]).
- Graph 2 is a larger LR deterministic automaton.
- Graph 3 is an all graph, i.e. all nodes are connected pairwise.
- Tree 1 is a syntax tree of a CLaX program (normal layout, not specialized tree layout).
- Tree 2 is a syntax tree with annotations (normal layout, not specialized tree layout).
- Tree 3 is a large syntax tree with annotations (normal layout, not specialized tree layout).
- Tree 4 is a binary tree of level 12 (normal layout, not specialized tree layout).

Example	|*Nodes*|	|*Edges*|	*Time for Loading (sec.)*	*Time for Positioning (sec.)*
Graph 1	12	35	3	1
Graph 2	131	287	9	*
Graph 3	20	190	22	1.5
Tree 1	154	153	2.5	*
Tree 2	614	613	10	4
Tree 3	2763	2762	24	3
Tree 4	4095	4094	30	*

Table 15: Statistics: Times for Loading and Positioning
* means "not measurable", i.e. less than 1 sec.

7 Related Work

During the work in the project COMPARE, we tested some visualization tools and algorithms. Each of these tools has certain advantages and disadvantages, but none of these tools combined speed on large graphs with an appropriate folding mechanism. Nevertheless, these excellent tools gave us many inspirations.

We mentioned already the **Edge** tool (see [PaTi90], [MaPa91]), that has the most common features with the **VCG** tool. Another visualization tool that works similar than the **VCG** tool or the **Edge** tool is **daVinci** (see [FrWe93]). This X11 tool reads a specification of a graph written in the style of a functional language. The DAG tool and the DOT tool are described in [GNV88], [GKNV93] and [KoEl91]. They allow the production of high quality graphs for printing. The **Graph**^{Ed} (see [Hi93]) is a graph editor that includes a large collection of algorithm to create, analyze and lay out graphs interactively. **D-ABDUCTOR** is described in [Mis94]. This tool is very powerful for the visualization of compound graphs.

8 Conclusions

VCG is a tool that allows to visualize complex graphs in a compact way and in good performance. It can be used to show the compiler functionality to prepare presentations and to help on compiler debugging. The GDL specification language of the tool is general such that the tool can be adapted to many applications.

The tool is intended to be used in combination with a program system that produces graphs. It is not an interactive graph editor. The algorithms to lay out the graph are rather simple and use heuristics, but they are very fast. Thus, the visualization of a graph may differ from the intuition, but these cases were seldom in our experiences. The layout algorithms can still be improved (see [BET94]). Usually, the readability of large graphs is improved by the layout algorithm. We believe that the tool is a good compromise between performance and legibility of the visualization.

There is a mailing list `vcg-users@cs.uni-sb.de` that distributes mail to all users of the VCG tool. If you want to be added to this list, please send a request to `sander@cs.uni-sb.de`. Then, you will be informed about bugs and new versions of the tool.

References

[BET94] Di Battista, G.; Eades, P.; Tamassia, R.; Tollis I. G.: "Algorithms for Drawing Graphs: an Annotated Bibliography", Computational Geometry: Theory and Applications, no. 4, pp. 235-282 *available via ftp from* `ftp.cs.brown.edu`, *file* `/pub/papers/compgeo/gdbiblio.tex.Z`, 1994

[FrWe93] Fröhlich, Michael; Werner, Mattias: Das interaktive Graph Visualisierungssytem daVinci V1.2, technical report (in German), University of Bremen, Germany, Fachbereich Mathematik und Informatik 1993

[GKNV93] Gansner, Emden R.; Koutsofios, Eleftherios; North, Stephen C.; Vo, Kiem-Phong: A Technique for Drawing Directed Graphs, IEEE Transactions on Software Engineering, Vol. 19, No. 3, pp. 214-230, March, 1993

[GNV88] Gansner, Emden R.; North, Stephen C.; Vo, Kiem-Phong: DAG – A program that draws directed graphs, Software – Practice and Experience, Vol. 17, No. 1, pp. 1047-1062, 1988

[HeSa93] Heckmann, Reinhold; Sander, Georg: Trafola-H Reference Manual, *in* Hoffmann, Berthold; Krieg-Brückner, Bernd, Editors: Program Development by Specification and Transformation, Lecture Notes in Computer Science 680, pp. 275-313, Springer Verlag 1993

[Hi93] Himsolt, Michael: "Konzeption und Implementierung von Grapheditoren", Dissertation (in German), Universitaet Passau, Germany, 1993 *published with* Berichte aus der Informatik, Verlag Shaker, Aachen 1993

[KoEl91] Koutsofios, Eleftherios; North, Stephen C.: Drawing graphs with dot, technical report, AT&T Bell Laboratories, Murray Hill NJ, 1992

[MaPa91] Manke, Stefan; Paulisch, Frances Newbery: Graph Representation Language: Reference Manual, 1991

[Mis94] Misue Kazuo: D-ABDUCTOR 2.30 User Manual, Institute for Social Information Science, Fujitsu Laboratories Ltd, 1994

[PaTi90] Paulisch, Frances Newbery; Tichy, W. F.: "EDGE: An Extendible Graph Editor", Software, Practice & Experience, vol. 20, no. S1, pp.63-88, 1990

[Sa94] Sander, Georg: Graph Layout throught the VCG Tool, technical report A03/94, Universit"at des Saarlandes, FB 14 Informatik, 1994 *available via ftp from* `ftp.cs.uni-sb.de`, *file* `/pub/graphics/vcg/doc/tr-A03-94.ps.gz`

[Sa95] Sander, Georg: Graph Layout throught the VCG Tool, *in* Tamassia, Roberto; Tollis, Ioannis G., Editors: Graph Drawing, DIMACS International Workshop GD'94, Lecture Notes in Computer Science 894, pp. 194 -205, Springer Verlag 1995

[STM81] Sugiyama, Kozo; Tagawa, Shojiro; Toda, Mitsuhiko: Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics SMC-11, No. 2, pp. 109-125, Feb. 1981

[SPG86] SunView Programmer's Guide (Revision A of 17 February 1986)

[WiMa92] Wilhelm, Reinhard; Maurer, Dieter: Übersetzerbau: Theory, Konstruktion, Generierung, Springer Verlag 1992

[Pet91] Peterson, Chris D.: X11 Athena Widget Set – C Language Interface (Revision notes to X11 R5, 1991)

Index