aiSee

Graph Visualization

User Manual for Windows and Linux – Version 3.4.3

AbsInt Angewandte Informatik GmbH
aisee@absint.com
http://www.absint.com

Release: Version 3.4.3
Date: 22 December 2009
Status: Release
Reference: aiSee-manual-absint

Copyright notice:
© 2008 by AbsInt Angewandte Informatik GmbH

All rights reserved. This document, or parts of it, or modified versions of it, may not be copied, reproduced or transmitted in any form, or by any means, or stored in a retrieval system, or used for any purpose, without the prior written permission of AbsInt Angewandte Informatik GmbH.

The information contained in this document is subject to change without notice.

LIMITATION OF LIABILITY:

Every effort has been taken in manufacturing the product supplied and drafting the accompanying documentation.

AbsInt Angewandte Informatik GmbH makes no warranty or representation, either expressed or implied, with respect to the software, including its quality, performance, merchantability, or fitness for a particular purpose. The entire risk as to the quality and performance of the software lies with the licensee.

Because software is inherently complex and may not be completely free of errors, the Licensee is advised to verify his work where appropriate. In no event will AbsInt Angewandte Informatik GmbH be liable for any damages whatsoever including – but not restricted to – lost revenue or profits or other direct, indirect, special, incidental, cover, or consequential damages arising out of the use of or inability to use the software, even if advised of the possibility of such damages, except to the extent invariable law, if any, provides otherwise.

AbsInt Angewandte Informatik GmbH also does not recognize any warranty or update claims unless explicitly provided for otherwise in a special agreement.

Contents

1 Introduction

aiSee is a tool that automatically calculates a customizable layout of graphs specified in GDL (Graph Description Language). This layout is then displayed, and can be interactively explored, printed, and exported to various formats.

aiSee reads a textual and human-readable graph specification and visualizes the graph. Its design has been optimized to handle large graphs automatically generated by applications (e. g. compilers).

aiSee was developed to visualize the internal data structures typically found in compilers. Today it is widely used in many different areas:

- Genealogy (family trees, evolution diagrams, pedigrees)
- Business management (organization charts, process diagrams)
- Bioinformatics (gene expression graphs, metabolic and signaling pathways, consensus genetic maps, protein interaction maps, shared GeneOntology annotations, literature co-citation relations)
- Software development (call graphs, control flow graphs, validation of hardware traces)
- Hardware design (circuit diagrams, finite state diagrams)
- Database management (entity relationship diagrams)
- Informatics (finite state automata, algorithm visualization)
- Web design and Web site optimization (sitemaps, visitor movement graphs)
- Network analysis (P2P networks, IRC networks, social networks, the Internet)
- Linguistics (syntax trees, grammar graphs, term variant graphs)
- Criminology (fraud networks, felony event flow diagrams)

There are many applications that offer GDL interfaces. Some of them are freely available on the Internet. Visit www.aisee.com/apps for further information.

2 Overview

The *Usage* section contains the basics the user needs to know for using **aiSee**. Upon completing it the user is able to do the following:

- Navigate through graphs
- Scale them
- Access additional information associated with nodes
- Print graphs

The *Advanced Usage* section describes subgraphs, regions of nodes, and User Actions. Subgraphs and regions are used to structure graphs, thereby facilitating interactive exploration of a graph. User Actions are used for communication with other applications.

The *GDL* section is written for programmers who want to produce graph specifications from their application program which can then be visualized using **aiSee**.

The *Overview of the Layout Phases* section provides an introduction to the internals of **aiSee** and the underlying graph layout algorithms.

3 Usage

This chapter describes the basics of using **aiSee**.

3.1 Starting aiSee

Under Windows, **aiSee** can be invoked either from the command line in a window running the MS-DOS command interpreter or like any other Windows program, e.g. from the **Start**–>**Programs** pop up menu, from the **Start**–> **Run** menu, or by double-clicking on the **aiSee** program icon.

Under Linux, **aiSee** can be invoked either from the command line, or by (double-)clicking on its icon in a graphical file browser.

3.1.1 Calling aiSee from the Command Line

aiSee can be called without any graph specifications, with a single specification or with a list of specifications.

```
aisee3 [options] [filename]
```

For the available command line options, see p. 23.

3.2 Exiting aiSee

aiSee is quit by pressing **Ctrl** + **W** on the keyboard or by choosing **Close** in the **File** menu. If you have several **aiSee** windows open, this will only close the currently active window. To close all **aiSee** windows at once, use **Ctrl** + **Q** or choose **Quit** in the **File** menu.

3.3 aiSee Window

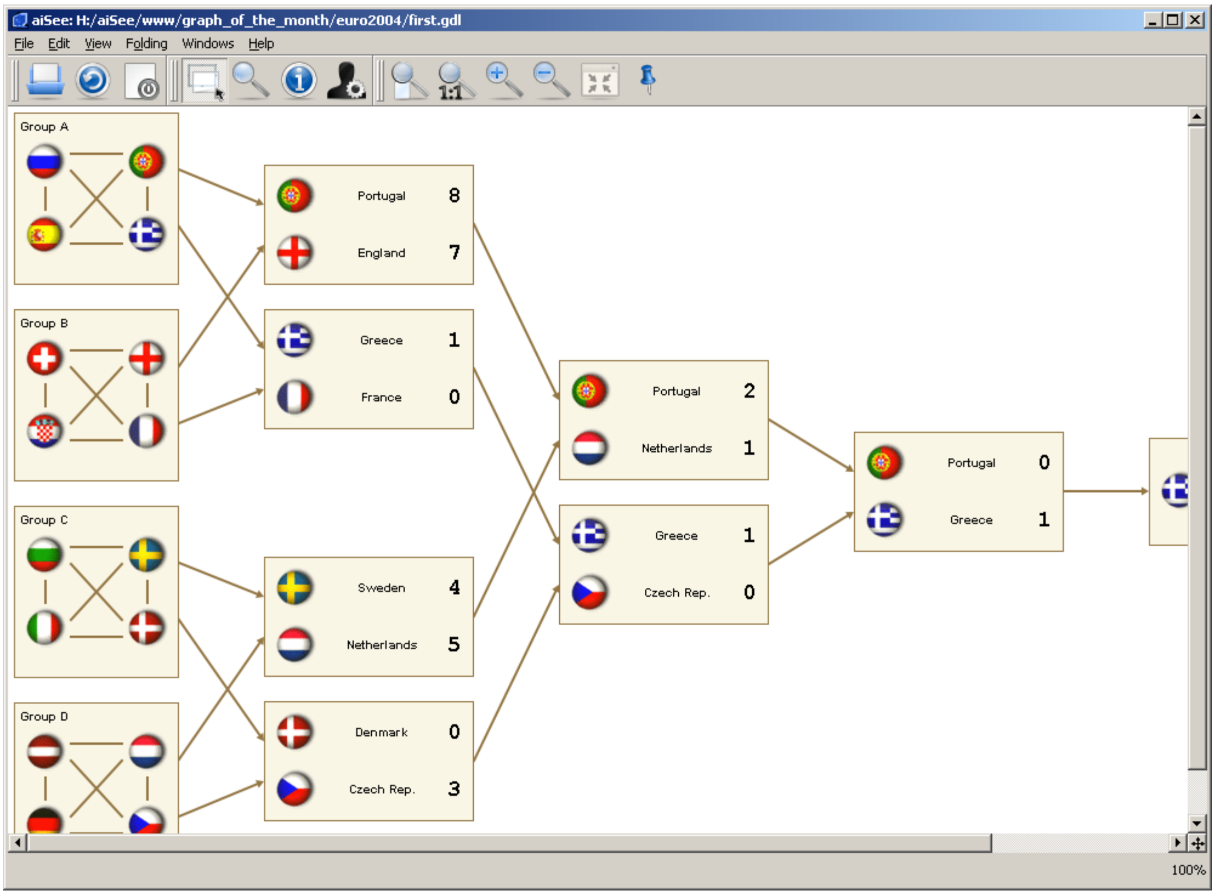


Figure 3.1: **aiSee** Window

The **aiSee** window consists of a title bar, a menu bar, a toolbar, a graph window (with scrollbars, when needed), and a status line below the graph window where short messages are printed.

3.4 Usage Modes

aiSee offers several main modes for exploring a graph or interacting with it.

- Selection Mode for selecting a node, several nodes, or a graph area
- Scaling Mode for zooming in and out
- Node Information Mode for viewing additional information associated with nodes
- User Actions Mode for communication with other applications

3.5 Navigating Through a Graph

There are several different ways to navigate through a graph.

3.5.1 Keys

The easiest way to explore a graph is by using the arrow keys. The display window is moved over the graph a few pixels in the corresponding direction.

For faster scrolling, use **PgUp**, **PgDown**, **Home** and **End**. Alternatively, you can use the arrow keys while holding down the **Shift** key.

The **o** gets you to the graph origin, i.e. it positions the upper left corner of the display window in the upper left corner of the graph.

3.5.2 Mouse Pointer

In all modes, you can drag-and-drop the graph by using the *middle* mouse button. When the button is pressed and held followed by moving the mouse, the graph moves in the same direction as the mouse until the button is released.

Instead of the middle mouse button, you can also hold down the **<** or the **>** key on your keyboard.

3.5.3 Scrollbars (Fine-Tuning)

The vertical and horizontal scrollbars for the graph window are used to scroll the graph window over the graph. The scrollbars are enabled automatically whenever the scaling factor is such that the graph does not completely fit into the graph window.

3.5.4 Panning Mode for Exploring Large Graphs

Another method of exploring large graphs is to use the Panning Mode.

This mode allows you to temporarily zoom out so that the entire graph fits the window. The part of the graph that has been zoomed out of is marked by a highlighted rectangle, while the rest of the graph is shown dimmed. Dragging the highlighted rectangle using the mouse enables you to quickly move to a different place in the graph. Left-clicking then allows you to zoom back in.

The Panning Mode is accessible either via the corresponding button in the toolbar or via the small button in the lower right corner of the graph window.

To quit the Panning Mode, either left-click to zoom into the currently highlighted part of the graph, or press **Esc** to return to your original position prior to entering the Panning Mode.

3.5.5 Following Edges

When working with very large graphs where the nodes are connected by long edges it is useful to navigate through the graph by taking advantage of the graph structure. This is done by following edges through the graph.

The **Follow Edge** operation works like this:

1. Select a node and press **e**. One edge that starts or ends at this node is highlighted.
2. Press **Tab** or use the mouse wheel to highlight the next edge, until the edge you wish to follow is highlighted.
3. Press **e** or right-click to follow the highlighted edge, i.e. to center the node at the other end of the edge in the graph window.
4. From the new node, another edge can be selected to be followed in the same manner.

Alternatively, you can start the **Follow Edge** by selecting an edge. Pressing **e** will then highlight the node at the end of the edge that is farther away from your current position. You can then proceed to follow further edges in the manner described above, by using **Tab** or the mouse wheel to select an edge and pressing **e** to follow it.

3.5.6 Show Neighbors

Select a node and press the **n** key to show all direct neighbors of that node — connected by a visible, invisible or hidden edge.

Hidden edges are not to be confused with invisible edges. Invisible edges are edges that are present in the current layout but have their `linestyle` set to `invisible` in the GDL specification. Hidden edges are edges that are present in the GDL specification, but not in the current layout. Edges can be hidden because edge classes can be hidden or because of folded or boxed subgraphs (see subgraphs on p. 20).

Each neighbor is annotated by an arrow head in order to distinguish outgoing edges from incoming ones.

Click on a node in the results list to center and mark that node in the main **aiSee** window. If the node is inside a folded subgraph, that subgraph will be centered and marked instead (but not unfolded).

You can also double-click on a node in the results list to list its neighbors.

3.5.7 Searching for Nodes

Press **Ctrl** + **F** to open the **Find dialog** box.

Click on a node in the results list to center and mark that node in the main **aiSee** window. If the node is inside a folded subgraph, that subgraph will be centered and marked instead (but not unfolded).

You can also double-click on a node in the results list to list its neighbors.

3.6 Scaling the Graph

If not specified otherwise in the graph specification, **aiSee** starts with an absolute scale factor of 100% (referred to as “normal”). To scale a graph one of the following three methods can be used.

3.6.1 Keys

The easiest way to scale a graph is by using one of the following keys in the graph window:

Key	Scale factor
+	141 % of the current scale factor
−	70 % of the current scale factor
0[1]	Sets the scale factor to normal (100%).
m	Maximum aspect: the scale factor is set so that the whole graph is entirely visible.

Another way to scale the graph is by using the zoom bar in the bottom right corner of the window. (To easily scale to exactly 100displayed to the left of the bar.)

3.7 Markers

Markers allow you to temporarily save your current position over the graph along with the current scaling factor for quick access later. This is especially useful for jumping between different points of interest in a huge graph.

Note: markers are lost once you quit **aiSee**, reload the current graph or load a new one.

- To set a new marker, press **.** (the period key).
- To see a list of markers that have been set, press **F3**. The marker list will show up in a separate window. Here, you can:
  - Jump to a marker (i.e., load the position and scaling factor represented by that marker in the main **aiSee** window) by double-clicking on it.
  - Rename a marker by performing two single left clicks on its name, followed by typing in a new name.
  - Remove a marker from the list by selecting it with a single left click and using the button **Remove**.
- To quickly jump between markers, you can also use the dropdown menu in the lower right corner of the **aiSee** window.

---

[1] This is the number zero.

3.8 Node Information

Apart from a node label, up to three pieces of additional information, i. e. text can be associated with a node.

The Node Information Mode enables this information to be accessed.

- To switch to this mode, click on the corresponding icon in the **aiSee** toolbar or simply select a node and press **i**.
- Clicking in an open information window causes it to close again.
- Leaving the Node Information Mode causes all information windows to be temporarily hidden but not closed, i.e., once you switch back to the Node Information Mode, the same information windows will be automatically opened again.

3.9 File Operations

All file operations are supported by a file selector dialog box to select a file name.

The submenu **File** offers the following file operations:

- **Open... (Ctrl + o)**
  Opens the file selector box to select a file name. A new graph specification is read from the file selected, a layout calculated, followed by the graph being displayed. Note: the graph will be opened in a separate **aiSee** window.
- **Reload (g)**
  This operation causes the file containing the current graph specification to be read again, the layout recalculated and the graph displayed again. It can also be invoked by pressing the **g** key.
- **Print...**
  The **Export** dialog box is opened.

3.9.1 File Selector Dialog Box

A file selector dialog box appears for all file operations.

To switch to a directory or select a file, double-click on the corresponding entry using the left mouse button.

4 Advanced Usage

4.1 Grouping of graph elements

Graphs consist of nodes and edges. **aiSee** offers various ways of grouping nodes and edges. Grouping is done either statically, i. e. in the graph specification, or dynamically, i. e. interactively. If not stated otherwise, the operations mentioned in this section are all located in the **Folding** menu of the menu line.

4.1.1 Grouping of edges

A group of edges is a set of edges belonging to the same edge class. The user con expose/hide edges by enabling/disabling edge classes. Edges of enabled edge classes and target nodes are drawn in the graph window. Edges of disabled edge classes and all nodes accessible via these edges are not drawn and are not considered in the layout calculation of the graph.

4.1.2 Grouping of nodes

A group of nodes is either

- A subgraph
- A region specified by paths (path region), or
- A region of neighboring nodes (neighbor regions)

A group of nodes can be treated in a special way. It can be:

- Folded into a summary node,
- Shown in a box,
- Shown as a cluster,
- Wrapped, or
- Represented exclusively.

A summary node is usually drawn in a special shape and/or background color so as to be recognized as a summary node. It represents all the nodes that are hidden by the folding operation. A summary node can be:

- unfolded,
- unfolded into a box,

- unfolded and represented as a cluster, or
- unfolded and wrapped.

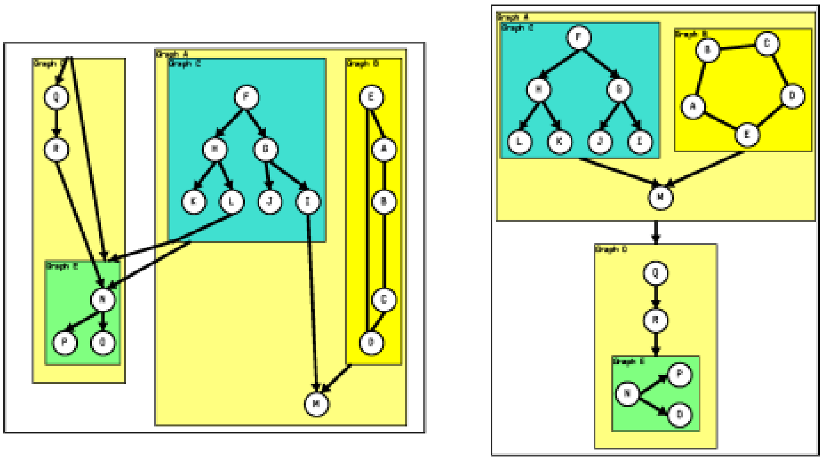
aiSee 3 so far only supports these operations for subgraphs, but not for regions.

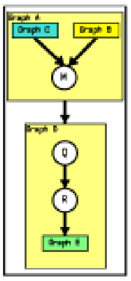
4.1.3 Example

Figure 4.1 shows a nested graph in different grouping and folding stages. This graph consists of five subgraphs: Graph A to Graph E. Graph B and C are subgraphs of Graph A. Graph E is a subgraph of Graph D.

1. All graphs are clustered. In this case it is not possible to choose a different layout for subgraphs.
2. All graphs are boxed. Boxed graphs are drawn in independent frames. Edges between nodes in different frames are substituted by edges between frames. For example, edge I–>M is represented by the substitute edge Graph C–>M and edge L–>Graph E is represented by substitute edge Graph A–>Graph D.

   Note: Different layout algorithms or layout parameters can be chosen within boxes. For example, graph B is drawn in a force-directed layout, graph C in a tree layout with a top down orientation and graph E right-left-oriented.
3. The frames of graphs B, C, and E are now folded. Nodes inside these frames, i. e. the nodes of the corresponding subgraphs, disappear.
4. The frames of graphs A and D are unfolded and the formerly substituted edges are shown. Edge Graph C–>N is represented by the substitute edge Graph C–>Graph E, since N is a node of the folded graph E.
5. The summary nodes of graphs B, C, and E are unfolded. No frames and summary nodes any longer exist in this rendering. Therefore edges like Graph E–>N are substituted. Substituted edges are introduced that start/end at the root node of a frame/subgraph. For example, edge Graph E–>N is now represented by the substitute edge F–>N since F is the root node of graph C.
6. Graphs A and D are boxed again, but graphs B, C, and E remain unfolded.
7. Graphs B, C, and E are wrapped. This operation doesn't change the layout.

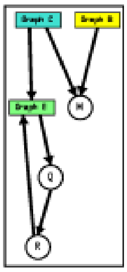




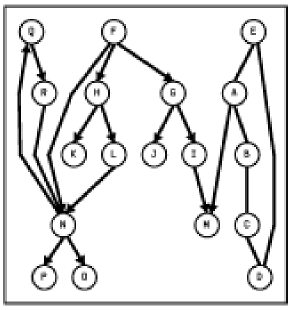
1) All subgraphs clustered

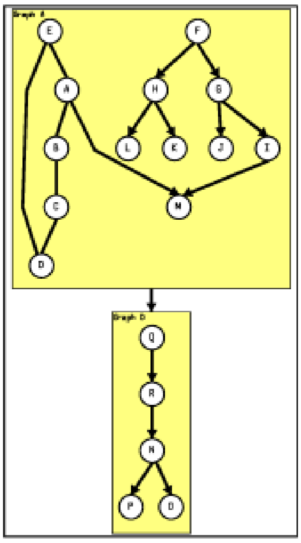
2) All subgraphs boxed

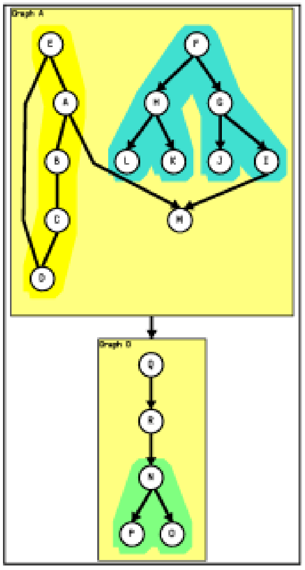
3) Graph B, C and E folded



4) Graph A and D folded







5) All subgraphs unfolded

6) Graph A and D boxed

7) Graph B, C and E wrapped

Figure 4.1: Several Representations of Groups of Nodes

4.1.4 Subgraph

A graph can be partitioned into nested subgraphs. Subgraphs can be defined only statically, i. e. in the graph specification. In the GDL specification, subgraphs should be specified in such a manner so that the user can identify subgraphs and summary nodes and thus make use of all the operations that work on subgraphs.

Operations: Subgraphs can be

- Folded into a summary node (**f**),
- Nested, i.e. unfolded into a box (**b** or double-click on the subgraph),
- Nested recursively (**Shift** + **b** or **Shift** + double-click).

4.2 Representation of Groups of Nodes

A group of nodes (a subgraph, path region or neighbor region) can be

- Folded into a summary node,
- Presented in a box,
- Represented as a cluster,
- Wrapped, or
- Represented exclusively.

4.2.1 Folding

The nodes belonging to a group are hidden. They are represented by a single node called the summary node. Summary nodes are usually drawn in a different shape and/or color, so that they can be recognized and selected for the reverse operation.

Targets
The fold operation can be chosen for a

- Subgraph (menu item **Fold Subgraph** or press the **f** key),
- Box (menu item **Fold Box**).

These operations are accessible from the context menu that appears upon a right-click on a node or subgraph.

Summary nodes
A summary node represents all the nodes of a group that was folded dynamically, i.e. by a folding operation, or statically, i.e. in the graph specification. A summary node can be selected and

- Unfolded into a box (**b** or double-click on the subgraph)
- Unfolded recursively (**Shift** + **b** or **Shift** + double-click)

4.2.2 Box

A boxed group of nodes is surrounded by a frame, i. e. drawn in a nested box. The nodes inside the box are independent of the rest of the graph, i. e. there are no edges connecting nodes outside of the box with nodes inside the box or vice versa. The layout for the graph inside a box can differ from the layout of the outer graph.

The layout calculation for graphs with boxes is fast since the nodes inside the box are independent of the rest of the graph and the layout of the box is calculated separately.

Targets
The box operation can be chosen for a

- Subgraph (menu item **Box Subgraph**),
- Path region (menu item **Box Region...**), or
- Neighbor region (menu item **Box Neighbors...**).

These operations are accessible from the context menu that appears upon a right-click on a node or subgraph.

Reverse Operation
The reverse operation is **Unfold/Unbox** (or press the **u** key).

4.2.3 Cluster

A clustered group of nodes is surrounded by a frame. In contrast to a box, edges from nodes outside the frame are drawn to nodes inside the frame and vice versa.

Here the layout is calculated as if the graph were unfolded, the only difference being that the nodes are placed according to the restrictions induced by the common frame. This often results in rather poor layout quality, layout calculation frequently being slow on account of the restrictions. Consequently, the boxed layout is preferable whenever possible.

Note: Clustering is an **experimental** feature. The layout of clustered subgraphs is quite challenging by nature. **aiSee** includes an experimental implementation of clustered layout which is currently not supported nor maintained. Nevertheless, many **aiSee** users find this feature quite useful.

Note: In this version, clustering cannot be enabled via the user interface, but only in the GDL specification of a graph.

4.2.4 Wrapping

All nodes and edges belonging to a group are wrapped using the same color. The layout is not changed.

Note: Nested wrapping is not supported.

Note: In this version, wrapping cannot be enabled via the user interface, but only in the GDL specification of a graph.

4.2.5 Summary Nodes

A summary node appears after a folding operation. It represents all the nodes grouped by the folding operation.

Usually summary nodes can be distinguished from normal graph nodes (e. g. they are drawn in a different shape and/or color) so that they can be recognized as summary nodes. Then they can be selected and unfolded.

A summary node can be

- Unfolded (menu item **Unfold/Unbox** or press the **u** key),
- Unfolded into a box (menu item **Unfold into Box** or press the **x** key),
- Unfolded and represented as a cluster (menu item **Unfold into Cluster**, press the **t** key),
- Unfolded and wrapped (menu item **Unfold and Wrap** or press the **w** key),
- Drawn exclusively (menu item **Exclusive**).

4.3 Command Line Options

- `--export` *`filename`*
  Directly exports the computed layout into the file named *`filename`*. Interactive displaying of the graph is turned off. The file type is automatically derived from the file extension. For example, `--export` *`out.svg`* directly exports the graph to SVG. Currently, the following formats are supported:
    - SVG
    - PNG
    - PS
    - PDF
    - XPM
    - BMP
- `--scale` *`N`*
  Scales the graph to *`N`* percent.
- `--area` *`W`*`x`*`H`*`+`*`X`*`+`*`Y`*
  When used in combination with `--export`, this option enables an image part to be exported instead of the entire image. The image part to be exported is specified by a bounding rectangle with the width *`W`* and the height *`H`*, while *`X`* and *`Y`* specify an offset from the graph origin, i.e. from the upper left corner of the layout.

4.4 User Actions

The concept of user actions allows to execute system commands from within **aiSee**. The User Actions Mode enables these commands to be accessed. Currently, only the User Action 3 is supported that requires one node to be selected as argument. To execute the command specified in the GDL specification, select a node and press **3**.

One improvement in **aiSee** 3 is that the command to be executed no longer has to be specified globally for the entire graph. Much rather, a different command can be specified for each particular node. In other words, there is now an attribute **useractioncmd3** for nodes.

Support for User Actions 1, 2 and 4 has been dropped in the current version. Instead, **aiSee** now implements more sophisticated communication via TCP.

4.4.1 Communication

aiSee 3 can be started in Server Mode by using the command line option `--server` *`port number`*.

Communication is done on a per line basis, i.e. a command followed by a newline character is sent to the server.

Upon a successful operation, the server returns ”OK” or the requested string. Otherwise, an error message prefixed by ”ERROR: ” is returned.

4.4.2 Commands

- `open` *`name`*
  Load the file named *`name`*, lay out the graph and display it in the current window.
- `opennew` *`name`*
  Load the file named *`name`*, lay out the graph and display it in a new window.
- `attach` *`name`*
  Connect to the window named *`name`*.
- `close`
  Close the current aiSee window.
- `exit`
  Completely exit aiSee (close all windows).
- `fit`
  Set the scale factor so that the entire graph fits in the graph window (maximum aspect).
- `zoom` *`factor`*
  Set the scaling factor to *`factor`* percent.
- `resize` *`width height`*
  Resize the aiSee window. The parameters specify the width and the height of the display window in pixels, respectively.

- `move` *`x y`*
  Move the aiSee window. The parameters specify the position of the window in relation to the root screen in pixels, i.e. the x and y coordinates of the upper left corner of the window.
- `center` *`x y`*
  Center the coordinates *`x y`* in the graph window.
- `scroll` *`dx dy`*
  Scroll the window over the graph by *`dx`* and *`dy`* pixels.
- `center` *`title`*
  Center the node titled *`title`* in the graph window.
- `set` *`title attr value`*
  Set the value of the attribute *`attr`* of the graph item titled *`title`* to *`value`*. So far, the following attributes are supported:
  - *`color`*
  - *`textcolor`*
  - *`bordercolor`*
  - *`shape`*
  - *`label`*

  For the available values, see www.aisee.com/manual/unix/47.htm and www.aisee.com/manual/unix/45list.htm#shape
- `get` *`title`* `[$]`*`attr`*
  Retrieve the value of the attribute *`attr`* of the node titled *`title`*. If the attribute is prefixed with a $, the attribute value is returned as a string (if possible).

  Attributes retrievable so far: cf. ”set” command.
- `signal` *`event`*
  Request certain events to be reported. So far, a left-click on a node is recognized (in certain Usage Modes). If the user clicks on the node, its title prefixed by ’LC ’ is reported.
- `nosignal` *`event`*
  Do not report events.
- `update`
  Redraw graph.
- `autoupdate [on|off]`
  Update graph after each command. The default value is ¡on¿. If autoupdate is disabled, send an ”update” command after modifications.
- `contextmenu clear`
  Removes all entries from the user context menu.
- `contextmenu add` *`text id`*
  Adds an entry to the user context menu. The *`text`* may not contain spaces, use underbars instead, aiSee will later replace them with spaces.
- `dialog` *`command`* `[arguments]`

Used to create and display small dialog boxes

- `dialog information` *`text`*
  `dialog question` *`text`*
  Open a dialog box and display text.
    - ’information’ only displays an OK button,
    - ’question’ displays a YES and a NO button

  Returns ”OK” if OK or YES resp. was clicked, otherwise ”ERROR: canceled”.
- `dialog create` *`name title`*
  Start creation of a dialog with reference *`name`* and title *`title`* and enter dialog mode.
- `dialog show` *`name`*
  Display a dialog and wait for user interaction. Returns ”OK” if OK was clicked, otherwise the reason for closing prefixed by an ”ERROR: ”.
- `dialog get` *`name field`*
  Return the value of the field *`field`* of the dialog *`name`*.
- `dialog set` *`name field value`*
  Set the value of the field *`field`* of the dialog *`name`* to *`value`*.
- `dialog clear` *`name field`*
  Clear contents of the field *`field`* in the dialog *`name`*.

  While in dialog mode, the following commands are recoqnized:
- `end`
  Exit dialog mode
- `add` *`type field`*
  Add field of type *`type`* to the dialog named *`field`*. Known field types:
    - `label`
    - `lineedit`
    - `listbox`
    - `mlistbox`
    - `spinbox`
- `break`
  Start a new row of widgets.
- `clear` *`field`*
  Remove the contents of *`field`*
- `set` *`field value`*

Add value to the `field`'s contents.

∗ for ”lineedit” and ”label”, the old value will be replaced by the new one.

∗ for ”listbox” and ”mlistbox”, the new value will be appended to the old one.

4.5 Reducing Layout Time

aiSee was designed to explore large graphs. However, the layout of large graphs may require considerable time. Thus, there are many possibilities to speed up the layout algorithm: the graph can be folded, iterations can be limited, and time limits can be specified.

The first step in visualizing a large graph is avoiding computing the layout of parts of the graph that are currently not of interest. These parts should be specified as initially folded in the specification. Folding makes the visible part of the graph smaller, thus enabling the layout to be calculated faster and the quality of the layout improved. It is useful to first try one of the fast algorithms (*dfs*, *minbackward*, *tree*), then the medium fast methods (*normal*, *mindepth*, *maxdepth*, ...) before resorting to the slower methods (*mindepthslow*, *maxdepthslow*).

In order to further reduce layout time, some layout phases should be omitted or the maximum number of iterations of some layout phases limited. However, this usually decreases the quality of the layout. First, the crossing reduction phase 2 (option `-nocopt2`, attribute **crossing_phase2**) should be skipped as it usually takes the most time. Crossing reduction phase 2 is indicated by `B` in the message window (see p. **??**). Next, the iterations for the crossing reduction can be limited (`-cmax` option, **cmax** attribute) or another crossing weight selected (`-bary` option, etc., **crossing_weight** attribute). Normally, it is not necessary to switch off local crossing optimization, because this step is quite fast and effective.

If the graph is very unbalanced, the pendulum method may require considerable time (indicated by `m` in the message window). In this case, the number of iterations for the crossing reduction should be limited (`-pmax` option, **pmax** attribute). If the straight-line fine-tuning phase takes too long (indicated by `S` in the message window), its maximum iteration number should be limited (`-smax` option, **smax** attribute).

Other parameters usually needn't be changed, because the corresponding phases are quite fast. Bending reduction in particular improves layout quality greatly and is so fast that the `-bending` option is hardly needed.

Fast mode (`-fast` option) drastically reduces *all* iteration limits and may result in poor layout quality.

The drawing of splines is slow, consequently it should be avoided on large graphs.

It is also possible to set a time limit (`-timelimit` option). If the time limit is exceeded, the fastest possible mode for the current and the following iteration phases is selected. Selecting a time limit does not mean that the layout has actually finished when the time has elapsed: Layout may be faster if the graph is small and/or its structure is simple. It may be much slower, because even the fastest possible methods may take some time. The time limit is real time, thus the result of the layout may depend on the load of the computer. Thus, given a time limit, two identical trials needn't result in identical layouts.

5 Graph Description Language (GDL)

GDL is an ASCII text representation of a graph. It describes a graph in terms of nodes, edges, subgraphs and attributes. A subgraph is described as a normal graph except that it is specified inside another graph, meaning graph specifications can be nested. **aiSee** provides special operations for subgraphs such as folding (see p. 20) to a summary node (see p. 22) , boxing (see p. 21) , clustering (see p. 21) , or wrapping (see p. 21) .

Graphs, nodes and edges may have attributes that specify details of their appearance on the screen such as colors, sizes, shapes etc.

There is always only one top-level graph.

It is also possible to specify regions that are to be initially folded after starting **aiSee**.

aiSee also accepts the `#line` directives of the C preprocessor. The macro processing facilities of the C preprocessor offers some tricky possibilities for graph specifications. For example, using a macro directive would enable different languages for node labels to be chosen.

5.1 Graph Format

A graph (the top-level graph or a subgraph) is specified by

`graph:  {`
  $<$ *list of graph_entries* $>$
`}`

A *graph_entry* is one of the following:

- *`subgraph`* (see p. 29) , i. e. a subgraph is specified in the same manner a top-level graph is specified. There is no special keyword for subgraphs.
- *`node`* (see p. 30)
- *`edge`* (see p. 30)
- *`attribute`* (see p. 32)
- *`region`* (see p. 33)

The delimiter between items of a list is one or more whitespace characters (spaces, tabs, line feeds, carriage returns). There is no order required for nodes, edges, subgraphs, or attributes. However, depending on the layout algorithm selected, the order of nodes may influence the layout.

5.2 Node Format

A node is specified by

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

The title of a node is any valid C string. Strings have to be enclosed in quotes and may contain the normal C escapes (e. g. `\"`, `\n`, `\f`, ...).

5.3 Edge Format

There are several different kinds of edges:

- ordinary edges (see p. 30)
- back edges (see p. 30)
- near edges (see p. 31)
- left near edges (see p. 31)
- right near edges (see p. 31)
- bent near edges (see p. 31)
- left bent near edges (see p. 32)
- right bent near edges (see p. 32)

5.3.1 Ordinary Edge Format

An ordinary edge is specified by

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

5.3.2 Back Edge Format

Back edges are drawn in the opposite direction as compared to ordinary edges. For instance, if the layout algorithm tries to give all ordinary 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 have to be reversed. In this case, the layout algorithm prefers back edges before selecting any other edge to be reversed.

A back edge is specified by

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

5.3.3 Near Edge Format

Near edges are drawn so that their source and target nodes are directly neighbored at the same level. This implies that a node cannot have more than two near edges, i.e. one on the left and one on the right with respect to the orientation of the graph. Near edges are drawn as short horizontal lines not crossed by any other edges or nodes.

Hint: Invisible near edges can be used to group nodes at one level.

A near edge is specified by

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

Left Near Edge Format: Same as near edge, only that the target node is placed directly neighbored to the left of the source node at the same level.

```
leftnearedge:  {
   source: < title of source node >
   target: < title of target node >
   < list of edge attributes >
}
```

Right Near Edge Format: Same as near edge, only that the target node is placed directly neighbored to the right of the source node at the same level.

```
rightnearedge:  {
   source: < title of source node >
   target: < title of target node >
   < list of edge attributes >
}
```

5.3.4 Bent Near Edge Format

Bent near edges consist of a horizontal part (same as near edges), a bend point and a vertical part. An edge label (if any) is placed only at the bend point. So bent near edges connect nodes across one or more levels yet leave the source node at the left or right side.

A near edge is specified by

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

Left Bent Near Edge Format: Same as bent near edge, only that the horizontal part of the edge leaves the source node on the left.

```
leftbentnearedge:  {
   source: < title of source node >
   target: < title of target node >
   < list of edge attributes >
}
```

Right Bent Near Edge Format: Same as bent near edge, only that the horizontal part of the edge leaves the source node on the right.

```
rightbentnearedge:  {
   source: < title of source node >
   target: < title of target node >
   < list of edge attributes >
}
```

5.4 Attribute Format

Attributes are specified as follows:

< *attribute keyword* > : < *attribute value* >

All attributes are optional except for the `title` attribute of nodes and the `source` and `target` attribute of edges. The order of attributes is irrelevant.

The attribute value depends on the attribute and is one of the following:

- String, e. g. `title: "My family tree"`
- Integer, e. g. `width: 700`
- Floating point number, e. g. `scaling: 1.2`
- Keyword, e. g. `shape: rhomb`
- Combination of the above, e.g. `fontname: "helvB10"`
- File name, e.g. `iconfile: "myicons/butterfly.ppm"`

5.4.1 Default Node and Edge Attributes

It is possible to specify default values for all node or edge attributes in a top-level graph or subgraph. The default attributes are specified as follows:

- `node.`< *attribute keyword* > `:` < *attribute value* >
- `edge.`< *attribute keyword* > `:` < *attribute value* >

These default specifications can appear anywhere in a top-level graph or subgraph specification and apply to all nodes and edges (including back edges, near edges and bent near edges) respectively where the corresponding attribute is not set explicitly. This default specification applies until another default specification for the same attribute appears or until the end of the subgraph or top-level graph specification is reached.

5.4.2 Default Summary Node and Edge Attributes

Regions of nodes and edges can be folded (see p. 17). As a result, a summary node is displayed for all nodes of a region of nodes. In addition, summary edges to this summary node are displayed for sets of edges to nodes of the region. It is possible to specify the attributes for these summary nodes and edges that come from a folding operation. This allows folded regions to be given a different appearance than normal nodes and edges. The attributes for these summary nodes or replacement edges are specified as follows:

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

5.5 Region Format

It is not only possible to fold regions dynamically, i. e. interactively with the mouse, but also statically, i. e. in the specification of the graph.

The format of a region specification is:

```
region:  {
  state: < state of the region (folded, boxed, exclusive) >
  source: < list of strings specifying start nodes >
  target: < list of strings specifying end nodes >
  class: < list of integers specifying edge classes >
  range: < integer specifying the neighbor region range >
}
```

The delimiter between items of a list is one or more whitespace characters. The attributes **target**, **class** and **range** are optional.

range is used to specify a neighbor region, in which case it gives the maximum number of edges to be followed in order to reach all the nodes of the neighbor region (see p. **??**).
A region cannot have a range and target nodes specified at the same time.

5.6 Examples of GDL Specifications

5.6.1 A Cyclic Graph

These examples show a number of small cyclic graphs without any labels but with different edge types. The titles are displayed in the nodes. The last example in this sequence shows how the **anchor** edge attribute is used.

The layout of the following three example specifications are shown in Figure 5.1.

Example 1: Ordinary Edges
aiSee tries to give all edges the same orientation. But since the graph is cyclic, one edge has to be reverted (edge D–>A).

```
01 graph:  {
02   // list of nodes
03   node:  { title: "A" }
04   node:  { title: "B" }
05   node:  { title: "C" }
06   note:  { title: "D" }
07   node:  { title: "E" }
08   // list of edges
09   edge:  { source: "A"  target: "B" }
10   edge:  { source: "A"  target: "C" }
11   edge:  { source: "C"  target: "D" }
12   edge:  { source: "D"  target: "E" }
13   edge:  { source: "D"  target: "A" }
14 }
```

Example 2: Back Edge
The edge to be reverted can be specified as a back edge.

```
11   backedge:  { thickness: 3  source: "C"  target: "D" }
```

Example 3: Near Edges
Near edges can be used to express a close relationship between two nodes and to place nodes right- or left-neighbored.

```
09   nearedge:  { source: "A"  target: "B" }
10   nearedge:  { source: "A"  target: "C" }
12   nearedge:  { source: "D"  target: "E" }
```

The layout of the following three example specifications are shown in Figure 5.2.

Example 4: Bent Near Edge
In some situations, edges are to be horizontally anchored, like near edges, yet the target node is not to be at the same level. These edges have to have a bend point. This is why bent near edges are used.

```
09   bentnearedge:  { source: "A"  target: "B" }
```

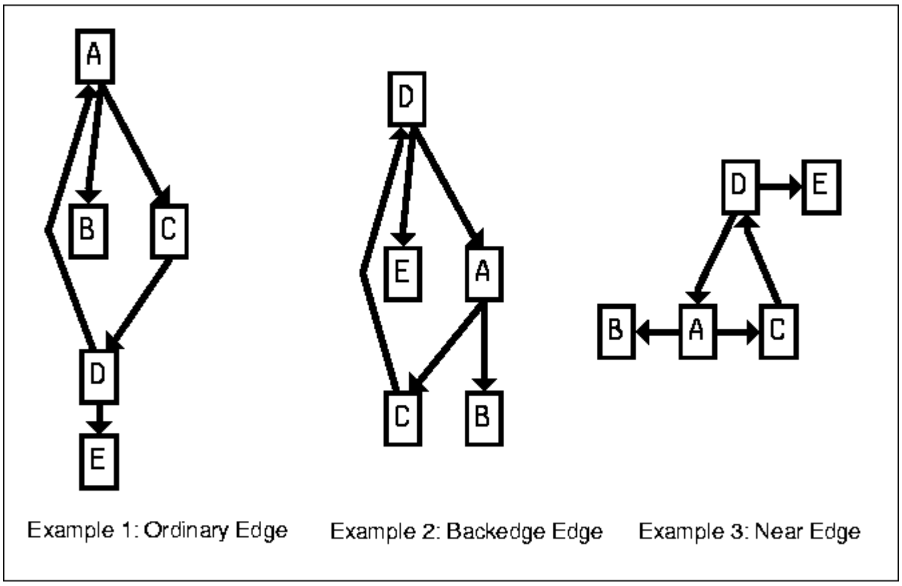


Figure 5.1: Ordinary, Back and Near Edges

```
bentnearedge:  { source: "D"  target: "E" }
```

Example 5: Right-Bent Near Edge
The left or right versions of bent near edges or near edges can be used if it is important to have edges anchored to a particular side of a node.

```
rightbentnearedge:  { source: "A"  target: "B" }
leftnearedge:  { source: "A"  target: "C" }
rightnearedge:  { source: "D"  target: "E" }
```

Example 6: Anchor
The **anchor** edge attribute can be used if a node label consists of one or more fields that are on separate lines where outgoing edges are to start. For example, if node D represents a struct with two fields whose first field is to point to node E and the second one to node A, then the edges are anchored to the corresponding anchor points 1 and 2. Counting of anchor points starts with 1 at the top of a node and increases from top to bottom.

Anchor points and near edge specifications cannot be used together, consequently the specification for Example 6 looks like this:

```
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
```

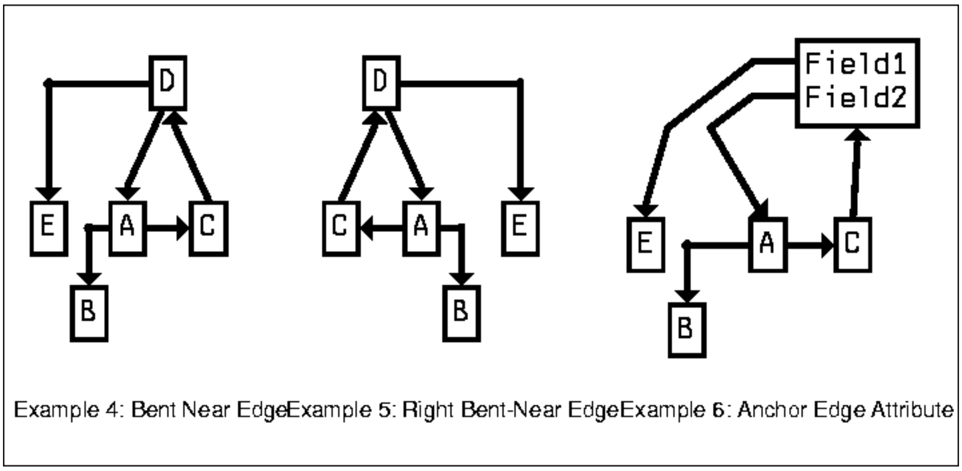

D
E
A
C
B
D
C
A
E
B
Field1
Field2
E
A
C
B
Example 4: Bent Near Edge
Example 5: Right Bent-Near Edge
Example 6: Anchor Edge Attribute

Figure 5.2: Bent Near, Right-Bent Near and Anchor Edges

```
    bentnearedge:  { source: "A"  target: "B" }
    nearedge:  { source: "A"  target: "C" }
    backedge:  { source: "C"  target: "D" }
    edge:  { source: "D"  target: "E" anchor: 1}
    edge:  { source: "D"  target: "A" anchor: 2}
}
```

5.6.2 Control Flow Graph

The following three graphs show the control flow graph of a procedural program. The nodes contain the text of statements as labels. Not all edges have labels. The visualized control flow comes from the following nonsense program, which consists of a procedure `test` and a main routine in a pseudo imperative language.

```
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 7: Control Flow Graph 1

This example shows a simple visualization of the control flow graph. The graph is shown in Figure 5.3.

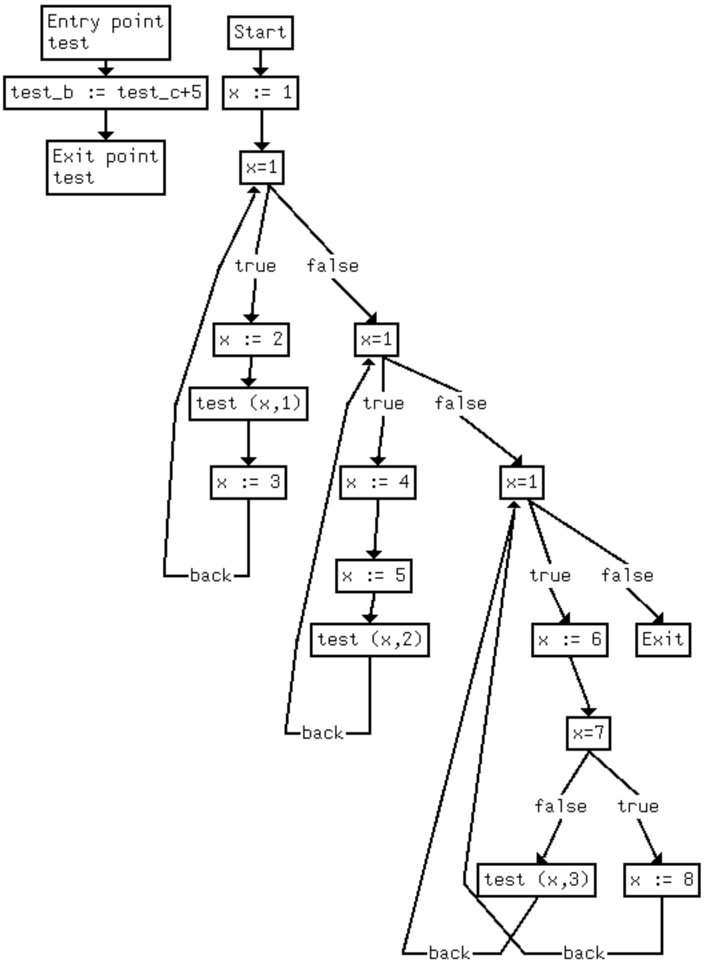


Figure 5.3: Control Flow Graph 1 – Simple Version

```
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:  { source:"18" target:"1" }
  edge:  { source:"0" target:"18" }
  edge:  { source:"12" target:"17" label: "false" }
  edge:  { source:"8" target:"12" label: "false" }
  edge:  { source:"16" target:"12" label: "back" }
  edge:  { source:"15" target:"12" label: "back" }
  edge:  { source:"13" target:"14" }
  edge:  { source:"14" target:"16" label: "false" }
  edge:  { source:"14" target:"15" label: "true" }
  edge:  { source:"12" target:"13" label: "true" }
  edge:  { source:"4" target:"8" label: "false" }
  edge:  { source:"11" target:"8" label: "back" }
  edge:  { source:"10" target:"11" }
  edge:  { source:"9" target:"10" }
  edge:  { source:"8" target:"9" label: "true" }
  edge:  { source:"3" target:"4" }
  edge:  { source:"7" target:"4" label: "back" }
  edge:  { source:"6" target:"7" }
  edge:  { source:"5" target:"6" }
  edge:  { source:"4" target:"5" label: "true" }
  edge:  { source:"2" target:"3" }
}
```

Example 8: Control Flow Graph 2

This example shows an improved visualization of the control flow graph. The graph is shown in

Figure 5.4. The start, exit and branch nodes are drawn in different shapes so that they can be better recognized. The edges closing a cycle are specified as back edges in order to see the uniform flow of control in the other edges. The edges at the branch nodes are anchored at the left and right via the bent near edge specification.

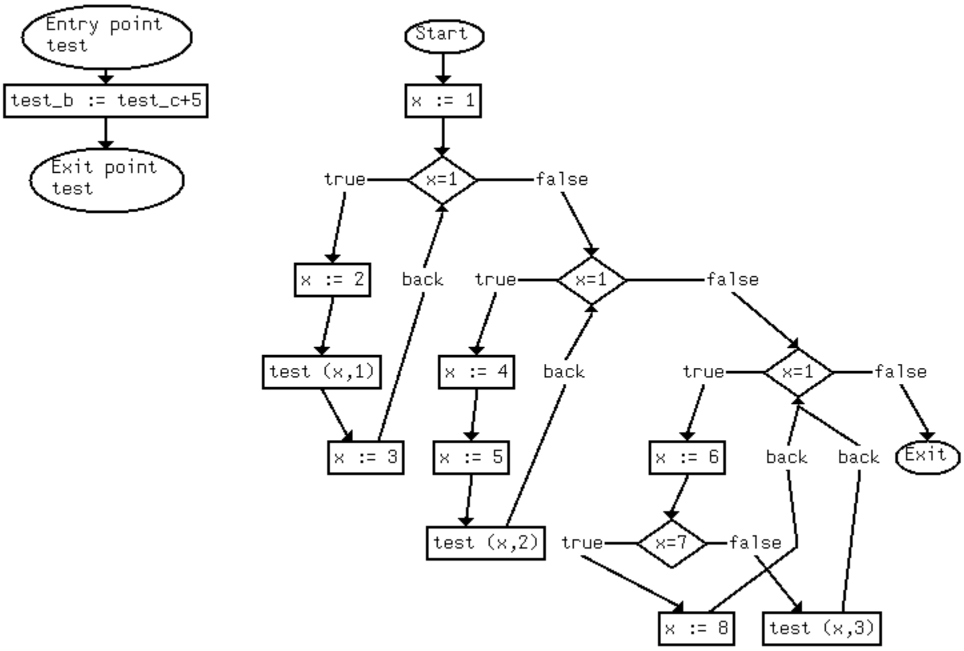


Figure 5.4: Control Flow Graph 2 with Rhomb-shaped Nodes and Near Edges

```
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 }
21    node:  { title:"3" label: "x := 1" }
22    node:  { title:"2" label: "Start" shape: ellipse }
23    node:  { title:"1" label: "Exit point\ntest" shape: ellipse }
24    node:  { title:"0" label: "Entry point\ntest" shape: ellipse }
25    edge:  { source:"18" target:"1" }
26    edge:  { source:"0" target:"18" }
27    bentnearedge:  { source:"12" target:"17" label: "false" }
28    bentnearedge:  { source:"8" target:"12" label: "false" }
29    backedge:  { source:"16" target:"12" label: "back" }
30    backedge:  { source:"15" target:"12" label: "back" }
31    edge:  { source:"13" target:"14" }
32    bentnearedge:  { source:"14" target:"16" label: "false" }
33    bentnearedge:  { source:"14" target:"15" label: "true" }
34    bentnearedge:  { source:"12" target:"13" label: "true" }
35    bentnearedge:  { source:"4" target:"8" label: "false" }
36    backedge:  { source:"11" target:"8" label: "back" }
37    edge:  { source:"10" target:"11" }
38    edge:  { source:"9" target:"10" }
39    bentnearedge:  { source:"8" target:"9" label: "true" }
40    edge:  { source:"3" target:"4" }
41    backedge:  { source:"7" target:"4" label: "back" }
42    edge:  { source:"6" target:"7" }
43    edge:  { source:"5" target:"6" }
44    bentnearedge:  { source:"4" target:"5" label: "true" }
45    edge:  { source:"2" target:"3" }
46 }
```

Example 9: Control Flow Graph 3

This example shows another improved visualization of the control flow graph of Example 7. The graph is shown in Figure 5.5. Here an orthogonal layout is used so that the graph looks like a typical flowchart. For orthogonal layout, a large down factor and near factor and an up factor of zero is recommended. This improves the layout of long vertical edges. Add the following lines to Example 8 after line *01*:

```
manhattan_edges:   yes
layout_downfactor: 100
layout_upfactor:   0
layout_nearfactor: 0
xlspace:           12
```

5.6.3 The Effect of the Layout Algorithms

The following sequence of layouts shows the same graph visualized by different layout algorithms. The graph is cyclic, so the algorithm *tree* can't be used.

A key problem is selecting the nodes that appear at the top level of the graph. The layout algorithm

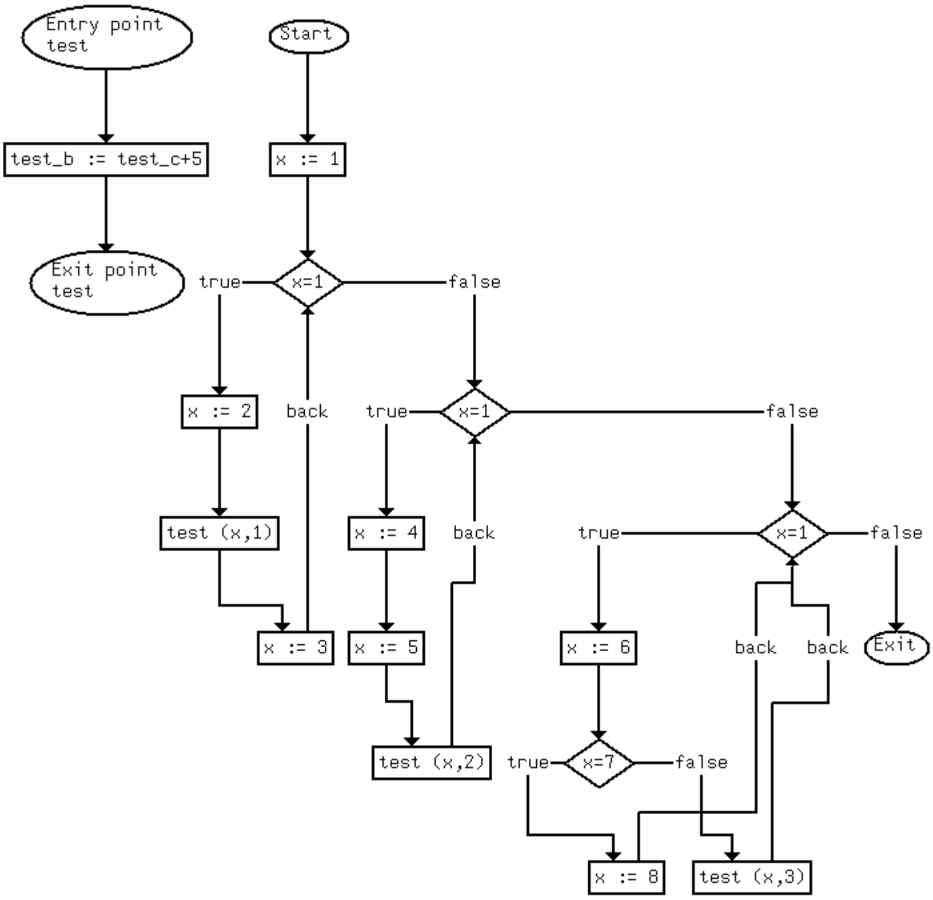


Figure 5.5: Control Flow Graph 3 with Manhattan Edges

looks for candidates that have no incoming edges yet at least one outgoing edge. If such nodes do not exist as in this example (Example 10), the algorithms *mindegree*, *minindegree*, *maxoutdegree* are the most appropriate algorithms.

The fine-tuning phase eliminates long edges, meaning the tuned graph is more compact. Note that the tuned graph created by *maxdephtslow* need not be of maximum depth because fine-tuning may have reduced the depth further. The tuned graph created by *mindepthslow* needn't be of minimum depth, either. All these partitioning algorithms are heuristics.

Example 10: A cyclic graph

```
graph:  {
  xspace: 25  xlspace: 15
  node:  { title: "0"  label: "Start" }
  node:  { title: "1" }  node:  { title: "2" }  node:  { title: "3" }
  node:  { title: "4" }  node:  { title: "5" }  node:  { title: "6" }
  node:  { title: "7" }  node:  { title: "8" }  node:  { title: "9" }
  node:  { title: "0" }
  edge:  { source: "0"  target: "1" }
  edge:  { source: "0"  target: "2" }
```

```
    edge:  { source: "0"  target: "3" }
    edge:  { source: "0"  target: "4" }
    edge:  { source: "0"  target: "5" }
    edge:  { source: "0"  target: "9" }
    edge:  { source: "1"  target: "3" }
    edge:  { source: "2"  target: "4" }
    edge:  { source: "3"  target: "5" }
    edge:  { source: "5"  target: "0" }
    edge:  { source: "9"  target: "0" }
    edge:  { source: "0"  target: "6" }
    edge:  { source: "6"  target: "7" }
    edge:  { source: "7"  target: "8" }
    edge:  { source: "8"  target: "0" }
}
```

Here follows a discussion of all the different hierarchical layout algorithms with and without fine-tuning for this example.

- *normal*, *finetuning: no*
  The normal layout algorithm breaks the cycle so that only one reverted edge is necessary (see left of Figure 5.6).
- *normal*, *finetuning: yes*
  Compared to the previous layout, the fine-tuning phase has balanced the position of the node 9. The long edge 8–>Start is not balanced since this would create additional reverted edges (see right of Figure 5.6).

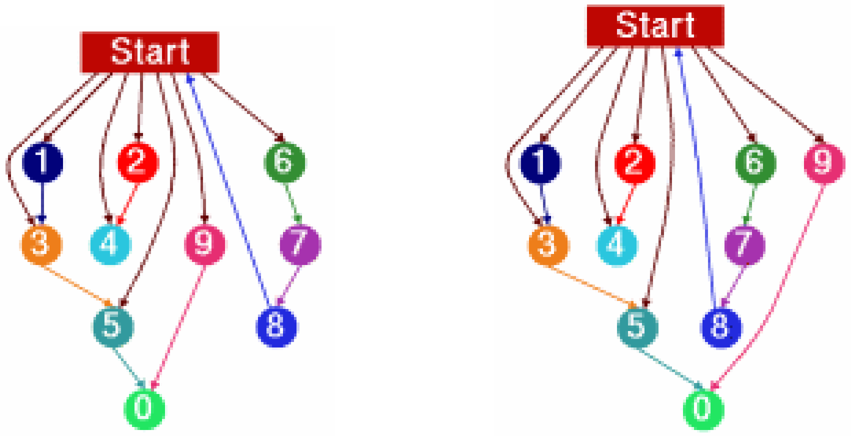


Figure 5.6: Example 10, *normal* layout, with and without finetuning

- *dfs*, *finetuning: no*
  The layout algorithms *dfs*, *mindepth*, *minindegree*, *maxoutdegree* and *maxdegree* happen to result in the same layout (see left of Figure 5.7).
- *dfs*, *finetuning: yes*
  The algorithms *dfs*, *mindepth*, *minindegree*, *maxoutdegree* and *maxdegree* happen to result in the same layout (see right of Figure 5.7).

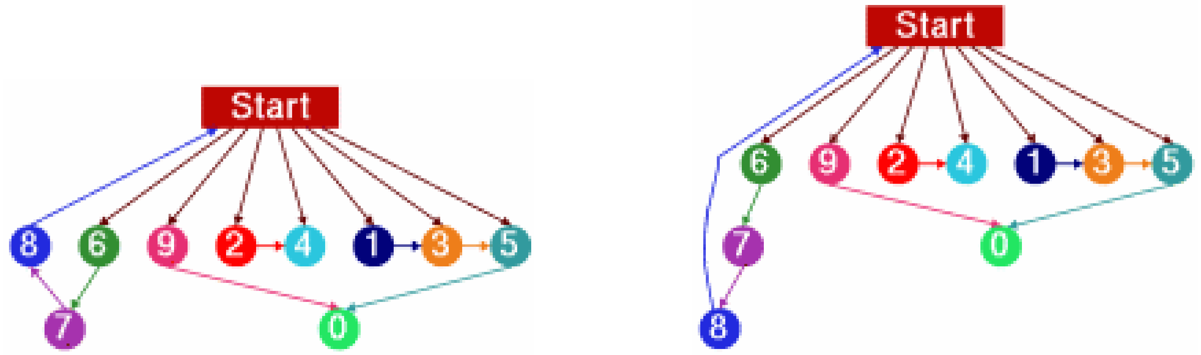


Figure 5.7: Example 10, *dfs* layout, with and without finetuning

- *minbackward*, *finetuning: no*
  This is almost the same layout as for the *normal* layout algorithm. Again only one reverted edge is necessary. The layout algorithm *maxdepth* without fine-tuning results in the same layout (see left of Figure 5.8).
- *minbackward*, *finetuning: yes*
  Compared to the layout without fine-tuning, here the long edge 8–>Start is partially eliminated and the position of node 9 is balanced again (see right of Figure 5.8).

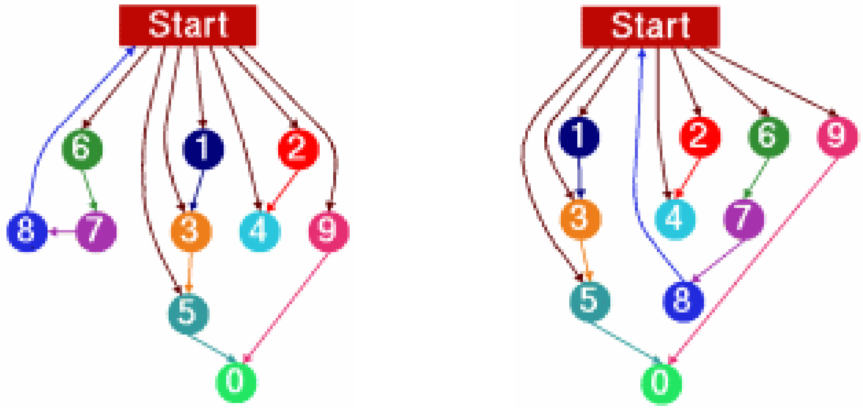


Figure 5.8: Example 10, *minbackward* layout, with and without finetuning

- *maxdepth*, *finetuning: no*
  Same as *minbackward* without fine-tuning (see left of Figure 5.8).
- *maxdepth*, *finetuning: yes*
  The long edge 8–>Start is now fully eliminated. Here, the fine-tuning phase is allowed to revert additional edges (see left of Figure 5.9).
- *mindepth*, *finetuning: no*
  The layout algorithms *dfs*, *mindepth*, *minindegree*, *maxoutdegree* and *maxdegree* happen to result in the same layout (see left of Figure 5.7).
- *mindepth*, *finetuning: yes*
  Compared to the previous layout the long edge 8–>Start is eliminated. The algorithms *dfs*,

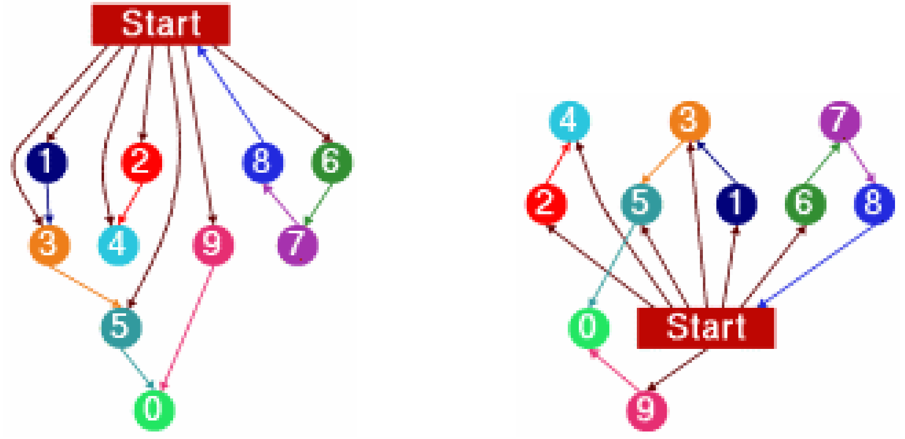


Figure 5.9: Example 10, *maxdepth*; and *mindegree* layout with finetuning

mindepth, *minindegree*, *maxoutdegree* and *maxdegree* happen to result in the same layout (see right of Figure 5.7).

- *maxdepthslow*, *finetuning: no*
  This depth six layout is in fact of maximum depth as compared to all the other variants (see left of Figure 5.10).
- *maxdepthslow*, *finetuning: yes*
  The fine-tuning phase eliminates the long edge Start–>6. Thus, the layout is no longer of maximum depth. Fine-tuning may destroy the maximum depth property (see right of Figure 5.10).

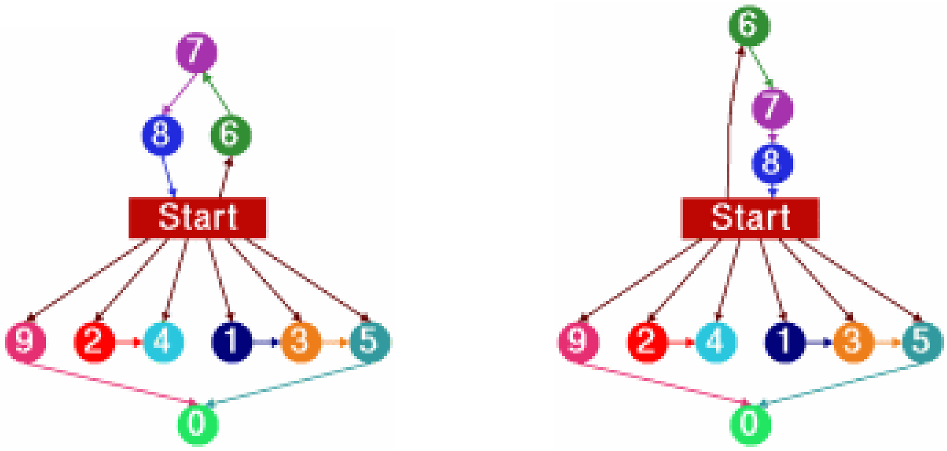


Figure 5.10: Example 10, *maxdepthslow* layout, with and without finetuning

- *mindepthslow*, *finetuning: no*
  Graphs that are of minimum depth tend to have many nodes at the top level. Compared to all untuned graphs, this layout is of minimum depth. It should be noted, however, that the algorithm *mindepth with fine-tuning* is able to produce a flatter layout (see left of Figure 5.11).

- *mindepthslow*, *finetuning: yes*
  Compared to the previous layout, the long edge 8–>Start is balanced again (see right of Figure 5.11).

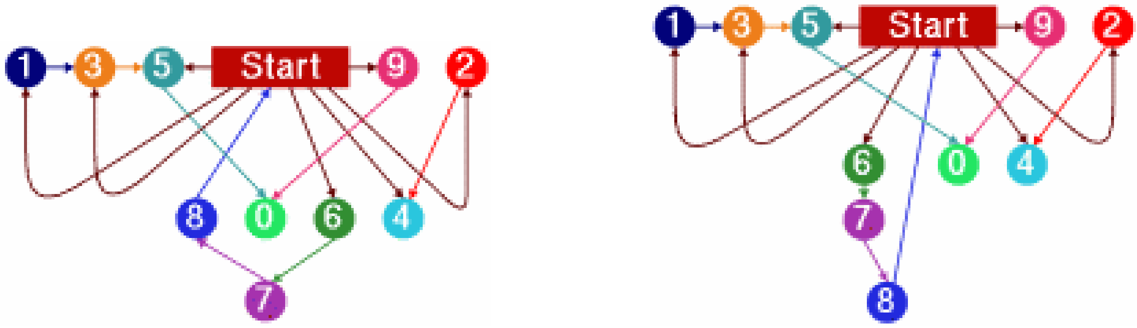


Figure 5.11: Example 10, *mindepthslow* layout, with and without finetuning

- *maxindegree*, *finetuning: no*
  Here node 3 is placed at the level zero because it has the maximum indegree. Node 0 is not chosen for level zero because it doesn't have any outgoing edges (see left of Figure 5.12).
- *maxindegree*, *finetuning: yes*
  Once again the long edge 8–>Start of the previous algorithm is eliminated by balancing the position of node 8. The algorithms *dfs*, *mindepth* and *minindegree* happen to result in the same layout (see right of Figure 5.12).

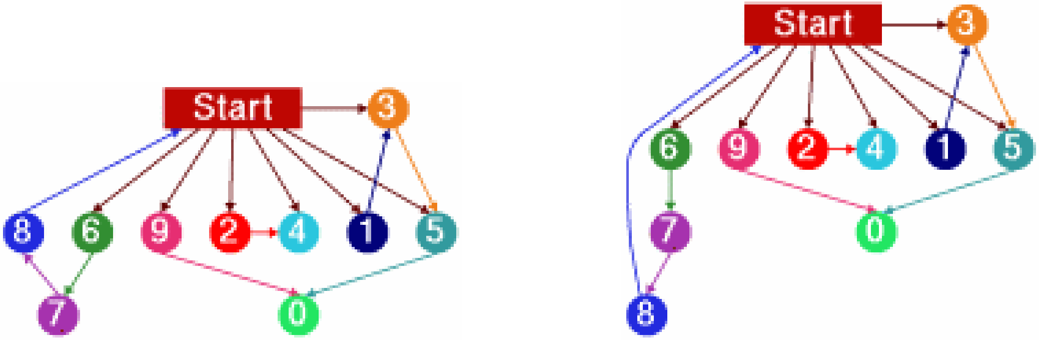


Figure 5.12: Example 10, *maxindegree* layout, with and without finetuning

- *minindegree*, *finetuning: no*
  The layout algorithms *dfs*, *mindepth*, *minindegree*, *maxoutdegree* and *maxdegree* happen to result in the same layout (see left of Figure 5.7).
- *minindegree*, *finetuning: yes*
  The algorithms *dfs*, *mindepth*, *minindegree*, *maxoutdegree* and *maxdegree* happen to result in the same layout (see right of Figure 5.7).
- *maxoutdegree*, *finetuning: no*
  The layout algorithms *dfs*, *mindepth*, *minindegree*, *maxoutdegree* and *maxdegree* happen to result in the same layout (see left of Figure 5.7).

- *maxoutdegree*, *finetuning: yes*
  The algorithms *dfs*, *mindepth*, *minindegree*, *maxoutdegree* and *maxdegree* happen to result in the same layout (see right of Figure 5.7).
- *minoutdegree*, *finetuning: no*
  Nodes 4 and 0 with a minimum outdegree of zero cannot be start nodes because start nodes have to have at least one successor otherwise they would create single-node components of the spanning tree. Start nodes can be any other nodes except Start, from which 1, 2, and 6 happen to be selected (see left of Figure 5.13).
- *minoutdegree*, *finetuning: yes*
  The long edges Start–>1, Start–>2 and Start–>6 are eliminated (see right of Figure 5.13).

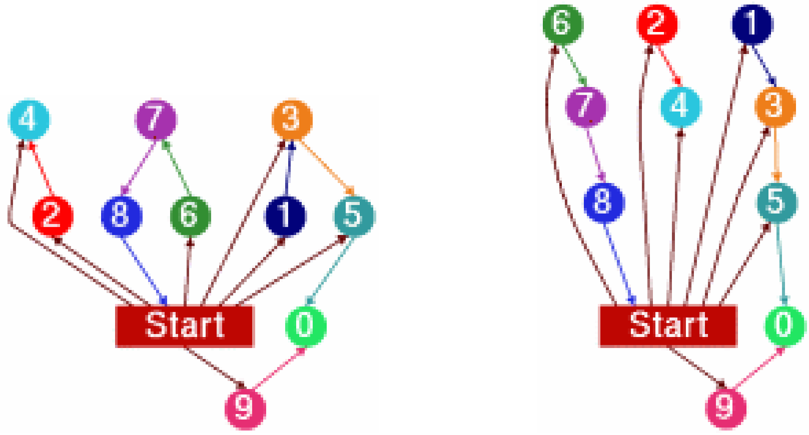


Figure 5.13: Example 10, *minoutdegree* layout, with and without finetuning

- *maxdegree*, *finetuning: no*
  The Start node has the maximum number of incoming and outgoing edges, so it is selected as the start node of the spanning tree, i. e. it appears at the topmost level. The layout algorithms *dfs*, *mindepth*, *minindegree*, *maxoutdegree* and *maxdegree* happen to result in the same layout (see left of Figure 5.7).
- *maxdegree*, *finetuning: yes*
  Compared to the previous layout the long edge 8–>Start is eliminated. The algorithms *dfs*, *mindepth*, *minindegree*, *maxoutdegree* and *maxdegree* happen to result in the same layout (see right of Figure 5.7).
- *mindegree*, *finetuning: no*
  The candidates for start nodes of the spanning tree are 1, 2, 6, 7, 8 and 9 because they have a minimum degree of two. 1, 2 and 6 happen to be selected as the start nodes. Note that nodes 4 and 0 are not candidates for start nodes because they do not have outgoing edges. The layout algorithms *minoutdegree* and *mindegree* happen to result in the same layout (see left of Figure 5.13).
- *mindegree*, *finetuning: yes*
  Compared to the previous layout, the long edges Start–>1 Start–>2 and Start–>6 are eliminated. This changes the structure of the layout entirely (see right of Figure 5.9).

5.6.4 Tree Layout

The following example shows a typed syntax tree. This tree can either be drawn by the specialized algorithm for "downward laid-out trees" or by the normal algorithms. When the normal algorithms are used to draw a tree, it is advisable to increase the layout down factor in order to obtain good results (see Figure 5.14). If the layout down factor is not used, the incoming edges draw the nodes too much in the direction of the parent node.

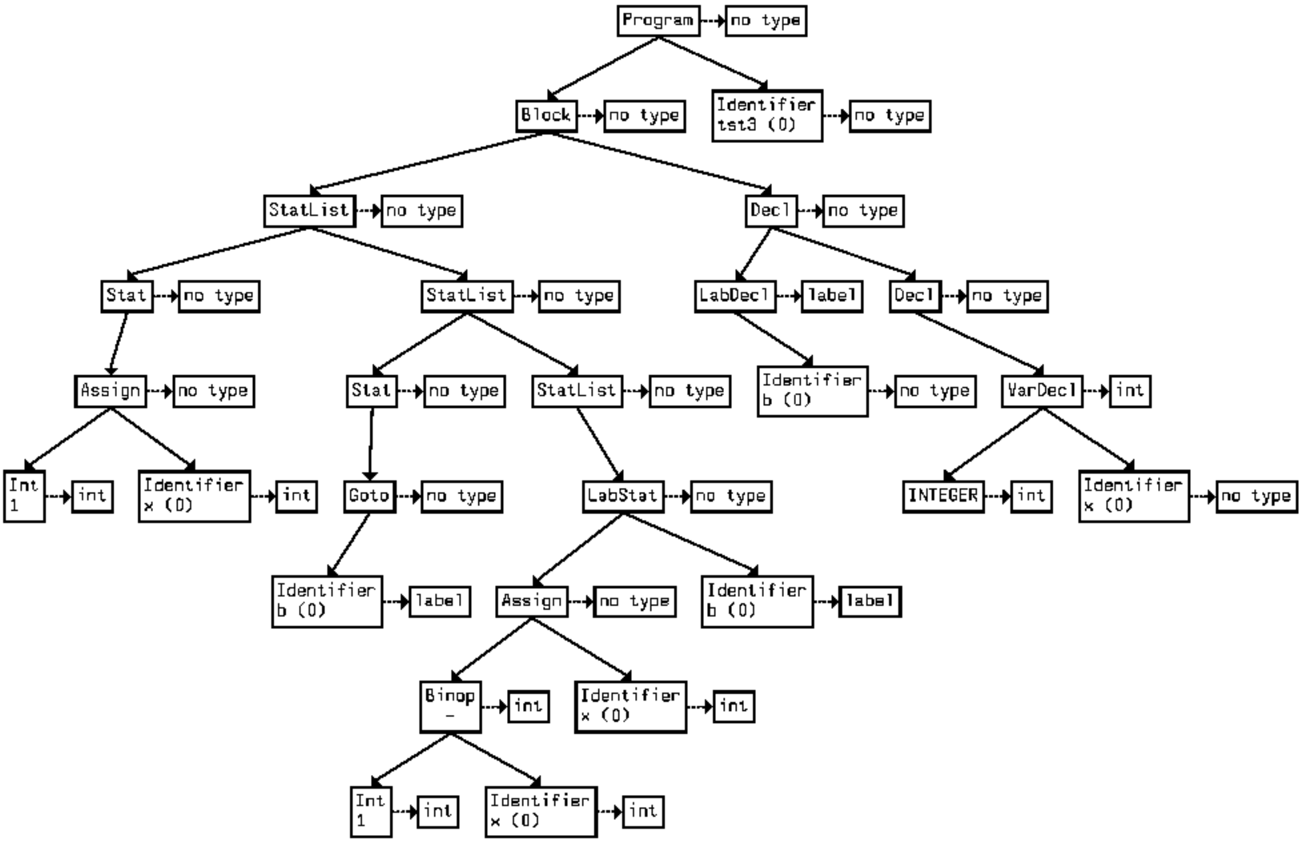


Figure 5.14: Example 11, *normal* layout

A nice layout is achieved by the specialized tree algorithm with a tree factor of 0.9 (see Figure 5.15).

If an orthogonal layout is needed, the attribute **smanhattan_edges** can be used. For trees, this attribute is more appropriate than the standard Manhattan layout with **manhattan_edges** (see Figure 5.16).

Example 11: Syntax Tree

```
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" }
```

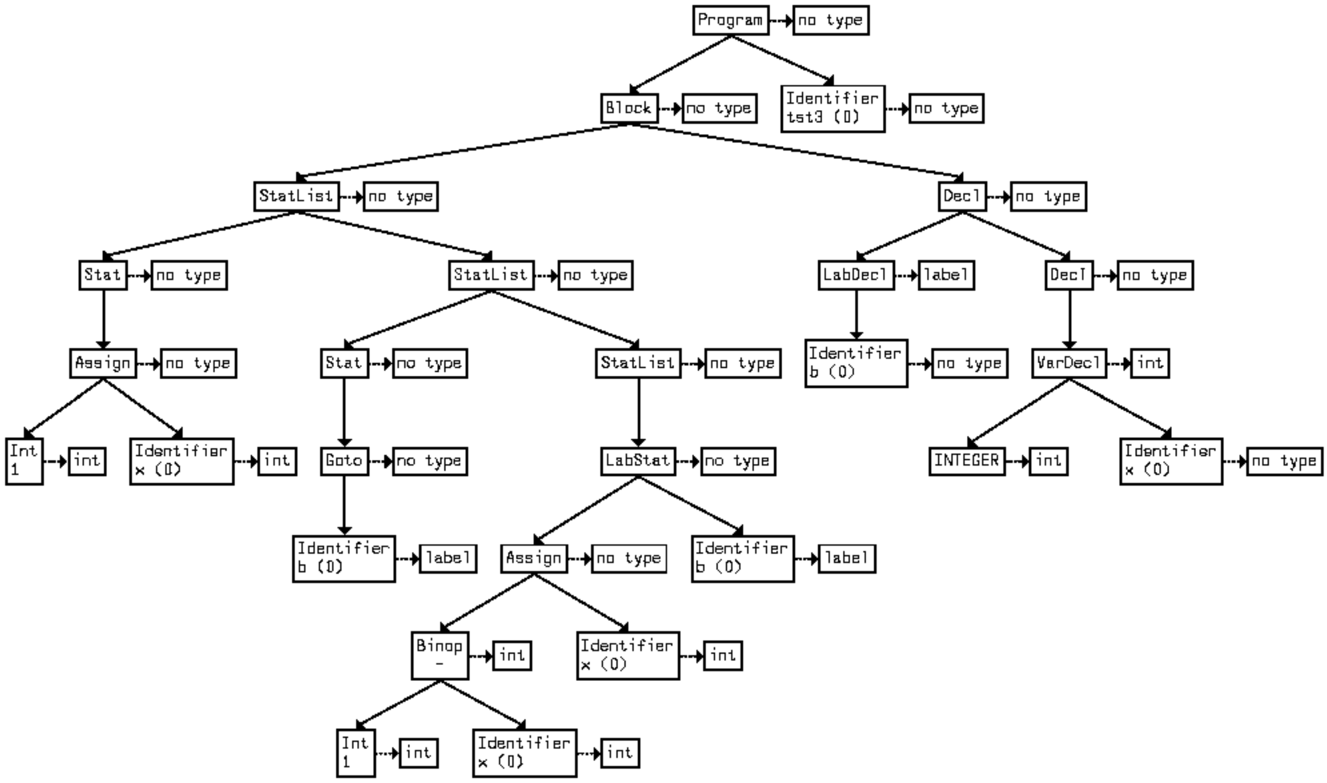


Figure 5.15: Example 11, *tree* layout, *treefactor: 0.9*

```
...
node:  { title:"T0" label: "no type" }
node:  { title:"T1" label: "no type" }
node:  { title:"T2" label: "int" }
...
edge: { source:"503304" target:"503240" }
edge: { source:"503304" target:"502952" }
...
nearedge: { source:"503160" target:"T0" linestyle: dotted }
nearedge: { source:"503240" target:"T1" linestyle: dotted }
nearedge: { source:"502952" target:"T2" linestyle: dotted }
...
}
```

5.6.5 Combination of Features

The following example (see Figure 5.17) is taken from [GKNV93] and shows the dependencies of different shell programs. A combination of **aiSee** features has been used to visualize it. There is a time scale to indicate the origin of the programs. The shells themselves are nodes that have to be placed at the same rank as their birth dates. The **level** attribute is used to set the nodes at these

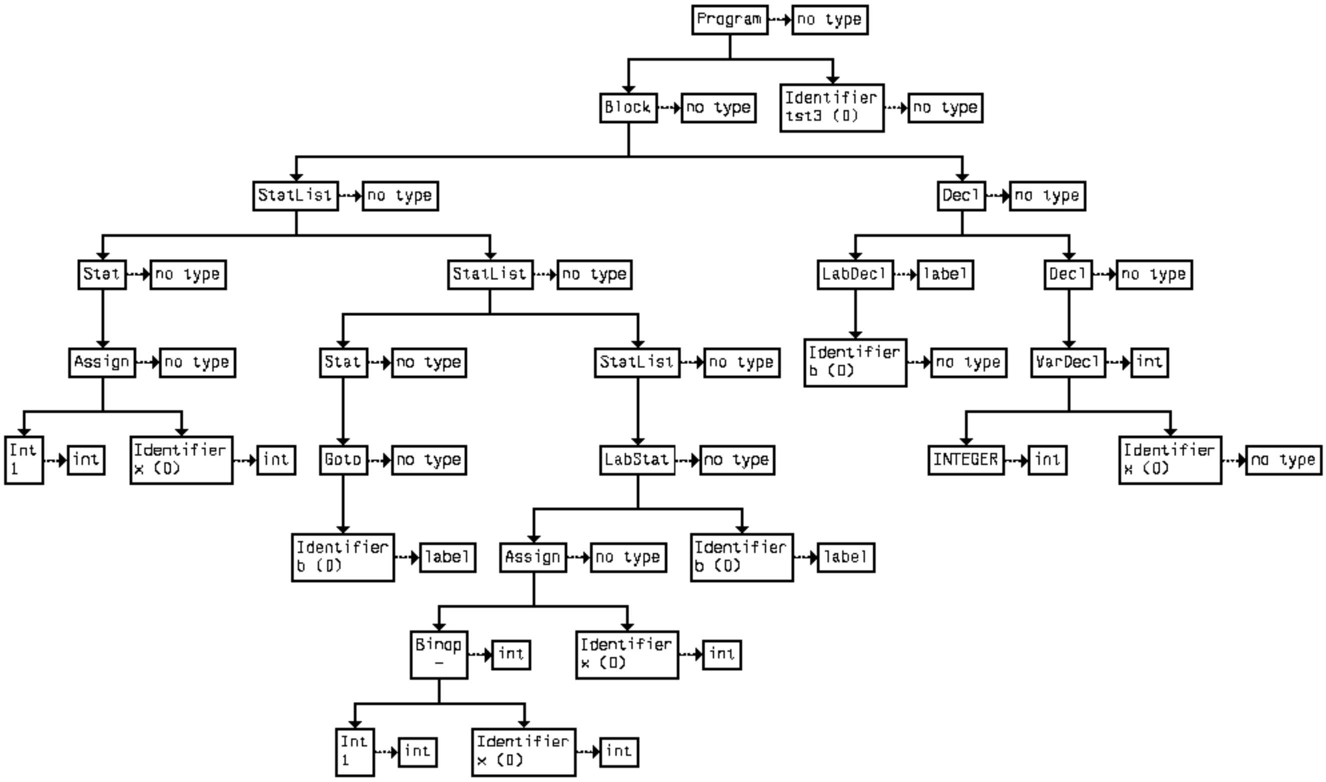


Figure 5.16: Example 11, *tree* layout, *smanhattan_edges: yes*

positions. In addition, the time axis should be positioned at the left side of the shell dependence graph. This is achieved by the **horizontal_order** attribute at some of the nodes. However, this attribute doesn't work unless the graph is connected, which is why three invisible edges are created.

Like any other edges, invisible edges influence the positions of the nodes. They pull their adjacent nodes together. To avoid this effect on the invisible edges, the priority of the invisible edges is set to zero and the priority of the visible edges to 100. There are many ways to change the priority. The **priority** attribute can be set, and the factors **layout_downfactor**, **layout_downfactor** and **layout_downfactor** as well. The real priority of a downward edge is the product *layout_downfactor* x *priority*.

The Bourne shell should be positioned to the left of the Mashey shell and the csh shell to the right of the Mashey shell. Therefore, the level two nodes receive a horizontal order. However, csh is on level three, and only its edge crosses level two. Therefore, the **horizontal_order** attribute is set for this edge as well. Now the edge is drawn to the right of the Mashey shell.

Default attribute specifications are used for the height, width and border width of nodes and for the style of edges in order to reduce the amount of specification. The various shell types are differentiated by using ellipses for the variations of the Korn shell, triangles for C shells and a rhomb for the tcl shell. The graph is acyclic, which is why the layout algorithm *minbackward* is used. Edges are drawn by splines.

Example 12: Development of Shells

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

   // First the time scale

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

   node: { title: "1972" level: 1 horizontal_order: 1}
   node: { title: "1976" level: 2 horizontal_order: 1 }
   node: { title: "1978" level: 3 }
   node: { title: "1980" level: 4 }
   node: { title: "1982" level: 5 horizontal_order: 1 }
   node: { title: "1984" level: 6 }
   node: { title: "1986" level: 7 }
   node: { title: "1988" level: 8 }
   node: { title: "1990" level: 9 }
   node: { title: "future" level: 10 horizontal_order: 1 }

   edge: { source: "1972" target: "1976" }
   edge: { source: "1976" target: "1978" }
   edge: { source: "1978" target: "1980" }
   edge: { source: "1980" target: "1982" }
   edge: { source: "1982" target: "1984" }
   edge: { source: "1984" target: "1986" }
   edge: { source: "1986" target: "1988" }
   edge: { source: "1988" target: "1990" }
   edge: { source: "1990" target: "future" }

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

   edge: { source: "ksh-i" target: "Perl" linestyle: invisible priority: 0 }
   edge: { source: "tcsh" target: "tcl" linestyle: invisible priority: 0 }
   nearedge: { source: "1988" target: "rc" linestyle: invisible }
   nearedge: { source: "rc" target: "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"   level: 1 horizontal_order: 2 }
   node: { title: "Mashey" level: 2 horizontal_order: 3 }
   node: { title: "Bourne" level: 2 horizontal_order: 2 }
```

```
  node: { title: "Formshell" level: 3 }
  node: { title: "csh" level: 3 shape: triangle }
  node: { title: "esh" level: 4 horizontal_order: 2 }
  node: { title: "vsh" level: 4 }
  node: { title: "ksh" level: 5 horizontal_order: 3 shape: ellipse }
  node: { title: "System V" level: 5 horizontal_order: 5 }
  node: { title: "v9sh" level: 6 }
  node: { title: "tcsh" level: 6 shape: triangle }
  node: { title: "ksh-i" level: 7 shape: ellipse }
  node: { title: "KornShell" level: 8 shape: ellipse }
  node: { title: "Perl" level: 8 }
  node: { title: "rc" level: 8 }
  node: { title: "tcl" level: 9 shape: rhomb }
  node: { title: "Bash" level: 9 }
  node: { title: "POSIX" level: 10 horizontal_order: 3 }
  node: { title: "ksh-POSIX" level: 10 horizontal_order: 2 shape: ellipse }

  edge: { source: "Thompson"   target: "Mashey"     }
  edge: { source: "Thompson" target: "Bourne" }
  edge: { source: "Thompson" target: "csh" horizontal_order: 4 }
  edge: { source: "Bourne" target: "ksh" }
  edge: { source: "Bourne" target: "esh" }
  edge: { source: "Bourne" target: "vsh" }
  edge: { source: "Bourne" target: "System-V" }
  edge: { source: "Bourne" target: "v9sh" }
  edge: { source: "Bourne" target: "Formshell" }
  edge: { source: "Bourne" target: "Bash" }
  edge: { source: "csh" target: "tcsh" }
  edge: { source: "csh" target: "ksh" }
  edge: { source: "Formshell" target: "ksh" horizontal_order: 4 }
  edge: { source: "esh" target: "ksh" }
  edge: { source: "vsh" target: "ksh" }
  edge: { source: "ksh" target: "ksh-i" }
  edge: { source: "System-V" target: "POSIX" }
  edge: { source: "v9sh" target: "rc" }
  edge: { source: "ksh-i" target: "KornShell" }
  edge: { source: "ksh-i" target: "Bash" }
  edge: { source: "KornShell" target: "Bash" }
  edge: { source: "KornShell" target: "POSIX" }
  edge: { source: "KornShell" target: "ksh-POSIX" }
}
```

5.7 Graph Attributes

This section describes the whole list of graph attributes. Each attribute is listed together with its type, default value and where it can be used, i. e. in the top-level graph, a subgraph specification or in both.

- **amax**
  Type: *integer*
  Default value: automatic
  Attribute of: *top-level graph*
  Description:
  This attribute specifies the number of iterations that are animated after relayout. Specifying 0 means animation is turned off.
- **arrow_mode**
  Type: *fixed*, *free*
  Default value: *free*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This attribute specifies two modes for drawing arrow heads.
  - *fixed*
    This arrow mode should be used if port sharing (see **port_sharing** on p. 66) is used because then only a fixed set of rotations for the arrow heads are used. Here the arrow head is rotated only in increments of 45 degrees, and only one arrow head occurs at each port.
  - *free*
    Here each arrow head is rotated individually for each edge. If a node has many incoming edges, this parameter can lead to a confusing image.
- **attraction**
  Type: *integer*
  Default value: 60
  Attribute of: *top-level graph*
  Description:
  This attribute applies only to the *forcedir* layout algorithm.

  Here it is part of the force-directed spring embedder during attractive impulse calculation. It specifies the constant proportional to the attractive forces acting on a node.

  This constant and its repulsive counterpart (see p. 68) enable the length of edges to be controlled. For example, if only attractive and repulsive forces are working on a node, an edge length of n pixels is achieved for edges with priority of 1 (see edge attribute priority on p. 88) by specifying n^2 for the attributes **attraction** and **repulsion**. Usually the values for these two attributes are of the same order of magnitude.

  For details, see p. 101.
- **bmax**
  Type: *integer*

Default value: 100
Attribute of: *top-level graph*, *subgraphs*
Description:
This attribute sets the maximum number of iterations of the phase reducing edge bends. Edge bends are used to prevent edges from being drawn across nodes. Reducing the number of iterations reduces layout calculation time, however the layout quality may suffer.

- **border**
  Type:
    x : *integer* pixels or
    y : *integer* pixels
  Default value: **x** : 600, **y** : 600
  Attribute of: *top-level graph*
  Description:
  This attribute applies only to the *forcedir* layout algorithm.

  Depending on the specification of the layout parameters for the *forcedir* layout algorithm it is possible for nodes to move far away from one another (especially single nodes not connected to the main graph) This prevents nodes from being placed "infinitely" far from one another.

  These two attributes enable a rectangle to be specified within which the graph is drawn.

  For details, see p. 101.

- **bordercolor**
  Type: *black*, *blue*, *red*, ...
  Default value:
    Same as the value of the **textcolor** attribute for summary nodes
  Attribute of: *subgraphs*
  Description:
  Specifies the color for borders of summary nodes, boxes and frames of clusters. For details on colors, see p. 90. See also **color** (p. 54) and **textcolor** attribute (p. 73).

- **borderstyle**
  Type: *continuous*, *dashed*, *dotted*, *double*, *invisible*, *solid*, *triple*
  Default value: *continuous*
  Attribute of: *subgraphs*
  Description:
  This attribute specifies the line style used for drawing the borders of a summary node. See also edge attribute **linestyle** (p. 87).

- **borderwidth**
  Type: *integer*
  Default value: 2 pixels
  Attribute of: *subgraphs*
  Description:
  This attribute specifies the thickness of the border of a summary node in pixels.

- **classname**
  Type: *integer* : *string*
  Default value: 1 : "1", 2 : "2", 3 : "3", ...

Attribute of: *top-level graph*
Description:
This attribute enables the names of edges classes to be introduced. These names appear in the **Select Edge Classes** dialog box. For details on edge classes, see p. **??**.

- **cmin**, **cmax**
  Type: *integer*
  Default value: 0 for **cmin**, *infinite* for **cmax**
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  The **cmin** attribute sets the minimum number of iterations that are performed for reducing crossings using crossing weights. The normal method stops when two consecutive checks no longer cause the number of crossings to be reduced. However, this number of crossings might be a local minimum, meaning the number of crossing might decrease even more after some more iterations.

  The **cmax** attribute sets the maximum number of iterations of the crossing reduction phase. A reduction of this value causes the layout process to be speeded up. The default value *infinite* means that the method is iterated as long as any improvement is possible.

- **color**
  Type: *black*, *blue*, *red*, ...
  Default value:
  white for *top-level graph*
  white for *subgraphs*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  - *top-level graph*
    Here the **color** attribute specifies the background color of the graph window.
  - *subgraphs*
    Here it specifies the background color of subgraphs. This color is valid as the background color in summary nodes, boxes, clusters and as a wrapping color.

  For details on colors, see p. 90. See also attributes **textcolor** (p. 73) and **bordercolor** (p. 53).

- **colorentry**
  Type: *integer* : *integer integer integer*
  Default value: no default value defined
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This attribute enables the default color map to be filled and changed. A color is a triple of integer values for the red, green and blue part. Each integer ranges from 0 (color part turned off) to 255 (color part turned on), e. g. 0 0 0 specifies the color black and 255 255 255 specifies the color white. For instance, `colorentry 75 : 70 130 180` sets map entry 75 to steel blue. This color can be used by merely specifying 75 wherever a color value is expected. For more details on colors, see p. 90.

- **crossing_optimization**
  Type: *yes*, *no*

Default value: *yes*
Attribute of: *top-level graph*, *subgraphs*
Description:
yes activates the crossing optimization phase, which works locally. It is a postprocessing phase after normal crossing reduction. It tries to optimize locally by exchanging pairs of nodes to reduce the number of crossings.

- **crossing_phase2**
  Type: *yes*, *no*
  Default value: *yes*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  yes activates crossing reduction phase two. In this phase, nodes having equal crossing weights are permuted. Note that this is the most time-consuming phase of crossing reduction.
- **crossing_weight**
  Type: *bary*, *median*, *barymedian*, *medianbary*
  Default value: *bary*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This attribute specifies the weight to be used for crossing reduction. There is no general recommendation as to which is the best method. A guideline might be to use *bary* if the degree of the nodes is large and *median* or one of the hybrid methods *barymedian* or *medianbary* if the degree is small, the degree of a node being the total of incoming and outgoing edges at a node. See p. 105 for details.
  - *bary*
    The barycenter is used for calculating the weights during crossing reduction. This is the fastest method for graphs with nodes whose average degree is very large.
  - *median*
    The median center is used for calculating the weights during crossing reduction.
  - *barymedian*
    These weights are the combination of barycenter and mediancenter weights, with barycenter having priority and mediancenter only being used for nodes whose barycenter weights are equal.
  - *medianbary*
    These weights are the combination of barycenter and mediancenter weights, with the mediancenter having priority.
- **dirty_edge_labels**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  yes forces a fast layout of edge labels, which may result in overlapping of labels. Dirty edge labels cannot be used if splines are used for edge drawing.
- **display_edge_labels**

Type: *yes*, *no*
Default value: *no*
Attribute of: *top-level graph*, *subgraphs*
Description:
yes: edge labels are displayed, *no*: edge labels are not displayed.

- **edges**
  Type: *yes*, *no*
  Default value: *yes*
  Attribute of: *top-level graph*
  Description:
  Deprecated. *no* suppressed the drawing of edges in the top-level graph and in all nested subgraphs.

- **energetic**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*
  Description:
  This attribute applies only to the *forcedir* layout algorithm.

  Apart from the forces of the spring embedder and the magnetic fields, the local energy level of a node can be taken into account in deciding whether the node should move or not. Setting this attribute to *yes* causes the local energy present at a node to be considered during layout.

  For details, see p. 101.

- **energetic attraction**, **energetic repulsion**, **energetic gravity**, **energetic crossing**, **energetic overlapping**, **energetic border**
  Type: *float*
  Default values: see list below
  Attribute of: *top-level graph*
  Description:
  This attribute applies only to the *forcedir* layout algorithm.

  Apart from the forces of the spring embedder and the magnetic fields, the local energy level of a node can be taken into account in deciding whether the node should move or not. The behavior of the local energy present at a node can be influenced via the following attributes:

  - **energetic attraction**, Default value: 70.0
    Weight of the attractive energy of edges
  - **energetic repulsion**, Default value: 70.0
    Weight of the repulsive energy between nodes
  - **energetic gravity**, Default value: 0.3
    Weight of the gravitational energy of a node
  - **energetic crossing**, Default value: 80.0
    Weight of the global energy of an edge crossing
  - **energetic overlapping**, Default value: 80.0
    Weight of the global energy of a node overlapping

  - **energetic border**, Default value: 70.0
    Weight of the border energy of a node

  Gauging layout quality can be done as follows: The better the layout, the lower the total of all the energy values mentioned above.

  For details, see p. 101.

- **equal_y_dist**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  If this attribute is enabled (*yes*), then the vertical distance in a hierarchical layout is equal among all levels.

- **fast_icons**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*
  Description:
  Deprecated. *yes*: causes icon file loading to be faster, which may negatively impact the quality of the drawing if not all the icon colors are present. For details on pictures in nodes, see p. 91.

- **fdmax**
  Type: *integer*
  Default value: 300
  Attribute of: *top-level graph*
  Description:
  This attribute only applies to the *forcedir* layout algorithm.

  Here it is used in the simulated annealing part of the algorithm. It specifies the upper hard limit for the number of iterations performed. The algorithm stops when the global temperature drops below a threshold value or when the limit specified here is reached.

  For details, see p. 101.

- **finetuning**
  Type: *yes*, *no*
  Default value: *yes*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  no: switches off the fine-tuning phase of the graph layout algorithm. The fine-tuning phase tries to give all edges the same length. It tries to improve the ranks of nodes in order to avoid very long edges (see p. 103).

- **focus**
  Type: no type
  Default value: no value
  Attribute of: *subgraphs*

Description:
This attribute sets the focus for the summary node of a subgraph, i. e. if the **status** of the subgraph is *folded* at startup, then the summary node of the subgraph for which **focus** was specified is centered in the graph window.

Note: The **focus** can also be specified for nodes (see node attribute p. 78). It goes without saying that the **focus** attribute should appear only once in a graph specification.

- **fontname**
  Type: *string*
  Default value: default font drawn by turtle graphics routines
  Attribute of: *subgraphs*
  Description:
  This attribute specifies a pixel font different from the default font and used for drawing the text labels of summary nodes. This font is given by the name of the **aiSee** font file containing the font description, e.g. a 12-point Times Roman font can be specified via `fontname: "timR12"`. Note: If the font file is not in the current directory the environment variable `AISEEFONTS` has to be set to the directory containing the font description files. For details on fonts, see p. 92.

- **fstraight_phase**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*
  Description:
  yes: forces straight edges that are not anchored at the same position on the border of the nodes. This is useful only if no port sharing (see p. 66) is selected, because bends are avoided by correcting the port position.

- **gravity**
  Type: *float*
  Default value: 0.0625
  Attribute of: *top-level graph*
  Description:
  This attribute only applies to the *forcedir* layout algorithm.

  It is used for impulse calculation. Only using a simulation of a spring embedder would force unconnected components of a graph to move further and further apart from one another, as there would be no attractive forces acting between them. That is why gravity is introduced as a counterforce.

  This attribute specifies the constant which is proportional to the gravitational force acting on a node. This constant controls the strength of the gravitational force, e. g. a value of zero cancels out the influence of any gravitational force.

  For details, see p. 101.

- **height**
  Type: *integer*
  Default value:
  Top-level graph: (height of root screen - 100) pixels

Subgraphs: (height of the label for summary nodes) pixels
Attribute of: *top-level graph*, *subgraphs*
Description:
The meaning of this attribute depends on where it is specified:

- *top-level graph*
  This attribute specifies the height of the display window in pixels.
- *subgraphs*
  Here it specifies the height of the summary node.

See also graph attribute **width** (p. 75).

- **hidden**
  Type: *integer*
  Default value: none
  Attribute of: *top-level graph*
  Description:
  This attribute specifies the edge class to be hidden. To hide more than one edge class, repeat this attribute for each additional edge class. For more details, see p. **??**.

  Edges in such a class are ignored during layout calculation and are not drawn. Nodes that are only accessible (forward or backward) via edges of a hidden class are not drawn. However, nodes that are not accessible at all are drawn (see p. 60).

  Note the difference between hiding edge classes and the edge line style *invisible* (see p. 88). Hidden edges do not exist in the layout. Edges with the *invisible* line style exist in the layout, i. e. they influence the layout, meaning they need space and may produce crossings, for example.

- **horizontal_order**
  Type: *integer*
  Default value: none (corresponds to -1)
  Attribute of: *subgraphs*
  Description:
  In a hierarchical layout, this attribute specifies the horizontal position of the summary node within a level (see p. 74). The nodes specified via horizontal positions are ordered according to these positions within the levels. Nodes without this attribute are inserted into this ordering by the crossing reduction mechanism (see p. 105).

  Note: Connected components are handled separately during crossing reduction, thus it is not possible to intermix nodes of different connected components in one ordering sequence. For example, one connected component consists of nodes A, B, C and another of nodes D, E, all nodes being positioned at the same level. Thus, for instance, it is not possible to specify the following horizontal order at level 0: A, D, C, E.

  Note further: If the algorithm for downward laid-out trees is used the specified horizontal order is retained only within nodes that are children of the same node, i. e. in case of downward laid-out trees it is not possible to specify a horizontal order for the entire level.

- **iconcolors**
  Type: *integer*

Default value: 32
Attribute of: *top-level graph*
Description:
Deprecated. This attribute specifies the size of the color map used for colors in bitmap files. For details on pictures in nodes, see p. 91.

- **iconfile**
  Type: *string*
  Default value: no default value
  Attribute of: *subgraphs*
  Description:
  This attribute specifies the bitmap file (format raw PBM, PPM) to be displayed in the summary node of the folded subgraph. Note: if the bitmap file to be displayed is not in the current directory the environment variable `AISEEICONS` can be set to the directory containing the bitmap file. For details on pictures in nodes, see p. 91.
- **icons**
  Type: *yes*, *no*
  Default value: *yes*
  Attribute of: *top-level graph*
  Description:
  no: disables displaying of icons in nodes.
- **ignore_singles**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  yes: hides all nodes of the remaining graph which would appear singly and unconnected. These nodes have no edges at all and drawing them sometimes results in an ugly layout of the remaining graph. The default setting is to show all nodes (*no* option).
- **importance**
  Type: *integer*
  Default value: 0 (which means infinity)
  Attribute of: *subgraphs*
  Description:
  Deprecated. This was the central attribute when it came to filtering in fish-eye views as it enabled the importance of a summary node of a folded subgraph to be specified via an integer.

  Low integers signified less important nodes which were filtered out first by a filtering fish-eye view. High integer numbers signified nodes that were important, them being rarely filtered out. A value of 0 represented an infinite importance, the result being that these nodes were never filtered out. This attribute existed for nodes too, see p. 79.
- **info1**, **info2**, **info3**
  Type: *string*
  Default value: empty string for all three

Attribute of: *subgraphs*
Description:
These attributes enable three additional text fields to be specified for a summary node of a folded subgraph. The same set of attributes exists for nodes (see node attribute **info1**, p. 80). These additional information fields can be selected interactively from the submenu of the Information menu (see p. **??**).

- **infoname**
  Type: *integer* : *string*
  Default value: 1 : "0", 2 : "1", 3 : "2"
  Attribute of: *top-level graph*
  Description:
  This attribute enables names for the additional information fields available for each node to be introduced. These names appear in the submenu of the menu item **Information** in the **Misc** menu of the menu line. For details on additional information fields, see p. **??** and p. 60.
- **inport_sharing**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*, *subgraphs*
  Description
  See graph attribute **port_sharing** (p. 66).
- **label**
  Type: *string*
  Default value: *empty string*
  Attribute of: *subgraphs*
  Description:
  This string is displayed inside the summary node of a folded subgraph. If no label is spezified the value of the **title** of the subgraph is used. If there is no label or title specified for the subgraph then the file name of the graph spezification is used.

  Note: This text may contain control characters, e.g. "\n" (newline character), that influence the size of the node (see 5.12)
- **invisible**
  Type: *integer*
  Default value: no default value
  Attribute of: *top-level graph*
  Description:
  This attribute is a synonym for the graph attribute **hidden** (p. 59).
- **late_edge_labels**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This attribute controls the moment when edge labels are drawn.
  - *yes*

  The graph is first partitioned and then edge labels are introduced.

  - *no*
    In this case the algorithm first creates labels and then partitions the graph. This option yields a more compact layout, but may result in more crossings.

- **layoutalgorithm**
  Type: *normal*, *maxdepth*, *mindepth*, ...
  Default value: *normal*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This attribute specifies the basic layout algorithms, there being two main categories. The first fourteen algorithms describe variations of a hierarchical layout, whereas the last algorithm implements a force-directed layout. The variations differ in the way nodes are selected for the various levels in the hierarchical layout.

  - *normal*
    This algorithm is the default algorithm. It tries to give all edges the same orientation and is based on the calculation of strongly connected components. The algorithms based on depth first search are faster.
  - *dfs*
    This algorithm uses a depth first search for layout calculation, but does not enforce additional constraints pertaining to the degree of nodes. It is faster than the *normal* layout algorithm. The result is heavily dependent on the initial order of the nodes in the specification.
  - *maxdepth/mindepth*
    These two algorithms are based on depth first search and are both fast heuristics. *maxdepth* tries to increase the depth of the layout. *mindepth* tries to increase the width of the layout.
  - *maxdepthslow/mindepthslow*
    These slower algorithms might be better if the fast heuristics offered by the algorithms *maxdepth* and *maxdepth* do not provide satisfying results. *maxdepthslow* tries to increase the depth of the layout and *mindepthslow* the width of the layout.
  - *maxindegree / minindegree*
    maxindegree schedules nodes with a maximum of incoming edges first, i. e. these nodes are positioned early. *minindegree* schedules nodes with a minimum of incoming edges first.
  - *maxoutdegree / minoutdegree*
    maxoutdegree schedules nodes with a maximum of outgoing edges first, i. e. these nodes are positioned early. *minoutdegree* schedules nodes with a minimum of outgoing edges first.
  - *maxdegree / mindegree*
    maxoutdegree schedules nodes with a maximum of the sum of incoming and outgoing edges first, i. e. these nodes are positioned early. The algorithm *minoutdegree* schedules nodes with the minimum of the sum of incoming and outgoing edges first.

- *minbackward*
  Instead of calculating strongly connected components, *minbackward* performs a topological sorting to assign ranks to the nodes. This algorithm is fast if the graph is acyclic.
- *tree*
  The *tree* algorithm is a specialized method for downward laid out (see p. 103). It is very fast for these tree-like graphs and results in a balanced layout.
- *forcedir*
  This layout algorithm computes a force-directed placement. It is a nonhierarchical layout method that works well for undirected graphs. Some features of hierarchical layout like near edges are disabled for this layout algorithm.

  The layout calculation for this algorithm can be controlled via the following graph attributes:

  - Spring embedder forces

    attraction (p. 52), **repulsion** (p. 68)
  - Gravitational forces
    gravity (p. 58)
  - Simulated annealing
    fdmax (p. 57),
    tempmin, **tempmax** (p. 72),
    temptreshold (p. 73), **tempscheme** (p. 72), **tempfactor** (p. 71)
  - Random influence
    randomfactor (p. 67), **randomrounds** (p. 67), **randomimpulse** (p. 67)
  - Magnetic forces
    magnetic_field1, **magnetic_field2** (p. 64),
    magnetic_force1, **magnetic_force2** (p. 65)
  - Energy level
    energetic (p. 56)
    energetic attraction (p. 56), **energetic repulsion** (p. 56)
    energetic gravity (p. 56), **energetic crossing** (p. 56)
    energetic overlapping (p. 56), **energetic border** (p. 56)
  - Boundary rectangle
    border x (p. 53), **border y** (p. 53)

- **layout_downfactor**, **layout_upfactor**, **layout_nearfactor**
  Type: *integer*
  Default value: 1 for all three attributes
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  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. These attributes have no effect if the layout algorithm *tree* is used.

If the **layout_downfactor** is large as compared to the **layout_upfactor** and **layout_nearfactor**, then the positions of the nodes are mainly determined by the edges pointing downwards.

If the **layout_upfactor** is large as compared to the **layout_downfactor** and **layout_nearfactor**, then the positions of the nodes are mainly determined by the edges pointing upwards.

If the **layout_nearfactor** is large, then the positions of the nodes are mainly determined by the edges pointing sidewards.

- **level**
  Type: *integer*
  Default value: none (corresponds to -1)
  Attribute of: *subgraphs*
  Description:
  This attribute is a synonym for **vertical_order** (p. 74).

- **linear_segments**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  yes switches linear segment layout on. This layout favors straight long vertical edges. See also `-linseg` options (p. **??**) and `-linsegmax` (p. **??**)

- **loc**
  Type: { **x**: <*integer*> **y**: <*integer*> }
  Default value:
    Top-level graph: { x:0 y:0 } for both
    Subgraphs: unspecified for both
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  For details, see **x** (p. 75).

- **magnetic_field1**, **magnetic_field2**
  Type:
    no, *top_to_bottom*, *bottom_to_top*, *left_to_right*, *right_to_left*,
    polar, *circular*, *orthogonal*, *polcircular*
  Default value: *no*
  Attribute of: *top-level graph*
  Description:
  This attribute only applies to the *forcedir* layout algorithm.

  Magnetic fields are part of impulse calculation. Forces that originate from a simulation of a spring embedder neglect the directions of edges. In directed graphs edges should point in a uniform direction, consequently magnetic forces are introduced, with edges being interpreted as magnetic needles that align according to a magnetic field.

  Two independent magnetic fields are possible. These two attributes specify the kind of magnetic field for each. If two fields are specified, the edges are influenced by both. The attributes

magnetic_force1 and **magnetic_force2** (p. 65) influence the strength of each field.

For details, see p. 101.

- **magnetic_force1**, **magnetic_force2**
  Type: *integer*
  Default value: 1 for both
  Attribute of: *top-level graph*
  Description:
  This attribute only applies to the *forcedir* layout algorithm.

  These two attributes specify the constant factors that are multiplied by the corresponding magnetic forces of the two magnetic fields.

  See also **magnetic_field1** and **magnetic_field2** (p. 64).

  For details, see p. 101.

- **manhattan_edges**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  yes switches orthogonal layout on. Orthogonal layout (or Manhattan layout) means that all edges consist of horizontal or vertical line segments. Vertical edge segments might be shared by several edges, while horizontal edge segments are never shared. This results in aesthetic layouts for flowcharts. If orthogonal layout is used, the priority phase and straight phase are also used by default (see **priorty_phase**, p. 67 and **straight_phase**, p. 71).

- **margin**
  Type: *integer*
  Default value: *0* if the value of **borderwidth** (p. 53) is zero, *3* otherwise
  Attribute of: *subgraphs*
  Description:
  Specifies the horizontal and vertical offset between the border of a summary node and its label in pixels. Useful for rectangular nodes only.

- **near_edges**
  Type: *yes*, *no*
  Default value: *yes*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  no: All near edges are treated as normal edges in the graph layout.

- **nodes**
  Type: *yes*, *no*
  Default value: *yes*
  Attribute of: *top-level graph*
  Description:
  Deprecated. *no* suppressed the drawing of nodes in the top-level graph and all nested subgraphs.

- **node_alignment**
  Type: *top*, *center*, *bottom*
  Default value: *center*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  For hierarchical layout this attribute specifies the vertical alignment of nodes at the horizontal reference line of levels.
  - *top*: The tops of all nodes of a level have the same y coordinate.
  - *center*: All nodes of a level are centered.
  - *bottom*: The bottoms of all nodes of a level have the same y coordinate.
- **orientation**
  Type: *top_to_bottom*, *bottom_to_top*, *left_to_right*, *right_to_left*
  Default value: *top_to_bottom*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This attribute specifies the orientation of the graph. All explanations in this section are given in relation to the default orientation.
- **outport-sharing**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  See graph attribute **port-sharing** (p. 66).
- **pmin**, **pmax**
  Type: *integer*
  Default value:
    0 for **pmin**
    100 for **pmax**
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  pmin sets the minimum number of iterations of the pendulum method. Like crossing reduction, this method stops when the “imbalance weight” stops decreasing. However, an increase in imbalance weight might be a local phenomenon, meaning that the imbalance might decrease much more after a few more iterations.

  pmax sets the maximum number of iterations of the pendulum method. Reducing this factor increases layout calculation speed.
- **port_sharing**, **inport_sharing**, **outport_sharing**
  Type: *yes*, *no*
  Default value:
    no (since **aiSee** 2.1.89) for **port-sharing**,
    no for **inport-sharing** and **outport-sharing**
  Attribute of: *top-level graph*, *subgraphs*
  Description:

no suppresses the sharing of ports by edges at nodes. **inport-sharing** enables the port sharing of incoming edges only, with **outport-sharing** enabling the port sharing of outgoing edges only.

Generally speaking, if multiple edges are adjacent to the same node, and the arrow heads of all these edges have the same appearance (color, size, etc.), these edges may share a port at a node. This means that only one arrow head is drawn, and all edges meet at this arrow head. This enables many edges to be located adjacent to one node without getting confused by too many arrow heads. If no port sharing is used, each edge gets its own port.

- **priority_phase**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  yes switches on the priority phase. This phase replaces the normal pendulum method with a specialized method: It forces long vertical edges to be straight, just like the straight phase (see p. 71). In fact, the straight phase is a fine-tuning of the priority phase, the priority phase being recommended for an orthogonal layout (see **manhattan_edges**, p. 65).

- **randomfactor**
  Type: *integer*
  Default value: 70
  Attribute of: *top-level graph*
  Description:
  This attribute only applies to the *forcedir* layout algorithm.

  If randomized rounds have been specified (see **randomrounds**, p. 67) then a node is placed with a probability of *randomfactor* percent during a round. This factor should be close to 100 in order to prevent the process from stopping too early.

  For details, see p. 101.

- **randomimpulse**
  Type: *integer*
  Default value: 32
  Attribute of: *top-level graph*
  Description:
  This attribute only applies to the *forcedir* layout algorithm.

  It specifies the strength of the random impulse vector. If the *forcedir* algorithm should behave like a *simulated annealing* algorithm, this constant should be large and a slow temperature scheme should be chosen. Otherwise a small value is preferable for the **randomimpulse** attribute.

  For details, see p. 101.

- **randomrounds**
  Type: *integer*
  Default value: -1
  Attribute of: *top-level graph*

Description:
This attribute only applies to the *forcedir* layout algorithm.

This attribute specifies the number of randomized rounds during impulse calculation. It should only be used for the first few rounds so as to add a random impulse. Afterwards, the random impulse would delay completion of calculation.

For details, see p. 101.

- **repulsion**
  Type: *integer*
  Default value: 60
  Attribute of: *top-level graph*
  Description:
  This attribute only applies to the *forcedir* layout algorithm.

  Here it is part of the force-directed spring embedder during repulsive impulse calculation. It specifies the constant that is inversely proportional to the attractive forces acting on a node.

  This constant and its attractive counterpart (see **attraction** on p. 52) enabled the length of edges to be controlled. Usually the values for these two attributes are of the same order of magnitude.

  For details, see p. 101.

- **rmin**, **rmax**
  Type: *integer*
  Default value:
  0 for **rmin**
  100 for **rmax**
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  rmin sets the minimum number of iterations for rubberbanding. This works in a manner similar to the pendulum method.

  rmax sets the maximum number of iterations for rubberbanding. Reducing this factor increases layout calculation speed.

- **scaling**
  Type: a *float* value ¿ 0.0 or *maxspect*
  Default value: 1.0
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This attribute specifies the scaling factor for graph representation. A scaling factor of 1.0 means normal size. *maxspect* scales a graph so that the entire graph fits into the graph window.

  When specified for the top-level graph, this attribute determines the size of the entire graph including all the subgraphs. When specified for a subgraph it determines the scaling factor of the summary node of the folded subgraph. The size of a boxed subgraph is not affected, however the size of the subgraph nodes may still be affected.

  For details, see **shrink/strech** (p. 69).

- **shape**
  Type:
    box, *rhomb*, *ellipse*, *circle*, *triangle*,
    trapeze, *uptrapeze*, *hexagon*, *lparallelogram*, *rparallelogram*
  Default value: *box*
  Attribute of: *subgraphs*
  Description:
  Specifies the shape of the summary node of a folded subgraph (see node attribute **shape** (p. 82) for a description of shapes).
- **shrink**, **stretch**
  Type: *integer*
  Default value: 1 for both
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  These two attributes specify the shrinking and stretching factors for the representation of the top-level graph. The scaling of the graph as a percentage is given by the formula ((*stretch* / *shrink*) * 100).

  For instance, (*stretch*, *shrink*) = (1, 1) or (2, 2) or (3, 3) or ... is normal size, (*stretch*, *shrink*) = (1, 2) is half size, and (*stretch*, *shrink*) = (2, 1) is double size. The scaling factor can also be specified via the **scaling** (p. 68).

  - *top-level graph*
    When these attributes are specified for the top-level graph, they determine the size of the entire graph including all the subgraphs.
  - *subgraphs*
    When specified for a subgraph they determine the scaling factor of the summary node of the folded subgraph. The size of a boxed subgraph is not affected, however the size of the subgraph nodes may still be affected. For details, see node attribute **shrink** (p. 83).
- **smanhattan_edges**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  yes selects a specialized orthogonal layout: All horizontal edge segments between two levels share the same horizontal line, i. e. not only vertical edge segments are shared (as in the Manhattan layout, see p. 65). However, horizontal edge segments are shared by several edges, too. This looks nice for trees but might be confusing in general.
- **smax**
  Type: *integer*
  Default value: 100
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This attribute sets the maximum number of iterations of the straight-line recognition phase. This value is not of any use unless the straight-line recognition phase is switched on, see

straight_phase on p. 71. It can be used to improve the Manhattan layout or the layout with the priority phase turned on.

- **splinefactor**
  Type: *integer*
  Default value: 70
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This factor determines the bending of splines. A factor of 100 indicates very sharp bending, a factor of 1 indicating very flat bending. Useful values range from 30 to 80. The default value changes to 10 if manhattan edges are enabled.

- **splines**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This attribute specifies whether splines are used to draw edges. Polygon segments are used to draw edges by default, because this is faster. The spline drawing routine is quite slow. Splines are mainly used to prepare high-quality PostScript output for small graphs.

- **spreadlevel**
  Type: *integer*
  Default value: 1
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This parameter only influences the *tree* algorithm. Spreading of the uppermost nodes of large balanced trees would increase the width of the tree to such an extent that the tree would no longer fit in a window. Consequently, the spread level specifies the minimum level (rank) where nodes are spread. Nodes of levels above the spread level are not spread.

- **state**
  Type: *boxed*, *clustered*, *exclusive*, *folded*, *unfolded*, *wrapped*,
  Default value: *unfolded*
  Attribute of: *subgraphs*
  Description:
  This attribute specifies the initial state of a subgraph, i. e. it describes the way a subgraph is displayed the first time a graph is visualized. The appearance of the subgraph (its state) can be changed interactively later on (see p. 17).

  - *boxed*
    The subgraph is surrounded by a frame, i. e. drawn in a box. The nodes in side the box are independent of the rest of the graph, i. e. there are no edges connecting nodes outside the box with nodes inside the box and vice versa. For details, see p. 21.

  - *clustered*
    The subgraph is surrounded by a frame. In contrast to a box, edges from nodes outside the frame are drawn to nodes inside the frame and vice versa. This is an experimental feature. For details, see p. 21.

  - *exclusive*
    The subgraph is shown exclusively. All other nodes of the graph are not visible. Only edges between nodes of a group are visible. Of course, this value should appear only once in a graph specification. For details, see p. **??**.
  - *folded*
    The nodes of a subgraph are hidden. They are represented by a single node, called a summary node. For details, see p. 20.
  - *unfolded*
    This is the default setting.
  - *wrapped*
    All nodes and edges belonging to the subgraph are wrapped using the same color. For details, see p. 21.

- **straight_phase**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  yes switches on the straight phase. This is an additional phase that tries to avoid bends in long edges. Long edges are drawn as long straight vertical lines. Thus, this phase is not very appropriate for normal layout, however it is recommended when an orthogonal layout is selected (see **manhattan_edges**, p. 65).
- **stretch**
  Type: *integer*
  Default value: 1
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  See graph attribute **shrink** (p. 69).
- **subgraph_label**
  Type: *yes*, *no*
  Default value: *yes*
  Attribute of: *subgraphs*
  Description:
  New. *no* switches off the displaying of the subgraph label when the subgraph is boxed or clustered. If the subgraph is folded into a summary node, the label is still displayed.
- **subgraph_labels**
  Type: *yes*, *no*
  Default value: *yes*
  Attribute of: *top-level graph*
  Description:
  Deprecated. *no* switched off the displaying of subgraph labels globally.
- **tempfactor**
  Type: *float*
  Default value: 1.3

Attribute of: *top-level graph*
Description:
This attribute only applies to the *forcedir* layout algorithm.

It is used in the simulated annealing part of the algorithm. It specifies the temperature scheme factor used for exponential and reverse exponential temperature schemes.

For details, see p. 101.

- **tempmin**
  Type: *integer*
  Default value: 1
  Attribute of: *top-level graph*
  Description:
  This attribute only applies to the *forcedir* layout algorithm.

  Here it is used in the simulated annealing part of the algorithm. It specifies the lower limit of the temperature range.

  For details, see p. 101.

- **tempmax**
  Type: *integer*
  Default value: 128
  Attribute of: *top-level graph*
  Description:
  This attribute only applies to the *forcedir* layout algorithm.

  Here it is used in the simulated annealing part of the algorithm. It specifies the upper limit of the temperature range.

  For details, see p. 101.

- **tempscheme**
  Type: *integer*
  Default value: 1
  Attribute of: *top-level graph*
  Description:
  This attribute only applies to the *forcedir* layout algorithm.

  There are local and global temperature schemes. In global temperature schemes all nodes have the same temperature.

  - 1 (local temperature temp_speed)
    Local adaptive temperature scheme with speedup during cooling.
  - 2 (local temperature temp_normal)
    Local adaptive temperature scheme with no speedup.
  - 3 (global temperature temp_linear)
    Linear curve.
  - 4 (global temperature temp_hyperbolical)
    Hyperbolic curve: very fast descent, then a low temperature for an extended period of time

  - 5 (global temperature temp_exponential)
    Exponential descending temperature, i. e. a small temperature for an extended period of time.
  - 6 (global temperature temp_logarithmic)
    Logarithmic descending, i. e. a small temperature for an extended period of time.
  - 7 (global temperature temp_reverse_exponential)
    Reverse exponential descending temperature, i. e. a high temperature for an extended period of time.
  - 8 (global temperature temp_reverse_logarithmic)
    Reverse logarithmic descent, i. e. a high temperature for an extended period of time.

  For details, see p. 101.

- **temptreshold**
  Type: *integer*
  Default value: 3
  Attribute of: *top-level graph*
  Description:
  This attribute only applies to the *forcedir* layout algorithm.

  Here it is used in the simulated annealing part of the algorithm. It specifies the threshold value for the global temperature. The algorithm stops if the global temperature drops below the value specified here.

  For details, see p. 101.

- **textcolor**
  Type: *black*, *blue*, *red*, *...*
  Default value:
  black for summary nodes
  Attribute of: *subgraphs*
  Description:
  Specifies the color for text labels of summary nodes. For details on colors, see p. 90. See also **color** (p. 54) and **bordercolor** (p. 53).

- **textmode**
  Type: *center*, *left_justify*, *right_justify*
  Default value: *center*
  Attribute of: *subgraphs*
  Description:
  This attribute specifies the alignment of text within a summary node frame.

- **title**
  Type: *string*
  Default value: name of the graph specification file
  Attribute of: *subgraphs*
  Description:
  This attribute specifies the name associated with the subgraph. If no title is specified the name of the file containing the graph specification is used. Note: Titles have to be unique

throughout a graph specification, meaning there can be only one subgraph at most without a title specification.

The name of a subgraph is used to identify this graph, so that the subgraph can be the source and target of an edge specification. These edges start or end at the summary nodes of folded subgraphs. If the subgraph is visualized unfolded, these edges start or end at the root of the subgraph or at the root of the first subgraph in the subgraph.

- **treefactor**
  Type: *float*
  Default value: 0.5
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  The *tree* algorithm 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 have a larger gradient, this possibly improving the readability of the tree. Note: It is not obvious whether spreading results in a denser or wider layout. A tree factor exists for each tree, enabling maximu density of the entire tree.

- **useractioncmd1**, **useractioncmd2**, **useractioncmd3**, **useractioncmd4**
  Type: *string*
  Default value: empty string for all four
  Attribute of: *top-level graph*
  Description:
  These attributes enable four commands to be specified in a graph specification, that are executed in the User Action Mode. In this aiSee version, only User Action 3 is supported.

  For details on user actions, see section User Actions (p. 24) and also be refered to **useractionname** on p. 74.

- **useractionname**
  Type: *integer* : *string*
  Default value: 1 : "User Action 1", 2 : "User Action 2", 3 : "User Action 3",
  4 : "User Action 4"
  Attribute of: *top-level graph*
  Description:
  This attribute enables names for the user actions to be introduced.

  These names are currently unused. In future versions, they could be used to refer to particular user actions in various submenus.

  For details on user actions, see section User Actions (p. 24) and **useractioncmd** on p. 74.

- **vertical_order**
  Type: *integer* or *maxlevel*
  Default value: none (corresponds to -1)
  Attribute of: *subgraphs*
  Description:
  In a hierarchical layout, this attribute specifies the vertical position of a summary node of a folded subgraph. *maxlevel* tries to position the node as the maximum calculated level. Generally for all nodes, their vertical position is called their level or rank (see p. 103). **level**

is a synonym for **vertical_order**.

All nodes of level 0 form the uppermost layer (the first layer), if the orientation is top-down. Nodes of level 1 form the second layer, etc.

The level specification is not in effect unless automatic layout is being calculated. Layout is calculated automatically if there is at least one node without a specified location (see **loc** attribute for nodes (p. 81) and summary nodes (p. 64).

Note: The level specification may conflict with a *near edge* (p. 31) specification, because the source and target node of a near edge have to have the same level. In this case, the level specification of the source or the target node of the near edge is ignored.

- **width**
  Type: *integer*
  Default value:
    Top-level graph: (width of root screen − 100) pixels
    Subgraphs: (width of the label for summary nodes) pixels
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  The meaning of this attribute depends on its location.

  - *top-level graph* This attribute specifies the width of the display window in pixels.

  - *subgraphs* Here it specifies the width of the summary node in pixels.

  See also graph attribute **height** (p. 58)

- **x**, **y**
  Type: *integer*
  Default value:
    Top-level graph: 0 pixels for both
    Subgraphs: unspecified for both
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  Deprecated. The meaning of these attributes differs depending on whether they are specified for the top-level graph or for subgraphs.

  - *top-level graph*
    Here these attributes specify the position of the graph window in relation to the root screen, i. e. the x and y coordinates of the upper left corner of the graph window are specified in pixels. The origin of the root screen is in the upper left corner.

  - *subgraphs*
    Here they specify the x and y coordinates (in pixels) of the summary node in relation to the upper left corner of the graph window.

  The positions can also be specified via the **loc** attribute (p. 64).

- **xbase**, **ybase**
  Type: *integer*
  Default value: 5 pixels for both attributes
  Attribute of: *top-level graph*, *subgraphs*

Description:
xbase, **ybase** specify the horizontal and vertical offset between the graph window and the upper left-hand corner of the graph, i. e. the position of the origin of the coordinate system in relation to the upper left-hand corner of the virtual window.

In subgraph specifications they are the offsets from the frame of the box containing the subgraph.

- **xmax**, **ymax**
  Type: *integer*
  Default value:
  (width of the root screen − 90) pixels for **xmax**
  (height of the root screen − 90) pixels for **ymax**
  Attribute of: *top-level graph*
  Description:
  Deprecated. These attributes specified the maximum size of the virtual window used to display the graph (see Figure 5.18). This is usually larger than the displayed part. The width and height of the displayed part cannot be larger than **xmax** and **ymax**. Only the parts of the graph inside the virtual window are drawn. The virtual window can be moved over the potentially infinite coordinate system by special positioning commands (see p. 13).

  Note: It is advisable to set **xmax**, **ymax** so they do not exceed the size of the root screen so as to get good performance.

- **xraster**, **yraster**
  Type: *integer*
  Default value: 1 pixel
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  These attributes specify the horizontal and vertical raster distance for the position of nodes. The center of a node is aligned to this raster.

- **xlraster**
  Type: *integer*
  Default value: 1 pixel
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This attribute is the horizontal raster for the positions of the line control points (the dummy nodes). It should be a divisor of xraster.

- **xlspace**
  Type: *integer*
  Default value:
  ($\frac{1}{2}xspace$)pixels, if polygons are used for edge drawing
  ($\frac{4}{5}yspace$) pixels, if splines are used
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  This attribute describes the horizontal distance between lines at the points where they cross levels. Note: It is advisable to set **xlspace** to a larger value, if splines are used in order to

prevent sharp bends.

- **xspace**,**yspace**
  Type: *integer*
  Default value:
    20 pixels for **xspace**,
    70 pixels for **yspace**
  Attribute of: *top-level graph*, *subgraphs*
  Description:
  xspace, **yspace** specify the minimum horizontal and vertical distance between nodes.

5.8 Node Attributes

This section describes the entire list of node attributes. Each attribute is listed together with its type and default value.

- **bordercolor**
  Type: *black*, *blue*, *red*, ...
  Default value:
  Same as the value of the **textcolor** attribute for nodes
  Attribute of: *node*
  Description:
  Specifies the color for node borders. For details on colors, see p. 90. See also node attributes **color** (p. 78) and **textcolor** (p. 83).
- **borderstyle**
  Type: *continuous*, *dashed*, *dotted*, *double*, *invisible*, *solid*, *triple*
  Default value: *continuous*
  Attribute of: *subgraphs*
  Description:
  This attribute specifies the line style used for drawing the borders of a node. See also edge attribute **linestyle** (p. 87).
- **borderwidth**
  Type: *integer*
  Default value: 2 pixels
  Attribute of: *node*
  Description:
  This attribute specifies the width of the border of a node in pixels.
- **color**
  Type: *black*, *blue*, *red*, ...
  Default value: *white* or transparent
  Attribute of: *node*
  Description:
  Here the **color** attribute specifies the background color of a node. For details on colors, see p. 90. See also node attributes **textcolor** (p. 83) and **bordercolor** (p. 78).
- **focus**
  Type: no type
  Default value: no value
  Attribute of: *node*
  Description:
  This attribute sets the focus for the node for which it is specified. After startup this node is centered in the graph window.

  Note: the **focus** can also be specified for summary nodes (see graph attribute **focus**, p. 57). It goes without saying that the **focus** attribute should appear only once in a graph specification.
- **fontname**
  Type: *string*

Default value: default font is drawn by turtle graphics routines
Attribute of: *node*
Description:
This attribute specifies a pixel font different from the default font and used for drawing the text labels of summary nodes. The font is given by the name of the **aiSee** font file containing the font description, for example a 12-point Times Roman font can be specified via `fontname: "timR12"`. Note: If the font file is not in the current directory the environment variable `AISEEFONTS` has to be set to the directory containing the font description files. For details on fonts, see p. 92.

- **height**
  Type: *integer*
  Default value: (node label height) pixels
  Attribute of: *node*
  Description:
  This attribute specifies the height of a node including the border. See also node attribute **width** (p. 84).

- **horizontal_order**
  Type: *integer*
  Default value: none (corresponds to -1)
  Attribute of: *node*
  Description:
  In a hierarchical layout, this attribute specifies the horizontal position of a node within a level (see node attribute **vertical_order**, p. 84). Nodes specified by horizontal positions are ordered according to these positions within levels. Nodes without this attribute are inserted in this ordering by the crossing reduction mechanism (see p. 105).

  Note: Connected components are handled separately during crossing reduction, thus it is not possible to intermix nodes of different connected components in one ordering sequence. For example, one connected component consists of nodes A, B, C and another one of nodes D and E, which are all positioned at the same level. Then, for instance, it is not possible to specify the following horizontal order at level 0: A, D, C, E.

  Note further: If the algorithm for downward laid-out trees is used, the specified horizontal order is only retained within nodes that are children of the same node, i. e. in case of downward laid-out trees it is not possible to specify a horizontal order for the entire level.

- **iconfile**
  Type: *string*
  Default value: no default value
  Attribute of: *node*
  Description:
  This attribute specifies the bitmap file (format: raw PBM, PPM) to be displayed in the node. Note: If the bitmap file to be displayed is not in the current directory the environment variable `AISEEICONS` can be set to the directory containing the bitmap file. For details on pictures in nodes, see p. 91.

- **importance**

Type: *integer*
Default value: 0 (which means *infinity*)
Attribute of: *node*
Description:
Deprecated. This was the central attribute when it came to filtering in fish-eye views as it enabled the importance of a node to be specified via an integer. Low integers signified less important nodes which were filtered out first by a filtering fish-eye view. High integer numbers signified nodes that were important, them being rarely filtered out. A value of 0 represented an infinite importance, the result being that these nodes were never filtered out. This attribute also existed for summary nodes of folded subgraphs, see **importance** (p. 60).

- **info1**, **info2**, **info3**
  Type: *string*
  Default value: empty string for all three
  Attribute of: *node*
  Description:
  These attributes enable three additional text fields to be specified for a node. The same set of attributes exists for summary nodes (see node attribute **info1**, p. 60). These additional information fields can be selected interactively from the submenu of the **Information** menu (see p. **??**).

  When exporting a graph to SVG (or HTML), a URL can be specified in the info3 field of a node that is visited when the user clicks on that node in the SVG (or PNG or BMP) image. The URL specification has to be in the format `info3: "href:URL"`, meaning that if the content of an `info3` field does not start with `href:`, it will be interpreted as a normal information field rather than as a hyperlink.

  In addition to a URL, the following optional hyperlink attributes can be specified:

  - `target`
  - `onMouseOver`
  - `onMouseOut`
  - `onMouseDown`
  - `onMouseUp`
  - `onMouseMove`
  - `onClick`

  These attributes can be specified in any order directly after the URL itself. The specifications must be separated by semicolons. Examples:

```
info3:  "href:http://www.aisee.com/svg/"
info3:  "href:javascript:myfunction(42,'Hello world!')"
info3:  "href:http://www.aisee.com;target:_blank;onClick:foo('bar')"
info3:  "href:#; onMouseOver:foo(1); onMouseOut:foo(0)"
```

  As of aiSee 2.1.96, the URL specification can be omitted:

```
info3:  "onMouseOver:foo(1);onMouseOut:foo(0)"
```

If aiSee spots a mouse event handler specification in the GDL source of a graph, it automatically includes a reference to an external JavaScript file when exporting the graph to SVG:

```
<svg:script xlink:href="SVGFileName.js" language="JavaScript">
</svg:script>
```

This allows JavaScript functions and global variables to be conveniently specified in the external JS file rather than in the SVG file itself. Thanks to this approach, you can easily make changes to your graphs and re-export them to SVG without having to copy & paste JavaScript code from old SVG files into new ones each time.

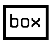
- **invisible**
  Type: *yes*, *no*
  Default value: *no*
  Attribute of: *node*
  Description:
  Hides a node after computing the layout.
- **label**
  Type: *string*
  Default value: empty string
  Attribute of: *node*
  Description:
  This string is displayed inside a node. If no label is specified the value of the node attribute **title** is used.

  Note: This string may contain control characters, e. g. “`\n`” (newline character), that influence the size of the node. See p. 94 for more details.
- **level**
  This is a synonym for **vertical_order**.
- **loc**
  Type: { **x**: <*integer*> **y**: <*integer*> }
  Default value: no default value
  Attribute of: *node*
  Description:
  This attribute specifies the location of the node, i. e. the x and y coordinates in relation to the coordinate system of the graph. The origin is in the upper left corner. For example, the specification `loc: { x: 100 y: 200 }` places a node at location (100, 200) in the coordinate system.

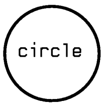
  The location of nodes are not valid unless locations are specified for all nodes, otherwise **aiSee** calculates appropriate x and y coordinates according the layout algorithm chosen.
- **margin**
  Type: *int*
  Default value:
    3, if **borderwidth** (p. 78) > 0

0, if **borderwidth** = 0
Attribute of: *node*
Description:
Specifies the horizontal and vertical offset between the border of a node and its label in pixels. Useful for rectangular nodes only.

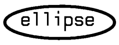
- **scaling**
  Type: *float* ¿ 0.0
  Default value: 1.0
  Attribute of: *node*
  Description:
  This attribute specifies the scaling factor of a node. To completely hide a node, use the attribute **invisible** (p. 81). For further details, see node attribute **shrink** (p. 83).
- **shape**
  Type:
  box, *rhomb*, *ellipse*, *circle*, *triangle*,
  trapeze, *uptrapeze*, *hexagon*, *lparallelogram*, *rparallelogram*
  Default value: *box*
  Attribute of: *node*
  Description:
  This attribute specifies the frame shape of a node.
  Note: Drawing ellipses is slower than drawing other shapes.
  - *box*

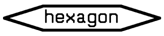
  - *circle*

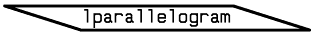
  - *ellipse*

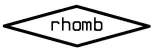
  - *hexagon*

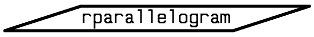
  - *lparallelogram*

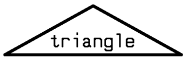
  - *rhomb*

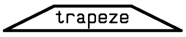
  - *rparallelogram*

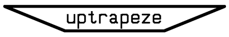
  - *triangle*



  - *trapeze*



  - *uptrapeze*



- **shrink**, **stretch**
  Type: *integer*
  Default value: 1 for both
  Attribute of: *node*
  Description:
  These attributes specify the shrinking and stretching factor of a node. The values of the node attributes **width**, **height**, **borderwidth** and the size of the **label** is scaled by ((*stretch* / *shrink*) * 100) percent. The scale value can also be specified by the node attribute **scaling** (p. 82).

  Note: The actual scaling factor of a node is determined by the scale factor of a node in relation to the scale factor of the graph, i. e. if the scaling factor for the graph is (*stretch*, *shrink*) = (2, 1) and (*stretch*, *shrink*) = (2, 1) for the node, the node is scaled by a factor of 4 as compared to normal size.

- **textcolor**
  Type: *black*, *blue*, *red*, ...
  Default value: *black*
  Attribute of: *node*
  Description:
  Specifies the color for text labels of nodes. For details on colors, see p. 90. See also node attributes **color** (p. 78) and **bordercolor** (p. 78).

- **textmode**
  Type: *center*, *left_justify*, *right_justify*
  Default value: *center*
  Attribute of: *node*
  Description:
  This attribute specifies the alignment of text within a node frame.

- **title**
  Type: *string*
  Default value: no default
  Attribute of: *node*
  Description:
  This specifies the unique string identifying the node. This attribute is mandatory for the node specification.

- **useractioncmd3**Type: *string*
  Default value: empty string
  Attribute of: *node*

Description:
This attribute enables a command to be specified for a node that can be executed in the User Actions Mode by selecting the node and pressing **3**.

For details on user actions, see section User Actions (p. 24).

- **vertical_order**
  Type: *integer* or *maxlevel*
  Default value: none (corresponds to -1)
  Attribute of: *node*
  Description:
  In a hierarchical layout, this attribute specifies the vertical position of a node. *maxlevel* tries to position the node at the maximum calculated level. Generally speaking, the vertical position of nodes is called *level* or *rank* (see p. 103), with **level** being a synonym for **vertical_order**.

  All level 0 nodes form the uppermost layer (the first layer), if the orientation is top-down. Level 1 nodes form the second layer, etc.

  The level specification is not in effect unless if an automatic layout is being calculated. The layout is calculated automatically if there is at least one node without a specified location (see **loc** attribute for nodes (p. 81) and summary nodes (p. 64).

  Note: The level specification may conflict with a *near edge* (p. 31) specification, because the source and target node of a near edge have to have the same level. In this case, the level specification of the source or target node of the near edge is ignored.

- **width**
  Type: *integer*
  Default value: (node label width) pixels
  Attribute of: *node*
  Description:
  This attribute specifies the width of a node including the border. See also **height** node attribute (p. 79).

5.9 Edge Attributes

This section describes the whole list of edge attributes. Each attribute is listed together with its type and default value.

- **anchor**
  Type: *integer*
  Default value: no default value
  Attribute of: *edge*
  Description:
  An anchor point describes the vertical position in a node where an edge starts. This is useful if node labels are several lines long and outgoing edges are related to label lines. (For instance, this allows a nice visualization of structs containing pointers as fields). A node with anchored edges can only have one near edge at maximum. Further, if anchored edges occur, the graph `orientation` is always `top_to_bottom`.

- **arrowcolor**, **backarrowcolor**
  Type: *black*, *blue*, *red*, ...
  Default value: value of the **color** attribute for both
  Attribute of: *edge*
  Description:
  Respectively the color of the arrow head or backarrow head. For details on colors, see p. 90. See also edge attributes **textcolor** (p. 88) and **color** (p. 86).

- **arrowsize**, **backarrowsize**
  Type: *integer*
  Default value:
  10 pixels for the **arrowsize** and
  0 pixels for the **backarrowsize**
  Attribute of: *edge*
  Description:
  The arrow head is a right-angled, isosceles triangle. These two attributes respectively specify the length of the cathetuses of the arrow head and back arrow head.

- **arrowstyle**, **backarrowstyle**
  Type: *”none”*, *”solid”*, *”line”*, *”circle”*, ...
  Default value:
  ”solid” for the **arrowstyle** and
  ”none” for the **backarrowstyle**
  Attribute of: *edge*
  Description:
  Each edge has two arrow heads, one arrow head pointing to the target node, called normal arrow head, and the other one “pointing” to the source node, called back arrow head. **arrowstyle** is the style of the normal arrow head and **backarrowstyle** the style of the back arrow head. The styles are:
  - *”none”*
    No arrow head is drawn.

  - *"solid"*
    Arrow head is a triangle filled with **arrowcolor**, or the edge color if no **arrowcolor** is specified.
  - *"line"*
    Arrow head is not filled and drawn without the hypotenuse of the triangle forming the head.
  - *"circle"*
    Arrow head is a circle.
  - *"filled circle"*
    Arrow head is a circle filled with **arrowcolor**, or the edge color if no **arrowcolor** is specified.
  - *"half circle"*
  - *"half filled circle"*
  - *"box"*
  - *"filled box"*
  - *"half box"*
  - *"half filled box"*
  - *"rhomb"*
  - *"filled rhomb"*
  - *"half rhomb"*
  - *"half filled rhomb"*
  - *"diamond"*
  - *"half diamond"*
  - *"slash"*
  - *"dslash"*

  The default edges have a *solid* arrow head and *no* back arrow head.

- **class**
  Type: *integer*
  Default value: 1
  Attribute of: *edge*
  Description:
  This attribute specifies the edge class to which an edge belongs. For details, see p. **??** and the grouping of nodes in general (p. 17).
- **color**
  Type: *black, blue, red, ...*
  Default value: *black*
  Attribute of: *edge*
  Description:

Here the **color** attribute specifies the color of the edge. For details on colors, see p. 90. See also edge attributes **textcolor** (p. 88) and **arrowcolor** (p. 85).

- **fontname**
  Type: *string*
  Default value: default font is drawn by turtle graphics routines
  Attribute of: *node*
  Description:
  This attribute specifies a pixel font which is different from the default font and to be used for drawing the edge labels. This font is given by the name of the **aiSee** font file that containing the font description, for example a 12-point Times Roman font can be specified via `fontname: "timR12"`. Note: If the font file is not in the current directory the environment variable `AISEEFONTS` has to be set to the directory containing the font description files. For details on fonts, see p. 92.

- **horizontal_order**
  Type: *integer*
  Default value: unspecified
  Attribute of: *edge*
  Description:
  In a hierarchical layout, this attribute specifies the horizontal position of long edges. This is only of interest if an edge crosses several levels (see node attribute **horizontal_order**, p. 79). This attribute specifies the point where the edge crosses the level.

  The nodes specified by horizontal positions are ordered according to these positions within a level. The horizontal position of a long edge crossing a level specifies the two nodes of the level between which the edge has to be drawn. Other edges not possessing this attribute are inserted in this ordering by the crossing reduction mechanism (see p. 105).

- **label**
  Type: *string*
  Default value: empty string
  Attribute of: *edge*
  Description:
  This attribute specifies the text label of an edge. It is drawn if the graph attribute **display_edge_labels** (p. 55) is set to *yes*.

  Note: This string may contain control characters, e. g. `\n` (newline character), that influence the size of the label. See p. 94 for more details.

- **linestyle**
  Type: *continuous*, *dashed*, *dotted*, *double*, *invisible*, *solid*, *triple*
  Default value: *continuous*
  Attribute of: *edge*
  Description:
  This attribute specifies the style in which an edge is drawn. The following possibilities are offered:
  - *continuous*
    The edge is drawn as a solid line (——).

  - *dashed*
    The edge consists of single dashes (– – –).
  - *dotted*
    The edge consists of single dots (· · ·).
  - *double*
    The edge consists of a solid double line (═══).
  - *invisible*
    The edge is not drawn. The attributes of its shape (color, thickness) are ignored.
  - *solid*
    The edge is drawn as a solid line (same as *continuous* parameter).
  - *triple*
    The edge consists of a solid triple line (≡≡≡)

- **priority**
  Type: *integer*
  Default value: 1
  Attribute of: *edge*
  Description:
  The positions of the nodes are mainly determined by the incoming and outgoing edges. The edges can be imagined as rubberbands pulling a node to its position. The priority of an edge corresponds to the strength of the imaginary rubberband acting on it.

- **source**
  Type: *string*
  Default value: no default
  Attribute of: *edge*
  Description:
  This attribute specifies the title of the source node of an edge. It is mandatory for the edge specification.

- **target**
  Type: *string*
  Default value: no default
  Attribute of: *edge*
  Description:
  This attribute specifies the title of the target node of an edge. It is mandatory for the edge specification.

- **textcolor**
  Type: *black*, *blue*, *red*, ...
  Default value: value of the **color** attribute
  Attribute of: *edge*
  Description:
  Specifies the color of the text label of an edge. For details on colors, see p. 90. See also node attributes **color** (p. 86) and **arrowcolor** (p. 85).

- **thickness**

Type: *integer*
Default value: 2
Attribute of: *edge*
Description:
Specifies the thickness of an edge.

5.10 Colors

aiSee has a color map of 256 colors of which 254 can be used. 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 being specified by their color map index number. The color map is changed by specifying a sequence of color entry attributes. For instance,

```
colorentry 32:  210 218 255 // alice blue
colorentry 33:  205  92  92 // indian red
colorentry 34:   46 139  87 // sea green
```

introduce the colors “alice blue”, “indian red” and “sea green” with color index 32, 33 and 34. If blue, a default color, is to be used specification can be done via `color: blue` or `color: 1`. In order to use sea green as a node color, `color: seagreen` cannot be specified since “seagreen” is unknown to **aiSee** as a name. Its index number `color: 34` has to be specified instead.

The default colors can be overwritten, however this is even more tricky. Specifying

```
colorentry 1:  210 218 255 // alice blue
colorentry 2:  205  92  92 // indian red
colorentry 3:   46 139  87 // sea green
```

causes the default colors blue, red and green to be overwritten. Now if `color: blue` is specified, the color alice blue appears.

Table 5.1 shows the default color map.

No.	Color name	RGB triple	No.	Color name	RGB triple
0	*white*	255 255 255	16	*lightblue*	128 128 255
1	*blue*	0 0 255	17	*lightred*	255 128 128
2	*red*	255 0 0	18	*lightgreen*	128 255 128
3	*green*	0 255 0	19	*lightyellow*	255 255 128
4	*yellow*	255 255 0	20	*lightmagenta*	255 128 255
5	*magenta*	255 0 255	21	*lightcyan*	128 255 255
6	*cyan*	0 255 255	22	*lilac*	238 130 238
7	*darkgrey*	85 85 85	23	*turquoise*	64 224 208
8	*darkblue*	0 0 128	24	*aquamarine*	127 255 212
9	*darkred*	128 0 0	25	*khaki*	240 230 140
10	*darkgreen*	0 128 0	26	*purple*	160 32 240
11	*darkyellow*	128 128 0	27	*yellowgreen*	154 205 50
12	*darkmagenta*	128 0 128	28	*pink*	255 192 203
13	*darkcyan*	0 128 128	29	*orange*	255 165 0
14	*gold*	255 215 0	30	*orchid*	218 112 214
15	*lightgrey*	170 170 170	31	*black*	0 0 0

Table 5.1: Color Codes of the Default Color Map

Table 5.2 lists the color attributes available in GDL.

Name	Attribute of	Default value
color	Nodes, top-level graph background, subgraph background, summary nodes	`white`
color	Edges, summary edges	`black`
textcolor	Nodes, subgraphs, summary nodes	`black`
bordercolor	Nodes, subgraphs, summary nodes	The value of *textcolor*
arrowcolor	Edges, summary edges	The same color as the edge itself
backarrowcolor	Edges, summary edges	The same color as the edge itself

Table 5.2: GDL Color Attributes

Though there is no attribute for specifying the background color of additional node information windows, it should be noted that this color is not bound to the RGB constant `255 255 255`, but rather to the value of the first entry of the color map (which happens to be `255 255 255` by default). Thus, for example, specifying

```
colorentry 0:  210 218 255
```

sets the background color for all additional node information windows to alice blue. However, this color then also appears everywhere else where the color white is used, so you might have to also specify

```
colorentry 32:  255 255 255
```

and use the color number 32 instead of white.

The ASCII formfeed character `\f` (see p. 94) followed by two digits signifying a color changes the color for the characters following in a string, thus enabling the first 100 colors of the color map to be accessed.

5.11 Icons and Additional Fonts

5.11.1 Icons

Instead of text labels, **aiSee** also offers the option of displaying icon files in nodes. The icon files have to be in PNG, PBM, PGM, or PPM format (raw). They can be attached to summary nodes or ordinary nodes by using the **iconfile** attribute in subgraph (p. 60) or node (p. 79) specifications. Note: In order to prevent the title text from being drawn over the icon displayed, the label attribute of the node or summary node has to be set to the empty string, i. e. `label: ""`.

The search path for the icon files can be specified by the environment variable `AISEEICONS`.

Some additional graph attributes influence the drawing of icons, see **icons** (p. 60), **fast_icons** (p. 57) and **iconcolors** (p. 59).

5.11.2 Fonts

Apart from the default font, **aiSee** offers the fonts described in Table 5.3.

These pixel fonts are compatible with Adobe PostScript fonts. When exporting **aiSee** files to PostScript, the original PostScript fonts can be used (providing your PostScript printer is equipped with these fonts available).

They can be specified by the **fontname** attribute for summary nodes (p. 58), ordinary nodes (p. 78), and edge labels (p. 87). The string value of this attribute is the font file name of the font to be used. The font file name is composed of the file name base and size:

`fontname:` *<filename_base><size>*

The available font sizes are: 8, 10, 12, 14, 18 and 24 points. For example, a size 12 Times Roman font is specified via `fontname: "timR12"`.

Filename base	Corresponding PostScript font
`courB`	Courier Bold
`courBO`	Courier Bold Oblique
`courO`	Courier Oblique
`courR`	Courier
`helvB`	Helvetica Bold
`helvBO`	Helvetica Bold Oblique
`helvO`	Helvetica Oblique
`helvR`	Helvetica
`timB`	Times Bold
`timBI`	Times Bold Italic
`timI`	Times Italic
`timR`	Times Roman
`symb`	Symbol
`ncenB`	New Century Schoolbook Bold
`ncenBI`	New Century Schoolbook Bold Italic
`ncenI`	New Century Schoolbook Italic
`ncenR`	New Century Schoolbook

Table 5.3: Available **aiSee** Pixel Fonts

The search path for the **aiSee** font files can be specified by the environment variable `AISEEFONTS`.

5.11.3 Compatibility with SVG

We highly recommend using the bitmap fonts when exporting graphs to SVG format. The standard vector font is translated into `<line>` elements, whereas the bitmap fonts are translated into `<text>`. SVG treats `<text>` as text rather than outlines, enabling the user to search SVG images for node labels by using the Find dialog in the SVG viewer. Also, when bitmap fonts are used, the output SVG file is much smaller and loads much faster.

An iconfile specification is translated into an `<image>` element. The `<image>` element indicates to the SVG viewer used that the contents of the icon file specified are to be rendered into a given rectangle within the current user coordinate system. SVG viewers usually support at least PNG, JPEG and SVG format icon files.

Tables 5.4 and 5.5 show how the various bitmap fonts are translated into SVG.

Filename base	SVG font-family
`courB`	`"Courier New-Bold"`
`courBO`	`"Courier New-BoldOblique"`
`courO`	`"Courier New-Oblique"`
`courR`	`"Courier New"`
`helvB`	`"Helvetica-Bold"`
`helvBO`	`"Helvetica-BoldOblique"`
`helvO`	`"Helvetica-Oblique"`
`helvR`	`"Helvetica"`
`timB`	`"Times-Bold"`
`timBI`	`"Times-BoldItalic"`
`timI`	`"Times-Italic"`
`timR`	`"Times"`
`symb`	`"Symbol"`
`ncenB`	`"NewCenturySchlbk-Bold"`
`ncenBI`	`"NewCenturySchlbk-BoldItalic"`
`ncenI`	`"NewCenturySchlbk-Italic"`
`ncenR`	`"NewCenturySchlbk"`

Table 5.4: **aiSee** Pixel Fonts and the Corresponding SVG Fonts

GDL font size	SVG font-size
`08`	`"10"`
`10`	`"12"`
`12`	`"14"`
`14`	`"16"`
`18`	`"20"`
`24`	`"26"`

Table 5.5: **aiSee** Pixel Font Sizes and the Corresponding SVG Font Sizes

\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 5.6: ISO Latin 1 Character Set

5.12 Character Set

It is possible to switch the text colors or underlining within text output, e. g. drawing of labels or info fields. This is controlled by special characters in the strings. Note: The ASCII value of the control characters depends on the operating system and C compiler. The following control characters are allowed:

- `"\n"` (ASCII code 10)
  Newline: drawing of text continues at the beginning of the next line.
- `"\t"` (ASCII code 9)
  Tab: corresponds to eight space characters.
- `"\a"` (ASCII code 7)
  Beep: produces an audible or visible alert. The position of the next character is not to be changed.
- `"\b"` (ASCII code 8)

Backspace: goes one character back and continues drawing there.

- `"\f"` (ASCII code 12)
  Formfeed: is used in combination with an additional parameter to change the current form of output.
  - `"\fu"` (ASCII codes 12 117) starts underlining.
  - `"\fI"` (ASCII codes 12 73) starts italics typeface.
  - `"\fb"` (ASCII codes 12 98) starts bold typeface.
  - `"\fB"` (ASCII codes 12 66) starts very bold typeface.
  - `"\fn"` (ASCII codes 12 110) stops underlining, italics and bold typefaces, i. e. returns to normal typeface.
  - **Selecting special characters**
    aiSee now supports UTF-8. However, for backwards compatibility reasons, certain special characters (all characters of the ISO Latin 1 character set) can also be ”encoded” rather than entered directly, as specified in table 5.6. For example:
    - `"\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).
  - **Changing of current color for drawing characters** The formfeed character followed by two digits signifying the color chosen from table 5.1 changes the color for the characters following in a string, thus enabling the first 100 colors of the map to be accessed. Example:
    - `"\f00"` (ASCII codes 12 48 48) sets the color to white
    - `"\f31"` (ASCII codes 12 51 49) sets the color to black

5.13 Remarks

All titles of graphs and nodes have to be unique.

A node can only have two near edges. If more than two near edges are specified for a node, the remaining near edges are converted into normal edges.

A node with anchored edges can only have one near edge at maximum. Further, if anchored edges occur, the orientation is always *top_to_bottom*.

The level of nodes (also of summary nodes) is not recognized unless the whole graph is drawn automatically, menaing unless at least one node has no specified location. Normally, all level 0 nodes form the uppermost layer, with nodes of other levels forming the next layer underneath. The level specification may conflict with a *near edge* specification, because the source and target node of a near edge have to have the same level. In this case, the level specification of the source or target node of the near edge is ignored.

5.14 GDL's Grammar

This section presents the grammar of GDL (Graph Description Language) in EBNF (Extended Bacchus Naur Form).

- Terminals are enclosed in “**double quotes**” and printed in bold.
- Nonterminals are written in *italics*
- Finite iterations are specified by (...)*

graph : “**graph:** {” (*graph_entry*)* “}”
graph_entry : *graph_attribute*
| *graph*
| *node*
| *edge*
| *node_defaults*
| *edge_defaults*
| *foldnode_defaults*
| *foldedge_defaults*
| *backedge*
| *nearedge*
| *lnearedge*
| *rnearedge*
| *bentnearedge*
| *lbentnearedge*
| *rbentnearedge*
| *region*
graph_attribute : *graph_attribute_name* “**:**” *attribute_value*
graph_attribute_name : any attribute listed in section 5.7
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*)* “}”
lnearedge : “**leftnearedge:** {” (*edge_attribute*)* “}”
rnearedge : “**rightnearedge:** {” (*edge_attribute*)* “}”
bentnearedge : “**bentnearedge:** {” (*edge_attribute*)* “}”
lbentnearedge : “**leftbentnearedge:** {” (*edge_attribute*)* “}”
rbentnearedge : “**rightbentnearedge:** {” (*edge_attribute*)* “}”
region : “**region:** {” (*region_attribute*)* “}”
node_attribute : *node_attribute_name* “**:**” *attribute_value*
edge_attribute : *edge_attribute_name* “**:**” *attribute_value*
region_attribute : “**source**” “**:**” *string_list*
| “**target**” “**:**” *string_list*

| “**state**” “**:**” *enum_status*
| “**class**” “**:**” *integer_list*
| “**range**” “**:**” *integer_value*
node_attribute_name : any attribute listed in section 5.8
edge_attribute_name : any attribute listed in section 5.9
attribute_value : *integer_value*
| *float_value*
| *string_value*
| *enum_value*
integer_list : (*integer_value*)*
integer_value : any integer constant in **C** style
float_value : any float constant in **C** style
string_list : (*string_value*)*
string_value : “`"`” (*character*)* “`"`”
enum_value : any possible key word value for a graph, node or edge attribute
enum_status : any possible key word value for the graph attribute **state** (see p. 70)
character : any printable **ASCII** character

Note that `graph:`, `node:` and `edge:`, etc. are keywords. Therefore, no whitespace[1] character is allowed before these colons.

Integers are sequences of digits. Floating point numbers consist of a sequence of digits followed by a dot ‘.’, followed by a sequence of digits. C style comments (`/* ... */`) and C++ style comments (`//...`) are allowed.

5.15 Animation of layout phases (aka smooth transitions)

If a new layout is calculated for a graph and if animation of layout phases is turned on, the nodes move smoothly from their initial positions in the layout to the newly calculated ones, enabling the user to keep visual track of layout changes. The GDL graph attribute `amax: <Int>` specifies the maximum number of animation steps. Specifying 0 means animation is turned off.

---

[1] A whitespace is a blank, tab, linefeed or newline character

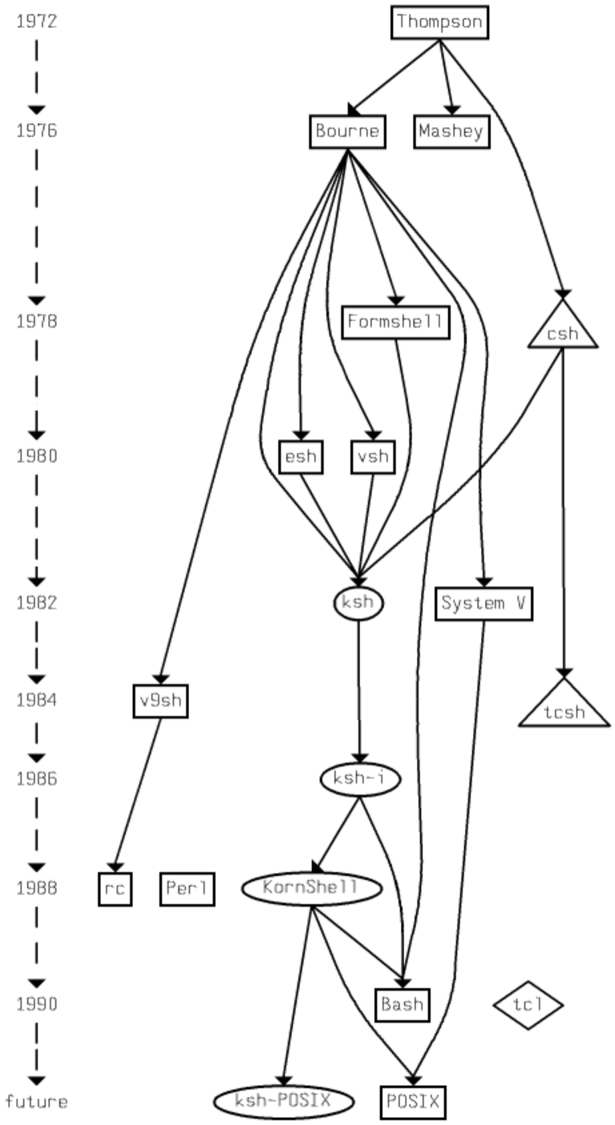


Figure 5.17: Development of Shells

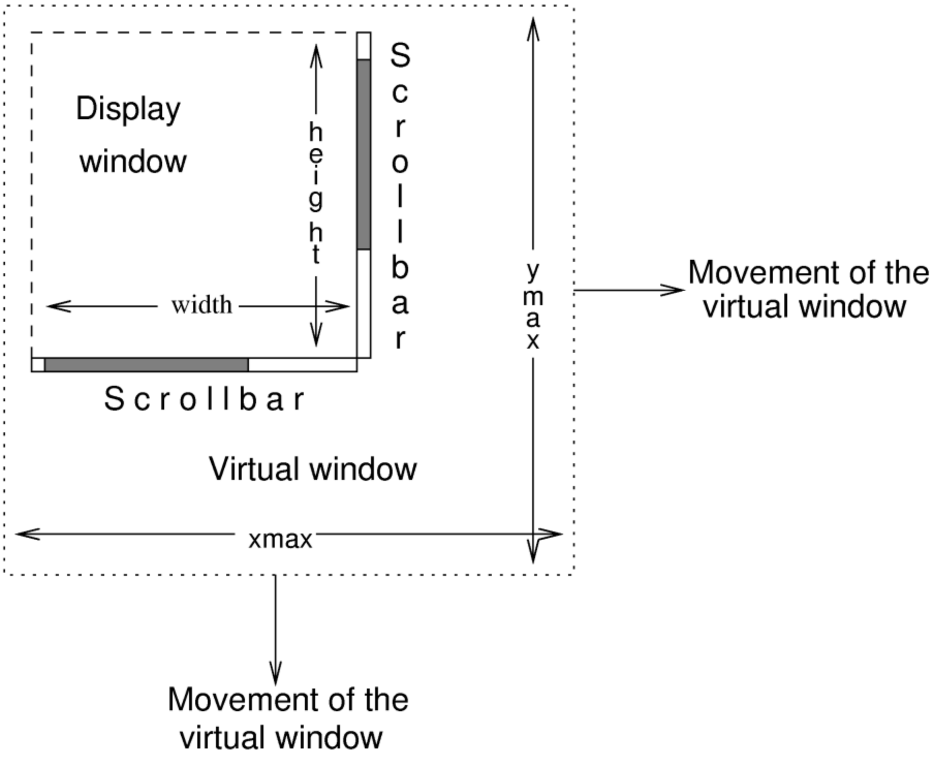


Figure 5.18: Displayed Window and Virtual Window

6 Overview of the Layout Phases

The task of **aiSee** is to parse a graph specification, calculate a layout according to the layout algorithm chosen, and draw the resulting layout in a window. The specification given as input to **aiSee** is a readable ASCII text. The output window can be used to browse through the graph, shrink or magnify the graph, fold parts of the graph, and export the graph to a bitmap or PostScript file. Graph folding potentially results in a new layout of the graph. The layout can be extensively influenced by edge, node and graph attributes or by different layout algorithms. **aiSee** offers force-directed and hierarchical layout schemes.

6.1 Parsing

The first phase is to parse the specification and construct internal data structures representing the graph. The specification may contain attributes denoting initially folded parts of the graph.

6.2 Grouping of Nodes and Edges – Folding Phase

aiSee offers various ways of grouping nodes and edges (see p. 17).

Folding a graph enables the graph to be inspected in a more efficient manner. Folding the unimportant parts gives the more important parts more space for visualization. Folding parts of the graph also improves **aiSee**'s performance. It should be noted that nested foldings are possible.

In graph specifications, the folding of regions or subgraphs may interfere with the hiding of edges. In this case, first the summary node of the folded region or subgraph is calculated, followed by hiding the edges.

6.3 Force-Directed Layout

Force-directed layout schemes are usually selected for undirected graphs, this being ideal for simulating physical and chemical models.

aiSee combines the following four ideas in its force-directed layout algorithm:

- **Spring forces**
  A spring embedder is simulated. The nodes of a graph are regarded as electrically charged particles that repel one another, the edges being regarded as springs connecting the particles. Particles that are far away from one another attract each other by spring forces, particles that are too close repel one another.

- **Magnetic forces**
  Spring forces do not take the direction of edges into account. In directed graphs all edges should have a uniform direction to point in. Here the edges are interpreted as magnetic needles that align themselves according to a magnetic field.
- **Gravitational forces**
  The problem with spring forces is that they are only effective in connected graphs. In unconnected graphs simulating a spring embedder makes unconnected nodes move away from one another as there are only repulsive forces but no attractive forces. That is why gravitational forces are introduced. All nodes are attracted to the bary center of all the other nodes.
- **Simulated annealing**
  The simulated annealing model is oriented to the physical process of annealing, which often leads to very regular structures (e.g. like crystals).

  A global energy level is computed for a graph which is the sum of all energy levels of the nodes. The energy level at a node is determined from the forces acting it, much like the elongation of the springs. The spring embedder tries to minimize the global energy level by moving the nodes in the direction of the forces.

  Nodes are randomly moved so as to avoid being trapped at a local energy minimum. At the beginning this is done more vigorously, with random movement being ceased towards the end in order to stabilize the final layout. The amount of random movement depends on the ”temperature”, which is controlled by a temperature scheme.

  aiSee also supports the concept of local temperatures for a node. The temperature takes the local situation of the graph into account (see **energetic** graph attribute, p. 56) and regulates how much and how often a node is randomly moved.

The force-directed placement algorithm consists of four phases:

- **Initialization phase**
- **First iteration phase**
- **Optional second iteration phase**
- **Final improvement phase**

The two iteration phases are conceptionally the same. They simply sequentially simulate the two magnetic fields acting on the system. The second iteration phase is omitted if there is only one magnetic field.

One iteration phase consists of a loop of iteration steps that are executed until the global temperature value has fallen below a specified threshold value or until a maximum number of iterations is reached.

In each iteration step the new impulses of the nodes (force directions) are calculated, the nodes moved to their new positions according to the impulses, and the global temperature adjusted.

In the final improvement phase, the node positions can be rastered. This is done by moving a node to its closest raster point after each iteration step.

Finally the minimum x and y coordinates are calculated and the entire layout is moved so that the minimum coordinates are just zero. This step is necessary because all nodes move during the iteration phase, meaning they could have all moved away from the origin of the coordinate system. Finally the start and end points of the edges are calculated.

For more details, see [Sa96].

6.4 Hierarchical Layout

6.4.1 Rank Assignment

After folding, all the visible nodes are determined. If all the visible nodes have been specified by the user using valid coordinates, the graph is drawn immediately. However, if the coordinates of at least one node are missing, an appropriate layout has to be calculated. The first pass places the nodes in discrete ranks. All nodes of the same rank appear at the same vertical position.

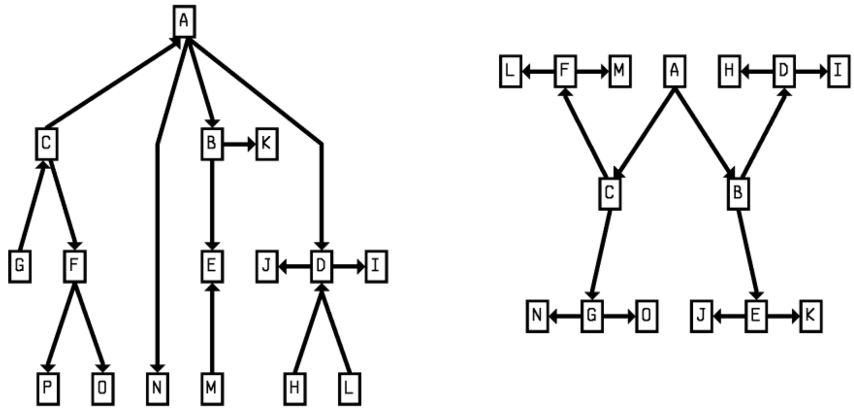
There are many possibilities for assigning rank. The normal method is to calculate a spanning tree by determining the strongly connected components of the graph. All edges should be oriented top-down. A heuristic tries to find a minimum set of edges that cannot be oriented top-down.

A faster method is to calculate the spanning tree of a graph by depth first search (DFS). However, the order in which the nodes are visited has a substantial influence on the layout. The initial order of the nodes is the order given by the graph specification. **aiSee** offers various versions of such methods:

- *dfs*:
  Calculates the spanning tree by one single DFS traversal. This is the fastest method, but the quality of the result might be poor for some graphs.
- *maxdepth*:
  Calculates the spanning tree by DFS using the initial order and the reverted initial order, followed by choosing the deepest spanning tree. This results in more levels, i.e. the graph is larger in the y direction.
- *mindepth*:
  Takes the flatter spanning tree of both DFS's. This results in fewer levels, or more nodes at the same levels, meaning the graph is larger in the x direction.
- *maxdepthslow*, *mindepthslow*:
  Whereas the above algorithms are fast heuristics for increasing or decreasing the depth of the layout, *maxdepthslow* and *mindepthslow* actually calculate a good order so as to obtain a maximum or minimum spanning tree. However, they pose one disadvantage: they are rather slow. Warning: A minimum spanning tree does not necessarily mean that the depth of the layout is minimal. However, good heuristics involves obtaining a flat layout (see the examples on p. 40).
- *maxdegree*, *mindegree*, *maxindegree*, *minindegree*
  maxoutdegree, *minoutdegree*:
  These algorithms combine DFS with node sorting. The sorting criteria are the number of incoming edges, the number of outgoing edges, and the number of edges all at the same node. Node sorting may have various effects and can sometimes be used as a fast alternative

to *maxdepthslow* or *mindepthslow*.

- *minbackward*:
  Instead of calculating strongly connected components, **aiSee** can also perform topological sorting to assign ranks to nodes. This is much faster, however it requires that the graph be acyclic.
- *tree*:
  This method is very fast, however it can't be used unless the graph is a forest of *downward laid-out trees*. A downward laid-out tree has the following structure: Each node at rank l has at most one adjacent edge coming from a node of an upper rank $k < l$. A node may have edges pointing to nodes at the same level and edges coming from nodes of lower ranks $k > l$. The direction of the edges may be arbitrary, but the picture of the layout (if the arrow heads are ignored) has to be a tree (see Figure 6.1). The assignment of ranks is done by DFS. Then, the graph is checked to determine whether it is a forest of downward laid-out trees. If this is not the case, the standard layout is used as a fallback solution. Crossing reduction (see next section) is not necessary for downward laid-out trees, meaning a very fast positioning algorithm can be used.



Structurally this is not a tree (e.g. many edges point to node D). However, the layout is tree-shaped, thus it is a "downward laid-out tree".

Structurally this is a tree, however the layout is not a "downward laid-out tree".

Figure 6.1: Downward Laid-out Trees and Structural Trees

A further possibility for influencing the layout is edge priority. Higher priority edges are preferable when calculating the spanning tree. After partitioning, a fine-tuning phase tries to improve the ranks in order to avoid very long edges.

6.4.2 Crossing Reduction

This pass calculates a good order of the nodes within levels ao as to avoid edge crossings. The first step is to separate connected components of the graph and to handle each component separately.

The crossing reduction algorithm calculates the weights of the nodes in keeping with the possible crossings to the left and the right, and reorders the nodes of a level according to these weights. The ordering of nodes within one level influences the weights of the adjacent levels, consequently this is performed iteratively until a user-defined maximum is reached or no more improvements are recognized. This is phase 1 of the crossing reduction.

If the weights of some nodes are equal a permutation of these nodes is tried. Sometimes a permutation enables crossings to be reduced even further (optional phase 2 of crossing reduction).

There are four possibilities to calculate weights for crossing heuristics. The default weights are *barycenter* weights [STM81], while *mediancenter* weights [GNV88] are sometimes more appropriate, especially if the average degree (number of edges) of nodes is small. *barymedian* weights are the combination of *barycenter* and *mediancenter*, where *barycenter* is considered first and *mediancenter* is only used for nodes whose *barycenter* weights are equal. Conversely, *medianbary* weights are the combination of *barycenter* and *mediancenter*, where *mediancenter* is considered first. The weights can be selected interactively in the Layout dialog box, or statically in the GDL specification. See graph attribute *crossing_weight* (p. 55).

However, the final result needn't necessarily be optimal as crossing reduction is only a heuristic.

Finally, a local optimization phase tries to improve the layout by exchanging directly neighbored nodes. See graph attribute *crossing_optimization* (p. 54)

6.4.3 Coordinate Calculation

Coordinate calculation follows after partitioning nodes into levels and ordering the nodes within the levels. Nodes can be aligned at the bottom or at the top of a level or centered at a level, with there being a minimum distance between levels (**yspace**). This influences the y coordinates. The x coordinate has to be calculated such that there is a minimum distance between nodes (**xspace**) and a minimum distance between the bend points of edges (**xlspace**). Furthermore, the layout should be balanced such that the edges are short and straight.

This achieved by using a special method for downward laid-out trees or performing two general iteration phases: The first phase simulates a physical pendulum, the nodes being the balls and the edges the strings. The balls hanging on the strings swing to and fro, i.e. the nodes move within their level and influence the neighboring nodes until the layout is sparse enough and each node has sufficient space to be favorably positioned.

Next the nodes are centered with respect to their edges. This phase simulates a rubberband network: The edges pull on a node with a power proportional to their length, the result being that 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.

An optional fine-tuning phase tries to recognize long edges and draws them as vertical long line segments. This is useful for the orthogonal layout methods.

Unfortunately, both physical simulations needn't be convergent, meaning they may be iterated infinitely often without resulting in a stable layout. However, these cases are seldom. The number of iterations is restricted in order to prevent infinite execution, the `t` message indicating that a timeout has occurred.

6.4.4 Edge Bending

If a graph contains nodes of different sizes, an edge starting at a very small node may be drawn through a neighbored large node. This situation is avoided by bending edges at certain points. In addition, if an orthogonal layout method is selected, the edges are bent so that only orthogonal line segments exist.

6.5 Drawing

Finally, the graph is drawn in a window or exported to a file. Edges can be drawn as polygon segments or splines, however spline drawing is slower (indicated by the `d` message).

Export into PostScript, SVG, or bitmap formats (BMP, PNG, etc.) is also possible.

Bibliography

[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, Department of Mathematics and Computer Science, 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

[Hi93] Himsolt, Michael: "Konzeption und Implementierung von Grapheditoren", Dissertation (in German), Universitaet Passau, Germany, 1993 published with *Berichte aus der Informatik*, Verlag Shaker, Aachen, Germany, 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 through the VCG Tool", technical report A03/94, Saarland University, Department 14, Computer Science, 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 through 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

[Sa96] Sander, Georg: *Visualisierungstechniken für den Compilerbau*, Dissertation, Pirrot Verlag & Druck 1996

[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

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

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

Index