Angewandte Mathematik und Informatik
Universität zu Köln

Report No. 96.242

The Tree Interface – Version 1.0 User Manual

by
Sebastian Leipert

1996

Partially supported by DFG-Grant Ju204/7-2, Forschungsschwerpunkt "Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen" and ESPRIT Long Term Research Project Nr. 20244 (ALCOM-IT).

Institut für Informatik
Universität zu Köln
Pohligstraße 1
50969 Köln

1991 Mathematics Subject Classification: 90CXX,68NXX
Keywords: Graphical Interface, Software devellopment, Graph Drawing

The Tree Interface

Version 1.0

A Tool For Drawing Trees

by

Sebastian Leipert

User Manual

September 1996

Institut für Informatik der
UNIVERSITÄT ZU KÖLN

Sebastian Leipert
Institut für Informatik
Universität zu Köln
Pohligstraße 1
50969 Köln
Germany
E-mail: leipert@informatik.uni-koeln.de

Writing this manual has been partially supported by

- DFG-Grant Ju204/7-2, Forschungsschwerpunkt "Effiziente Algorithmen für diskrete Probleme und ihre Anwendungen".
- ESPRIT Long Term Research Project Nr. 20244 (ALCOM-IT).

Contents

Chapter 1

Introduction

1.1 The *Tree Interface*

The *Tree Interface* is a tool specially designed to draw binary and general rooted trees, as they occur during an algorithmic process. It therefore covers three main areas of drawing trees: simply drawing a tree as a result of any kind of process, emulating a tree growing process after the computation is finished and drawing a tree during a computational process.

The first problem is easy to handle. The user might have some kind of tree stored in a file along with some information at every node and simply wants to take a look at it. So this case is kept very simple and loading such a file means painting the tree.

The second case is almost as easy to handle as the first case, although the file structure, which stores the tree, is larger. The main idea is to emulate the process that grew the tree after the process is finished. This might be preferred before a runtime simulation for several reasons. On the one hand, a process might last to long and the user is forced to keep an eye on the screen while nothing actually happens. On the other hand, a process might be very fast and produces a large tree, so that it is not possible to follow the action, that takes place. Instead of following the action online, it might be a good idea to reproduce the visualization of the process afterwards and then either speed this visualization up or slow it down. The *Tree Interface* provides a comfortable tool which enables the user to reproduce such visualizations.

Besides, situations may occur, where it is not possible at all to visualize a process. This eg. is typical for parallel algorithms which are run on multiple processor machines. Here the user has no other choice but emulating the process after it has been finished.

Nevertheless, the user might need a runtime simulation. This can be achieved by this tool as well in two different ways. The first is simply piping the output of some program to the *Tree Interface*. This is rather easy to handle, since the user just has to make sure that his output obeys a few rules (see section 3.6). The second option is to include the users program into the environment of the *Tree Interface* such that the program can be started by clicking on a button in the menu bar. This possibility involves some work for the user and is restricted to C++ implementations only, but offers a nice handling once the work is done. The necessary changes, that need to be programmed are described in detail in chapter 5.

The implementation of the *Tree Interface* is based on the *Graph Interface* which itself is based on the *Graphical Front End GFE* Version 1.0 both written by Joachim Kupke in C++. *GFE* itself is a library based on the *MotifApp* Application Framework and the library written for *OSF/Motif* by Young [You92]. For accurate manipulating and deriving our source code, we expect the user to have the library *libApp.a*, written by Young, the *GFE* and the *Graph Interface* library written by Kupke and of course the *X/Motif Libraries*.

1.2 The node-positioning algorithm

Drawing a tree consists of two stages: determining the position of each node, and actually rendering the individual nodes and their interconnecting branches. Therefore the heart of the *Tree Interface* is an algorithm for computing the coordinates of the nodes of the tree, while the actual drawing of the nodes is left to the *Graph Interface*.

The algorithm used by the *Tree Interface* bases on a publication by Walker [Wal90]. It is a node-positioning algorithm for general trees, which gives similar results as the famous algorithm presented by Reingold and Tilford [RT81] for binary trees and even better results for general trees.

The *Tree Interface* considers only rooted, directed trees, which are trees with one root and hierarchical connections from the root to its offsprings. No node may have more than one father. Since the *Tree Interface* draws general trees, there does not exist any restriction on the number of offsprings each node has. Binary trees are special trees, where each node has either two offsprings or no offsprings at all.

Walkers algorithm does not support a common practice, to distinguish between the left and the right son of a node in a binary tree, so the tree can be drawn to preserve this left to right distinction. Such trees are called ordered trees, and the algorithm does not support ordered tree drawing.

With the help of Walkers algorithm, the *Tree Interface* satisfies the following aesthetic rules while the drawing of the tree occupies as little space as possible (see [RS83, WS79]):

- Nodes at the same level of the tree should lie along a straight line. The straight lines defining the **levels** should be parallel. The user should be aware of the fact that it is therefore not possible to line all leaves of a parse tree in a horizontal line.
- A parent should be centered over its offsprings.
- A tree and its mirror image should produce drawings that are reflections of one another.
- A subtree should be drawn the same way regardless of where it occurs in the tree. This is necessary since in some applications, one wishes to examine large trees in order to find repeated patterns and the search for patterns is facilitated by having isomorphic subtrees drawn isomorphically. Therefore small subtrees should not appear arbitrarily positioned among larger subtrees:
  - Small interior subtrees should be placed out evenly among larger subtrees, while the larger subtrees are adjacent at a larger level.
  - Small subtrees at the far left or far right should be adjacent to larger subtrees.

Every node is identified by a set of (x, y) coordinates, determining a point in the plane. Those coordinates of the nodes are computed with respect to the aesthetic rules listed above, observing the following important values:

Level Separation is the fixed distance of adjacent levels of the tree. This value is used in order to determine the y-coordinates of the nodes.

Sibling Separation is the minimum distance between adjacent siblings of the tree.

Subtree Separation is the minimum distance between adjacent subtrees of a tree.

All values have a standard value set to 4, except the *Level Separation* value which has the default value 2, but all values can be influenced by the user (see 2.4.3 and 6.4.2)

In the presentation of a tree, edges are normally drawn as lines while nodes are drawn as circles. Apart from the fact that both appearances can be changed (see 2.4.3, 6.3.5 and 6.3.6), the *Tree Interface* normally draws nodes as circles centered above their coordinates having a radius of 1. Of course the radius of the nodes can be changed by the user in order to vary the layout of the trees, but internally the *Tree Interface* handles the space within a range of 1 of the node defining point as safety area. This strategy forbids drawings where nodes intersect each other and is achieved by simply adding 2 to the *Separation* values presented above (so if eg. the *Level Separation* = 2, then it has actually a value of 4).

This manual does not go into detail of the node-positioning algorithm. The reader is therefore referred to [Wal90]. Nevertheless, the code of our implementation was enclosed in our appendix in section 7.3 and we remark that the time needed by this algorithm is in $O(n)$ where n is the number of nodes in the tree.

1.3 The *Tree Interface* Window

When calling the *Tree Interface*, a window will appear on the screen, that handles all features of the *Tree Interface*. This window is divided into several parts, that will be referred to throughout this manual. Therefore a brief description of the window is given in this section.

Besides the menu bar, that handles most of the applications of the *Tree Interface* (see chapter 6 for a complete reference of all commands), the window is divided into two parts (see figure 1.1):

- the *display area*
- and the *draw area*.

The *display area* is the smaller part of the window, placed just below the menu bar. Its task is to show all kinds of printed messages to the user. Such messages can be warnings, *main* informations of nodes (see 4.2.2), confirmations of commands and a various number of different informations. Even the user is able to print out strings in the display area within a few applications (see chapter 5).

Actually, the display area consists of two different displays: a left display and a right display. When the *Tree Interface* is started, only the right display is visible. The left display will appear next to the right window, as soon any information is printed into it. Normally the ratio between the two displays is 50 percent, but it can be manipulated by the user at will (see 2.4.3.2).

The right display is endowed with scroll bars (only visible if necessary) and keeps all information displayed up to a certain extend. This supports the user to keep track of the displayed information, while the left display shows only one information at a time. As soon as new information is printed into the left display, all information shown in the left display before is removed.

The *draw area* is the large display underneath the display area. This area displays all paintings of trees and is the most referred feature of the *Tree Interface* window throughout this manual. Since it shows the drawings of the trees, its size is normally much larger than the size of the display area. Nevertheless, it is possible to change the ratio of the draw area and the display area with the help of a *resize button*. This very small button is on the far right side of the *Tree Interface* window just between the two described areas. In order to use this button, the mouse cursor has to be positioned right on top of this button. To indicate, that the user has been successful in doing this, the cursor shape changes from the arrow shape to cross hair shape. Pressing either the left or the middle mouse button and keeping it pressed, the ratio of the two areas can be resized.

The draw area shows the nodes of the tree in any kind of colour that was chosen by the user. Unfortunately, the *Graph Interface* is not able to present different kinds of shadings yet, so for good application we strongly suggest to use a colour monitor. Three colours, that are used by the *Tree Interface* by default are to be mentioned here:

Standard Colour is by default indian red. It is used for normal presentation of nodes.

Standard Highlight is by default green1. It is used to highlight nodes.

Standard Shade is by default snow2. It is used to insinuate nodes, that are not present yet.

The background colour is normally wheat, while the edge colour is normally black.

Of course all colours can be changed by the user at will (see 2.4.3, 6.3.4 – 6.3.6), so it should be guaranteed that the drawings can be adapted to the users aesthetical beliefs.

To be more precise, Standard Colour, Standard Highlight and Standard Shade are not only names of colours, they are names for **node categories**. A node categorie is a conglomeration of descriptions of the appearance of a node, including:

- colour of the node,
- font used to show the number of the node,
- colour of the font,
- a flag whether the number has to be shown,
- a flag whether the node is drawn as a circle or a square,
- a flag whether the node is drawn filled or not,

- a custom name.

So every node is assigned to one of the 20 node categories that are in the *Tree Interface* present, the default value is always Standard Colour. Since the conspicuous attribute of a node categorie is the colour of the node, node categories are often referred as node colours. This should also explain, why the default node categorie is called Standard Colour.

Not only the colour of a node categorie can be changed as mentioned above, but also every single feature of it can be replaced, in order to satisfy special purposes. In order to change the different features, the reader is referred to 2.4.3, 6.3.4 – 6.3.6.

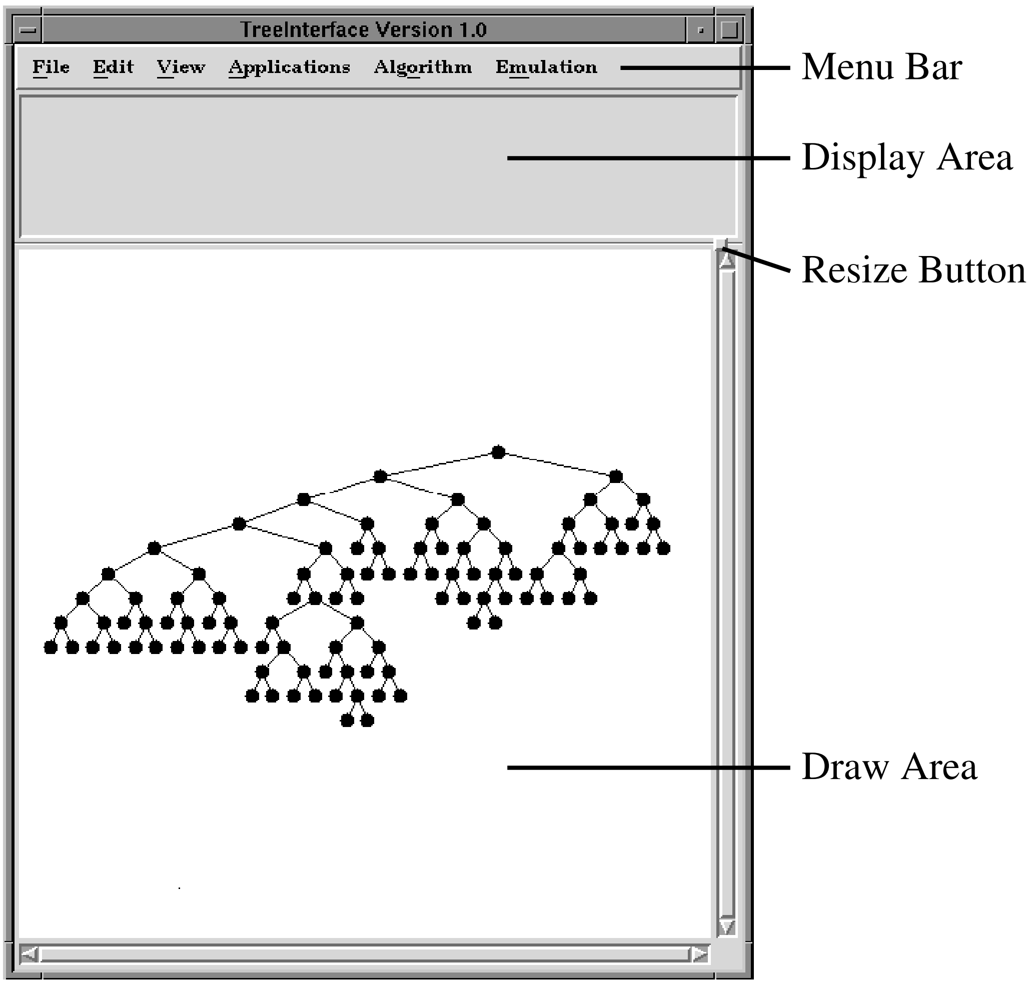
TreeInterface Version 1.0
File Edit View Applications Algorithm Emulation
Menu Bar
Display Area
Resize Button
Draw Area

Figure 1.1: The *Tree Interface* window

Chapter 2

Installing the *Tree Interface*

In order to install the *Tree Interface* we expect the user to have a package of the *Tree Interface* including an executable corresponding to his operating system. These packages are currently available under the following adress:

`http://www.informatik.uni-koeln.de/ls_juenger/projects/vbctool.html.`

Along with the executable, the user will find the following files in his package:

`vbcAppDefaults`: a file looked up by the *Tree Interface* when the interface is started using the fast installation (see 2.1).

`GFE`: the same file as vbcAppDefaults, supposed to be used for a firm installation of the *Tree Interface*.

`GRAPHResource`: a subdirectory containing several files used by *Tree Interface*.

`startVbcTool`: a shell-script needed for instant use of the *Tree Interface*.

`vbctool`: the executable of the *Tree Interface*. The name stands vor **V**isualization of **B**ranch and **C**ut algorithms and is historically motivated, since it was first used in the **ABACUS** system of Stefan Thienel [ST95].

2.1 Fast installation

In case that the user prefers a simple and fast installation that provides all functionalism, all files of the package have to be kept in one directory with the `GRAPHResource` directory as subdirectory. Type

```
startVbcTool
```

and the *Tree Interface* can be used immediately. Nevertheless, for intense use of the *Tree Interface*, we strongly recommend a firm installation of the package as described in the next section 2.2.

When using the *Tree Interface* with the help of the script `startVbcTool`, the *Tree Interface* depends on the existence of the file `vbcAppDefaults`. The user is free to manipulate this file at his will for changing size and appearance of the *Tree Interface*. Since the file `vbcAppDefaults` is manipulated as the file `GFE` we refer to the section 2.3.

2.2 Firm installation

In order to install the *Tree Interface* the user has to establish two different subdirectories:

1. `app-defaults`
2. `GRAPHResource`

The first one has to be placed into the main directory of the user, while the `GRAPHResource` directory that comes along with the package of the *Tree Interface*, can be placed anywhere. The user just has to make sure that the following two facts are established:

1. The environmental variable `XAPPLRESDIR` has to be set. If the C shell is used, the following line has to be written into the `.login` file:

   ```
   setenv XAPPLRESDIR $HOME/app-defaults/
   ```

   If the Bourne shell or Korn shell is used, the following lines have to be established:

   ```
   setenv XAPPLRESDIR
   XAPPLRESDIR=$HOME/app-defaults/; export XAPPLRESDIR
   ```

2. The path of the `GRAPHResource` subdirectory has to be set in the files that are contained in the directory `app-defaults` as `ResourceDirectoryPath` (see 2.3.2).

2.3 The directory `app-defaults`

The directory `app-defaults` has to contain a file called `GFE`. This file `GFE` comes along with the package of the *Tree Interface* and is looked up by the *Tree Interface* every time the interface is called. `GFE` mainly carries information about the appearance of the interface, eg. the size of the windows or the fonts used, which enables *Motif* to build up the window correctly.

This manual gives only a short introduction to the user for changing sizes or fonts and concentrates on structures which are of more importance for the user.

2.3.1 Changing sizes, fonts and colours

The *Tree Interface* consist of different *elements* as the *draw area*, the *display area* (see also 1.3), the menu bar and pop up menus. Every element can be addressed by its name. Together with the commands `width`, `height`, `fontlist`, followed by a proper value, it is possible to achieve appropriate changes.

Syntax: `*.<element>[*|.]<command>: <value>`

Values for width and height are nonnegative integer values indicating a number of pixels. The value of a font is a string. The strings describing the fonts are listed in the files `ListOfAdobeFonts` and `ListOfAllFonts` in the directory `GRAPHResource` (see 2.4.5).

Examples:

- `*.drawArea.width: 700` set the width of the draw area to 700 pixels.
- `*.Color/Context*fontList:`
  `-daewoo-gothic-medium-r-normal--0-0-100-100-c-0-ksc5601.1987-0`
  changes the font of the *Color/Context Chooser*. Try it, you get a nice result.

2.3.2 Adding the path of the `GRAPHResource` directory

In order to run the *Tree Interface*, the user has to add the path of his `GRAPHResource` directory in the file `GFE`, stored in the directory `app-defaults`. This file contains the command

```
*.ResourceDirectoryPath.labelString:
```

This command has to be followed by the path of the `GRAPHResource` directory.

If the command is missing or the path is wrong, the *Tree Interface* will crash after calling it, leaving with the message

```
NO RESOURCE-FILE CALLED GRAPHResource/GRAPHStandardResource.rsc
```

2.3.3 Changing Labels and Mnemonics

As described in chapter 6, all commands in the menu bar can be activated by pressing the *Meta* key and the key of the underlined letter of the commands name at the same time. The name of a command, written on the button of the menu is called a **label**, the underlined letter is called a **mnemonic**. All labels and mnemonics are listed in the `GFE` file and can be changed by the user. This is especially useful if the name of the *User Algorithm* has to be changed or more algorithms and therefore more buttons have to be added to the menu.

Warning! It is not possible to add new menu buttons via the `GFE` file to the *Tree Interface*. It is only possible to change their labels and their mnemonics.

We now give an example how to change the menu button of *User Algorithm*. Considering that the reader has an original `GFE` file, he will find the following two lines:

```
*User Algorithm.labelString: User Algorithm
User Algorithm.mnemonic: U
```

Those can be change for instance into the following:

```
*User Algorithm.labelString: New Name
User Algorithm.mnemonic: e
```

which results in a button with the label *New Name* and the first ‘e’ underlined.

2.3.4 Hot Keys

As described in chapter 6, some commands in the submenus of the main menu can be activated by pressing the *Ctrl* key and some letter key, without activating the menu bar button first. Such a key combination is called a **hot key** or **accelerator**. All hot keys are listed in the `GFE` file and can be changed by the means of the user.

A hot key description in the `GFE` file bases on two different informations:

the accelerator text: A string written next to a label of a command in the menu button indicating any user of the *Tree Interface* that this command can by activated by a hot key. Typically the hot key itself is written here.

the accelerator: The hot key of the command.

The hot key description is best described by the use of an example. Considering that the reader has an original `GFE` file, he will find the following two lines:

```
*Stop Emulation.acceleratorText:  Ctrl+O
Stop Emulation.accelerator:  Ctrl<Key>O
```

describing the hot key of the command *Stop Emulation*. The first line describes the text which is written next to the label so the button appears with the following information:

`Stop Emulation Ctrl+O`.

The second line then describes the hot key of the command.

2.4 The directory `GRAPHResource`

The directory `GRAPHResource` can be placed anywhere the user wishes to. It just has to be made sure that the path of this directory is added to the `GFE` file (see 2.3.2). This directory contains four files which are established for the use of *Graph Interface*, the base class of the *Tree Interface*. Those files are indicated by the prefix `GRAPH` at the begining of their name and are read every time by the *Tree Interface* when it is started. Furthermore it contains a list of all fonts that can be used (for changing fonts see 2.3.1 and 2.4.3).

Below, a short description of all files is given. Special attention should be drawn to `GRAPHStandardResource.rsc` file, where important information about the appearance of a tree can be modified.

2.4.1 `GRAPHFonts.ft`

The `GRAPHFonts.ft` file contains a list of fonts, that is used by the *Tree Interface*. This list is originally used by the *Graph Interface* and is a lookup for the *Tree Interface*, in order to generate the fonts.

The fonts listed in `GRAPHFonts.ft` are used by the *Tree Interface* in the font selection menus, where the user can choose different fonts for eg. the numbering of nodes.

2.4.2 `GRAPHHeader.ps`

The file `GRAPHHeader.ps` contains a header for postscript files. Every time a postscript file is generated by the *Tree Interface* (see 6.1.3 for postscript generation), the header is copied into this file.

2.4.3 `GRAPHStandardResource.rsc`

The file `GRAPHStandardResource.rsc` is the most interesting file for the user. Here, certain features of the *Tree Interface* can be switched on or off and most of all, the appearance of the nodes can be modified.

Comments in this file are indicated by an `#`. All other lines carry a command, that precedes every line and is understood by the *Tree Interface*. The command then has to be followed by certain values. We now give a description of all commands.

2.4.3.1 `DisplayAreaVisible`

Syntax: `DisplayAreaVisible: <value>`

where `<value>` is either `True` or `False`. If the value is `True`, the display area (see 1.3) will be visible when calling the *Tree Interface*.

2.4.3.2 `DisplayAreaRatio`

Syntax: `DisplayAreaRatio: <value>`

where `<value>` should be an integer value between 0 and 100. If the display area (see 1.3) is visible, `<value>` describes the ratio of the left and right display windows in percent. More precisely, `<value>` describes the share of the left window in percent, if it is visible.

2.4.3.3 `DrawAreaBackgroundColor`

Syntax: `DrawAreaBackgroundColor: <value>`

The `<value>` for this command is a string covering the name of a colour. All possible names can be looked up in the `GRAPHrgb.txt` file (see 2.4.4). This value sets the background colour of the draw area (see 1.3)

2.4.3.4 `LineWidthScaleNumberOfValues`

Syntax: `LineWidthScaleNumberOfValues: <value>`

The `<value>` should be a nonnegative integer value. The *Graph Interface* uses several scalers, whose exactness can be modified by this value. This does not affect the *Scaler...* that has been specially designed for the *Tree Interface*(see 6.4.2).

2.4.3.5 `LoadFileFilter`

Syntax: `LoadFileFilter: <value>`

where `<value>` is a string describing a path. The file filter then lists only those files, which match the description in the path.

2.4.3.6 `TreeLevelSeparationValue`

Syntax: `TreeLevelSeparationValue: <value>`

where `<value>` is a nonnegative integer value restricted by 32. It sets the value of the *Level Separation* (see 1.2). The user does not need to worry about expanding the values. If the value is smaller than 0, the default value is 0, while the default value is 32, if the value is too large.

2.4.3.7 `TreeSubtreeSeparationValue`

Syntax: `TreeSubtreeSeparationValue: <value>`

where `<value>` is a nonnegative integer value restricted by 32. It sets the *Subtree Separation* value (see 1.2). The user does not need to worry about expanding the values. If the value is smaller than 0, the default value is 0, while the default value is 32, if the value is too large.

2.4.3.8 `TreeSiblingSeparationValue`

Syntax: `TreeSiblingSeparationValue: <value>`

where `<value>` is a nonnegative integer value restricted by 32. It sets the *Sibling Separation* (see 1.2). The user does not need to worry about expanding the values. If the value is smaller than 0, the default value is 0, while the default value is 32, if the value is too large.

2.4.3.9 `TreeNodeRadiusValue`

Syntax: `TreeNodeRadiusValue: <value>`

where `<value>` is a nonnegative integer value. It sets the radius of all node categories (see 1.3). The integer value is interpreted as 10 times the radius. So if `<value>` = 28 then the radius of all node categories is set to 2.8. While the minimum value is 5, the maximum value is restricted by any of the above mentioned separation values, since the following inequality must hold:

$$\texttt{<value>} \leq \min\left\{\frac{\texttt{<sepvalue>}+2}{2} \mid \texttt{<sepvalue>} \text{ is separation value}\right\}.$$

Again, as used for the separation values, the user does not need worry about expanding bounds. If the value is smaller than 5, the default value is 10, hence the radius is 1. If the value to larger, then the default value is $\min\left\{\frac{\texttt{<sepvalue>}+2}{2} \mid \texttt{<sepvalue>} \text{ is separation value}\right\}$.

2.4.3.10 `VisualizeAlgorithmTimeIntervall`

Syntax: `VisualizeAlgorithmTimeIntervall: <value>`

where `<value>` is a nonnegative integer value. It sets the length of a time interval in milliseconds. The interval is used when visualizing algorithms. It describes the time that a node is highlighted, that means is drawn in Standard Highlight. If eg. a interval length of 1000 is chosen, every node will be highlight one second.

2.4.3.11 `Nodes <number>`

Syntax:

```
Nodes <number>:  -<colour> -<fontcolour> -<font> -<number flag><filled
                 flag><circle flag> (-(-)<customname>)
```

This command describes the appearance of the nodes in the node categorie `<number>`. `<number>` therefore is an integer value between 1 and 20, accessing one of the 20 different node categories. The different values are listed below:

`<colour>` is a string covering the name of a colour as they are listed in the `GRAPHrgb.txt` (see 2.4.4). Sets the colour of all nodes of the node categorie `<number>`.

`<fontcolour>` is a string covering the name of a colour as they are listed in the `GRAPHrgb.txt` (see 2.4.4). Sets the colour of the numbers of the node of the node categorie `<number>`.

`<font>` is a string covering the name of a font as they are listed in the `GRAPHFonts.ft` (see 2.4.1). Sets the font of the numbers of the node out of the node categorie `<number>`.

`<number flag>` is either the character 'y' or 'n'. Set 'y' if the nodes of the node categorie `<number>` have to be drawn with numbers, else set 'n'.

`<filled flag>` is either the character 'y' or 'n'. Set 'y' if the nodes of the node categorie `<number>` have to be drawn filled, else set 'n'.

`<circle flag>` is either the character 'y' or 'n'. Set 'y' if the nodes of the node categorie `<number>` have to be drawn as circles, else set 'n' if the nodes have to be drawn as squares.

`<customname>` is a string giving the node categorie `<number>` a name. This information is not necessary and can be omitted. If it exists, the name will appear in the pop up menu called by the command *Nodes...* (see 6.3.5), so it can be selected. If there is no `<customname>` listed, it is not possible to change the appearance of the nodes of this node categorie via the command *Nodes...* while running the *Tree Interface*. Placing two times a minus in front of the string has the same result as if there was no `<customname>` listed.

Warning! The *Tree Interface* expects all 20 descriptions of its 20 node categories. None of them may be omitted. If the user does not provide all of them, the *Tree Interface* might crash.

2.4.3.12 Example

The following line:

```
Nodes 2:  -green1 -black -CourierMedium -nyy -Standard Highlight
```

describes the appearance of the node categorie 2. All nodes of this node categorie are painted in a bright colour called green1. They are drawn as filled circles without their number. In case that a demand for drawing the nodes with numbers exists (eg. by using the command *Nodes...* or setting the `<number flag>` to 'y'), the numbers would be written in black *Courier Medium*. The custom name of the node categorie is `Standard Highlight` and therefore can be accessed via the command *Nodes...*.

2.4.4 `GRAPHrgb.txt`

The `GRAPHrgb.txt` file contains a list of colours, that is used by the *Tree Interface*. This list is originally used by the *Graph Interface* and is a lookup for the *Tree Interface*, in order to generate the colours that are given by name. Every colour can be generated by a combination of three nonnegativ integer values. Therefore the *Tree Interface* reads this list in order to find the corresponding combination, when a certain colour name was given.

Warning! Several colours appear more than once in this list, only their names are slightly different (eg `dark slate gray` and `DarkSlateGray` both having the colour code `47 79 79`). The *GFE* accepts only the first detected name of such a colour when it scans the file `GRAPHrgb.txt`. So the user should make sure that the name he uses for a certain colour comes first. Therefore the user is free to modify the `GRAPHrgb.txt` at his will.

2.4.5 `ListOfAdobeFonts` / `ListOfAllFonts`

The files `ListOfAdobeFonts` and `ListOfAllFonts` list all possible fonts. This is only thought as a lookup for the user if he wishes to include certain fonts that are not present in the *Tree Interface*. This lookup might also come in handy, if fonts have to be changed in the file `GFE` (see 2.3.1).

Chapter 3

File structure of the *Tree Interface*

3.1 General construction of a file

The *Tree Interface* distinguishes between two types of files:

1. A user defined file format, which is going to be used in the users algorithm.
2. A file format adapted to the needs of the *Tree Interface* for simple tree drawing or emulating tree building processes, including reading from standard input.

While the latter one will be checked fully by the *Tree Interface*, the first one will be directed to the users algorithm, without being seen by the *Tree Interface*. In this case, the user has to make sure, that the file has a correct format.

3.1.1 Identifiers

A *Tree Interface* formatted file contains 5 lines with some general information about the data that comes along with this file.

- The **first** line must contain the following line:

  ```
  #TYPE: COMPLETE TREE
  ```

  This indicates the file to be in *Tree Interface* format.

- The **second** line contains the following informations:

  ```
  #TIME: SET
  ```

  or

  ```
  #TIME: NOT
  ```

  The first command indicates an emulation process, while the second one indicates a simple drawing of the tree.

- The **third** line contains the following information:

  ```
  #BOUNDS: SET
  ```

  or

  ```
  #BOUNDS: NONE
  ```

  The first command indicates that during an emulation process upper bounds and lower bounds are presented as global informations on the screen. The second command is used if no such bounds have to be shown during the emulation, or if the file does not contain an emulation process.
  Warning! If the reader just applies simple tree drawing, he should not skip this line! The second command should then be used.

- The **fourth** line contains the following information

  ```
  #INFORMATION: STANDARD
  ```

  or

  ```
  #INFORMATION: EXCEPTION
  ```

  The first command indicates that the file contains regular information of the nodes, including written information and node categorie information (see 1.3), while the second one indicates no information at all. In this case, the node numbers are stored as written information and the node categorie is by default Standard Colour.

- The **fifth** line contains the following information:

  ```
  #NODE_NUMBER: AVAILABLE
  ```

  or

  ```
  #NODE_NUMBER: NONE
  ```

  If the node number is available the next two lines will contain the number of nodes in the tree and the number of edges. If not, the *Tree Interface* first has to read the complete file, in order to find out how much memory has to be allocated. The next line will then contain the first relevant information of the tree.

3.1.2 The principle of the file format of a tree

A tree $T = (V, E)$ with V a set of nodes and E a set of edges, is given as an edge list. So the basic component of any file in *Tree Interface* format is this edge list together with features, that the user is willing to use. We therefore assume that the edges are numbered and that edges have the following appearance:

$$(i, j), \qquad i < j, \qquad \{i, j\} \subset V$$

where the node i is the father of node j and j is the son of node i.

The file structure therefore bases on the following simple line:

$$ij$$

indicating such a father-son relationship. The node numbered 1 is always the root of the tree and therefore does not have a father. According to the specifications, that were made in the identifiers (see 3.1.1), the line is modified in order to satisfy the needs of the *Tree Interface*. Those modifications are discussed below (see sections 3.2-3.4).

3.2 File format for simple trees with no informations

A simple tree, where the nodes do not contain any kind of information can be achieved with the following identifier:

```
#TYPE: COMPLETE TREE
#TIME: NOT
#BOUNDS: NONE
#INFORMATION: EXCEPTION
#NODE_NUMBER: AVAILABLE
```

According to the specification in the last line the next two lines contain

1. the number n of nodes in the tree
2. the number m of edges in the tree

```
#TYPE: COMPLETE TREE
#TIME: NOT
#BOUNDS: NONE
#INFORMATION: EXCEPTION
#NODE_NUMBER: AVAILABLE
7
6
1 2
1 3
2 4
2 5
3 6
3 7
```

Figure 3.1: Example of a file, that draws a simple tree.

The edges of the tree are listed on the next m lines, obeying the father son relationship, such that the first number is the father and the second one is the son. For an example see figure 3.1. As described in 3.1.1, it is not necessary to have the node number available. If this is the case, the listing of the edges has to be continued directly after the identifier.

It is not necessary to have the edges ordered as in the example 3.1. The edges can be listed in any arbitrary order. Observe that different listings produce the same tree, but **not** the same drawing.

3.3 File format for trees with informations

A tree, where the nodes carry node categorie informations and informations such as written information, that can be shown via the *Node-Information-Window* (see 4.1), needs the following identifier.

```
#TYPE: COMPLETE TREE
#TIME: NOT
#BOUNDS: NONE
#INFORMATION: STANDARD
#NODE_NUMBER: AVAILABLE
```

According to the specification in the last line, the next two lines contain

1. the number n of nodes in the tree,
2. the number m of edges in the tree.

The following lines contain the list of edges, a list of string informations and a list of node categorie informations. Since node categories mainly differ through their colours, those informations are as well referred as colour informations. The order in which informations, colour informations and edge descriptions appear is arbitrary. Informations may appear first, at the end or they may be mixed with the edge descriptions. Especially the latter one satisfies the needs of algorithmic output. Some nodes may not contain written informations, so it is not necessary to include an extra line for them. If a node does not even have a node categorie information, the corresponding line can be left out as well. The node then simply is assigned to the default node categorie Standard Colour. Furthermore, if a node is assigned to a non existing node categorie, it will be assigned as well to the default node categorie.

So the user will need exactly m lines, where the edges are listed, and at most $2n$ lines for the informations of the nodes.

Observe that the description of the edges again obeys the father son relationship, such that the first number is the father and the second one is the son. Observe further that as described in 3.1.1, it is not necessary to have the number of nodes available. If this is the fact, the user leaves the two lines with node number and edge number out.

In order to indicate what kind of information a line currently holds, the following identifiers are set at the beginning of each line:

e is set when the line holds an **edge** description.

n is set when the line holds written **node** information.

c is set when the line holds node **categorie** information.

```
#TYPE: COMPLETE TREE
#TIME: NOT
#BOUNDS: NONE
#INFORMATION: STANDARD
#NODE_NUMBER: AVAILABLE
7
6
n 1 \inode 1\iinformation included by node 1
n 2 \inode 2\iinformation included by node 2
e 1 2
e 1 3
c 1 1
c 2 1
n 3 \inode 3\iinformation included by node 3
n 4 \inode 4\iinformation included by node 4
e 2 4
c 3 8
c 4 9
e 2 5
e 3 6
e 3 7
n 5 \inode 5\iinformation included by node 4
c 5 5
n 6 \inode 6\iinformation included by node 6
n 7 \inode 7\iinformation included by node 7
c 6 8
```

Figure 3.2: Example of a file, drawing a tree with node information.

The example 3.2 shows a tree with node informations. The format of the informations will be discussed in chapter 4. Therefore the reader is referred to that chapter. Observe that the node numbered seven does not have a node categorie information. It will therefore be assigned to the default node categorie Standard Colour.

3.4 File format for emulation processes

A file carrying information about an emulation process needs the following identifier.

```
#TYPE: COMPLETE TREE
#TIME: SET
#BOUNDS: SET
#INFORMATION: STANDARD
#NODE_NUMBER: NONE
```

Important is the change in the second line, which tells the *Tree Interface* that this file contains an emulation process. The third line could be also set to:

```
#BOUNDS: NONE
```

indicating that no bounds are to be set. The fifth line is normally set to:

```
#NODE_NUMBER: NONE
```

since such files are in general the output of some algorithmic process, where the number of nodes produced by the algorithm is not known when the process starts writing into the file.

Every following line starts with a time label, indicating at what time the information, stored in the line, was written by the process into the file. The time label has the following syntax:

hour:minutes:seconds.hundredth of a second

Examples :

```
00:00:12.11     // 12 seconds and 110 milliseconds
02:12:00.00     // 2 hours and 12 minutes
```

The time label is followed by an identifier which is closely related to the public commands in the *Tree Interface* that are needed to adapt the users algorithm to the *Tree Interface* (see also chapter 5). The following list describes all possible identifiers and it names the functions of the *Tree Interface* that are called when the identifiers are read:

A **adds** information to existing information of a node.
Calls `theTreeInterface->AddNodeInfo`.

I a node gets new **information**.
Calls `theTreeInterface->SetNodeInfo`.

L prints out the new **lower** bound on the screen.
Calls `theTreeInterface->LowerBound`.

N a new **node** is added into the tree.
Calls `theTreeInterface->NewNode`.

P **paints** a node in a new colour.
Calls `theTreeInterface− >PaintNode`.

U prints the new **upper** bound on the screen.
Calls `theTreeInterface− >UpperBound`.

```
#TYPE: COMPLETE TREE
#TIME: SET
#BOUNDS: SET
#INFORMATION: STANDARD
#NODE_NUMBER: NONE
00:00:00.00 N 0 1 5
00:00:00.00 U 200.1
00:00:00.00 L 0
00:00:00.01 I 1 \inode 1\iinformation included by node 1
00:00:00.50 N 1 2 5
00:00:00.50 I 2 \inode 2\iinformation included by node 2
00:00:02.00 N 1 3 5
00:00:02.00 I 3 \inode 3\iinformation included by node 3
00:00:03.00 N 1 4 5
00:00:03.00 I 4 \inode 4\iinformation included by node 4
00:00:06.00 P 1 15
00:00:06.00 U 178.52
00:00:08.01 I 1 \inODE 1\iinformation included by node 1
00:00:09.00 N 2 5 6
00:00:09.00 I 5 \inode 5\iinformation included by node 5
00:00:09.20 N 2 6 5
00:00:09.20 L 23.7
00:00:09.20 I 6 \inode 6\iinformation included by node 6
00:00:10.01 I 2 \inODE 2\iinformation included by node 1
00:00:10.20 P 3 14
00:00:10.21 U 162.789
00:00:10.22 L 34.7
00:00:10.41 I 3 \inODE 3\iinformation included by node 1
00:00:10.61 I 4 \inODE 4\iinformation included by node 1
00:00:10.80 N 3 7 5
00:00:10.80 I 7 \inode 7\iinformation included by node 7
00:00:11.00 N 3 8 5
00:00:11.00 I 8 \inode 8\iinformation included by node 8
00:00:11.20 N 6 9 5
00:00:11.20 I 9 \inode 9\iinformation included by node 9
00:00:11.40 N 6 10 5
00:00:11.40 I 10 \inode 10\iinformation included by node 10
00:00:13.00 N 8 11 5
00:00:14.00 P 4 18
00:00:14.00 U 123.456
00:00:14.00 I 11 \inode 11\iinformation included by node 11
00:00:14.00 L 89.667
00:00:15.00 N 8 12 5
00:00:15.00 I 12 \inode 12\iinformation included by node 12
00:00:19.00 N 8 13 5
00:00:20.00 P 2 12
00:00:22.00 P 12 9
00:00:22.50 I 13 \inode 13\iinformation included by node 13
00:00:23.01 I 12 \iHELLO! This is node 12\i
00:00:25.00 N 8 14 5
00:00:25.00 I 14 \inode 14\iinformation included by node 14
00:00:25.00 U 100.2
00:00:25.00 L 99.8
00:00:27.00 N 8 15 5
00:00:27.00 I 15 \inode 15\iinformation included by node 15
00:00:28.00 P 1 1
00:00:29.50 A 13 \imake it longer\i add more information
00:00:30.50 A 14 \imake it longer\i add more information
00:00:31.00 A 15 \imake it longer\i add more information
00:00:31.00 U 100
00:00:31.00 L 100.3
```

Figure 3.3: Example of a file, executing an emulation process.

All those identifiers cause the call of one of these functions. Therefore the identifiers have to be followed by the input of theses functions. All informations are **separated** by a blank.

The following list gives a brief description of the values that have to follow the identifiers:

A	int char∗	Number of the node, information of the node.
I	int char∗	Number of the node, information of the node.
L	double	The size of the lower bound.
N	int int int	Number of the father, number of the new node and number of the new nodes colour.
P	int int	Number of the node, number of the nodes colour.
U	double	The size of the upper bound.

An example of such a file is shown in figure 3.3.

3.5 File format for *User Algorithm*

This file format has to be specified by the user. If the first line is **not equal** to the string

```
#TYPE: COMPLETE TREE
```

then this file will be automatically referred to the users algorithm and the menu bar button for executing this algorithm will be activated.

This strategy of not checking such a file at all offers as much freedom as possible to the user but is also a source for easy crashes of the *Tree Interface*. We therefore strongly suggest to check every file carefully, that is passed to the users algorithm.

For further information the reader is referred to the chapter 5.

3.6 Reading input from `standard input`

Besides reading trees from files and adapting user defined algorithms to the *Tree Interface*, it is as well possible to read input from `standard in`, which comes in hand especially when piping the output of any user's program to the *Tree Interface*. For those, who are not familiar with piping standard output of any program to the standard input of another program, we remember that the syntax is as follows:

```
<usersprogram> <options> | vbctool
```

The functionality and therefore the format of the standard input is quite similar to the one described in the section 3.4, where the file format for emulating trees was described, except that in this case the *Tree Interface* does not need time labels, since it works *online* (a word we hear in these days quite often).

Nevertheless a special identifier, the **$** sign, is used to mark all lines concerning all kinds of informations of the tree. This enables the user to have information, that he wants to be printed in the display area during the drawing of the tree, to be merged with the general tree informations. The *Tree Interface* simply prints all information stored in a line, which is not preceded by a $, into the display area.

After starting the *Tree Interface* as described, the pull down menu `Online` has to be opened, and the menu bar button `Standard In` has to be pressed. This enables the *Tree Interface* to read from standard input from that moment on. Since all information written by the user's program to standard output is buffered, the *Tree Interface* reads all information written to standard out (see also 6.7).

The information concerning the painting of the tree written to the buffer has to start with the following identifier:

```
$#TYPE: COMPLETE TREE
$#TIME: SET
$#BOUNDS: NONE
$#INFORMATION: STANDARD
$#NODE_NUMBER: NONE
```

Observe that, although this is not an emulation process and no time labels are set, the *Tree Interface* expects that time is `SET`. This is due to the fact, that the *Tree Interface* when reading from standard input uses the same set of commands as when emulating a process. The only information in the identifier, which is optional therefore is the third line. The user can also use:

```
$#BOUNDS: SET
```

The fifth line is always set to

```
$#NODE_NUMBER: NONE
```

since the *Tree Interface* never expects in a tree growing process that the total number of nodes is available.

The above mentioned identifier may be preceded by some information which is marked without the $ sign as information reserved to be printed in the display area, and it may be followed by such information. On the other hand the user should observe **not to print lines which do not concern the identifier within the identifier**. Once the identifier was piped, the user is free to merge display information and tree information at will.

Every following line concerning the painting of the tree starts with an $ followed by one of the identifiers as described in section 3.4 and every identifier has to be followed by an information as described in the same section 3.4. So the information written to standard output by the user's program looks pretty much the same as the file format for emulation processes, with the only exception that instead of the time labels a $ is printed at the beginning of every line (an example is shown in figure 3.4). We therefore omit further discussions about this format and turn our attention to some other important information: Indicating the end of a process.

After the user's program has written all information to standard out, it has to close the information with the following line:

```
$#END_OF_OUTPUT
```

This line indicates the *Tree Interface* that all information was read and stops it from reading further information from standard input. After this line was read, all other features of the *Tree Interface* are accessible again. Forgetting to write this line could cause serious trouble, since almost all features of the *Tree Interface* will be closed until reading from standard input has been finished.

```
Information printed into the Display Area
Information printed into the Display Area
$#TYPE: COMPLETE TREE
$#TIME: SET
$#BOUNDS: SET
$#INFORMATION: STANDARD
$#NODE_NUMBER: NONE
Information printed into the Display Area
$N 0 1 5
$U 200.1
$L 0
$I 1 \inode 1\iinformation included by node 1
$N 1 2 5
$I 2 \inode 2\iinformation included by node 2
$N 1 3 5
$I 3 \inode 3\iinformation included by node 3
$N 1 4 5
$I 4 \inode 4\iinformation included by node 4
$P 1 15
$U 178.52
Information printed into the Display Area
Information printed into the Display Area
$I 1 \inODE 1\iinformation included by node 1
$N 2 5 6
$I 5 \inode 5\iinformation included by node 5
$N 2 6 5
$L 23.7
$I 6 \inode 6\iinformation included by node 6
$I 2 \inODE 2\iinformation included by node 1
$P 3 14
$U 162.789
$L 34.7
$I 4 \inODE 4\iinformation included by node 1
$N 3 7 5
$I 7 \inode 7\iinformation included by node 7
$N 3 8 5
$I 8 \inode 8\iinformation included by node 8
$N 6 9 5
$I 9 \inode 9\iinformation included by node 9
$N 6 10 5
$I 10 \inode 10\iinformation included by node 10
$N 8 11 5
$P 4 18
$U 123.456
$I 11 \inode 11\iinformation included by node 11
$L 89.667
$N 8 12 5
Information printed into the Display Area
Information printed into the Display Area
$I 12 \inode 12\iinformation included by node 12
$N 8 13 5
$P 2 12
$P 12 9
$I 13 \inode 13\iinformation included by node 13
$I 12 \iHELLO! This is node 12\i
$N 8 14 5
$I 14 \inode 14\iinformation included by node 14
$U 100.2
$L 99.8
Information printed into the Display Area
$N 8 15 5
$I 15 \inode 15\iinformation included by node 15
$P 1 1
$A 13 \imake it longer\i add more information
$A 14 \imake it longer\i add more information
$A 15 \imake it longer\i add more information
$U 100
$L 100.3
Information printed into the Display Area
Information printed into the Display Area
Information printed into the Display Area
```

Figure 3.4: Example of a file as used in standard input.

Chapter 4

Node Information

4.1 The Node Information Window

The *Node Information Window* is a tool to show the informations stored at a node. In order to activate the Node Information Window the mouse cursor has to be positioned at a node and the right mouse button has to be pressed. This pops up a window which shows the information stored at the node.

Pressing the right mouse button while the cursor is in the draw area will always pop up such a Node Information Window. If the cursor is not positioned at a node, the node next to the cursor is chosen, in order to show its information. The user can open up to 10 different Node Information Windows. It is not possible and probably not useful to open more Node Information Windows.

The Node Information Window is an always active feature. As long as there is a tree drawn in the draw area, node informations can be shown by the use of the right mouse button. This feature is also active in the user algorithm. If an information changes throughout an algorithmic process, the information shown in the window is updated and the size of the window is adapted to size of the new information.

The Node Information Window is divided into two parts, showing the *main* information as well as the *general* information. The main information is placed at the top position of the window into a framed area, while the general information is placed underneath it (see also 4.2.2).

4.2 String format

The layout of the Node Information Window is fully dependent on the user. The *Tree Interface* does not make any suggestions how to present the information. Showing informations is understood in terms of a modified `printf` instruction. This gives the user maximum freedom how to present the informations of the nodes.

4.2.1 Layout

In order to achieve a specific layout in the Node Information Window, the following for $C++$ programmers well known control characters have to be used:

\t the tabulator character,

\n the new line character.

Those characters can be placed anywhere in the information array of the node. As an example, observe the following line:

```
Number:\t\t 12345\nUpper Bound:\t 6789
```

gives the following output on the Node Information Window:

```
Number:          12345
Upper Bound:     6789
```

4.2.2 *Main* and *general* information

The *Tree Interface* offers the user the opportunity to distinguish between *main* information and *general* information. This is done for a better presentation of the informations of a node and for a better orientation in the tree.

If trees have to be drawn with a large amount of informations for every node, some of the informations should probably be highlighted in order to see them directly without reading through the complete text. Furthermore, if the user searches for a special node, it is useful to filter informations, so that the nodes can be apprehended faster, especially when informations and trees tend to be large.

The main information will be highlighted in the *Node Information Window*. All information that is considered to belong to the main information will be written in a framed box at the top of the window. The general information is written below the main information. Furthermore, the main information is the **only** information of a node that is shown in the left display of the display area (see 1.3) when the Browser Mode is activate (see 6.4.3.

Both main and general information are stored in the same string of characters. To indicate the main information within the string the following control character is used:

\i

The beginning as well as the end of every main information is indicated by this control character. The order in which general and main information appear in the input files is arbitrary. Besides it is not necessary that main or general information form a consecutive sequence. So the main information may be divided into several parts, each part indicated by a pair of `\i` control characters.

As an example, observe the following possible information of a node:

```
\iNode 1\iinformation included by node 1\i\nInfo 1:\t\t 12345\i\nInfo
2:\t\t 1234\nInfo 3:\t\t stringinformation\n Information 4:\t TRUE\nI
nformation 5:\t containing an Information\n\t\t with an additional li
ne
```

The information of the example is considered to be a string of characters as it appears typically in one line in an input file (see also 3.3 and 3.4). Special attention should be drawn to the fact, that the main information does not appear in a consecutive sequence. Although this information is hard to read while it is stored in a string, it will result the following presentation of the information in the Node Information Window:

```
Node 1
Info 1:                 12345
```

```
information included by node 1
Info 2:                 1234
Info 3:                 stringinformation
Information 4:          TRUE
Information 5:          containing an Information
                        with an additional line
```

4.2.3 Array bounds

The total length of an information that a node is allowed to carry is restricted by the *Tree Interface*. The total length of the information, including main and general information, may not exceed a number of 1024 characters. The length of the main information may not exceed a number of 128 characters.

In general, this bounds should leave enough space for presenting the information of nodes, since the control characters `\t` and `\n`can be used. In case that more space is needed by the user in order to express himself, two global defined constants, defining the array bounds have to be changed and the source code of the *Tree Interface* has to be recompiled.

In the file `def.glb`, which is supplied together with all files of the *Tree Interface*, the following two lines will be found:

```
#define INFOSIZE 1024
#define SHORTSIZE 128
```

`INFOSIZE` describes the length of the arrays, holding complete informations of the nodes, while `SHORTSIZE` describes the length of the arrays holding only the main information. After making appropriate changes, the program has to be compiled again.

Chapter 5

Adapting algorithms to the *Tree Interface*

For adapting algorithms to the *Tree Interface* we provide a package containg the libraries and headers of the GFE the GraphInterface and the *Tree Interface*. This package contains also a few source files, that have to be manipulated by the user. This package does not contain any commercial software! Hence this package does not provide the *Motif Application Library* of the *Xm Library*. However, the user will need this software to implement the here described manipulation of our software. If the reader does not have these libraries, he should check if it is sufficient to pipe the output of a programm into the *Tree Interface*. This feature of the *Tree Interface* probably covers most requierements.

The *Tree Interface* is designed to behave like a window program. The philosophy of such a program is to let the user not worry about a *main* function, it has already a main function. To be concrete, this *main* function exists in the library of Young [You92]. The user therefore only writes subroutines, which will be adapted to the *Tree Interface*. Theses subroutines are strictly limited to tree drawing algorithms.

In the following, the reader will be shown how to do this. It will be easy, if just one program has to be adapted, but we will also discuss what to do if two or more different subroutines have to be included. In case that the users program has a `main` function, its function name has to be replaced by an appropriate name like `clientmain`.

5.1 Adapting one subroutine to the *Tree Interface*

This section fully describes the adaption of a subroutine to the *Tree Interface*. This subroutine can be called any time by loading a file, which does not fit the file formats described in chapter 3, causing the button *Users Algorithm* in the menu *Algorithm* to be activated and pressing the button starts the subroutine. We strongly remind the reader of the fact, that every file he loads, which does not fit to the *Tree Interface* formatted files, will be assigned to the users algorithm (see 3.1). So it has to be made sure, that all files are checked by the users algorithm.

5.1.1 The `VBCCmd.cc` File

When adding a subroutine to the *Tree Interface*, the file `VBCCmd.cc` shown in figure 5.1 has to be manipulated. Of special interest should be the function `doit`. Here, the user has to place his own function call instead of the function `clientmain`.

This adaption should be of no problem and has the following effect on the *Tree Interface*: every time, a user opens the menu *Algorithm* and clicks on the menu entry, the function `doit` will be called and hence the users application will be started.

The `VBCCmd.cc` file furthermore offers the option to use either a filename or a pointer to the file in order to communicate with the file as the user wishes.

After adapting the `VBCCmd.cc`, the function calls of the *Tree Interface*, that can be used by any subroutine in order to draw trees, will be introduced and discussed in the following sections. Special attention should be paid to the use of the function `dispatchEvents` described in subsection 5.1.3. The use of this function guarantees, that the *Tree Interface* will be accessible while the process of the users algorithm is still continuing.

In order to use the function calls of the *Tree Interface*, it is necessary to include the file `TreeInterface.h` to the users subroutines. This offers the programmer to use the external pointer

```
theTreeInterface
```

in order to reference the public functions of the *Tree Interface*.

Example:

```
theTreeInterface->dispatchEvents()
```

calls the function `dispatchEvents` of the *Tree Interface*.

5.1.2 The function `AddNodeInfo`

The complete function call is:

```
AddNodeInfo(int node, char* information)
```

This function adds more information to already existing information of a node. The node is accosted via its positive integer number, while the information has to be *Tree Interface* formatted (see 4.2). It is possible to add main information as well as general information with this call. The information carried by this command will be added at the end of the existing information. The user can apply this call as often as he wishes and only has to make sure that the internal array bounds are not expanded.

```
#include "TreeInterface.h"
#include "VBCCmd.h"
#include "branchIncludes.h"
#include <string.h>


/**************************************************************
                          VBCCmd
**************************************************************/


VBCCmd::VBCCmd(char* name,int active)
        : NoUndoCmd(name,active)

   // Constructor
{
}



/**************************************************************
                       setFileName
**************************************************************/


void VBCCmd::setFileName(FILE* ptr,char* filename)

   // The name of the file that has been read as input is set.
{
   strcpy(_FileName,filename);
   _FilePtr = ptr;
}




/**************************************************************
                          doit
**************************************************************/


void VBCCmd::doit()

   // The Procedure is called when the button in the menue bar,
   // that belongs to this command, is pressed. It will then call
   // a user defined function.
{
                                            // Print a message if necessary.
   printf("Execute Clients Program.\n");
                                            // Make sure that old trees are
                                            // removed.
   theTreeInterface->Tree()->clean_tree();
                                            // Make appropriate deactivation
                                            // calls.
   theTreeInterface->DeactivationCall();

                                            // Call the clients function.
   char* arg[2];
   arg[1] = _FileName;
   clientmain(1,arg);
}
```

Figure 5.1: The file VBCCmd.cc.

5.1.3 The function `dispatchEvents`

The complete function call is:

```
dispatchEvents()
```

This function is derived from the *GFE*. It **has to be applied** by the user. This function enables the *Tree Interface* to check for any kind of event as mouse button inputs or key clicks. If the user **does not** implement this function in his source code, it is not possible to apply any of the features of the *Tree Interface* after the users algorithm was started, until the algorithm has come to a stop.

The function `dispatchEvents` checks what kind of events have taken place between two calls of `dispatchEvents` and then executes them. Therefore the user should jump at this function very often in order to get a fluent and nice handling of the *Tree Interface* during the execution of the users algorithm. We therefore suggest to implement this function call always in the inner loops of the algorithm.

5.1.4 The function `FinishAlgorithm`

The complete function call is:

```
FinishAlgorithm(int boolean, int returnvalue)
```

This function can be used to indicate that the user algorithm has been finished. If the `boolean` value is set to 0 = `FALSE`, nothing will happen, even the `returnvalue` is ignored. If the `boolean` value is set 1 = `TRUE`, this subroutine will print out the messages:

Algorithm has come to a stop.

and

Program exited normally

if the `returnvalue` = 0, or

Program exited with "returnvalue".

if the `returnvalue` $\neq 0$. Furthermore the function will reactivate the command button of the user algorithm so the algorithm can be run again.

5.1.5 The function `LowerBound`

The complete function call is:

```
LowerBound(double bound)
```

This function prints the value of a lower bound in the right display of the display area.

5.1.6 The function `SetNodeInfo`

The complete function call is:

```
SetNodeInfo(int node, char* information)
```

This function sets the information of a node. The node is accosted via its node number, while the information has to be *Tree Interface* formatted (see 4.2). It is possible to add main information as well as general information with this call. The user has to make sure, that the internal array bounds are not expanded.

If information has been stored in a node before, this call first deletes all existing information, before storing the new information.

5.1.7 The functions `NewNode`

The complete function call is:

```
NewNode(int father, int newNode, int nodeCategorie)
```

or

```
NewNode(int father, int newNode, int nodeCategorie, int draw)
```

Both functions add a new node to an existing tree. The first call always causes a repaint of the tree, while the latter does not cause a repaint, if the value of `draw` is set to 0 = `FALSE`. This might come in handy when the tree, that is build by an instance of the algorithm starts to get large, whereas computing the new coordinates and repainting the tree grows linear with the size of the tree. Since this slows down the execution of the algorithm, it might be useful to repaint the tree only eg. every 10 nodes.

To include a new node into the tree, first mention the number of its father, second the number of the node and third the number of the node categorie, which this node should belong to. This node categorie number is an integer value between 1 and 20. The features of the different node categories can be looked up and manipulated by the user in the file `GRAPHStandardResource.rsc` (see 2.4.3). In case that the new node is the root of the tree, which means that `newNode` = 1, the value for `father` = 0.

5.1.8 The function `PaintNode`

The complete function call is:

```
PaintNode(int node, int nodeCategorie)
```

With the help of this function, a current node categorie of a node can be changed in order to apply different colours or shapes to a node. The node is accosted via its node number, while the node categorie number is an integer value between 1 and 20. The features of the different node categories can be looked up and manipulated by the user in the file `GRAPHStandardResource.rsc` (see 2.4.3).

5.1.9 The function `RepaintTree`

The complete function call is:

```
RepaintTree()
```

This function repaints a tree, which is especially necessary, when using the function `NewNode` (see 5.1.7) while suppressing a repaint after every newly introduced node.

5.1.10 The function `UpperBound`

The complete function call is:

```
UpperBound(double bound)
```

This function prints out the value of an upper bound in the right display of the display area.

5.1.11 The `printf` instructions

There are there different `printf` instructions at hand for a user of the *Tree Interface*: `printf`, `lprintf` and `rprintf`. With the help of the `printf` instruction, the user can show informations on the right message window of the display area. The `printf` is used as the normal ANSI C printf. The `rprintf` instruction supplies the same result as the `printf` instruction, whereas `lprintf` prints out strings on the left display of the display area.

5.1.12 An Example

This subsection shows a small example that can be called by the `doit` function shown in figure 5.1. `INFOSIZE` and `SHORTSIZE` are the defines for the array bounds as described in 4.2.3.

5.1.12.1 UserExample.h

```
/*****************************************************************

            Filename   :  UserExample.h

            Version    :  01.1995

            Author     :  Sebastian Leipert

            Language   :  C++

            Purpose    :  Contains the header of the client defined
                          function clientmain. This file has to be
                          included into the file VBCCmd.cc, in order
                          to call the user algorithm when pressing
                          the corresponding menue bar button.

*****************************************************************/


void clientmain(int argc, char *argv[]);
```

5.1.12.2 UserExample.cc

```
/*********************************************************************

            Filename    :  UserExample.cc

            Version     :  01.1995

            Author      :  Sebastian Leipert

            Language    :  C++

            Purpose     :  Example for a userdefined program.
                           Draws a simple tree.

*********************************************************************/


#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include "TreeInterface.h"
#include "branchIncludes.h"


void clientmain(int argc, char *argv[])
{

   int node_nb = 0;
   int edge_nb = 0;
   int tail = 0;
   int head = 0;

   char *filename = argv[1];
   char inputline[INFOSIZE];
   char *inputptr;
   char infochar;
   char bufshort2[SHORTSIZE];

   ifstream inClientFile(filename,ios::in);
   inClientFile.getline(inputline,INFOSIZE);
   inClientFile.getline(inputline,INFOSIZE);
   inClientFile >> node_nb >> edge_nb;

   int *father = new int [node_nb+1];
   father[1] = 0;

   int k = 1;
   while (k <= edge_nb)
   {
      inClientFile >> tail >> head;
      if (tail < head)
         father[head] = tail;
      k++;
   }



   for (int j = 1; j <= node_nb; j++)
   {



      for (int i = 1; i <= 1000; i++)
      {
            for (int l = 1; l ≤ 1000; l++);
                dispatchEvents();
        }


        theTreeInterface→NewNode(father[j],j,5);
        sprintf(bufshort2,"\\idual bound node: %d\\iTestInformation",j);
        theTreeInterface→SetNodeInfo(j,bufshort2);
        if (j > 20)
        {
            int node = j - 20;
            sprintf(bufshort2,"\\idual bound node: %d\\iNew TestInformation\\nContains one more line.",node);
            theTreeInterface→SetNodeInfo(node,bufshort2);
        }
        dispatchEvents();
        if (j > 5)
            theTreeInterface→PaintNode(j-5,4);
        if (j > 1)
            theTreeInterface→PaintNode(j-1,6);
    }


    delete[] father;

    theTreeInterface→FinishAlgorithm(TRUE,0);
}
```

5.2 Adapting two or more subroutines to the *Tree Interface*

In this section the case is considered to adapt not only one subroutine but two or more subroutines to the *Tree Interface*. To do so, five different tasks have to be solved by the user:

1. The introduction of extra menu buttons in the menu bar of the *Tree Interface*.
2. Including the extra subroutines into the *Tree Interface*, that cannot be covered by the use of the `VBCCmd.cc` file (see 5.1.1).
3. Reassure that the menu buttons call the corresponding subroutines when pressed.
4. Make appropriate changes to the loading command.
5. Manage the activation and deactivation of the menu buttons.

To achieve all goals, the user has to derive the *Tree Interface*. This includes some work, but is not so difficult to implement. We assume that the reader is familiar with C++, and has all libraries at hand. For a better understanding, we show the solution of the problem by giving a full description of an example.

5.2.1 The included programs

Assume, that we have three programs which have to be visualized with the help of the *Tree Interface*. In a first step, the functions described in the previous section 5.1 have to be adapted to the programs. Furthermore, it has to be made sure that none of the programs has a main function. This can be done by an appropriate renaming of the main functions.

After having done the described changes, we have three subroutines, that have to be adapted to the *Tree Interface*. In our example, they are called `clientmain`, `clientmain1` and `clientmain2`. The three subroutines are listed below in the files `branch1.cc`, `branch2.cc` and `UserExample.cc`. The file `UserExample.cc` is the same as in section 5.1. It is shown for completeness. The file `branchIncludes.h` serves as an interface between the subroutines and the derivation of the *Tree Interface*. Other solutions than this may of course be realized.

One of the subroutines can be added to the *Tree Interface* with the help of the `VBCCmd.cc` file as described in the previous section 5.1. This again will be the function `clientmain` of the file `UserExample.cc`.

5.2.1.1 branchIncludes.h

```
/*********************************************************************

          Filename   :  branchIncludes.h

          Version    :  01.1995

          Author     :  Sebastian Leipert

          Language   :  C++

          Purpose    :  Contains the headers of all client defined
                        functions. Other solutions than this may
                        of course be possible.

*********************************************************************/

#ifndef BRANCHINCLUDES_H
#define BRANCHINCLUDES_H


void clientmain(int argc, char *argv[]);

void clientmain1(int argc, char *argv[]);

void clientmain2(int argc, char *argv[]);

#endif
```

5.2.1.2 branch1.cc

```
/*********************************************************************

          Filename   :  branch1.cc

          Version    :  01.1995

          Author     :  Sebastian Leipert

          Language   :  C++

          Purpose    :  Example for a userdefined program.
                        Draws a simple tree.

*********************************************************************/
```

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include "TestInterface.h"
#include "branchIncludes.h"



void clientmain1(int argc, char *argv[])
{

   int node_nb = 0;
   int edge_nb = 0;
   int tail = 0;
   int head = 0;

   char *filename = argv[1];
   char inputline[INFOSIZE];
   char *inputptr;
   char infochar;
   char bufshort2[SHORTSIZE];

   ifstream inClientFile(filename,ios::in);
   inClientFile.getline(inputline,INFOSIZE);
   inClientFile.getline(inputline,INFOSIZE);
   inClientFile.getline(inputline,INFOSIZE);
   inClientFile >> node_nb >> edge_nb;

   int *father = new int [node_nb+1];
   father[1] = 0;

   int k = 1;
   while (k <= edge_nb)
   {
      inClientFile >> tail >> head;
      if (tail < head)
         father[head] = tail;
      k++;
   }



   for (int j = 1; j <= node_nb; j++)
   {



      for (int i = 1; i <= 500; i++)
      {
         for (int l = 1; l <= 500; l++);
            dispatchEvents();
      }


      theTreeInterface->NewNode(father[j],j,5);
      sprintf(bufshort2,"\\idual bound node:  %d\\iTestInformation",j);
      theTreeInterface->SetNodeInfo(j,bufshort2);
      if (j > 20)
      {
         int node = j - 20;
         sprintf(bufshort2,"\\idual bound node:  %d\\iNew TestInformation\\nContains one more line.",node);
         theTreeInterface->SetNodeInfo(node,bufshort2);
      }
```

```
        dispatchEvents();
        if (j > 5)
            theTreeInterface→PaintNode(j-5,4);
        if (j > 1)
            theTreeInterface→PaintNode(j-1,6);
    }

    delete[] father;

    theTreeInterface→FinishAlgorithm(TRUE,0);
}
```

5.2.1.3 branch2.cc

```
/*********************************************************************

            Filename    :  branch2.cc

            Version     :  01.1995

            Author      :  Sebastian Leipert

            Language    :  C++

            Purpose     :  Example for a userdefined program.
                           Draws a simple tree.

*********************************************************************/


#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include "TestInterface.h"
#include "branchIncludes.h"


void clientmain2(int argc, char *argv[])
{

    int node_nb = 0;
    int edge_nb = 0;
    int tail = 0;
    int head = 0;

    char *filename = argv[1];
    char inputline[INFOSIZE];
    char *inputptr;
    char infochar;
    char bufshort2[SHORTSIZE];

    ifstream inClientFile(filename,ios::in);
    inClientFile.getline(inputline,INFOSIZE);
    inClientFile.getline(inputline,INFOSIZE);
    inClientFile.getline(inputline,INFOSIZE);
    inClientFile >> node_nb >> edge_nb;

    int *father = new int [node_nb+1];
    father[1] = 0;

    int k = 1;
    while (k ≤ edge_nb)
    {
        inClientFile ≫ tail ≫ head;
        if (tail < head)
            father[head] = tail;
        k++;
    }



    for (int j = 1; j ≤ node_nb; j++)
    {



        for (int i = 1; i ≤ 100; i++)
        {
            for (int l = 1; l ≤ 100; l++);
                dispatchEvents();
        }


        theTreeInterface→NewNode(father[j],j,5);
        sprintf(bufshort2,"\\idual bound node:  %d\\iTestInformation",j);
        theTreeInterface→SetNodeInfo(j,bufshort2);
        if (j > 20)
        {
            int node = j - 20;
            sprintf(bufshort2,"\\idual bound node:  %d\\iNew TestInformation\\nContains one more line.",node);
            theTreeInterface→SetNodeInfo(node,bufshort2);
        }
        dispatchEvents();
        if (j > 5)
            theTreeInterface→PaintNode(j-5,4);
        if (j > 1)
            theTreeInterface→PaintNode(j-1,6);
    }


    delete[] father;

    theTreeInterface→FinishAlgorithm(TRUE,0);
}
```

5.2.1.4 UserExample.cc

```
/*********************************************************************

            Filename    :  UserExample.cc

            Version     :  01.1995

            Author      :  Sebastian Leipert

            Language    :  C++

            Purpose     :  Example for a userdefined program.
```

```
                              Draws a simple tree.

************************************************************************/


#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include "TreeInterface.h"
#include "branchIncludes.h"


void clientmain(int argc, char *argv[])
{

   int node_nb = 0;
   int edge_nb = 0;
   int tail = 0;
   int head = 0;

   char *filename = argv[1];
   char inputline[INFOSIZE];
   char *inputptr;
   char infochar;
   char bufshort2[SHORTSIZE];

   ifstream inClientFile(filename,ios::in);
   inClientFile.getline(inputline,INFOSIZE);
   inClientFile.getline(inputline,INFOSIZE);
   inClientFile >> node_nb >> edge_nb;

   int *father = new int [node_nb+1];
   father[1] = 0;

   int k = 1;
   while (k <= edge_nb)
   {
      inClientFile >> tail >> head;
      if (tail < head)
         father[head] = tail;
      k++;
   }



   for (int j = 1; j <= node_nb; j++)
   {



      for (int i = 1; i <= 1000; i++)
      {
         for (int l = 1; l <= 1000; l++);
            dispatchEvents();
      }


      theTreeInterface->NewNode(father[j],j,5);
      sprintf(bufshort2,"\\idual bound node: %d\\iTestInformation",j);
      theTreeInterface->SetNodeInfo(j,bufshort2);
      if (j > 20)
      {
         int node = j - 20;
         sprintf(bufshort2,"\\idual bound node: %d\\iNew TestInformation\\nContains one more line.",node);
         theTreeInterface->SetNodeInfo(node,bufshort2);
```

```
        }
        dispatchEvents();
        if (j > 5)
            theTreeInterface→PaintNode(j-5,4);
        if (j > 1)
            theTreeInterface→PaintNode(j-1,6);
    }

    delete[] father;

    theTreeInterface→FinishAlgorithm(TRUE,0);
}
```

5.2.2 Derivation of the *Tree Interface*

This subsection describes how the *Tree Interface* can be derived. Remarks on all important functions, that have to be overloaded by the user, are provided. Nevertheless, the reader should study the example carefully before starting an implementation on its own.

In the example shown in this section, the derived class from the class `TreeInterface` is called `TestInterface`. As in C++ implementations usual, we distinguish between a header file and a `.cc` file.

In a derivation of the *Tree Interface*, several functions have to be overloaded. Furthermore for each subroutine, that has to be included, a class derivated from the class `NoUndoCmd` has to be constructed. These derivated classes from `NoUndoCmd` handle the menu buttons in the menu bar and each of them starts a subroutine defined by the user. Since this subsection is restricted to the description of the functions of `TestInterface`, we expect for our example to have two derivated classes from `NoUndoCmd`, and leave the description of the construction to the following subsection. Those two classes are called `Command1` and `Command2`, where the first one handles the menu button of the subroutine `clientmain1` of the file `branch1.cc` while the latter handles the menu button of the subroutine `clientmain2` of the file `branch2.cc`.

We now give a brief description of the overloaded functions of the *Tree Interface*:

5.2.2.1 The function `graphicInit`

Syntax:

```
graphicInit()
```

This function initializes all features of the graphical surface. It is a virtual function of the *GFE* and **has** to be overloaded by the user. The `graphicInit` function of the *Tree Interface* has to be called in this function. Furthermore the user has to initialize all his derived classes from `NoUndoCmd`, that handle the menu buttons for calling the subroutines.

5.2.2.2 The function `checkFile`

Syntax:

```
checkFile(FILE* file_ptr, char* filename)
```

This function is called by the *Tree Interface* every time a file is loaded via the *Load...* command, if and only if the file does **not fit** the file format of the *Tree Interface* (see also chapter 3).

Here the file has to be checked, if it has to be handed to one of the subroutines. If this is the case, the user has to make sure that the filename and a file pointer is handed to the corresponding class by calling the function `setFileName`.

Furthermore the user has to control the menu bar buttons and activate or deactivate them in correspondance of the loaded problem. Observe, that it is not only possible to activate one menu bar button, but even two or more, if the file can be read by different subroutines.

The function `checkFile` returns an integer value indicating that `checkFile` has been successful or not. In case that `FALSE = 0` is returned, the file will be handed by the *Tree Interface* to the user algorithm and the corresponding menu button is activated (see also 3.5). If this should be prohibited, make sure that `TRUE = 1` is returned.

5.2.2.3 The function `addToAlgorithmMenue`

Syntax:

```
addToAlgorithmMenue(CmdList *List)
```

This function is called, when the menu bar is initialized after invoking the *Tree Interface*. Here the commands from the class `NoUndoCmd`, that handle the new menu bar buttons, are introduced into the menu bar. Every single one of these commands has to be added by the user within this function call.

5.2.2.4 The function `DeactivationCall`

Syntax

```
DeactivationCall()
```

Every time a subroutine was started in order to visualize an algorithm, some buttons in the menu bar have to be deactivated. This has to be done to prevent undefined conditions, when pressing different buttons while some process is still not finished.

The user has to implement in any case the `DeactivationCall` from the *Tree Interface* in his overloaded functions. Furthermore he can switch of his own menu bar buttons. This function is used in the implementation of the `NoUndoCmd` classes of the user.

5.2.2.5 The function `FinishAlgorithm`

Syntax

```
FinishAlgorithm(int boolean, int returnvalue)
```

This function is the previously described function of subsubsection 5.1.4. It is not necessary to overload the function. It mainly can be used to activate the last used `NoUndoCmd` button again (This is what the integer values `_command1Active` and `_command2Active` in our example are used for.) If the function is overloaded, it has to be made sure, that the `FinishAlgorithm` of the *Tree Interface* is called within the overloaded version.

5.2.2.6 TestInterface.h

```
/**********************************************************************

            Filename   :  TestInterface.h

            Version    :  01.1995

            Author     :  Sebastian Leipert

            Language   :  C++

            Purpose    :  This is the header of the clients interface.
                          It has to be derivated from the TreeInterface.
                          It handles every user defined function and is
                          the connection to the TreeInterface. This file
                          is the master of the clients derivation of
                          the TreeInterface.

**********************************************************************/


#ifndef TESTINTERFACE_H
#define TESTINTERFACE_H


#include "TreeInterface.h"
#include "Command1.h"
#include "Command2.h"

class TestInterface : public TreeInterface {

public:

   TestInterface();
   ~TestInterface();


                                    // Virtual Function from GFE.
   virtual int graphicInit();

                                    // Virtual Functions from TreeInterface.
   virtual int checkFile(FILE* file_ptr,char* filename);
   virtual void addToAlgorithmMenue(CmdList* List);
   virtual void DeactivationCall();
   virtual void FinishAlgorithm(int boolean, int returnvalue);


private:
    int _command1Active;
    int _command2Active;

    Command1* _command1;
    Command2* _command2;

};

extern TestInterface* theTestInterface;

#endif
```

5.2.2.7 TestInterface.cc

```
/*********************************************************************

            Filename    :  TestInterface.cc

            Version     :  01.1995

            Author      :  Sebastian Leipert

            Language    :  C++

*********************************************************************/


#include "TestInterface.h"
#include <iostream.h>
#include <fstream.h>


#define LINE_1 "#FIRST PROBLEM"
#define LINE_2 "#SECOND PROBLEM"


    TestInterface* theTestInterface = NULL;

/****************************************************************
                        TestInterface
****************************************************************/


TestInterface::TestInterface()
    :TreeInterface()

    // Constructor
{
    _command1 = NULL;
    _command2 = NULL;
    _command1Active = FALSE;
    _command2Active = FALSE;

    theTestInterface = this;

}



/****************************************************************
                        ~TestInterface
****************************************************************/
```

```
TestInterface::~TestInterface()

   // Destructor
{
   delete _command1;
   delete _command2;
}




/*****************************************************************
                         graphicInit
*****************************************************************/


int TestInterface::graphicInit()

   // Initializes all features of the graphical surface. This function
   // is a virtual function of the GFE and has to be overloaded by
   // the client.
{
                                          // Initialize the graphical
                                          // features of the Tree Interface.
   TreeInterface::graphicInit();

                                          // Initialize the new commands.
                                          // FALSE means, that their buttons
                                          // won't be active when the
                                          // TestInterface is started.
   _command1 = new Command1("Command1",FALSE);
   _command2 = new Command2("Command2",FALSE);

}




/*****************************************************************
                         checkFile
*****************************************************************/


int TestInterface::checkFile(FILE* file_ptr,char* filename)

   // This function is a virtual function of the TreeInterface.
   // It has to be overloaded by the user. This function will be
   // automatically called by the TreeInterface as soon as a file
   // does not fit TreeInterface's file format. The client can check
   // in this procedure, if the file format fits the requirements of
   // the client.
   //
   // If the file fits, the user can activate the menue buttons here
   // and set the filename in his command classes.
   //
   // If the file does not fit and the user returns false, the
   // FILE WILL BE SEND BY DEFAULT TO THE USERS ALGORITHM.
{
   char inputline_1[INFOSIZE];
   char *inputptr_1;
   char infochar;
   int j = 0;


   _command1Active = FALSE;
```

```
    _command2Active = FALSE;
    _command1→deactivate();
    _command2→deactivate();

    ifstream inClientFile(filename,ios::in);

                                         // Get the first line of the loaded
                                         // problem in order to check what
                                         // kind of problem it is.
    while ( (infochar = inClientFile.get()) ≠ '\n')
        inputline_1[j++] = infochar;
    inputline_1[j] = '\0';
    inputptr_1 = inputline_1;

    while (--j ≥ 0 )
        inClientFile.putback(inputline_1[j]);

    if (!strcmp(inputptr_1,LINE_1))
                                         // File has to be handled by the
                                         // first algorithm.
    {
                                         // Activate first command button.
        _command1→activate();
        _command1Active = TRUE;
                                         // Store the file name, so the file
                                         // can be accessed every time the
                                         // first command is called.
        _command1→setFileName(file_ptr,filename);
                                         // Print out a message, if necessary.
        printf("File ");
        printf(filename);
        printf(" has been loaded.\n");
                                         // DON'T FORGET TO RETURN TRUE,
                                         // or the file will be as well
                                         // adressed to the users algorithm.
        return TRUE;
    }
    else if (!strcmp(inputptr_1,LINE_2))
                                         // File has to be handled by the
                                         // second algorithm.
    {
                                         // Activate second command button.
        _command2→activate();
        _command2Active = TRUE;
                                         // Store the file name, so the file
                                         // can be accessed every time the
                                         // second command is called.
        _command2→setFileName(file_ptr,filename);
                                         // Print out a message, if necessary.
        printf("File ");
        printf(filename);
        printf(" has been loaded.\n");
                                         // DON'T FORGET TO RETURN TRUE,
                                         // or the file will be as well
                                         // adressed to the users algorithm.
        return TRUE;
    }
    else
                                         // The file will be adressed by
                                         // default to the users algorithm.
        return FALSE;
}


/***************************************************************
```

```
                        addToAlgorithmMenue
******************************************************************/


void TestInterface::addToAlgorithmMenue(CmdList* List)

    // The newly introduced commands have to be added as submenues to
    // the menue "Algorithms". This is done here.
{
    List->add(_command1);
    List->add(_command2);
}




/*****************************************************************
                        DeactivationCall
******************************************************************/


void TestInterface::DeactivationCall()

    // When an algorithm is started, some buttons have to be deactivated.
    // This prevents the client from starting some new action which might
    // cause an undefined state and leads to a segmentation fault.
{
    TreeInterface::DeactivationCall();

    _command1->deactivate();
    _command2->deactivate();
}




/*****************************************************************
                        FinishAlgorithm
******************************************************************/


void TestInterface::FinishAlgorithm(int boolean, int returnvalue)

    // When an algorithm is finished, this function has to be called.
    // It provides the user the opportunity to start the algorithm
    // again by activating the corresponding button again.
{
    TreeInterface::FinishAlgorithm(boolean,returnvalue);

    if (_command1Active)
        _command1->activate();
    if (_command2Active)
        _command2->activate();

}
```

5.2.3 Initialization of the derived class

Every class that is used by any program has to be initialized at some point. In this case, since the program that use the class `TestInterface` is not accessible, this cannot be done by manipulating the source code. There is another way to ensure that the `TestInterface` really is initialized. With the help of a so called *application file*, *MotifApp* guarantees that the `TestInterface` is properly initialized.

This application file is a `.cc` file which has to be compiled together with the other files. Besides its include files, it consists of one line where the `TestInterface` is initialized. Without this file the *GFE* cannot create an object of the type `TestInterface`.

5.2.3.1 TestApp.cc

```
/************************************************************************

            Filename    :  TreeApp.cc

            Version     :  01.1995

            Author      :  Sebastian Leipert

            Language    :  C++

            Purpose     :  Creates a new TestInterface Object.
                           Without implementing this file, the GFE cannot
                           initialize an object of the type TestInterface.

************************************************************************/


#include "Version.h"
#include "GFEApp.h"
#include "TestInterface.h"


TestInterface *testInterface = new TestInterface();
```

The reader probably has noticed, that the `TestApp.cc` file includes a file called `Version.h`. This file has to be created by the user himself and should look like the file presented in 5.2.3.2. `Version.h` is linked to the *GFE* (actually it is a *GFE* feature) and contains the following three main information for the *GFE*:

- The version of the program. The string defining `VERSION` is written on to the frame of the window of the program.
- The program file. The string defining `PROGRAM` tells the GFE, what file it should look up in the directory `app-defaults` (see also 2.3).
- The file-prefix. The string defining `PREFIX` tells the GFE, what the prefix $*$ of the files in the directory `*Resource` is. For instance, the prefix of the *Tree Interface* is `GRAPH` (compare also 2.4). The name of the directory is `GRAPHResource` and all files read by the *Tree Interface* start with `GRAPH`.

5.2.3.2 Version.h

```
////////////////////////////////////////////////////////////////////////
//
//
//        Filename :    Version.h
//
```

```
//      Version  :   07.04.94
//
//      Author   :   Diehl & Kupke
//
//      Language :   C++
//
//      Purpose  :
//
//
///////////////////////////////////////////////////////////////////////

#ifndef VERSION_H
#define VERSION_H

#define VERSION "TestInterface Version 1.0"
#define PROGRAM "GFE"
#define PREFIX "GRAPH"

#endif
```

In case that the user gets everything compiled, but cannot start the program, since it leaves shortly after calling it with a message like:

```
        NO RESOURCE-FILE CALLED GRAPHResource/GRAPHStandardResource.rsc
```

then probably the `Version.h` file was forgotten or not correct.

5.2.4 Adaption of the subroutines

This subsection describes how subroutines, that have been prepared as pictured in subsection 5.2.1, are adapted to a derived class of *Tree Interface*. It also shows how a class is implemented, that manages a command button in the menu bar. Actually, both tasks are handled by the class `NoUndoCmd.cc`, that has to be included in each subroutine.

For one of the subroutines, the file `VBCCmd.cc` can be used after appropriate changes (see 5.1.1). For every additional subroutine, a file similar to the `VBCCmd.cc` file has to be constructed and linked to the derivation of the *Tree Interface*.

In our example, the subroutine `clientmain` of the `UserExample` is added to the `VBCCmd.cc`. For the two subroutines `clientmain1` and `clientmain2` out of the files `branch1.cc` and `branch2.cc` two new `NoUndoCmd` classes named `Command1` and `Command2` are derived. Since the operating method is exactly the same as in the file `VBCCmd.cc` the reader is referred to the subsection5.1.1.

After the construction of the classes `Command1` and `Command2` the user should not forget to include the header files in the file `TestInterface.h`

5.2.4.1 VBCCmd.cc

```
/***************************************************************

        Filename    :  VBCCmd.cc

        Version     :  01.1995

        Author      :  Sebastian Leipert
```

```
          Language    :  C++

          Purpose     :  This is the .cc file of the user algorithm
                         command button.
                         It has to be derivated from NoUndoCmd.
                         It handles the function of the menue button
                         belonging to the user algorithm command.

*************************************************************/

#include "TreeInterface.h"
#include "VBCCmd.h"
#include "branchIncludes.h"
#include <string.h>


/************************************************************
                        VBCCmd
*************************************************************/


VBCCmd::VBCCmd(char* name,int active)
      : NoUndoCmd(name,active)

   // Constructor
{
}



/************************************************************
                      setFileName
*************************************************************/


void VBCCmd::setFileName(FILE* ptr,char* filename)

   // The name of the file that has been read as input is set.
{
   strcpy(_FileName,filename);
   _FilePtr = ptr;
}




/************************************************************
                         doit
*************************************************************/


void VBCCmd::doit()

   // The Procedure is called when the button in the menue bar,
   // that belongs to this command, is pressed. It will then call
   // a user defined function.
{
                                          // Print a message if necessary.
   printf("Execute Clients Program.\n");
                                          // Make sure that old trees are
                                          // removed.
   theTreeInterface->Tree()->clean_tree();
                                          // Make appropriate deactivation
                                          // calls.
   theTreeInterface->DeactivationCall();
```

```
                                        // Call the clients function.
   char* arg[2];
   arg[1] = _FileName;
   clientmain(1,arg);
}
```

5.2.4.2 Command1.h

```
/*********************************************************************

            Filename    :  Command1.h

            Version     :  01.1995

            Author      :  Sebastian Leipert

            Language    :  C++

            Purpose     :  This is the header of the first command button.
                           It has to be derivated from NoUndoCmd.
                           It handles the function of the menue button
                           belonging to the first command.

*********************************************************************/


#ifndef COMMAND1_H
#define COMMAND1_H

#include "NoUndoCmd.h"

class Command1 : public NoUndoCmd{

public:

   Command1(char*, int);

   void setFileName(FILE* ptr,char* filename);

protected:

   void doit();

private:

   char _FileName[INFOSIZE];
   FILE* _FilePtr;

};


#endif
```

5.2.4.3 Command1.cc

```
/*********************************************************************

            Filename    :  Command1.cc

            Version     :  01.1995
```

```
        Author      :  Sebastian Leipert

        Language    :  C++

        Purpose     :  See .h File.

***********************************************************************/

#include "TestInterface.h"
#include "Command1.h"
#include "branchIncludes.h"
#include <string.h>




/*************************************************************
                        Command1
*************************************************************/


Command1::Command1(char* name,int active)
       : NoUndoCmd(name,active)

   // Constructor
{
}




/*************************************************************
                      setFileName
*************************************************************/


void Command1::setFileName(FILE* ptr,char* filename)

   // The name of the file that has been read as input is set.
{
   strcpy(_FileName,filename);
   _FilePtr = ptr;

}




/*************************************************************
                          doit
*************************************************************/


void Command1::doit()

   // The Procedure is called when the button in the menue bar,
   // that belongs to this command, is pressed. It will then call
   // a user defined function.
{
                                             // Print a message if necessary.
   printf("Execute Clients Program.\n");
                                             // Make sure that old trees are
                                             // removed.
   theTreeInterface->Tree()->clean_tree();
                                             // Make appropriate deactivation
```

```
                                          // calls.
  theTestInterface->DeactivationCall();

                                          // Call the clients function.
  char* arg[2];
  arg[1] = _FileName;
  clientmain1(1,arg);
}
```

5.2.4.4 Command2.h

```
/************************************************************************

        Filename    :  Command2.h

        Version     :  01.1995

        Author      :  Sebastian Leipert

        Language    :  C++

        Purpose     :  This is the header of the first command button.
                       It has to be derivated from NoUndoCmd.
                       It handles the function of the menue button
                       belonging to the first command.

************************************************************************/


#ifndef COMMAND2_H
#define COMMAND2_H

#include "NoUndoCmd.h"

class Command2 : public NoUndoCmd{

public:

  Command2(char*, int);

  void setFileName(FILE* ptr,char* filename);

protected:

  void doit();

private:

  char _FileName[INFOSIZE];
  FILE* _FilePtr;

};


#endif
```

5.2.4.5 Command2.cc

```
/************************************************************************

        Filename    :  Command2.cc
```

```
            Version     :  01.1995

            Author      :  Sebastian Leipert

            Language    :  C++

            Purpose     :  See .h File.

*************************************************************************/

#include "TestInterface.h"
#include "Command2.h"
#include "branchIncludes.h"
#include <string.h>



/****************************************************************
                        Command2
****************************************************************/


Command2::Command2(char* name,int active)
        : NoUndoCmd(name,active)

   // Constructor
{
}




/****************************************************************
                      setFileName
****************************************************************/


void Command2::setFileName(FILE* ptr,char* filename)

   // The name of the file that has been read as input is set.
{
   strcpy(_FileName,filename);
   _FilePtr = ptr;
}




/****************************************************************
                          doit
****************************************************************/


void Command2::doit()

   // The Procedure is called when the button in the menue bar,
   // that belongs to this command, is pressed. It will then call
   // a user defined function.
{
                                          // Print a message if necessary.
   printf("Execute Clients Program.\n");
                                          // Make sure that old trees are
                                          // removed.
   theTreeInterface->Tree()->clean_tree();
                                          // Make appropriate deactivation
                                          // calls.
```

```
    theTestInterface→DeactivationCall();

                                        // Call the clients function.
    char* arg[2];
    arg[1] = _FileName;
    clientmain2(1,arg);
}
```

Chapter 6

Command Reference

Commands in the menu bar can be activated by either clicking them with the mouse, or pressing the *Meta_l* key and the key of the underlined letter of the commands name at the same time. This opens a submenu and all of the commands in the submenus can be either activated by the use of the mouse or the *Meta_l* key. Some of the commands, as eg. the *Load...* command can be activated by a possible key combination of the *Ctrl* key and some other letter key, such as *Ctrl+L* for the *Load...* command, without activating the menu bar button first. If this possibility exists, the combination is shown behind the command. If the user wishes so, he is able to change this combination. For further information the reader therefore is referred to 2.3.4.

We now give a brief description of the commands available in the *Tree Interface*. All commands that have "..." in their name generate a pop up menu when activated. A pop up menu is divided into a *control area* and an *action area*. The control area holds informations, scalers, buttons etc. for using the different features of the *Graph Interface* or the *Tree Interface*. The action area handles buttons as *Cancel*, *Apply*, *OK* and *Help*. If necessary, we give remarks on the pop up menu as well.

6.1 File

All commands shown here are derived from the *GFE*. They handle input and output and the *Quit* command.

6.1.1 Load...

The *Load* command of course handles the input. What kind of files will be accepted by the *Tree Interface* is discussed in chapter 3.

6.1.2 Save...

The *Save* command is a feature of the *GFE* which is of no use for the *Tree Interface*. Therefore the button is in a deactivated state and cannot be used.

6.1.3 Print..

The *Print* command generates a postscript file of the tree drawn in the draw area. The pop up menu gives the user the opportunity to choose the size of the output and a few various options.

6.1.4 Quit

The *Quit* command activates a small pop up menu, that checks for the possible wrong use of the button. Quitting means that all action that takes place, will be interrupted at that moment.

6.2 Edit

The command shown here is derived from the *GFE*.

6.2.1 Undo

One feature of the *OSF/Motif* library of Young [You92] is the distinction between commands that can be undone and commands that do not have this feature. The *Undo* command exactly does what it says. The *Tree Interface* only has one such command: the zoom command. All other commands do not have this feature.

Observe that the *Undo* command keeps only the last applied command "in mind". So it can only be used for undoing this last command. If this command was one not having the undo feature, the *Undo* command will be deactivated.

6.3 View

All commands shown here are derived from the *GFE*. They handle the general output on the screen as colour and size of the nodes.

6.3.1 Area Zoom

The *Area Zoom* command enables the user to choose an area of the size of a rectangle, that will be entirely drawn in over the complete draw area (see 1.3). To do so, the user activates the command via mouse button click or pressing the key combination *Ctrl+Z*.

The user then moves the mouse cursor into the draw area. Here the area, that has to be zoomed, is marked by pressing the left mouse button and keeping it pressed until the area is entirely surrounded by a rectangle. This rectangle is only drawn, while the mouse button is pressed. After releasing the mouse button, the area within the rectangle will be drawn in the size of the draw area.

To indicate that the *Area Zoom* command is active, the shape of the mouse cursor differs from the normal arrow shape as soon as the cursor is moved into the draw area. After pressing and releasing the left mouse button, the cursor will get its normal shape again.

The *Area Zoom* command is a command with an undo function. After finishing the zooming, the *Undo* command within the *Edit* menu is active. This enables the user to undo the *last* zooming. Observe, that if two zooming commands have been applied, it is not possible to undo both commands. In that case the following command is needed.

After zooming a part of the drawing, it is possible to drag the visible notch of the drawing over the entire area with the help of the middle mouse button (see also 6.8.2).

6.3.2 Fit To Window

The *Fit To Window* command can be applied after applying the *Area Zoom* command twice or more. It enables the user to get the drawing of the tree in the original size.

6.3.3 Repaint

The *Repaint* command paints the entire draw area again. This is of special purpose, when a lot of elements are deleted or drawn again while other elements are kept on the screen without being redrawn. Rounding problems then sometimes produce an offending screen output. Since the internal storage of the coordinates within the *GFE* is much more exact, it helps to use the *Repaint* command for completely redrawing the entire picture.

Nevertheless, this will be not much of a problem for the *Tree Interface*, since changing the tree will always cause a new computation of the coordinates of the nodes of the tree and therefore always results in drawing the complete tree again.

Observe that a repaint only repaints what is actually shown within the draw area. So if a repaint is demanded after the application of the *Area Zoom* command, only the zoomed area will be repainted.

6.3.4 Color/Context-Chooser...

The *Color/Context-Chooser* command changes the colour of the background. In the original design of the *GFE*, this command is supposed to handle all different kinds of contexts, such as background, foreground, different types of rectangles, circles and all kinds of various geometric objects, that are needed to realize the special purpose of a visual program. But since the *Tree Interface* just draws trees consisting of nodes and edges and a simple background, the user only gets the opportunity to change the colour of the background with this command.

Furthermore changing the colour of the nodes and edges is not provided by this command, since the *Graph Interface* supports this already by the commands *Nodes...* and *Edges...*.

When the *Color/Context-Chooser* is called a pop up menu will appear, where a context first has to be *selected*, before it can be *changed*. This results in the appearance of a second pop up menu, where the user actually can choose the colour.

Of course there is the possibility to change the colour of the background in general, without calling every time the *Color/Context-Chooser*. For further information the reader is referred to 2.3.1.

6.3.5 Nodes...

The *Nodes* command offers the user the opportunity to apply the following changes to a selected node categorie:

- colour,
- radius,
- nodes drawn as circles,
- nodes drawn as squares,
- nodes drawn as filled nodes,
- nodes drawn as lined nodes:
  - with solid line,
  - with dashed line,
  - with double dashed line,
  - with different line width,
  - with different join,
- nodes drawn with their numbers displayed,
  - with different fonts,
  - with different colours.

To achieve this, the user first chooses a node categorie and then presses the button *Change*. This causes a pop up menu to appear, which covers most of the above mentioned features. For changing the colour and the line, the user needs to press the button *Color* and for changing the font of numbers, the button *Font* has to be pressed. In both cases an appropriate pop up menu will appear to satisfy the needs of the user.

The user should be aware of the fact, that this feature of the *Graph Interface* just changes one node categorie at the time. This may cause some difficulties when a lot of different categories are used, eg. for representing a lot of different colours. So the reader is referred to two different features of this tool:

1. The appearance of the different nodes can be modified with the help of the `GRAPHStandardResource.rsc`. So the user just adapts the look of the node categories once to his needs instead of changing them every time, when he calls the program (see 2.4.3.11).
2. The most needed feature is changing the radius of the nodes, especially when the trees start to get larger. Therefore the *Scaler* (see 6.4.2) should be called, which changes the radius for all node categories.

6.3.6 Edges...

The *Edges* command offers the user the opportunity to apply the following changes to a selected edge categorie:

- colour
- solid line,
- dashed line,
- double dashed line,
- line width,
- join,
- cap.

To achieve this, the user first chooses an edge categorie and presses the button *Change*. A pop up menu then will appear, where the necessary changes can be activated.

6.4 Applications

All commands shown here are a feature of the *Tree Interface*. They handle the general applications for the tools of the *Tree Interface*.

6.4.1 Visualize Algorithm

The command *Visualize Algorithm* starts a simple run through the tree, assuming that the nodes were added to the tree according to the numbers they have. It first paints the complete tree in Standard Shade (see 1.3), then starts painting all nodes one by one in Standard Colour. By doing this, the next node which will be painted in Standard Colour, is drawn in Standard Highlight.

6.4.2 Scaler...

The command *Scaler* satisfies four different needs when drawing a tree nicely. There are three main separation values between nodes respected, when computing the coordinates of the nodes (see also 1.2): the sibling separation value, the level separation value and the subtree separation value. All three have a default value of 4 except the level separation value with a default value of 2, in order to achieve a nice and geometrically balanced drawing. By changing the corresponding entries of the `GRAPHStandardResource.rsc` file, the user may as well have chosen other separation values. Nevertheless, since large trees tend to be very wide, up to hundreds of nodes in one level by just a few levels in total, the drawings of such trees sometimes do not satisfy the users impression of an aesthetical drawing. So the *Scaler* gives the user a tool to correct the first choice of separation values.

In order to extend the tree in vertical direction, the user can choose a larger level separation value via the *Scaler*. If the spacing between siblings or subtrees has to be changed, the user can change sibling and subtree separation values. Furthermore it is possible to enlarge the radius of all nodes. Observe that this command changes the radius of all node categories at once, while the command described in 6.3.5 only changes one node categorie at the time.

The largest level separation value is bounded by the width of the tree divided by the number of levels. So if large separation values for siblings and subtrees are used, the maximum value, that can be chosen for the level separation, is larger as well. It is not possible to extend the tree further into vertical direction, because this would not make sense in terms of nice tree drawings. Furthermore, the maximal value for sibling and subtree separation is restricted by 32. The reason for setting this arbitrary bound is motivated by the fact that a lot of different values do not provide a lot of different drawings. To be more precisely: the layout of a tree where all separation values are 4 and the radius is set to 1 does not differ from the layout of the same tree, where all separation values are set to 8 and the radius of the nodes is 1.5.

The maximal value of the scaler for radius is always bounded by

$$\min\left\{\frac{\texttt{<sepvalue>}+2}{2} \mid \texttt{<sepvalue>} \text{ is separation value}\right\}$$

This is done for inhibiting drawings where the nodes intersect each other (although this can be achieved by changing the radius of a node categorie via the menu *Nodes...*). Any time the user changes separation values, the maximal values for the radius scaler is adapted if necessary, even when the pop up menu was not closed. If the current value of the radius then exceeds the new maximal value it will be reset to the maximal value.

6.4.3 Browser Mode

If the user activates the *Browser Mode*, he can *browse* through the tree by using the mouse and keeping its left button pressed. Anytime, the user then presses the left mouse button and *moves* the mouse, the node which is next to the cursor, is highlighted and its main information is shown in left display of the display area.

To indicate that the *Browser Mode* is active, the shape of the mouse cursor differs from the normal arrow shape, as soon as the cursor is moved into the draw area. The mouse cursor then has changed to crosshair shape.

The *Browser Mode* might be of special need if the trees are rather large and the user searches for a node with some special information. Using the Node Information Window then is rather time consuming and very unhandy.

6.4.4 Normal Mode

The *Normal Mode* command switches the *Browser Mode* of and changes the cursor in the draw area back to normal arrow shape. Using the left mouse button then has no effect anymore.

6.5 Algorithm

The command shown here is a feature of the *Tree Interface*. It handles a user defined application. Observe that in a derived application of the *Tree Interface*, this menu contains more buttons handling several different user defined applications.

6.5.1 User Algorithm

The command *User Algorithm* starts any users algorithm, which was adapted to the *Tree Interface*. For adapting an algorithm to this tool, the reader is referred to chapter 5.

6.6 Emulation

All commands shown here are a feature of the *Tree Interface*. They handle the afterward emulation of an algorithmic growth of a tree.

6.6.1 Start Emulation

This command starts the emulation. It will only be active, if a corresponding file has been loaded first (see 3.4).

6.6.2 Setup

The command *Setup* calls a pop up menu, which offers the user the following informations to the emulation process:

- It shows the complete time that the emulation needs.
- It shows the time which still has to be processed.

Furthermore it offers the following applications to the emulation process:

- to decide how much time the emulation process should need,
- single step processing,
- to show the complete tree during the process, where the nodes, that have not been processed yet, are drawn in Standard Shade.

All applications can be used either before or during the emulation process. If the user decides to use one of the features during the process, we strongly suggest to interrupt the process first.

6.6.3 Interrupt

This command interrupts an emulation. It will only be active, if the process was started first.

6.6.4 Continue

This command continues an emulation after it was interrupted.

6.6.5 Stop Emulation

This command stops an emulation process.

6.7 Online

The command shown here is a feature of the *Tree Interface*. It handles reading input from standard input.

6.7.1 Standard In

This command actually enables the *Tree Interface* to read from standard input. Before the button *Standard In* was not pressed, the *Tree Interface* is not able to read from standard input. So the convenient strategy when piping program output into the *Tree Interface* is as follows:

1. Call a program `<userprogram>` and pipe its output to the *Tree Interface*.
2. As soon as the window of the *Tree Interface* appears, press the button *Standard In*. Since all output of `<userprogram>` was buffered, the *Tree Interface* now starts reading the input from standard input.

The user has to make sure that the output of `<userprogram>` fits the formal file format of the *Tree Interface* (see in any case section 3.6).

6.8 Mouse-button commands

This section gives an overview of the commands that are bound to mouse buttons. We expect a mouse with three buttons.

6.8.1 Left Button

The left button has been overloaded by two features,

- the area zoom command (see 6.3.1) and
- the browser mode (see 6.4.3).

Since both features use the same mouse button, none of them is available in the first hand. They have to be activated before they can be used. For the explicit description of the use we refer to the corresponding subsections.

6.8.2 Middle Button

The middle button has been overloaded with a **dragging feature**. After using the area zoom command (see 6.3.1) the user is able to drag the small visible notch over the entire drawing. To do so, place the cursor into the draw area, press the middle mouse button and keep the button pressed while moving the mouse. The result is the same as if the picture was dragged into the direction where the cursor was moved.

6.8.3 Right Button

The right mouse button can be used to open up a `Node Information Window`. If the cursor is placed into the draw area and the right mouse button is pressed, such an information window will apear, displaying the information of the node next to the cursor (see also 4.1).

Chapter 7

Appendix

The appendix includes a selection of files of the implementation of the *Tree Interface*.

7.1 def.glb

This file contains all defines, that are used anywhere in the implementation of the *Tree Interface*.

```
/***************************************************************

             Filename    :  def.glb

             Version     :  01.1995

             Author      :  Sebastian Leipert

             Language    :  C++

             Purpose     :  Global file containing all definitions,
                            that are used in the TreeInterface and
                            all included classes.

***************************************************************/

#ifndef DEF_GLB
#define DEF_GLB

                                          // Boolean definitons.
#define TRUE 1
#define FALSE 0

                                          // Array sizes used for
                                          // node informations.
#define SHORTDIM 16
#define INFOSIZE 1024
#define SHORTSIZE 128

                                          // Temporary array size used for
                                          // storing trees of unknown size.
#define ARRAYSIZE 100

                                          // Number of Dialogwindows.
#define DIALOGNUM 10

                                          // Number of node categories.
```

```
#define COLOURCOUNT 20

                                        // Default node categories.
#define STANDARDCOLOUR 1
#define STANDARDHIGHLIGHT 2
#define STANDARDSHADE 3

                                        // Default edge categorie.
#define STANDARDEDGECOLOUR 1

                                        // Maximal separation value.
#define MAXSEPARATION 32

                                        // Lines of the headers of the
                                        // files in TreeInterface format.
#define FIRSTLINE "#TYPE: COMPLETE TREE"
#define SECONDLINE_1 "#TIME: SET"
#define THIRDLINE_1 "#BOUNDS: SET"
#define FOURTHLINE_1 "#INFORMATION: STANDARD"
#define FOURTHLINE_2 "#INFORMATION: EXCEPTION"
#define FIFTHLINE_1 "#NODE_NUMBER: NONE"
#define FIFTHLINE_2 "#NODE_NUMBER: AVAILABLE"

#endif
```

7.2 TreeInterface.h

This section contains the header of the main interface of the *Tree Interface*. Here all threads are linked together and controlled. This interface is also important for users, who want to derive the *Tree Interface*. In order to do so, they have to construct a derived class from the class `TreeInterface`. Furthermore the class `TreeInterface` handels all customer function calls that can be used in subroutines included to the *Tree Interface* by the user. Since the implementatory file is way to large, we decided to suppress its depictment in this manual.

```
/*****************************************************************

            Filename    :  TreeInterface.h

            Version     :  01.1995

            Author      :  Sebastian Leipert

            Language    :  C++

            Purpose     :  Interface for drawing trees.
                           A drived class from GraphInterface.

*****************************************************************/


#ifndef TREEINTERFACE_H
#define TREEINTERFACE_H

#include "GFE.inc"
#include "GraphInterface.h"
#include "queue.h"
#include "tree.h"
#include "GFEWindow.h"
#include "TreeDiagArray.h"
#include "VisualizeAlgoCmd.h"
#include "TreeScalerCmd.h"
#include "TreeSearcherCmd.h"
```

```
#include "SwitchToSearcherCmd.h"
#include "NormalModeCmd.h"
#include "EmulationCmd.h"
#include "EmlSetupCmd.h"
#include "EmlInterruptCmd.h"
#include "EmlStopCmd.h"
#include "EmlContinueCmd.h"
#include "StandardInputCmd.h"
#include "VBCCmd.h"
#include <X11/cursorfont.h>


class TreeInterface : public GraphInterface {

public:

   TreeInterface();
   ~TreeInterface();



/****************************************************************
       Virtual Functions from the GFE-ClientInterface.
****************************************************************/

   // Virtual functions of the GFE that have been overloaded by
   // the TreeInterface.

   virtual int graphicInit();
                                   // Initializes all graphical fonts.

   virtual int menuInit(MenuBar*);
                                   // Initializes the menubar.

   virtual int load(FILE*,char*);
                                   // Manages the load command.



/****************************************************************
          Virtual Functions from TreeInterface.
****************************************************************/

   // Functions introduced for an easy handling of a derivation
   // of the TreeInterface. They can be overloaded by the customer
   // at will. No special purpose in the TreeInterface itself.

   virtual int checkFile(FILE* file_ptr,char* filename){ return FALSE; };
                                   // Checks a file for customer purposes.
   virtual void addToAlgorithmMenue(CmdList* List){};
                                   // Adds menue buttons into the
                                   // menue "Algorithms".



/****************************************************************
          Special Purpose Virtual Functions from TreeInterface.
****************************************************************/

   // Functions can be overloaded by the customer in a derivation
   // of the TreeInterface. Functions are needed in the TreeInterface.

   virtual void DeactivationCall(){ _visualizeAlgo->deactivate();
                                     _branchAndCutCmd->deactivate(); };
                                   // Deactivate command buttons
```

```
                                          // in the menue bar.
    virtual void ActivationCall(){ _treeScaler→activate();
                                           _switchToSearcherCmd→activate(); };
                                          // Activate command buttons
                                          // in the menue bar.



/*************************************************************
         Customer function calls from TreeInterface.
*************************************************************/

    // Functions introduced for adapting customer subroutines to
    // the TreeInterface.

    int SetNodeInfo(int node,char* information);
                                          // Sets the information of a node.
    int AddNodeInfo(int node,char* information);
                                          // Adds information to already
                                          // existing information of a node.
    void PaintNode(int node,int colour);
                                          // Sets a colour of a node
    void NewNode(int father, int newNode,int colour);
                                          // Introduces a new node. Includes
                                          // repainting the tree.
    void NewNode(int father, int newNode,int colour,int draw);
                                          // Introduces a new node. Repainting
                                          // the tree is dependent on the
                                          // boolean value of draw.
    void UpperBound(double bound);
                                          // Prints out the upper bound.
    void LowerBound(double bound);
                                          // Prints out the lower bound.
    void RepaintTree();
                                          // Repaints a tree.



/*************************************************************
        Virtual Customer function calls from TreeInterface.
*************************************************************/

    // Function introduced for adapting customer subroutines to
    // the TreeInterface. Can be overloaded by the client in a
    // derivation of the TreeInterface.

    virtual void FinishAlgorithm(int boolean, int returnvalue);
                                          // After finishing a customers
                                          // subroutine, this function
                                          // activates command buttons
                                          // and starts valid clean ups.



/*************************************************************
                        Return Values.
*************************************************************/

    // Public functions returning values.

    double ActualRadius(){ return _actualRadius; };
                                          // Returns the actual radius of
                                          // the nodes.
    tree* Tree(){ return &_T; };
                                          // Returns a pointer to the tree.
```

```
/**************************************************************
         Public commands for the Emulation Process.
**************************************************************/

   // Public commands used only for the emulation process.
   // They only supply pointers to the different Cmd classes,
   // which supports the Cmd classes to interact each other.

   EmulationCmd* emulationCmd(){ return _emulationCmd; };
                                   // Returns the pointer to the start
                                   // command of the emulation process.
   EmlInterruptCmd* emlInterruptCmd(){ return _emlInterruptCmd; };
                                   // Returns the pointer to the
                                   // interrupt command.
   EmlStopCmd* emlStopCmd(){ return _emlStopCmd; };
                                   // Returns the pointer to the
                                   // stop command.
   EmlContinueCmd* emlContinueCmd(){ return _emlContinueCmd; };
                                   // Returns the pointer to the
                                   // continue command.
   EmlSetupCmd* emlSetupCmd(){ return _emlSetupCmd; };
                                   // Returns the pointer to the
                                   // setup command.



/**************************************************************
           Further public commands of TreeInterface
**************************************************************/

   // Public commands used in different classes that are included
   // by the TreeInterface.

   void NewActualRadius(double rad){ _actualRadius = rad; };
                                   // Sets a new radius for the nodes
   void MaxCoordValues(double *x,double *y){ (*x) = _T.get_max_X_coord();
                                             (*y) = _T.get_max_Y_coord(); };
                                   // Gets the maximal coordinates
                                   // of the tree. Used for estimating
                                   // the size of coordinate systems
   void decreaseOpenWin(int number){ _treedialog->decreaseOpenWin(number); };
                                   // Decreases the number of open
                                   // Node Information Window.




private:

   TreeDiagArray* _treedialog;

   tree _T;

   int _node_nb;
   int _branchAndCutActive;

   Cursor _cursor;

   VisualizeAlgoCmd* _visualizeAlgo;
   TreeScalerCmd* _treeScaler;
   SwitchToSearcherCmd* _switchToSearcherCmd;
   TreeSearcherCmd* _treeSearcherCmd;
   NormalModeCmd* _normalModeCmd;
```

```
    VBCCmd* _branchAndCutCmd;

    EmulationCmd* _emulationCmd;
    EmlSetupCmd* _emlSetupCmd;
    EmlInterruptCmd* _emlInterruptCmd;
    EmlStopCmd* _emlStopCmd;
    EmlContinueCmd* _emlContinueCmd;

    StandardInputCmd* _standardInputCmd;

    double _actualRadius;
    double _upperBound;
    double _lowerBound;


    void Mybutton1Motion(XEvent *event);
    void Mybutton1Release(XEvent *event);
    void button3Press(Point<CoordType>);
    void SpaceBarPressEvent(XEvent *event);


    void readCompleteTree(char*);
    void NewTreePaint(int);

    static void button1MotionEventHandler(Widget widget,
                                  XtPointer clientData,
                                  XEvent *event,
                                  Boolean *continueToDiapatch);

    static void buttonReleaseEventHandler(Widget widget,
                                  XtPointer clientData,
                                  XEvent *event,
                                  Boolean *continueToDiapatch);

    static void keyPressSpaceBarEventHandler(Widget widget,
                                  XtPointer clientData,
                                  XEvent *event,
                                  Boolean *continueToDiapatch);


};

extern TreeInterface* theTreeInterface;

#endif
```

7.3 tree.*

This section contains a class called `tree`, beeing a derived class from a class called `basic_tree`. While the class `basic_tree` mainly supports basic interrogations, the class `tree` includes the positioning algorithm of Walker [Wal90]. It therefore can be understod in terms of a datatstructure including a large algorithm. Since this algorithm is the heart of the *Tree Interface*, we decided to include booth, the header and the implementatory file into the appendix.

7.3.1 tree.h

```
/*********************************************************************
```

```
        Filename    :  tree.h

        Version     :  01.1995

        Author      :  Sebastian Leipert

        Language    :  C++

        Purpose     :  A derived class from basic_tree. Computes
                       and manages the coordinates of all nodes
                       of a tree for a nice layout. Uses an
                       algorithm by John Q. Walker published in:
                       Software-Practice and Experience, vol. 20(7),
                       685-705 (July 1990) under the titel:
                       A Node-positioning Algorithm for General Trees.

*********************************************************************/


#ifndef TREE_H
#define TREE_H

#include <iostream.h>
#include <fstream.h>
#include "basic_tree.h"
#include "queue.h"

class tree : public basic_tree {
   friend char *operator>>(char*, tree&);
   friend node* allocate_tree(char* ,tree &,int &);
   friend int enter_edge(ifstream &,tree &,int);
   friend int set_node_info(int,char*,tree &);
   friend int add_node_info(int,char*,tree &);
public:

   tree();
   ~tree();


   void positiontree(double*, double*);
   void newPositionLevel(int, double*, double*);

   void enter_new_node(int, int, int);
   void clean_tree();

   int get_level_count() const;
   int get_actual_level_separation() const;
   double get_max_X_coord() const;
   double get_max_Y_coord() const;


   static int get_sib_separation();
   static int get_level_separation();
   static int get_subtree_separation();

   void set_separation_values(int sibling,int level,int subtree);

private:

   double root_X_coord;
   double root_Y_coord;

   int level_count;
   int actual_level_sep;
   double max_X_coord;
```

```
    double max_Y_coord;


    static int sibling_separation;
    static int level_separation;
    static int subtree_separation;

    void compute_coord(node *,double*,double*,double*);
    void firstwalk(node &,int,node*[]);
    void secondwalk(node *, int, double,double*,double*,double*);

    void apportion(node &,int);
    node* get_left_most(node *,int,int);

    void get_max_XYvalue(node*, double*, double*, double*);

    void deapth_first_search(node *,int);
    void del_deapth_first(node *);
    int mean_node_size(node *,node *);
    void introduce_child(node *,node *);

};


#endif
```

7.3.2 tree.cc

```
/**************************************************************

            Filename   :  tree.cc

            Version    :  01.1995

            Author     :  Sebastian Leipert

            Language   :  C++

            Purpose    :  See .h File.

**************************************************************/


#include "tree.h"
#include <iostream.h>
#include <string.h>


/*********************************************************************
                              STATIC

               This part contains all static files
*********************************************************************/





int tree::sibling_separation = 4;
int tree::level_separation = 2;
int tree::subtree_separation = 4;


/*********************************************************************
                        get_sib_separation
```

```
**********************************************************************/


int tree::get_sib_separation()
{
   return sibling_separation;
}


/**********************************************************************
                    get_level_separation
**********************************************************************/


int tree::get_level_separation()
{
   return level_separation;
}


/**********************************************************************
                    get_subtree_separation
**********************************************************************/


int tree::get_subtree_separation()
{
   return subtree_separation;
}



/**********************************************************************
                    set_separation_values
**********************************************************************/


void tree::set_separation_values(int sibling,int level,int subtree)
{
   sibling_separation = sibling;
   level_separation = level+2;       // +2 for keeping distance
   actual_level_sep = level+2;
   subtree_separation = subtree;
}







/**********************************************************************
                         PUBLIC

             This part contains all public files
**********************************************************************/



/**********************************************************************
                         constructor
**********************************************************************/


tree::tree()
```

```
    : basic_tree()
{

    root_X_coord = 0.0;
    root_Y_coord = 0.0;

    level_count = 0;
    actual_level_sep = level_separation;
    max_X_coord = 0.0;
    max_Y_coord = 0.0;
}



/***********************************************************************
                              destructor
***********************************************************************/


tree::~tree()
{
    if (root ≠ NULL)
        del_deapth_first(root);
}





/***********************************************************************
                              positiontree
***********************************************************************/


void tree::positiontree(double *x_max, double *y_max)

    // Calls the procedure compute_coord for computing the coordinates
    // of the tree nodes. Returns the values of the maximum X and Y
    // coordinates in order to adjust a coordinate system of proper size
    // to a drawing of the tree.
{

    (*x_max) = 0.0;        // maximum x coordinate
    (*y_max) = 0.0;        // maximum y coordinate
    double x_min = 0.0;        // minimum x coordinate
    double help = 0.0;

    compute_coord(root,x_max,y_max,&x_min);

                                // The smallest x coordinate might be negative.
                                // If this is the case, the tree is moved into
                                // positive direction of the x-axis until no
                                // node has a negative x-coordinate. This
                                // is necessary, since the coordinate system,
                                // in which the tree is drawn, is only positive.
                                // Observe that the value of the maximum
                                // x-coordinate has to be adjusted as well.
    if (x_min < 0)
    {
        for (int i = 1; i ≤ node_nb; i++)
        {
            help = node_array[i]→get_Xcoord();
            node_array[i]→set_Xcoord(help - x_min);
        }
        (*x_max) -= x_min;
    }
    max_X_coord = (*x_max);
    max_Y_coord = (*y_max);
}





/************************************************************************
                        newPositionLevel
************************************************************************/


void tree::newPositionLevel(int new_scaler,double* x_max, double *y_max)

    // This function is specially introduced to fit the needs of the
    // TreeInterace. In case that a user of the TreeInterface cooses
    // a new level separation value via a popup menue called "Scaler",
    // it is not necessary to compute the coordinates completely new,
    // since only the y-coordinates have to be adapted. This is done
    // here in the function newPositionLevel.
    // Furthermore, since changing the level speration value changes the
    // size of the drawing, this function returns the maximal coordinate
    // values so the coordinate system can be addapted to the new
    // drawing of the tree.
{
    double help = 0.0;

    (*y_max) = 0.0;
    (*x_max) = max_X_coord;

                                        // For all nodes, compute the
                                        // y-coordinate again.
    for (int i = 1; i <= node_nb; i++)
    {
        help = node_array[i]->get_Ycoord();
        help = (help / actual_level_sep) * new_scaler;
        node_array[i]->set_Ycoord(help);
        if (help > (*y_max))
            (*y_max) = help;
    }
                                        // Memorize the new values.
    actual_level_sep = new_scaler;
    level_separation = new_scaler;
    max_Y_coord = (*y_max);
}





/************************************************************************
                        enter_new_node
************************************************************************/

void tree::enter_new_node(int father,int new_node,int colour)

    // This function is specially introduced to fit the needs of the
    // TreeInterace. The TreeInterface offers the option to add nodes
    // to already existing trees. So this public function provides
    // the TreeInterface a tool, for adding nodes.

{
    int newroot = FALSE;
```

```
                              // The tree is empty. Construct space
                              // to hold up to a hundred nodes. This
                              // is done since the computation of the
                              // coordinates is made fast with the help
                              // of such an array. Furthermore, we do
                              // not want every time that a node is
                              // added to the tree, rebuild that array.
if (root == NULL)
{
   array_size = 100;
   node_array = new (node*)[array_size+1];
   edge_array = new (edge)[array_size];

   for (int i = 1; i ≤ array_size; i++)
      node_array[i] = new node(i);
   root = node_array[1];
   newroot = TRUE;
}

                              // The tree is not empty, but the array
                              // is full. Add space for 100 more
                              // nodes to the array.
else if (root ≠ NULL && node_nb == array_size)
{
   node** new_node_array = new (node*)[array_size+101];
   edge* new_edge_array = new (edge)[array_size+100];

   for (int i = 1; i ≤ array_size; i++)
   {
      new_node_array[i] = node_array[i];
   }

   for (int k = 1; k ≤ (array_size-1); k++)
   {
      new_edge_array[k] = edge_array[k];
   }
                              // Changed after using gcc 2.7.0
   for (int j = array_size+1; j ≤ (array_size+100); j++)
      new_node_array[j] = new node(j);
   delete[] node_array;
   delete[] edge_array;
   node_array = new_node_array;
   edge_array = new_edge_array;

   array_size = array_size + 100;
}

if (father ≠ 0)
                              // The node is not the root.
{
   if (father < new_node)
                              // Include node into the structure
                              // of the tree.
   {
      node_array[new_node]→set_parent(node_array[father]);
      introduce_child(node_array[father],node_array[new_node]);
      edge_array[node_nb].set_first(father);
      edge_array[node_nb].set_second(new_node);
      node_nb++;
      edge_nb++;
   }
}
else if (newroot)
   node_nb++;

                              // Set the nodes colour.
```

```
    node_array[new_node]→set_colour(colour);
    node_array[new_node]→set_underlying_colour(colour);
}


/************************************************************************
                              clean_tree
************************************************************************/


void tree::clean_tree()

    // Empties all information stored by the class tree, if a new tree
    // has to be saved.
{
    node_nb = 0;
    edge_nb = 0;

    root = NULL;

    root_X_coord = 0.0;
    root_Y_coord = 0.0;

    level_count = 0;
    actual_level_sep = level_separation;
    max_X_coord = 0.0;
    max_Y_coord = 0.0;

    delete[] edge_array;
    delete[] node_array;

    edge_array = NULL;
    node_array = NULL;
}


/************************************************************************
                              get_level_count
************************************************************************/

int tree::get_level_count() const

    // Returns the number of levels in the tree.
{
    return level_count;
}



/************************************************************************
                        get_actual_level_separation
************************************************************************/

int tree::get_actual_level_separation() const

    // Returns the actual level_separation.
{
    return actual_level_sep;
}
```

```
/*********************************************************************
                              get_max_X_coord
*********************************************************************/

double tree::get_max_X_coord() const

     // Returns the maximal x-coordinate of the tree.
{
    if (root ≠ NULL)
        return max_X_coord;
    else
    {
        cout ≪ "tree::get_max_X_coord:  empty tree" ≪ endl;
        return 0.0;
    }
}


/*********************************************************************
                              get_max_Y_coord
*********************************************************************/

double tree::get_max_Y_coord() const

     // Returns the maximal x-coordinate of the tree.
{
    if (root ≠ NULL)
        return max_Y_coord;
    else
    {
        cout ≪ "tree::get_max_Y_coord:  empty tree" ≪ endl;
        return 0.0;
    }
}


/*********************************************************************
                               PRIVAT

                 This part contains all privat files
*********************************************************************/


/*********************************************************************
                             compute_coord
*********************************************************************/

void tree::compute_coord(node *check_node,double* x_max,
                        double*y_max,double*x_min)

   // Computes the X and Y coordinates of all nodes in the tree.
{
  if (check_node ≠ NULL)
  {
    level_count = 0;
                                    // Scan the complete tree in deapth-first
                                    // manner in order to determine the number
                                    // of levels of the tree and setting
                                    // previous y-coordinates.
    deapth_first_search(check_node,0);

    node **level_array;      // Array of pointers to nodes.
                                    // An entry at position i in level_array
                                    // contains a pointer to the last visited
                                    // node at this level (the so-called
                                    // previous node). If another node
                                    // is checked at level i, than level_array[i]
                                    // points to the left neighbor of the node.
    level_array = new (node*)[level_count+1];


    for (int i = 0; i ≤ level_count; level_array[i++] = NULL);

                                    // Do a post-order walk in order to assign to
                                    // every node a preliminary x-coordinate,
                                    // held in the field prelim_x of every node.
                                    // In adition, internal nodes are given
                                    // modifiers, which will be used to move
                                    // their offsprings to the right.
    firstwalk(*check_node,0,level_array);


    root_X_coord = 0;       // Set the x-coordinate of the root by default 0.
    root_Y_coord = 0;       // Set the y-coordinate of the root by default 0.

                                    // Do a pre-order walk in order to give each
                                    // node its final x-coordinate by summing
                                    // its preliminary x-coordinates and the
                                    // modifiers of all nodes ancestors.
    secondwalk(check_node,0,0,x_max,y_max,x_min);

    delete[] level_array;
  }
}




/*************************************************************************
                        firstwalk
*************************************************************************/


void tree::firstwalk(node &check_node,int level,node* level_array[])

  // Does a post-order walk in order to assign to every node a
  // preliminary x-coordinate, held in the field prelim_x of every node.
  // In adition, internal nodes are given modifiers, which will be
  // used to move their offsprings to the right.
{

                                    // Set the pointer to the previous node
                                    // at this level.
```

```
check_node.set_leftneighbor(level_array[level]);
level_array[level] = &check_node;

check_node.set_modifier(0);

if (check_node.is_leaf())
{
    if (check_node.has_left_sib())
                                    // Determine the preliminary x-ccordinate
                                    // based on the following facts:
                                    // the prelim_x of the left sibling,
                                    // the separation between the sibling nodes,
                                    // and the meansize of the left sibling
                                    // and the current node.
    {
        check_node.set_prelim(check_node.get_leftsibling()→get_prelim()
                                  + tree::get_sib_separation()
                                  + mean_node_size(check_node.get_leftsibling(),
                                                   &check_node));
    }
    else
                                    // There is no sibling on the left, that
                                    // we have to take care about.
        check_node.set_prelim(0);
}

else
                                    // This node is not a leaf, so call this
                                    // procedure recursively for each for
                                    // its children.
{
    node ∗left_most = NULL;
    node ∗right_most = NULL;
    double midpoint = 0;


    left_most = right_most = check_node.get_firstchild();
    firstwalk(∗left_most,level+1,level_array);

    while (right_most→has_right_sib())
    {
        right_most = right_most→get_rightsibling();
        firstwalk(∗right_most,level+1,level_array);
    }

    midpoint = (left_most→get_prelim() + right_most→get_prelim())/2;

    if (check_node.has_left_sib())
    {
                                    // Determine the preliminary x-ccordinate
                                    // based on the following facts:
                                    // the prelim_x of the left sibling,
                                    // the separation between the sibling nodes,
                                    // and the meansize of the left sibling
                                    // and the current node.
        check_node.set_prelim(check_node.get_leftsibling()→get_prelim()
                                  + tree::get_sib_separation()
                                  + mean_node_size(check_node.get_leftsibling(),
                                                   &check_node));

                                    // Since check_node has to be moved by the
                                    // value prelim_x to the right, all its
                                    // descendants have to do the same.
                                    // So set the modifier of the checked node by
                                    // prelim_x. In order to center check_node
                                    // over its children, subtract midpoint.
```

```
         check_node.set_modifier(check_node.get_prelim() - midpoint);

                                 // The subtree of check_node may still
                                 // overlay the subtree of its left sibling.
                                 // In order to find this out, the leftmost
                                 // descendants of check_node on each level
                                 // are examined and if necessary the complete
                                 // tree is moved to the right.
         apportion(check_node,level);
      }

      else
                                 // Center check_node over its children.
         check_node.set_prelim(midpoint);
   }

}




/*********************************************************************
                         secondwalk
**********************************************************************/


void tree::secondwalk(node *check_node, int level, double modsum,
                            double* x_max,double*y_max,double*x_min)

   // Rekursive procedure which does a pre-order walk in order to give
   // each node its final x-coordinate by summing its preliminary
   // x-coordinates and the modifiers of all nodes ancestors.
   // If the actual position of an interior node is right of its
   // preliminary place (stored in prelim_x), the subtree rooted at the
   // node must be moved to the right, so that the children of the node
   // are centered around the father. Rather than immediately readjust
   // all nodes of the subtree, each node remembers the distance to the
   // preliminary place in its modifier field mod. In this second pass
   // down the tree, the modifiers are accumulated and applied to every node.
{

                                 // Compute the x- and y coordinates of
                                 // the check_node.
   double temp_X = root_X_coord + check_node->get_prelim() + modsum;
   double temp_Y = root_Y_coord +
                                 (level * tree::get_actual_level_separation());

                                 // Set the x- and y-coordinates of check_node.
   check_node->set_Xcoord(temp_X);
   check_node->set_Ycoord(temp_Y);

                                 // Test whether the x- and y-coordinates
                                 // are maximal or minimal. This is needed
                                 // in order to adjust a coordinate system to
                                 // a drawing of the tree.
   get_max_XYvalue(check_node,x_max,y_max,x_min);

   if (check_node->has_child())
   {
      node *child_ptr = check_node->get_firstchild();
      secondwalk(child_ptr,level+1,
                  modsum + check_node->get_modifier(),x_max,y_max,x_min);
   }
   if (check_node->has_right_sib())
```

```
    {
        node *child_ptr = check_node→get_rightsibling();
        secondwalk(child_ptr,level,modsum,x_max,y_max,x_min);
    }
}



/*******************************************************************
                        apportion
*******************************************************************/


void tree::apportion(node & check_node,int level)

    // The subtree of a check_node may still overlay the subtree of its
    // left sibling. In order to find this out, the leftmost descendants
    // of check_node on each level are examined and if necessary, the complete
    // tree is moved to the right.
    // When moving a new subtree further and further to the right gaps may
    // open among smaller subtrees that were previously sandwiched between
    // larger trees. This so called the 'left-to-right gluing' problem,
    // which is cleaned up here. When moving a new large subtree to the
    // right, the distance it is moved is also apportioned to smaller interior
    // subtrees.
{

    node *leftmost = check_node.get_firstchild();
    node *neighbor = leftmost→get_leftneighbor();
    int compare_depth = 1;

    while (leftmost ≠ NULL && neighbor ≠ NULL)
                                        // Go down the levels of the subtree and
                                        // find for every level the leftmost
                                        // node of the subtree and its left neighbor.
                                        // Then compare the preliminary x-coordinates
                                        // of the leftmost and ist left neighbor
                                        // and if necessary we "move" the subtree
                                        // rooted at check_node by memorizing this
                                        // fact in check_nodes modifier field.
    {
                                        // Compute the location of check_node
                                        // and where it should be with respect to
                                        // the neighbor.
        double left_modsum = 0;
        double right_modsum = 0;
        node *ancestor_leftmost = leftmost;
        node *ancestor_neighbor = neighbor;

        for (int i = 0; i < compare_depth; i++)
        {
            ancestor_leftmost = ancestor_leftmost→get_parent();
            ancestor_neighbor = ancestor_neighbor→get_parent();
            right_modsum += ancestor_leftmost→get_modifier();
            left_modsum += ancestor_neighbor→get_modifier();
        }


                                        // Find the move_distance and apply it to
                                        // check_nodes subtree. Add appropriate
                                        // portions to smaller interior subtrees.
        double move_distance = neighbor→get_prelim() + left_modsum +
                              tree::get_subtree_separation() +
                              mean_node_size(leftmost,neighbor) -
                              leftmost→get_prelim() - right_modsum;
        if (move_distance > 0)
        {
                                    // Count interior sibling subtrees in
                                    // the left siblings of check_node.
            node *node_ptr = &check_node;
            int left_siblings = 0;

            while (node_ptr ≠ NULL && node_ptr ≠ ancestor_neighbor)
            {
                left_siblings++;
                node_ptr = node_ptr→get_leftsibling();
            }

            if (node_ptr ≠ NULL)
                                    // Apply appropriate portions to the subtrees
                                    // of the left siblings of check_node.
            {
                double portion = move_distance/left_siblings;
                node_ptr = &check_node;

                while (node_ptr ≠ ancestor_neighbor)
                {
                    node_ptr→set_prelim(node_ptr→get_prelim() + move_distance);
                    node_ptr→set_modifier(node_ptr→get_modifier()+move_distance);
                    move_distance -= portion;
                    node_ptr = node_ptr→get_leftsibling();
                }

            }

            else
            {
                                    // There is nothing to do, since
                                    // ancestor_neighbor and ancestor_leftmost
                                    // are not siblings of each other.
                                    // So moving subtrees has to be done
                                    // by an ancestor of check_node.
            }
        }

                                    // Now get the leftmost descendant of
                                    // check_node in the next lower level.
        compare_depth++;
        if (leftmost→is_leaf())
            leftmost = get_left_most(&check_node,0,compare_depth);
        else
            leftmost = leftmost→get_firstchild();
        if (leftmost ≠ NULL)
            neighbor = leftmost→get_leftneighbor();

    }

}


/*********************************************************************
                        get_left_most
*********************************************************************/


node* tree::get_left_most(node * check_node,int level,int depth)

    // Returns the leftmost descendant of check_node node at a given depth.
    // This is implemented using a post-order walk of the subtree
    // under check_node, down to the level of depth.
    // Level here is not the absolute tree level used in the two main
    // tree walks; it revers to the level below the node whose leftmost
    // descendant is beeing found.
{
    if (level ≥ depth)
        return check_node;

    else if (check_node→is_leaf())
        return NULL;

    else
    {
        node *rightmost = check_node→get_firstchild();
        node *leftmost = get_left_most(rightmost,level+1,depth);

        while (leftmost == NULL && rightmost→has_right_sib())
        {
            rightmost = rightmost→get_rightsibling();
            leftmost = get_left_most(rightmost,level+1,depth);
        }

        return leftmost;
    }
}




/***********************************************************************
                        get_max_XYvalue
***********************************************************************/

void tree::get_max_XYvalue(node* node_ptr, double* x_max,
                                    double* y_max, double* x_min)

    // Returns the maximal x- and y-coordinates as well as the minimal
    // x-coordinate found until now.
{
    if (node_ptr→get_Xcoord() > (*x_max))
        (*x_max) = node_ptr→get_Xcoord();
    if (node_ptr→get_Ycoord() > (*y_max))
        (*y_max) = node_ptr→get_Ycoord();
    if (node_ptr→get_Xcoord() < (*x_min))
        (*x_min) = node_ptr→get_Xcoord();
}



/***********************************************************************
                        deapth_first_search
***********************************************************************/

void tree::deapth_first_search(node *node_ptr,int level)

    // Recursive deapth-first-search procedure for computing a previous
    // y-coordinate and determining the number of levels of the tree.
{
    node *child_ptr = NULL;

    if (level > level_count)
        level_count = level;
    node_ptr→set_Ycoord(level);
    if (node_ptr→has_child())
    {
        child_ptr = node_ptr→get_firstchild();
        deapth_first_search(child_ptr,level+1);
        while (child_ptr→has_right_sib())
        {
            child_ptr = child_ptr→get_rightsibling();
            deapth_first_search(child_ptr,level+1);
        }
    }
}



/**********************************************************************
                          del_deapth_first
**********************************************************************/


void tree::del_deapth_first(node *node_ptr)

    // Recursive deapth-first-search procedure for a valid clean_up.
{
     node *child_ptr = NULL;
     node *right_child = NULL;


     if (node_ptr→has_child())
     {
         child_ptr = node_ptr→get_firstchild();
         right_child = child_ptr→get_rightsibling();
         del_deapth_first(child_ptr);
         while (right_child ≠ NULL)
         {
             child_ptr = right_child;
             right_child = child_ptr→get_rightsibling();
             del_deapth_first(child_ptr);
         }
     }
     delete node_ptr;
}



/**********************************************************************
                          mean_node_size
**********************************************************************/


int tree::mean_node_size(node * left_node,node * right_node)

    // Returns the mean size of the two passed nodes.
    // In this class a trivial calculation, since all nodes
    // are the same size.
{
    int node_size = 0;

    if (left_node ≠ NULL)
        node_size += left_node→get_radius();
    if (right_node ≠ NULL)
        node_size += right_node→get_radius();

    return node_size;
}
```

```
/*********************************************************************
                    introduce_child
*********************************************************************/

void tree::introduce_child(node *tail,node *head)

   // Introduces a new node head as child of its parent tail in
   // the tree.
{
   node *child = NULL;
   node *right_sib = NULL;

   if (tail→has_child())
   {
      child = tail→get_firstchild();
      right_sib = child→get_rightsibling();
      if (right_sib ≠ NULL)
      {
         child→set_rightsibling(head);
         head→set_rightsibling(right_sib);
         right_sib→set_leftsibling(head);
         head→set_leftsibling(child);
      }
      else      // right_sib == NULL
      {
         child→set_rightsibling(head);
         head→set_leftsibling(child);
      }
   }
   else
      tail→set_firstchild(head);
}
```

Bibliography

[You92] D. A. Young, *Object Oriented Programming with C++ and OSF/MOTIF*, Prentice-Hall (1992).

[RS83] E. M. Reingold and K. J. Supowit, *the complexity of drawing trees nicely*, Acta Informatica, 18, (4) (1983), 377–392.

[RT81] E. M. Reingold and J. S. Tilford, *Tidier drawing of trees*, IEEE Trans. Software Engineering, SE-7, (2) (1981), 223-228.

[ST95] S. Thienel, *ABACUS - A Branch–And–CUt System*, doctoral thesis, Universität zu Köln (1995).

[Wal90] J. Q. Walker II, *A Node-positioning Algorithm for General Trees*, Software-Practice and Experience, 20(7) (1990) 685–705.

[WS79] C. S. Wetherell and A. Shannon, *Tidy drawings of trees*, IEEE Trans. Software Engineering, SE-5, (5) (1979), 514–520.