The XGAP 4 Manual

by

Frank Celler

and

Max Neunhöffer

(email: max.neunhoeffer@math.rwth-aachen.de)

Contents

1 How to read this manual

This chapter tells you exactly, which part of this manual you have to read if you want to learn certain things about XGAP.

If you do not know anything about XGAP you might want to have a look at chapter 2. You can learn there, what XGAP is all about, what you can do with it as a user (doing no programming) and for what you can use XGAP within your own programs. You find also references to other sections of this manual to get into the details.

If you want to know how to install XGAP, then you should look into chapter 3.

If you know XGAP from its GAP3 version, you will consider chapter 10 useful. There you can quickly "update" your knowledge to the new GAP4 version and also find some technical details about the implementation, which is nearly completely new.

Perhaps you know already, that you can display subgroup lattices interactively with XGAP and you want to start with this right now. In this case, jump to chapters 4 and 5 immediately. However, you should make sure first that you have a working XGAP installed!

The XGAP library is described in these and the remaining chapters of this manual. The information is divided into the following parts:

subgroup lattices - examples

subgroup lattices - systematic description

graphic sheets
: Creating and managing graphic sheets and graphic objects. This is the lowest level.

user communication
: Handling of menus, dialogs, text selectors and popups.

graphic posets
: Display of posets in graphic sheets.

graphic graphs
: Display of mathematical graphs in graphic sheets. A lot of "routine work" for handling these things is already done and can be used in your applications!

There is a chapter for each of these parts.

2 What is XGAP?

In this chapter you find the answer to the above question beginning from a short overview up to a description of the technical concept.

2.1 Basics

The idea of XGAP is that GAP should be able to control graphics. A graphical user interface is sometimes easier to use than a text and command oriented one and there are mathematical applications for which it can be quite useful to visualize objects with computer graphics.

On the other hand it is not sensible to change the whole concept and user interface of GAP because it is not advisable to put all the facilities of GAP into a menu system. So XGAP is a separate C program running under the X Window System, which starts up a GAP job and allows normal command execution within a window. Note that the online help of GAP is available, however it will appear in a separate window.

In addition there is a library written in GAP, which makes it possible to open new windows, display graphics, control menus and do other graphical user communication in GAP via the separate C part.

Built on those "simple" windows and graphic objects are other libraries which display graphs and posets in a window and allow the user to move vertices around, select them and invoke GAP functions on mathematical objects which belong to the graphic objects.

One "application" of these libraries is a program to display subgroup lattices interactively. So XGAP works as a front end for mathematical operations on subgroup lattices. It is possible to "switch" between the graphics and the GAP commands. This means that you can for example use the graphically selected vertices resp. subgroups to do your own calculations in the command window. You can then display your results again as vertices in the lattice.

Of course there are other applications possible and the libraries are developed with code reusage in mind.

2.2 What you can do with XGAP

XGAP graphic sheets work graphic object oriented. This means that the basic graphic objects are not pixels but lines, rectangles, circles and so on. Although technically everything on the screen consists of pixels XGAP remembers the structure of your graphics via higher objects. This has advantages as well as disadvantages. Do not expect to be able to place pixel images into your XGAP graphic sheets. That is as of now **not possible** with XGAP and probably will never be, because it is not the idea of the design.

What you can do is create, move around and change lines, circles, text and so forth in graphic sheets. Your programs can communicate with the user via graphical user interfaces like mouse, menus, dialogs, and so on.

It is very easy to link this graphical environment with your programs in the mathematical environment of GAP. So you can very quickly implement visualizations of the mathematical objects you study. The user can select objects, choose functions from menus and ask for more information with a few mouse clicks.

A good example for this approach is the implementation of the interactive Todd-Coxeter-Algorithm to compute coset tables in finitely presented groups. It uses the graphical features of XGAP to give the user

quick and easy access to the algorithm by a few mouse clicks. This program was written by Ludger Hippe in Aachen using XGAP3 and is currently ported to XGAP4 and extended by Volkmar Felsch.

Another nice little example is in the `examples` subdirectory in the XGAP distribution. It was written by Thomas Breuer (Aachen) to demonstrate the features of XGAP. The user gets a small window with a puzzle and can solve it using the mouse. You can test this example by starting XGAP and `Read`ing the file `pkg/xgap/examples/puzzle.g`. You can do this by using

```
gap> ReadPkg("xgap","examples/puzzle.g");
gap> p := Puzzle(4,4);
```

You do not have to invent the wheel many times. For certain mathematical concepts like graphs, posets or lattices XGAP provides implementations which can be adapted to your special situation. You can use those parts of the code you like and substitute the other parts to adapt the behaviour of the user interface to your wishes.

2.3 How does it work?

XGAP consists of a C program `xgap` (in the following `xgap` in typewriter style refers to this C part) separate from GAP, and of some libraries written in the GAP language. `xgap` is started by the user and launches a GAP job in the background. It then talks to this GAP job. Especially it displays all the output which comes from GAP in the communication window and feeds everything the user types in this window into the GAP job.

But there is also some communication with the GAP job about the graphics that should be displayed. Because GAP has no concept of putting graphics on the screen, this part is done by `xgap`. Therefore there is a protocol between the GAP part of XGAP running in the GAP session and `xgap` which is embedded in the input/output stream. The user does not notice this. `xgap` stores all windows and graphic objects and does all the work necessary for displaying windows and managing user communication and so on.

The GAP part of XGAP also stores all graphical information, but in form of GAP objects. The user can inspect all these structures and use them in own programs. Changes in these structures are transmitted through the communications protocol to `xgap` and are eventually displayed on the screen. User actions like mouse clicks or keyboard events are caught by `xgap` and transmitted to the GAP job via function calls that are "typed in" as if the user had typed them. So the library can work on them and change the GAP objects accordingly.

Technically, XGAP is a share package and one of the first commands that is executed automatically within the GAP session is a `RequirePackage("xgap")` call. This reads in the extra XGAP libraries. They are found in the `pkg/xgap/lib` subdirectory, normally in the main GAP directory. The files contain the following:

`window.g`
: basic definitions for the communications protocol

`sheet.g[di]`
: graphic sheets and their operations

`color.g[di]`
: color information

`font.g[di]`
: text font information

`menu.g[di]`
: menus, dialogs, popups, and user communication

`gobject.g[di]`
: low level graphic objects and their operations

`poset.g[di]`
: graphic graphs and graphic posets

`ilatgrp.g[di]`
: graphic subgroup lattices

`meataxe.g[di]`
: support to display submodule lattices calculated within the C MeatAxe

The user normally does not need to know this or the details of it. However, it is important to understand that the program `xgap` is highly machine or at least operating system dependent. There is no generic way to access graphics across different platforms up to now. XGAP should run on all variants of Unix with the X Window System Version 11 Release 5 or higher. As of now XGAP does not run on Microsoft Windows or MacOS. It is also definitely **not** easily ported there, because some important features that are used within XGAP are missing there (such as pseudo terminals). There are currently no plans underways to do work in this direction. However, portations are very welcome! If you would like to put some work into this, please contact Max Neunhöffer (email: `max.neunhoeffer@math.rwth-aachen.de`).

2.4 Historical Remarks and Acknowledgements

A first program for drawing a diagram showing the lattice of subgroups of a finite group that had been calculated by a computer was implemented by H. Jürgensen in 1965 (see [FJ70]).

The design of XGAP was strongly influenced and in fact triggered by the QUOTPIC system of Derek Holt and Sarah Rees (see [HR91]) which allows to depict graphically knowledge about normal subgroups of a finitely presented group found by a variety of methods for the investigation of finitely presented groups. It seemed most desirable to allow to depict in a similar way the even wider variety of information on subgroups of groups that can be obtained by a system such as GAP.

Beginning 1993, Frank Celler developed the idea of an interface from GAP to graphic systems that allowed to actually write commands for graphical tasks in the GAP language and together with Susanne Keitemeier (see [Kei95]) wrote a first version of programs in XGAP for drawing diagrams representing posets of subgroups of finite and finitely presented groups. We most gratefully acknowledge the help of Sarah Rees in implementing the interactive lattice functions and in beta testing the GAP3 version of XGAP.

In 1998, Thomas Breuer, Frank Celler, Joachim Neubüser and Max Neunhöffer planned the new concepts for the GAP4 version. The implementation and portation to GAP4 was done mainly by Max Neunhöffer in 1998 and 1999. Michael Ringe added the link to his MeatAxe programs. We like to thank all those who have adapted the GAP library to the needs of the new XGAP, in particular Alexander Hulpke who has been extremely helpful with this task.

3 Installing XGAP

Installing XGAP should be easy once you have installed GAP itself. We assume here that you want to install XGAP in its standard location, which is in the "pkg" subdirectory of the main GAP4 installation.

3.1 Overview

You have to perform the following steps to install XGAP:

- Get the sources
- Unpack the sources with the *unzoo* utility
- Use the *configure* script to adjust everything to your specific system
- Compile the C part of XGAP
- Edit a certain startup script (if necessary) and install it in an executable location in your system

3.2 What you need to install XGAP

Being a graphical user interface to GAP, XGAP of course needs graphics. At the moment this means that you need the X window system in the Version 11 Release 5 or newer. So you **cannot** use XGAP on a Macintosh or a Microsoft Windows computer. On the other hand the type of Unix you use should not matter. Please contact Max Neunhöffer (email: `max.neunhoeffer@math.rwth-aachen.de`) or post to the gap-trouble mailing list, if you encounter problems with certain system configurations.

Because XGAP contains a C-part you need a C compiler.

3.3 Getting and unpacking the sources

You can download the sources from the same places as GAP. So the main FTP servers are:

```
ftp://ftp-gap.dcs.st-and.ac.uk/pub/gap/gap4/
ftp://ftp.math.rwth-aachen.de/pub/gap4/
ftp://ftp.ccs.neu.edu/pub/mirrors/ftp-gap.dcs.st-and.ac.uk/pub/gap/gap4/
ftp://pell.anu.edu.au/pub/algebra/gap4/
```

You need only one file with the name "xgap4r11.zoo" which is in the subdirectory for the share packages. When you installed GAP you used the utility *unzoo* to unpack the distribution. You will need this here again. See the GAP-manual for instructions on how to get and compile this. You now change your current directory to the `pkg` subdirectory of the location where you installed GAP (you typed an *unzoo*-command, then a new directory called "gap4" or something like that was created, this directory contains the `pkg` subdirectory). The standard location would be:

```
# cd /usr/local/lib/gap4/pkg
```

Now you extract the sources for the XGAP share package:

```
# unzoo -x xgap4r11.zoo
xgap/README     -- extracted as text
xgap/Makefile.in        -- extracted as text
xgap/configure  -- extracted as text
xgap/configure.in       -- extracted as text
xgap/init.g     -- extracted as text
xgap/cnf/Makefile       -- extracted as text
xgap/cnf/Makegap.in     -- extracted as text
xgap/cnf/Makegap.top    -- extracted as text
xgap/cnf/config.hin     -- extracted as text
xgap/cnf/configure.in   -- extracted as text
xgap/cnf/configure.out  -- extracted as text
xgap/doc/answers.tex    -- extracted as text
xgap/doc/diffgap3.tex   -- extracted as text
/bin/mkdir: cannot make directory ‘xgap’: File exists
...
```

Note that the warning is **not** serious.

The *unzoo* utility unpacks the files and stores them into the apropriate subdirectories. XGAP resides completely in the following subdirectory (assuming standard location):

```
/usr/local/lib/gap4/pkg/xgap
```

3.4 Configuring and Compiling the C part

You have to change your current working directory to the "xgap" subdirectory. You do this by

```
# cd xgap
```

if your current working directory is the one, where you used *unzoo*. There you invoke the *configure* script by:

```
# ./configure
creating cache ./config.cache
checking for make... make
checking host system type... i686-unknown-linux2.0.34
checking target system type... i686-unknown-linux2.0.34
checking build system type... i686-unknown-linux2.0.34
checking for gcc... gcc
checking whether the C compiler (gcc  ) works... yes
checking whether the C compiler (gcc  ) is a cross-compiler... no
checking whether we are using GNU C... yes
...
updating cache ./config.cache
creating ./config.status
creating Makefile
creating xgap.sh
```

... indicate omissions. This script tries to determine, which kind of operating system and libraries you have installed and configures the source accordingly. Normally this should produce some output but no error messages. The last step of the script produces some makefiles which are used to compile the code. You do this by typing

```
# make
if test ! -d bin;  then mkdir bin;  fi
if test ! -d bin/i686-unknown-linux2.0.34-gcc; \ # line broken for this manual!
                then mkdir bin/i686-unknown-linux2.0.34-gcc;  fi
cp cnf/configure.out bin/i686-unknown-linux2.0.34-gcc/configure
( cd bin/i686-unknown-linux2.0.34-gcc ; CC=gcc \ # line broken for this manual!
                ./configure --target=i686-unknown-linux2.0.34  )
creating cache ./config.cache
checking for gcc... gcc
checking whether the C compiler (gcc  ) works... yes
checking whether the C compiler (gcc  ) is a cross-compiler... no
checking whether we are using GNU C... yes
...
creating ./config.status
creating Makefile
creating config.h
make[1]: Entering directory \                   # line broken for this manual!
     '/usr/local/lib/gap4/pkg/xgap/bin/i686-unknown-linux2.0.34-gcc'
gcc -I. -g -O2 -I/usr/X11R6/include  -o xcmds.o -c ../../src.x11/xcmds.c
gcc -I. -g -O2 -I/usr/X11R6/include  -o utils.o -c ../../src.x11/utils.c
...
make[1]: Leaving directory \                    # line broken for this manual!
     '/usr/local/lib/gap4/pkg/xgap/bin/i686-unknown-linux2.0.34-gcc'
```

(a few lines were broken for typesetting purposes in this manual, the position is marked by a backslash)

Now all C sources are compiled and a binary executable is built. It is stored in a subdirectory of the "bin" subdirectory in your "xgap" directory. The name of this location has something to do with your installation. It could for example be

```
bin/i686-unknown-linux2.0.34-gcc/xgap
```

if you compile on a Linux system using the GNU-C-Compiler.

3.5 Installing the Startup Script

To make the startup of XGAP more convenient there is a startup script which contains also some configuration information like the position of your GAP installation. It is in the "xgap" directory and is called "xgap.sh". This file is automatically generated by the `configure` script and normally you should **not** have to change anything in it. Just copy it to some location that people have in their "PATH" environment variable, for example to "/usr/local/bin". This completes the installation.

If you want to change anything in the installation, you can also edit the script until the line

```
##  STOP EDITING HERE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

You can specify the directory where GAP is installed ("GAP_DIR"), the amount of memory that GAP should use as initial workspace ("GAP_MEM"), the name of the GAP-executable ("GAP_PRG") and the name of the XGAP-executable ("XGAP_PRG"). The first three are exactly the same things that you could edit in the main GAP startup script. After that you have the possibility to control the behaviour of the XGAP startup script. You can specify whether XGAP goes into the background ("DAEMON") and whether it prints out information about its parameters ("VERBOSE"). Note that it is possible to combine "DAEMON=YES" and "VERBOSE=YES" because the script actually runs in the foreground and only the C program is put into the background.

3.6 Installing in a different than the standard location

It could happen that you do not want to install XGAP in its standard location, perhaps because you do not want to bother your system administrator and have no access to the GAP directory. In this case you can unpack XGAP in any other location within a "pkg" directory with the *unzoo* command as described above. Let us call this directory "pkg" for the moment. You get an "xgap" subdirectory with all the files of XGAP in it. You follow the standard procedure with two exceptions:

Before you can configure and compile XGAP you need a symbolic link "cnf" which is in the directory where "pkg" is and points to the "cnf" directory in the main GAP directory. You can install this link directly after unzooing by (remember: you are in the "pkg" directory!):

```
ln -s /usr/local/lib/gap4/cnf ..
```

if "/usr/local/lib/gap4" is the location of the main GAP installation.

You can find out where the main GAP4 installation is by starting GAP as usual and looking at the variable `GAP_ROOT_PATHS` within GAP.

Note that you have to edit the startup script "xgap.sh" (see previous section) to adjust the paths for "XGAP_DIR" and "GAP_DIR", and possibly the name of the GAP executable "GAP_PRG". Enter your GAP installation directory for the variable "GAP_DIR" and the name of the directory that contains "pkg" for the variable "XGAP_DIR". The variable "GAP_PRG" has to contain the path to the GAP executable relative to the "bin" subdirectory of the main GAP installation. In most cases this value will already be correct. Note however that if GAP and XGAP are compiled on different machines, then it is possible that these directory names differ for the GAP and XGAP executables respectively.

The script will automatically launch GAP with two directories as library path such that all GAP and XGAP libraries will be found.

4 Subgroup Lattices - Examples

XGAP provides a graphical interface to the lattice or a partial lattice of subgroups of groups. For finitely presented groups it gives you easy access for example to the low index, prime quotient and Reidemeister-Schreier algorithms in order to build a partial lattice interactively. For other types of groups it provides easy access to many of the group functions (for example, the normalizer, normal subgroups, and Sylow subgroups).

This chapter explains how to use this interface by way of examples. Chapter 5 gives details about the various options and menus available. These two chapters will not describe how to write your own programs using the graphic extensions supplied by XGAP, see chapters 6 to 9 for details.

It is assumed that you have already started XGAP. On most systems you do this by typing

```
user@host:~> xgap
```

on the command line. Ask your system administrator if this does not work. This command will create a new window, the so called GAP window, in which GAP is awaiting your input. Depending on the window system and window manager you use, placing a new window on your screen might be done automatically or might require you to use the mouse to choose a position for the window and pressing the left mouse button to place the window.

The small arrow or cross you see on your screen is called a pointer. Although the device used to move this pointer can be anything, a mouse, a track ball, a glide-pad, or even something as exotic as a rat, we will use the term mouse to refer to this pointer device.

In case that some computation takes longer than expected, for instance the low index and the prime quotient can be quite time consuming, you can always interrupt a computation by making the GAP window active and pressing *CTRL-C* or selecting `Interrupt` in the `Run` menu. Again, making a window active is system and window manager dependent. In most cases you either have to move the pointer inside the GAP window or you have to click on the title bar of the GAP window.

Note that for each of the following examples there is a small GAP script in the `examples` subdirectory of the XGAP home directory which contains the necessary commands. However we consider it better for learning XGAP in a first time session if you type the commands by hand as suggested in the next few sections of this manual.

4.1 The Subgroup Lattice of the Dihedral Group of Order 8

This section gives you an example on how to use the function `GraphicSubgroupLattice` (see 5.1 for details), which will display the Hasse diagram of the subgroup lattice of a given group.

Using the dihedral group of size 8 as example the following will show you most features of the `GraphicSubgroupLattice` program. This exercise is best carried out in front of XGAP, trying the various commands yourself.

First you have to define a group in GAP, this example uses a dihedral group defined as polycyclic group.

```
gap> d8 := DihedralGroup(8);
<pc group of size 8 with 3 generators>
gap> SetName(d8,"d8");
```

Now you ask for a graphical display by

```
gap> s := GraphicSubgroupLattice(d8);
<graphic subgroup lattice "GraphicSubgroupLattice of d8">
```

XGAP will open a window containing a new graphic sheet, a menu bar (menus are described below) above the graphic sheet and a title. On most systems the title will be either below the graphic sheet or above the menu bar. The dimension of the graphic sheet is fixed, changing the size of the window will **not** change the size of the graphic sheet, see 5.8 how to resize the graphic sheet. It is possible that the graphic sheet is larger (depending on the lattice it might be much larger) than the window. In this case the window will contain so called scrollbars which allow you to select the portion of the graphic sheet which will be displayed.

XGAP first shows only the whole group (which is already selected) and the trivial subgroup, connected by a line indicating inclusion.

`ConjugacyClassesSubgroups` computes and returns the conjugacy classes of subgroups, so summing up the sizes of the classes tells you how many elements the lattice has.

```
gap> Sum( List( ConjugacyClassesSubgroups(d8), Size ) );
10
```

10 is small enough to use `AllSubgroups` without painting the screen black. After you have clicked this menu entry in the `Subgroups` menu you see the complete Hasse diagram in the graphic sheet.

The following initial remarks can be made about the graphical representation of the subgroup lattice:

- The vertex representing the trivial subgroup is labeled 1.
- Vertices representing subgroups of the same size are drawn at the same height. They are said to be "on the same level". In our example the subgroups that belong to the vertices 2, 4, 5, 8 and 9 all have size 2, and the subgroups of 3, 6, and 7 have size 4. The default behaviour is to place a vertex above another one if the size of the subgroup represented by the first vertex is larger than the size of the subgroup of the second, but see 5.1 for details. At the right edge of the graphic sheet each level is labeled with the index of the subgroups contained. See 5.3 for details on the labelling of levels.
- Vertices belonging to the same conjugacy class are placed closely together. In our example the subgroups of 4 and 5 form one conjugacy class.

The initial placement of the vertices chosen by `GraphicSubgroupLattice` might not be optimal or you might want to choose a different one in order to exploit certain features of the diagram. It is therefore possible to move the vertices around using the mouse.

The mouse together with the left mouse button can be used to move and select vertices. A selected vertex is represented by a thicker circle, colored red if your screen supports color. For example, in order to **move** vertex 4 use the mouse to place the pointer inside the circle around 4 and press the **left** mouse button. Keep the mouse button pressed and start moving the mouse. The vertex will now follow the pointer. Because of the height restrictions given by the size it is not possible to move 4 above 6 or below 1. It must always stay within its level. If you release the left mouse button vertex 4 will stay at its current position and the rest of the conjugacy class (in this example 5) will be moved to this new position.

In order to **select** vertex G place the pointer inside the diamond around G, press the **left** mouse button and release it immediately. Do not move the mouse while you hold down the left mouse button. Vertex G now has a slightly thicker boundary and is red if you have a color screen. There are two different ways to select more than one vertex, see 4.2 or 5.5.

On the top of the window, above the graphic sheet, you can see a list of menu names: `Sheet`, `Poset`, and `Subgroups`. In order to open any of these **pull down menus** place the pointer inside the button containing the menu name and press the left mouse button. Keep the button pressed. A pull down menu will be shown and by moving the pointer down you can choose a menu entry. By choosing an entry and then releasing the mouse button the entry is selected, the corresponding function is executed and the pull down menu is closed. If you release the mouse button while the pointer is outside the pull down menu the menu is closed without selecting any entry. Note that this behaviour is different from that of some other graphical user interfaces such as for example Windows.

Now select `Change Labels` from the `Poset` menu. If this entry is not available you have failed to select vertex G. After selecting `Change Labels` a small dialog box is opened asking for a label. Type in `D8` and press the *return* key or click on `OK`. The label of vertex G will now be changed to "$D8$". Note that in the X Window System you have to move the pointer on the text field if you want to edit the label.

In order to find out which vertex represents the centre of D_8, first select vertex $D8$ and then the menu entry `Centres` from the `Subgroups` menu. In case of a color screen, vertex $D8$ will be selected and colored red, and vertex 2 will be colored green. The color green indicates that vertex 2 is the result of a computation. There will also be a message in the GAP window saying that vertex 2 represents the centre of the group belonging to vertex $D8$.

```
#I  Centres (D8) --> (2)
```

Most of the menu entries in `Subgroups` should be self-explanatory, for details and the difference between `Closure` and `Closures` see 5.9.

If you have selected some vertices (in the example $D8$ is now selected), and you want to investigate the subgroups corresponding to these vertices further in GAP, the function `SelectedGroups` will return a list of these subgroups (note that you can also achieve calling this function by selecting `SelectedGroups to GAP` in the `Subgroups` menu):

```
gap> SelectedGroups(s);
[ d8 ]
```

On the other hand, the functions supplied via the `Subgroups` menu are by far not all functions applicable to groups. In order to show results of a computation in GAP in the diagram, you can use `SelectGroups`. The function `SelectGroups` allows you to mark any set of subgroups of D_8 in the diagram.

For instance, you can compute the lower central series of this (nilpotent) group in GAP.

```
gap> l := LowerCentralSeries(d8);
[ d8, Group([ f3 ]), Group([ <identity> of ... ]) ]
gap> SelectGroups(s,l);
```

This lower central series corresponds to the vertices $D8$, 2 and 1 which will now be selected. If, as it is not the case in this example, the subgroups are not yet depicted in the lattice, a warning appears in the GAP command window. You have to use `InsertVertex` to insert a new vertex into the lattice (note that you can also achieve this by selecting `InsertVertices from GAP` in the `Subgroups` menu, see section 5.6.1 for the complete description of this function or section 4.2 for an example).

To summarize the above: the function `SelectedGroups` can be used to transfer information from the diagram to GAP, the functions `SelectGroups` and `InsertVertex` can be used to transfer information from GAP to the diagram.

In order to finish this example, close the window by selecting `close graphic sheet` from the `Sheet` menu. This will close the window containing the Hasse diagram of D_8.

In this example you have learned, how to display the Hasse diagram of the subgroup lattice of a group using `GraphicSubgroupLattice`, how to use the mouse to move and select vertices, how to select a menu entry and how to transfer information between the Hasse diagram and GAP using `SelectGroups` and `SelectedGroups`.

In order to learn more about the menus `Sheet` and `Poset`, which were only mentioned very briefly, see 5.7, and 5.8.

4.2 A Partial Subgroup Lattice of the Symmetric Group on 6 Points

This section investigates the subgroup lattice of S_6.

```
gap> s6 := SymmetricGroup(6);
Sym( [ 1 .. 6 ] )
gap> SetName(s6,"S6");
gap> cc := ConjugacyClassesSubgroups(s6);;
gap> Sum(List(cc,Size));
1455
```

As there are 1455 subgroups, displaying the whole lattice of subgroups would not be helpful because there are simply too many. Therefore this example builds only a partial subgroup lattice. We assume that you are familiar with the general ideas, mouse actions and menus, which were discussed in 4.1.

We again start to build a partial lattice, by using `GraphicSubgroupLattice` (see 5.1). After you have entered

```
gap> s := GraphicSubgroupLattice(s6);
<graphic subgroup lattice "GraphicSubgroupLattice of S6">
```

XGAP will open a window containing a new graphic sheet with two connected vertices labeled 1 and G. Vertex 1 represents the trivial subgroup and vertex G the group S_6. Vertex G is already selected, so it will be red if your screen supports color.

XGAP can automatically write a protocol of all the subsequent actions you perform via mouse clicks. This is convenient because in comparison to normal GAP sessions you do not have the script of typed commands. You can activate this feature by selecting `Start Logging` from the `Subgroups` menu. XGAP prompts you for a filename via a file selector box. See 5.2 for details about this feature.

In order to find all subgroups of size 60, we cannot not use the `Subgroups` menu directly, so go back into the GAP window and extract the conjugacy classes of `cc` whose representatives have size 60.

```
gap> c60 := Filtered(cc,x->Size(Representative(x))=60);;
gap> s60 := List(c60,Representative);
[ Group([ (1,2)(3,4), (1,3,5) ]), Group([ (1,2)(3,4), (1,2,3)(4,5,6) ]) ]
```

We now use the function `InsertVertex` (see 5.6.1) to add these two subgroups to your partial lattice.

```
gap> for g in s60 do InsertVertex(s,g); od;
```

Note that we could have achieved this result with the entry `InsertVertices from GAP` in the `Subgroups` menu, see 5.6.1. The Hasse diagram now contains four vertices. The new vertices are **not** selected automatically. You can do this as mentioned above by clicking with the **left** mouse button on them. Selecting `Conjugate Subgroups` from the `Subgroups` menu adds the complete conjugacy classes. Please do this first for vertex 2 and then for vertex 3 such that the numbering of vertices in the following description is correct!

In order to find out what type of subgroups we are looking at, use another kind of menu not discussed so far, namely the "Information" menu. Place the pointer inside vertex 3, press the **right** mouse button and release it immediately. This will pop up a new window, containing some text describing vertex 3 (as mentioned above, depending on the window system and window manager, placing this window on the screen might require some interaction with the mouse).

```
Size        60
Index       12
IsAbelian   unknown
IsCentral   unknown
IsCyclic    unknown
IsNilpotent false
IsNormal    false
IsPerfect   true
IsSimple    unknown
IsSolvable  false
Isomorphism unknown
```

Note that GAP does not yet automatically draw the conclusion that a nonsolvable subgroup is also not abelian, cyclic or central. Place the pointer on top of the entry "Isomorphism" and press the **left** mouse button. After a while this entry is changed to

```
Isomorphism [ 60, 5 ]
```

telling you that the subgroup represented by vertex 3 is isomorphic to the alternating group on five symbols. The notation `[ 60, 5 ]` comes out of the small groups library and is the only information about the isomorphism type we can get from GAP4 up to now. Select *close* to close the "Information" menu. Repeat this with vertex 2, you will see that the subgroup of vertex 2 is also isomorphic to A_5, however these two A_5 inside S_6 are not conjugate in S_6. The "Information" menu is described in detail in 5.10.

Now we want to compute the normalizers of the elements of the conjugacy class containing the subgroup of vertex 3. You could either select vertex 3 and then *Normalizers* and repeat this process for the vertices 9 to 13, or you can first select the vertices 3, 9 to 13 and then select *Normalizers*. But how to select more than one vertex? If you first select 3 and then 9, vertex 3 will get deselected as soon as 9 gets selected. However, if you select vertex 3, place the pointer inside vertex 9, hold down the *SHIFT* key on your keyboard and then select vertex 9 using the left mouse button, vertex 9 will be selected in addition to vertex 3. Another method to select more than one vertex is to use the rubber band to catch vertices inside a rectangle. Place the pointer left and a bit higher than vertex 3 **outside** any other vertex. Press the **left** mouse button and hold it down. Now, using the mouse, move the pointer right and slightly below vertex 13. You see a rectangle, one corner at your start position and the other following the pointer. If vertices 3 and 9 to 13 are all inside this rectangle, release the mouse button. Now these vertices are selected. Select `Normalizers` from the `Subgroups` menu to compute and display the normalizers.

Now select vertex 3 and 4 and compute the intersection. The intersection is of size 10. Select this intersection and use `SelectedGroups` to get a GAP record describing the subgroup.

```
gap> l := SelectedGroups(s);
[ Group([ (2,3)(4,6), (1,2)(3,4) ]) ]
gap> u := l[1];
Group([ (2,3)(4,6), (1,2)(3,4) ])
```

In order to find out which subgroups of the complete lattice lie above the subgroup `u` you can use `Intermediate Subgroups`. You select the whole group in addition to `u` and choose `Intermediate Subgroups` in the `Subgroups` menu. You get 6 groups, some of them are already in the lattice, the others are added.

There is another feature we have not seen yet. Close the current graphic sheet and start again with a fresh one.

```
gap> Close(s);
gap> s := GraphicSubgroupLattice(s6);
<graphic subgroup lattice "GraphicSubgroupLattice of S6">
```

In order to compute a Sylow 2 subgroup select `Sylow Subgroup` from the `Subgroups` menu. A small dialog box will pop up asking for a prime, type in 2 and press *return* or click on `OK`. Now select this new vertex 2 representing the Sylow 2 subgroup and compute its normal subgroups. This is rather slow because the function checks for each new vertex if the corresponding subgroup is conjugate to an old one of the same size.

This is now the end of our partial investigation of the (partial) subgroup lattice of S_6, close the graphic sheet(s) using `close graphic sheet` of the `Sheet` menu. If you started the logging facility of XGAP as described above you now have a file (probably called `xgap.log` if you did not change the default) describing the actions we performed.

4.3 A Partial Subgroup Lattice of the Cavicchioli Group

This section investigates the following finitely presented group C_2, which was first investigated by Alberto Cavicchioli in [Cav86]:

$$\langle a, b \ ; \ aba^{-2}ba = b, (b^{-1}a^3b^{-1}a^{-3})^2 = a^{-1} \rangle.$$

In this example we will show a way to prove a finitely presented group to be infinite, and to find some big nonabelian factor groups of it.

The following GAP commands define C_2.

```
gap> f := FreeGroup( "a", "b" );  a := f.1;;  b := f.2;;
<free group on the generators [ a, b ]>
gap> c2 := f / [ a*b*a^-2*b*a/b, (b^-1*a^3*b^-1*a^-3)^2*a ];
<fp group on the generators [ a, b ]>
gap> SetName(c2,"c2");
```

We again assume that you are familiar with the general ideas, mouse actions and menus, which were discussed in 4.1 and 4.2.

In order to build a partial lattice of a finitely presented group, you again use the function `GraphicSubgroupLattice`. But if the first argument to `GraphicSubgroupLattice` is a finitely presented group the available menus are different from the example in the previous section. After you have entered

```
gap> s := GraphicSubgroupLattice(c2);
<graphic subgroup lattice "GraphicSubgroupLattice of c2">
```

XGAP will open a window containing a new graphic sheet. Compared to the interactive lattice of a permutation group as described in the previous section, there are the following differences:

– There is only one vertex instead of two. This vertex labeled G is the whole group C_2. There is no vertex for the trivial subgroup (yet).

– If you pull down the `Subgroups` menu, you will see that this menu is now very different. It gives you access to various algorithms for finitely presented groups but most of the entries from the last two examples are missing because most of the GAP functions behind these entries are not applicable to (infinite) finitely presented groups.

This example will show you how to prove that C_2 is infinite. First look at the abelian invariants in order to see what the commutator factor group is. In order to compute the abelian invariants pop up the "Information" menu. This is done in exactly the same manner as in the previous section. Place the pointer inside vertex

G, press the **right** mouse button and release it immediately. This "Information" menu is described in detail in 5.13.

```
Index              1
IsNormal           true
IsFpGroup          unknown
Abelian Invariants unknown
Coset Table        unknown
IsomorphismFpGroup unknown
Factor Group       unknown
```

This tells you what XGAP already knows about the group associated with vertex G. In order to compute the abelian invariants click onto this line. After a while this entry will change to

```
Abelian Invariants perfect
```

telling you that C_2 is perfect. So none of the `Subgroups` menu entries `Abelian Prime Quotient`, `All Overgroups`, `Conjugacy Class`, `Cores`, `Derived Subgroups`, `Intersection`, `Intersections`, `Normalizers` or `Prime Quotient` will compute any new subgroups.

In order to avoid accidents the menu entries `Abelian Prime Quotient`, `All Overgroups`, `Epimorphisms (GQuotients)`, `Conjugacy Class`, `Low Index Subgroups`, and `Prime Quotient` from the `Subgroups` menu are only selectable if exactly one vertex is selected because the functions behind these entries are in general quite time and space consuming.

Close the "Information" window and select `Low Index Subgroups` from the `Subgroups` menu. A small dialog box will pop up asking for a limit on the index. Type in 12 and press *return* or click on `OK`. In general it is hard to say what kind of index limit will still work, for some groups even 5 might be too much while for others 20 works fine, see also 45.9.1.

GAP computes 10 subgroups of index 11 and 8 subgroups of index 12. If you now start to check the abelian invariants of the index 12 subgroups you will find out that all subgroups represented by vertices 3 to 10 have a finite commutator factor group except the subgroup belonging to vertex 4 which has an infinite abelian quotient. Therefore the group C_2 itself is infinite.

Now we want to investigate C_2 a little further using GAP. Select vertices 3, 4, and 5 and switch to the GAP window. Use `SelectedGroups` to get the subgroups associated with these vertices.

```
gap> u := SelectedGroups( s );
[ Group([ a, b*a^2*b^-2, b*a*b^2*a^-1*b^-1*a^-1*b^-1, b^4*a^-2*b^-2,
      b^2*a^3*b^-1*a^-1*b^-2 ]),
  Group([ a, b^2*a*b^-1*a^-1*b^-1, b^3*a^-1*b^-1, b*a*b*a^3*b^-1 ]),
  Group([ a, b^2*a*b^-1*a^-1*b^-1, b*a^3*b^-2, b^4*a^-1*b^-3,
      b*a*b^3*a^-1*b^-1 ]) ]
```

`FactorCosetOperation` computes for each of these subgroups u_i the operation of C_2 on its cosets. It returns the result as a homomorphism of C_2 onto a permutation group. The operation on u_i is therefore a permutation representation of the factor group

$$C_2/Core(u_i)\cdot$$

Using `DisplayCompositionSeries` we can identify these factor groups.

```
gap> p := List( u, x -> FactorCosetOperation( c2, x ) );;
gap> l := List( p, Image );;
gap> for x  in l  do DisplayCompositionSeries(x);  Print("\n");  od;
G (2 gens, size 95040)
  M(12)
1 (0 gens, size 1)
```

```
G (2 gens, size 660)
  A(1,11) = L(2,11) ~ B(1,11) = O(3,11) ~ C(1,11) = S(2,11) ~ 2A(1,11) = U(2,11)
1 (0 gens, size 1)

G (2 gens, size 239500800)
  A(12)
1 (0 gens, size 1)
```

(This display can look a little different according to the GAP version you use.)

So C_2 contains the Mathieu group M_{12}, the alternating group on 12 symbols and $PSL(2,11)$ as factor groups. Therefore it would have been possible to find vertex 4 using `Epimorphisms (GQuotients)` instead of `Low Index Subgroups`. Close the graphic sheet by selecting the menu entry `close graphic sheet` from the `Sheet` menu and start with a fresh one.

```
gap> s := GraphicSubgroupLattice(c2);
<graphic subgroup lattice "GraphicSubgroupLattice of c2">
```

Select `Epimorphisms (GQuotients)` from the `Subgroups` menu. This pops up a menu similar to the "Information" menu (see 5.12).

```
Sym(n)
Alt(n)
PSL(d,q)
Library
User Defined
```

Select *PSL(d,q)*, which pops up a dialog box asking for a dimension. Enter `2` and click on *OK*. Then a second dialog box pops up asking for a field size. Enter `11` and click on *OK*. After a short time of computation the display in the `Epimorphisms (GQuotients)` menu changes and shows

```
PSL(2,11)      1 found
```

telling you, that GAP has found 1 epimorphism (up to inner automorphisms of $PSL(2,11)$) from C_2 onto $PSL(2,11)$. Click on *display point stabilizer* to create a new vertex representing a subgroup u such that the factor group of $C_2/Core(u)$ is isomorphic to $PSL(2,11)$. You could have clicked on *display* to create a new vertex representing the kernel of the epimorphism.

This is now the end of our partial investigation of the (partial) subgroup lattice of C_2, you have seen that C_2 is infinite and contains M_{12}, $Alt(12)$, and $PSL(2,11)$ as factor groups. Close the graphic sheet by selecting `close graphic sheet` from the `Sheet` menu.

4.4 A Partial Subgroup Lattice of the Trefoil Knot Group

This section investigates the following finitely presented group, the trefoil knot group K_3.

$$\langle a, b \ ; \ aba = bab \rangle$$

This examples shows some limitations of the methods available, in particular if infinite factors occur.

```
gap> f := FreeGroup( "a", "b" );
<free group on the generators [ a, b ]>
gap> k3 := f / [ f.1*f.2*f.1 / (f.2*f.1*f.2) ];
<fp group on the generators [ a, b ]>
gap> s := GraphicSubgroupLattice(k3);
<graphic subgroup lattice "GraphicSubgroupLattice">
```

If you compute the Abelian invariants of K_3 you will see that the commutator factor group is isomorphic to the infinite cyclic group. If you try to compute the derived subgroups it works! Just click on `Derived`

`Subgroups` in the `Subgroups` menu. A vertex appears in a level marked with `[ infinity, 1 ]`. However, there are not too many things you can do with such infinite index subgroups up to now, as we will illustrate below:

First produce some more subgroups by `Low Index Subgroups` (for example with index limit 5). If you now try to compare one of the new subgroups with the derived subgroup, this is possible. If you however try to calculate the intersection of one of the finite-index subgroups with the derived subgroups, GAP will run into an error:

```
Error the coset enumeration has defined more than 256000 cosets:
type 'return;' if you want to continue with a new limit of 512000 cosets,
type 'quit;' if you want to quit the coset enumeration,
type 'maxlimit := 0; return;' in order to continue without a limit,
...   (a few lines follow)
```

This can happen if the coset enumeration algorithm tries to enumerate the cosets of a subgroup with infinite index. This situation can also occur with other operations.

You can leave this break loop by entering the command `quit;` or by clicking `Leave Break Loop` in the `Run` menu of the main XGAP window.

Earlier you have computed the subgroups of index at most 5. There is one normal subgroup of index 2 belonging to vertex 6 and one of index 4 belonging to vertex 8. There is **no** line between those two vertices. Select both and click on `Compare Subgroups` in the `Subgroups` menu. A line appears and the line between vertices 8 and G vanishes. The reason for this is, that the `LowIndexSubgroupsFpGroup` call did not deliver the complete inclusion info. This can always happen for finitely presented groups in XGAP. In this case you have to compare the subgroups manually by `Compare Subgroups`. Note that this can mean large computations, especially if the indices are huge.

Now select vertex 10 and choose `Cores` from the `Subgroups` menu. You will get a new vertex 12 for an index 24 subgroup. Select the vertices 12 and G and choose `Intermediate Subgroups` from the `Subgroups` menu. You will get lots of new vertices. Note that some of them are duplicates of those which were already in the lattice. This is because comparison of subgroups can be quite expensive and is therefore **not** performed automatically in the case of finitely presented groups.

Select all vertices with a rubber band (click into the top left corner of the sheet, hold down the mouse and move the pointer to the lower right corner, then release the mouse button), and choose `Compare Subgroups` from the `Subgroups` menu. A few vertices will disappear and you get some messages in the GAP window about merging of vertices.

The display is also not fully correct with respect to conjugacy classes. `IntermediateSubgroups` does not return the complete information about conjugacy of subgroups. Because also conjugacy tests can be very expensive, they are also **not** performed automatically for finitely presented groups. Select `Test Conjugacy` from the `Subgroups` menu to trigger this test manually (note that all vertices are still selected!). The vertices belonging to conjugate subgroups are arranged together and if you move those containing the normal subgroup of index 24 above this one you recognize the subgroup lattice of the symmetric group on 4 points above that normal subgroup.

This is now the end of our partial investigation of the (partial) subgroup lattice of K_3, close the graphic sheet by selecting `close graphic sheet` from the `Sheet` menu.

4.5 A Partial Subgroup Lattice of a Finitely Presented Group

This section describes the investigation of the following finitely presented group:

$$G := \langle a, b \ ; \ a^6 = 1 \rangle$$

We will show especially how to deal with subgroups which have a very large index like those occuring in the prime quotient algorithm.

Define the group and open the subgroup lattice window:

```
gap> f := FreeGroup(2);
<free group on the generators [ f1, f2 ]>
gap> g := f/[f.1^6];
<fp group on the generators [ f1, f2 ]>
gap> s := GraphicSubgroupLattice(g);
<graphic subgroup lattice "GraphicSubgroupLattice">
```

First compute prime quotients by `Prime Quotient` in the `Subgroups` menu. You are asked for a prime and a class. Enter 2 and 7 respectively. You get lots of output in the GAP command window and seven new vertices. Some of the corresponding subgroups have huge indices. Note that these groups are only represented as kernels of epimorphisms within GAP. So explicit calculation of a coset table or a presentation could take very long or be absolutely impossible!

Now compute epimorphisms onto the symmetric group on 3 points by `Epimorphisms (GQuotients)` in the `Subgroups` menu, but use a polycyclic presentation as follows (the reason for this will be explained below):

```
gap> IdGroup(SymmetricGroup(3));
[ 6, 1 ]
gap> s3 := SmallGroup(6,1);
<pc group with 2 generators>
gap> IMAGE_GROUP := s3;;
```

This first determines the identification number of the symmetric group on 3 points within the small groups library, and then fetches this group as a polycyclic group. For groups of size less than 1000 this is often a good way to get a polycyclic presentation. Note that `SymmetricGroup(3)` leads to a permutation group. The last statement stores the group in a variable which can be used by XGAP.

Select vertex G, then click on `Epimorphisms (GQuotient)` in the `Subgroups` menu and select `User defined` in the window that pops up. This will always use the group stored in the global variable `IMAGE_GROUP`. GAP finds three epimorphisms. Display the three kernels by selecting *display* in the epimorphisms window.

Note that the new vertex 9 will be drawn on the line between vertices 2 and 3 because there is not yet a vertex in the level corresponding to index 6. You can move it aside by dragging it with the mouse to some better position within its level.

Now select vertices 8 and 11 and calculate the intersection of the two subgroups of indices 137438953472 and 6 respectively. GAP can calculate this intersection by calculating the subdirect product of the image groups of the epimorphisms (the index of the subgroup belonging to vertex 12 in G is 412316860416 which is three times of the index of the subgroup belonging to vertex 11). Note that this subdirect product can only be calculated because the two image groups are polycyclic groups. This is the reason why we needed S_3 as polycyclic group earlier.

This is now the end of our partial investigation of the (partial) subgroup lattice of G, close the graphic sheet by selecting `close graphic sheet` from the `Sheet` menu.

4.6 A Partial Subgroup Lattice of a Space Group

This section describes an investigation of an (infinite) space group provided by the catalogue CRYSTCAT of crystallographic groups and using the share package CRYST. Here you will see that subgroups of finite index and of finite size can occur at the same time in a graphic subgroup lattice of an infinite group. Note that you have to install these share packages to try this example.

Load the share packages, define the group and open the subgroup lattice window:

```
gap> RequirePackage("crystcat");;
gap> RequirePackage("cryst");;
gap> g := SpaceGroupBBNWZ(4,6,3,1,2);
SpaceGroupOnRightBBNWZ( 4, 6, 3, 1, 2 )
gap> s := GraphicSubgroupLattice(g);
<graphic subgroup lattice "GraphicSubgroupLattice of SpaceGroupOnRightBBNWZ( 4, \
6, 3, 1, 2 )">
```

This fetches the space group of dimension 4, associated crystal system number 6, $\mathbb{Q}$-class 3, $\mathbb{Z}$-class 1, and space group type 2 (see the CRYSTCAT documentation for an explanation of this).

Now we calculate some maximal subgroups with finite index and choose three of them:

```
gap> m := MaximalSubgroupClassReps(g,rec(latticeequal := true));;
gap> mm := m{[1..3]};
[ <matrix group with 7 generators>, <matrix group with 7 generators>,
  <matrix group with 7 generators> ]
```

Again refer to the CRYST share package documentation for an explanation of these commands. Insert this list of three subgroups into the lattice by selecting `InsertSubgroups from GAP` from the `Subgroups` menu.

Next calculate subgroups of infinite index but with finite size:

```
gap> w := WyckoffPositions(g);;
gap> ww := w{[1..3]};
[ < Wyckoff position, point group 11, translation := [ 0, 1/2, 0, 0 ],
    basis := [  ] >
    , < Wyckoff position, point group 11, translation := [ 0, 1/2, 0, 1/2 ],
    basis := [  ] >
    , < Wyckoff position, point group 11, translation := [ 0, 1/2, 1/2, 0 ],
    basis := [  ] >
     ]
gap> www := List(ww,WyckoffStabilizer);
[ <matrix group with 3 generators>, <matrix group with 3 generators>,
  <matrix group with 3 generators> ]
```

Insert these subgroups into the lattice by selecting `InsertSubgroups from GAP` from the `Subgroups` menu. They will be inserted in the level for groups of size 8.

Now you can compute the intersection of a subgroup with finite index and a subgroup with finite size, select for example vertex 2 and vertex 5 and choose `Intersection` from the `Subgroups` menu. You get a new vertex representing a subgroup of size 4.

If you now calculate the centralizers of the fifteen latticeequal maximal subgroups from above, you get among them four non-trivial cyclic subgroups:

```
gap> c := List(m,x->Centralizer(g,x));
[ <matrix group with 1 generators>, Group([  ]), Group([  ]), Group([  ]),
  <matrix group with 1 generators>, Group([  ]), Group([  ]),
  <matrix group with 1 generators>, Group([  ]),
  <matrix group with 1 generators>, Group([  ]), Group([  ]), Group([  ]),
  Group([  ]), Group([  ]) ]
```

Insert these into the graphic sheet by selecting `Insert Vertices from GAP` from the `Subgroups` menu. You will get four different new vertices representing groups with infinite index **and** infinite size. Each such vertex is placed into a level on its own, which is marked by `[ H1,` n`]` where n is replaced with subsequent natural numbers (see section 5.3 for details about levels). "H1" means Hirsch length 1, that is, each subnormal series of the group contains one and only one infinite cycle. In fact, since these are subgroups of space groups, it indicates that the translation subgroup is of dimension 1.

(Note that by calculating the point groups of these centralizers you can in fact see, that the infinite cyclic groups consist of translations only.)

Take two of these centralizers and calculate the closure by selecting `Closure` from the `Subgroups` menu. You will get a new subgroup of Hirsch length 2, which is also placed on a level of its own. Next select three of them, and calculate the closure. What do you observe? Also, select all four of them and calculate the closure. This time you get a subgroup of index 16, hence its level is marked by this finite index rather than a Hirsch number (which would be 4 here). Note that the finite index is used rather than the Hirsch length for this placement.

Finally, check, whether the centralizers are normal in the whole space group by clicking on the vertices with the right mouse button and choosing `IsNormal` in the "Information" window, which springs up. Now form the closures of each of them with each of the size 8 point stabilizers. You will get some other subgroups of Hirsch length 1. Both the centralizers and the point stabilizers are abelian. Is this also true for the closures?

This is now the end of our partial investigation of the (partial) subgroup lattice of G, close the graphic sheet by selecting `close graphic sheet` from the `Sheet` menu.

5 Subgroup Lattices - Systematic Description

In this chapter we give details about the various options and menus available in a systematic way.

5.1 GraphicSubgroupLattice

1 ▶ `GraphicSubgroupLattice(` *g* `)`

`GraphicSubgroupLattice` creates a new graphic sheet for the Hasse diagram of the subgroup lattice of *g*. The next sections describe how to select and move vertices and the following sections describe the available menus.

```
gap> GraphicSubgroupLattice( DihedralGroup(8) );
<graphic subgroup lattice "GraphicSubgroupLattice">
```

2 ▶ `GraphicSubgroupLattice(` *g*, *width*, *height* `)`

In this form `GraphicSubgroupLattice` creates a graphic sheet of initial dimensions *width* times *height*. However it is still possible to change these dimensions later using either the menu (described in 5.8) or the operation `Resize` for graphic sheets (see 6.1.14).

This function is the same for all types of groups. It only behaves differently according to some properties of the group. For example finitely presented groups are treated differently because other algorithms apply in this case and some of the standard ones are not feasible to be carried out.

In contrast to the GAP3 version of XGAP all graphic subgroup lattices are interactive so you can always remove vertices. For a finite group you can still get the full lattice by choosing `All Subgroups` for the whole group.

5.2 GraphicSubgroupLattice, Protocol of Group Theoretic Constructions

XGAP offers the possibility to write a protocol of the group theoretic constructions you perform via mouse clicks. This is convenient because otherwise you have no record of what you did. The normal GAP command script cannot supply this because the menu entries you select do not produce useful output there.

You start the protocol facility by selecting `Start logging` from the `Subgroups` menu. You are prompted for the file name of the protocol with a file selector box. After selecting `OK` in this box, all subsequent actions are logged in the protocol file. You can switch this feature off by selecting `Stop logging` from the `Subgroups` menu.

Note that when you select `SelectedGroups to GAP` from the `Subgroups` menu, the usual GAP logging is also directed to the XGAP log file. This is stopped when you select `InsertVertices from GAP` from the `Subgroups` menu. The idea of this is that you also see the GAP commands you issue within an XGAP session in the XGAP protocol file, if you temporarily work with the keyboard instead of the mouse. See 5.9.21 and 5.9.22 for a description of the menu entries.

5.3 GraphicSubgroupLattice, Labelling of Levels

We intend to represent subgroups of the same index at the same height of the graphic lattice. For this purpose we introduce "levels" as horizontal slices of a graphic subgroup lattice.

All vertices for subgroups of a certain finite index are placed in exactly one common level. But what about infinite indices?

Every level has a "level parameter". There are three types of levels: "finite index", "finite size", and "infinity". "finite index" means, that the index of the subgroups in this level is a certain, finite natural number, the level parameter is exactly this number. "finite size" means, that the size of the subgroups in this level is finite **and** the index is either not known (to GAP) or `infinity`. The level parameter is the size but with a negative sign. If the index is `infinity` and the size is not known, the third type applies: "infinity". In each "infinity" level only one vertex can exist.

This means that "finite index" takes precedence over "finite size" and "infinity", and "finite size" takes precedence over "infinity" if XGAP is in doubt which level to choose for a new vertex.

If the group in question is a space group provided by the CRYST share package, levels of type "infinity" are also labelled by the Hirsch length of the subgroup, which is the number of infinite cyclic factors in any given subnormal series. Note that this is only implemented for space groups as of this writing, although the Hirsch length is defined for a wider range of groups.

For every graphic subgroup lattice the levels are partially ordered by the following rules:

- A "finite index" level is greater than an "infinity" level.
- An "infinity" is greater than a "finite size" level.
- The "finite index" levels are totally ordered by descending indices.
- The "finite size" levels are totally ordered by ascending sizes.
- For space groups the "infinity" levels are partially ordered by the Hirsch lengths of the subgroups.

Note that in general different "infinity" levels are not comparable in this partial order!

On the screen, the levels are of course always totally ordered with respect to their height. XGAP ensures that this total ordering is always compatible with the abovementioned partial ordering. This means, that the user can permute "infinity" levels, as long as this does not violate the partial order by the Hirsch lengths and the principle, that a vertex "containing" another one is drawn higher on the screen.

Note that the level a vertex belongs to will be changed, when new information about the subgroup is available. This happens if the user chooses any entry in the information menu and new information is available thereafter.

5.4 GraphicSubgroupLattice, Moving Vertices

In order to move a vertex, place the pointer inside the vertex using the mouse, and press the **left** mouse button. Hold the mouse button pressed, and start moving the pointer by moving the mouse. The vertex will now follow the pointer, but it is not possible to move the vertex out of the boundaries of its level. or lower than a vertex of a group of smaller size. As soon as you release the left mouse button the vertex will stop following the pointer and, if the vertex was a member of a conjugacy class, the remaining elements of the class are moved.

If you hold down the *SHIFT* key before moving a vertex as described above then only the selected vertex is moved but not all members of the same class. You can realign the vertices of a class by selecting one of them and choose `Rearrange Classes` in the `Poset` menu (see 5.8.14).

5.5 GraphicSubgroupLattice, Selecting Vertices

Selected vertices are represented by a slightly thicker circle and, if your screen supports color, are colored red. There are five different ways to select or deselect a vertex or a bunch of vertices.

Place the pointer inside a vertex and press the **left** mouse button. Release the button immediately without moving the mouse. This will deselect all other vertices and select this vertex. If the vertex was already the only selected vertex it is deselected.

Place the pointer inside a vertex, hold down the *SHIFT* key, and press the **left** mouse button. Release the button immediately without moving the mouse, release the *SHIFT* key afterwards. If the vertex was deselected this action will select it in addition to any other selected vertices. If the vertex was already selected it will be deselected.

Place the pointer outside any vertex and press the **left** mouse button. Keep the button pressed and start moving the pointer. This will create a rubber band rectangle. One corner at the position where you pressed the mouse button, the opposite corner following the pointer. As soon as you release the left mouse button, all vertices inside the rectangle are selected, all vertices outside the rectangle are deselected.

Place the pointer outside any vertex, hold down the *SHIFT* key and press the **left** mouse button. Again you see a rubber band, however, now as soon as you release the mouse button, all vertices inside the rectangle are selected in addition to any other selected vertices.

To select vertices from the GAP command window call the function `SelectGroups` with a subgroup or a list of subgroups to select.

5.6 GraphicSubgroupLattice, Inserting Vertices

To insert new vertices into a graphic sheet one has to use the following operation:

1 ▶ `InsertVertex(` *sheet*`,` *grp*`,` *conj*`,` *hint* `)`
▶ `InsertVertex(` *sheet*`,` *grp* `)`

Inserts the group *grp* as a new vertex into the sheet. If the lattice is **not** the subgroup lattice of a finitely presented group, it checks, if the group is already in the lattice or if there is already a conjugate subgroup. Further, the new vertex is sorted into the poset. So `InsertVertex` checks for all vertices on higher levels, whether the new vertex is contained, and for all vertices on lower levels, whether they are contained in the new vertex. It then tries to add edges to the appropriate vertices. If the lattice is the lattice of a finitely presented group, nothing is done with respect to the connections of any vertex. `InsertVertex` returns a list with the new vertex as first entry and a flag as second, which says, whether this vertex was inserted right now or has already been there.

hint is a list of x coordinates which should give some hint for the choice of the new x coordinate. It can for example be the x coordinates of those groups which were parameter for the operation which calculated the group. *hints* can be empty but must always be a list!

If *conj* is a vertex we put the new vertex into the class of this vertex. Otherwise *conj* should either be `false` or `fail`.

You have access to the simpler form of the above functions via the `Subgroups` menu and the mouse. If you select `InsertVertices from GAP` in the `Subgroups` menu, the function `InsertVertex` is called for the group in the (automatic) variable `last`, or, if `last` contains a list of subgroups, for all of those subgroups. This is an easy way to insert results of calculations with GAP into the lattice. If `last` is neither a subgroup nor a list of subgroups, nothing happens.

5.7 GraphicSubgroupLattice, Sheet Menu

The `Sheet` menu will be pulled down if you place the pointer inside the `Sheet` button and press the left mouse button. Keep the button down and choose an entry by moving the pointer on top of this entry. Release the mouse button to select an entry.

1► `save as postscript`

Selecting this entry pops up a file selector. If you enter a file name and click on *OK*, this will save a description of the graphic sheet and graphic objects on the sheet as encapsulated postscript output. This output should be imported easily into other documents.

2► `close graphic sheet`

This entry will close the current graphic sheet and the window containing the sheet and all "Information" menus related to this sheet. On some window system or using some window manager the window itself has also a close button or menu entry. However, using the window close method will not close associated "Information" menus.

5.8 GraphicSubgroupLattice, Poset Menu

The `Poset` menu will be pulled down if you place the pointer inside the `Poset` button and press the left mouse button. Keep the button down and choose an entry by moving the pointer on top of this entry. Release the mouse button to select an entry.

This menu is a generic menu for any graphic poset and is not specific to subgroup lattices. So you find here options for the handling of general posets.

1► `Redraw`

The whole lattice will be redrawn. Use this option if some manipulation has disturbed the window.

2► `Show Levels`

If this menu entry is activated (after clicking it will have a small check sign at its right, clicking again deactivates it), there will be a little blue box under each level at the left edge of the graphic sheet. You can use these little boxes to change the height of a level by moving it up or down like a vertex. In cases where the order of the levels is not given by some "external source" like the index of the subgroups you can move levels by moving the corresponding blue box with pressed *SHIFT* button.

3► `Show Levelparameters`

This menu entry controls the display of the level parameters at the right edge of the graphic sheet. Normally these are displayed but you can switch it off with this menu entry.

4► `Delete Vertices`

Selecting the entry will delete all selected vertices. All edges from or to one of these are also deleted. However, inclusion information is preserved. This means that new edges are created from all vertices which were maximal in the deleted one to all vertices in which the deleted vertex was maximal. So you get the Hasse diagram of the poset which is the restriction of the former one to the not selected vertices.

5► `Delete Edge`

This menu entry is normally not selectable because it would destroy the Hasse diagram.

6► `Merge Classes`

This menu entry merges all classes within each level that contain a selected vertex. Afterwards `Rearrange-Classes` (see below) is performed. Use `Merge Classes` to unite classes within a level, but use it with care:

No further test is performed! If you unite classes of subgroups that are not conjugate, other errors may follow, because conjugacy tests are later on only performed by looking at the first subgroup in a given class!

7 ► `Magnify Lattice`

Selecting the entry will multiply the dimensions of the graphic sheet by the square root of 2 and enlarge the lattice accordingly.

8 ► `Shrink Lattice`

Selecting the entry will divide the dimensions of the graphic sheet by the square root of 2 and shrink the lattice accordingly.

9 ► `Resize Lattice`

Selecting this entry will pop up a dialog box asking for an x and y factor separated by a comma. You can enter integers or quotients. If you enter only one number this is used for x and y. The graphic sheet is then enlarged or shrinked and the lattice is resized accordingly.

10 ► `Resize Sheet`

This entry is similar to `Resize Lattice` except that only the graphic sheet is changed, the lattice remains unchanged. The numbers you enter must be integers and mean pixel numbers.

11 ► `Change Labels`

Selecting this entry will pop up a dialog box for each selected vertex asking for a new label. Clicking on *CANCEL* will cancel the relabelling of the remaining vertices but will not reset the already changed labels.

12 ► `Average Y Positions`

Selecting this entry will average the y coordinates of all vertices belonging to the same level. For graphic subgroup lattices this means all subgroups with the same index in the whole group.

13 ► `Average X Positions`

Selecting this entry will average the x coordinates of two or more selected vertices. This will only work if the corresponding subgroups do not have the same size and is used to align certain vertices vertically.

14 ► `Rearrange Classes`

Selecting this entry will clean up all classes which contain a selected vertex. You need this option if you have moved a vertex without its class (holding down the *SHIFT* key). The vertices in a class are arranged one next to the other horizontally without changing the order of the x coordinates. So you can permute the vertices within a class carelessly and then again get a nice picture with the new order by selecting this menu entry.

15 ► `Use Black&White`

Switches to black and white in case of a color screen or back to colors.

5.9 GraphicSubgroupLattice, Subgroups Menu

The `Subgroups` menu will be pulled down if you place the pointer inside the `Subgroups` button and press the left mouse button. Keep the button down and choose an entry by moving the pointer on top of this entry. Release the mouse button to select an entry.

Note that you can also get the `Subgroups` menu as a popup menu by clicking with the right mouse button into the graphic sheet of the subgroup lattice, but **not** on a vertex.

The result of a computation from any of the following entries is colored green, if your screen supports color. There will also be a short information message in the GAP window about the result.

In the following descriptions we use "vertices" as abbreviation for "subgroups associated with vertices".

The following descriptions do not apply to the case of finitely presented groups. See 5.12 for this case.

1 ► `All Subgroups`

For each selected vertex `All Subgroups` computes and displays all its subgroups. Requires at least one selected vertex. Use with care! This can cause huge computations! See also 37.19.1 in the GAP reference manual.

2 ► `Centralizers`

For each selected vertex `Centralizers` computes and displays its centralizer with respect to the whole group. Requires at least one selected vertex. See also 33.4.4 in the GAP reference manual.

3 ► `Centres`

For each selected vertex `Centres` computes and displays its centre. Requires at least one selected vertex. See also 33.4.5 in the GAP reference manual.

4 ► `Closure`

computes and displays the common closure of the selected vertices. Requires at least one selected vertex. See also 37.4.1 in the GAP reference manual.

5 ► `Closures`

computes and displays the closure of each pair of selected vertices. Requires at least two selected vertices. See also 37.4.1 in the GAP reference manual.

6 ► `Commutator Subgroups`

computes and displays the commutator subgroup of each pair of selected vertices. Requires at least two selected vertices. See also 37.11.2 in the GAP reference manual.

7 ► `Conjugate Subgroups`

computes and displays the conjugacy class (with respect to the whole group) of each selected vertex. Requires at least one selected vertex. See also 37.9.1 in the GAP reference manual.

8 ► `Cores`

For each selected vertex `Cores` computes and displays its core with respect to the whole group. Requires at least one selected vertex. See also 37.10.2 in the GAP reference manual.

9 ► `Derived Series`

For each selected vertex `Derived Series` computes and displays its derived series. Requires at least one selected vertex. See also 37.16.7 in the GAP reference manual.

10 ▶ `Derived Subgroups`

For each selected vertex `Derived Subgroups` computes and displays its derived subgroup. Requires at least one selected vertex. See also 37.11.3 in the GAP reference manual.

11 ▶ `Fitting Subgroups`

For each selected vertex `Fitting Subgroups` computes and displays its Fitting subgroup. Requires at least one selected vertex. See also 37.11.5 in the GAP reference manual.

12 ▶ `Intermediate Subgroups`

computes and displays all intermediate subgroups between two selected groups. Requires exactly two selected vertices. See also 37.16.18 in the GAP reference manual.

13 ▶ `Intersection`

computes and displays the common intersection of the selected vertices. Requires at least one selected vertex. See also 28.4.2 in the GAP reference manual.

14 ▶ `Intersections`

For each pair of selected vertices `Intersections` computes and displays the intersection of the two vertices. Requires at least two selected vertices. See also 28.4.2 in the GAP reference manual.

15 ▶ `Normalizers`

For each selected vertex `Normalizers` computes and displays its normalizer with respect to the whole group. Requires at least one selected vertex. See also 37.10.1 in the GAP reference manual.

16 ▶ `Normal Closures`

For each selected vertex `Normal Closure` computes and displays its normal closure with respect to the whole group. Requires at least one selected vertex. See also 37.10.4 in the GAP reference manual.

17 ▶ `Normal Subgroups`

For each selected vertex `Normal Subgroups` computes and displays the normal subgroups of the subgroup associated with this vertex. These new subgroups are not necessarily normal in the whole group. Requires at least one selected vertex. See also 37.18.7 in the GAP reference manual.

18 ▶ `Sylow Subgroups`

pops up a dialog box asking for a prime. After entering a prime p and pressing *return* or clicking *OK* it computes and displays a Sylow p-subgroup for each selected vertex. Requires at least one selected vertex. See also 37.12.1 in the GAP reference manual.

19 ▶ `SelectedGroups to GAP`

If the user selects this menu entry, the subgroups belonging to the selected vertices are put into a list which is stored into the variable `last`. This is equivalent to the statement `SelectedGroups(sheet);;` if `sheet` contains the graphic sheet object. If XGAP logging is on, then the normal GAP logging via `LogTo` is also directed to the XGAP log file.

20 ▶ `InsertVertices from GAP`

If the user selects this menu entry, the value of the variable `last` is used to insert new vertices into the graphic sheet. If `last` is equal to one subgroup, it is inserted via `InsertVertex`. If `last` is a list of subgroups, `InsertVertex` is called for all those subgroups. There is no error issued if one of the entries of `last` is no subgroup. If XGAP logging is on, then the normal GAP logging via `LogTo` is switched off! The idea of this is to switch the logging temporarily from XGAP logging to normal GAP logging between two clicks to `SelectedGroups to GAP` and `InsertVertices from GAP` respectively.

21 ▶ `Start Logging`

After clicking on this menu entry the user is prompted for a filename. From this point on all commands issued via mouse clicks in the subgroup menu are logged into that file, such that one can afterwards see "what happened" in the XGAP session. The information displayed is the same as in the info displays in the GAP window.

22 ▶ `Stop Logging`

A click onto this menu entry stops the XGAP logging.

These menu entries represent only a small selection of the functions of GAP which the authors of XGAP considered most frequently used. You can calculate other subgroups like for example prefrattini subgroups from the GAP command window. See sections 4.1 and 4.2 for examples how to transfer information from the graphical lattice of XGAP to GAP (via `SelectedGroups`, see 5.5) and vice versa (via `SelectGroups`, see 5.5, and `InsertVertex`, see 5.6).

5.10 GraphicSubgroupLattice, Information Menu

Note that this section does not deal with the case of a finitely presented group. See 5.13 for this case.

Placing the pointer inside a vertex (selected or not) and pressing the **right** mouse button pops up the "Information" menu. Clicking on any of the text lines will compute the corresponding property of the subgroup u associated with this vertex. Clicking on `all` will compute all properties, clicking on `close` will close the "Information" menu.

1 ▶ `Size`

computes and displays the size of u. See also 28.3.6 in the GAP reference manual.

2 ▶ `Index`

computes and displays the index of u in the whole group. See also 37.3.3 in the GAP reference manual.

3 ▶ `IsAbelian`

4 ▶ `IsCyclic`

5 ▶ `IsNilpotent`

6 ▶ `IsPerfect`

7 ▶ `IsSimple`

8 ▶ `IsSolvable`

computes and displays the corresponding property of u. See also 33.4.9, 37.14.1, 37.14.3, 37.14.5, 37.14.10, and 37.14.6 in the GAP reference manual.

9 ▶ `IsCentral`

10 ▶ `IsNormal`

computes and displays the corresponding property of u with respect to the whole group. See also 33.4.8 and 37.3.6 in the GAP reference manual.

11 ▶ `Isomorphism`

computes and displays the isomorphism type of u. This will only work if the size of u is small. See 48.7.5 in the GAP reference manual for details.

Note that the exact result of all these information displays is stored in the global variable `LastResultOfInfoDisplay` after each operation. So you can access this easily from the GAP command prompt. It is also returned as `last` value.

5.11 Vertex Shapes

The following vertex shapes can appear in an interactive lattice:

1► `circle`

Subgroup is not normal in whole group.

2► `diamond`

Subgroup is normal in whole group.

3► `rectangle`

Subgroup has an index that is too big for automatic calculation of normality. So it is not yet known whether this group is normal.

Automatic calculation is controlled by the following variable:

4► `GGLLimitForIsNormalCalc` V

Only for subgroups with index smaller than this number an automatic `IsNormal` test is performed when the vertex is added to the sheet.

5.12 GraphicSubgroupLattice for FpGroups, Subgroups Menu

The `Subgroups` menu will be pulled down if you place the pointer inside the `Subgroups` button and press the left mouse button. Keep the button down and choose an entry by moving the pointer on top of this entry. Release the mouse button to select an entry.

Note that you can also get the `Subgroups` menu as a popup menu by clicking with the right mouse button into the graphic sheet of the subgroup lattice, but **not** on a vertex.

The result of a computation from any of the following entries is colored green, if your screen supports color. In most cases there will also be short information message in the GAP window about the result.

Note that some of the menu entries make it necessary to compute presentations of subgroups using a modified Todd-Coxeter algorithm. This can be very time consuming and in some cases even impossible, if the index is too high.

In the following descriptions, we use "vertices" as abbreviation for "subgroups associated with vertices".

1► `Abelian Prime Quotient`

pops up a dialog box asking for a prime p. It then computes and displays the largest elementary abelian p quotient of the selected vertex. If no presentation for the subgroup associated to the vertex is known a presentation is first computed using a modified Todd-Coxeter algorithm. It then calls `PrimeQuotient` to compute the largest elementary abelian quotient. `Abelian PrimeQuotient` requires exactly one selected vertex.

2► `All Overgroups`

computes and displays all overgroups of the selected vertex. It first computes the permutation action of the whole group on the cosets of the subgroup associated with the selected vertex and then searches for all block systems. If the subgroup of the selected vertex is normal, then everything is calculated within the (finite) factor group in a better representation. `All Overgroups` requires exactly one selected vertex.

3► `Closure`

computes and displays the common closure of the selected vertices. Requires at least one selected vertex. See also 37.4.1 in the GAP reference manual.

4 ► `Compare Subgroups`

A non-empty set of vertices must be selected to choose this menu entry. All subgroups belonging to these vertices are compared pairwise, and the inclusion information is displayed in the lattice. It may happen that two or more vertices are merged if GAP notices, that the subgroups are equal.

5 ► `Conjugacy Class`

computes and displays the conjugacy class of the selected vertex. `Conjugacy Class` requires exactly one selected vertex.

6 ► `Cores`

computes and displays the cores of the selected vertices. `Cores` requires at least one selected vertex.

7 ► `Derived Subgroups`

computes and displays the derived subgroups of the selected vertices. If applied to a proper subgroup of the whole group it will only display those derived subgroups whose index is finite. `Derived Subgroups` requires at least one selected vertex.

8 ► `Epimorphisms (GQuotients)`

pops up another menu. Requires exactly one selected vertex.

```
Sym(n)
Alt(n)
PSL(d,q)
Library
User Defined
```

Click on any of these entries to try to find a quotient isomorphic to the symmetric group (`Sym(n)`), the alternating group (`Alt(n)`), the projective special linear group (`PSL(d,q)`), a group in a library supplied with XGAP (this will pop up a file selector), or a user defined group stored in the variable `IMAGE_GROUP`. After supplying additional parameters, for example, the degree of the symmetric group or the dimension and field of *PSL* using dialog boxes, the corresponding entry will change, for example to something like

```
Sym(3)        3 found
```

After one or more quotients were found click *display* to display them.

Note that in XGAP4 in fact the kernel of the epimorphism is marked whereas in XGAP3 this was not the case, even though the XGAP3 manual stated this.

In fact in XGAP3 a stabilizer of a permutation action on an orbit was put into the lattice.

In case that the image of the epimorphism is a permutation group you can get this functionality by clicking on *display point stabilizer* instead of *display*.

9 ► `Intermediate Subgroups`

computes and displays all intermediate subgroups between two selected groups. Requires exactly two selected vertices. See also 37.16.18 in the GAP reference manual.

10 ► `Intersection`

computes and displays the common intersection of the selected vertices. Requires at least one selected vertex. See also 28.4.2 in the GAP reference manual.

11 ► `Intersections`

computes and displays the pairwise intersections of the selected vertices. `Intersections` requires at least two selected vertices.

12 ▶ `Low Index Subgroups`

pops up a dialog box asking for index limit k. It will then do a low index subgroup search for subgroups of index at most k of the selected vertex using `LowIndexSubgroupsFpGroup`. If no presentation for the subgroup associated to the vertex is known a presentation is first computed using a modified Todd-Coxeter algorithm. `Low Index Subgroups` requires exactly one selected vertex.

13 ▶ `Normalizers`

computes and displays the normalizers of the selected vertices. `Normalizers` requires at least one selected vertex.

14 ▶ `Prime Quotient`

pops up a dialog box asking for a prime p and another dialog box asking for a class c. It then computes and displays the largest p-quotient of class c of the selected vertex. If no presentation for the subgroup associated to the vertex is known a presentation is first computed using a modified Todd-Coxeter algorithm. It then calls `PrimeQuotient`. `Prime Quotient` requires exactly one selected vertex.

15 ▶ `Test Conjugacy`

walks through all levels and tests for all pairs of classes, that contain a selected vertex, whether the groups in the classes are conjugates. If so, the classes are merged. After these calculations `Rearrange Classes` is called. Note that conjugacy calculations can take lots of time for finitely presented groups!

16 ▶ `SelectedGroups to GAP`

If the user selects this menu entry, the subgroups belonging to the selected vertices are put into a list which is stored into the variable `last`. This is equivalent to the statement `SelectedGroups(sheet);;` if `sheet` contains the graphic sheet object. If XGAP logging is on, then the normal GAP logging via `LogTo` is also directed to the XGAP log file.

17 ▶ `InsertVertices from GAP`

If the user selects this menu entry, the value of the variable `last` is used to insert new vertices into the graphic sheet. If `last` is equal to one subgroup, it is inserted via `InsertVertex`. If `last` is a list of subgroups, `InsertVertex` is called for all those subgroups. There is no error issued if one of the entries of `last` is no subgroup. If XGAP logging is on, then the normal GAP logging via `LogTo` is switched off! The idea of this is to switch the logging temporarily from XGAP logging to normal GAP logging between two clicks to "SelectedGroups to GAP" and "InsertVertices from GAP" respectively.

18 ▶ `Start Logging`

After clicking on this menu entry the user is prompted for a filename. From this point on all commands issued via mouse clicks in the subgroup menu are logged into that file, such that one can afterwards see "what happened" in the XGAP session. The information displayed is the same as in the info displays in the GAP window.

19 ▶ `Stop Logging`

A click onto this menu entry stops the XGAP logging.

These menu entries represent only a small selection of the functions of GAP which the authors of XGAP considered most frequently used. You can calculate other subgroups from the GAP command window. See sections 4.1 and 4.2 for examples how to transfer information from the graphical lattice of XGAP to GAP (via `SelectedGroups`, see 5.5) and vice versa (via `SelectGroups`, see 5.5, and `InsertVertex`, see 5.6).

5.13 GraphicSubgroupLattice for FpGroups, Information Menu

Placing the pointer inside a vertex (selected or not) and pressing the **right** mouse button pops up the "Information" menu. Clicking on any of the text lines will compute the corresponding property of the subgroup u associated with this vertex. Clicking on `close` will close the "Information" menu.

1 ► `Index`

displays the index of u in the whole group.

2 ► `IsNormal`

checks if u is normal in the whole group.

3 ► `IsFpGroup`

checks if u is a finitely presented group. Note that a subgroup of a finitely presented group that is defined by a coset table or as kernel of an epimorphism is **not** automatically known to GAP as a finitely presented group. This means, that certain algorithms can not be applied. Use `IsomorphismFpGroup` (see 45.10.1 in the GAP reference manual) to calculate a finitely presented group and an isomorphism onto it, if some calculation does not work automatically.

4 ► `Abelian Invariants`

computes and displays the abelian invariants of u.

5 ► `Coset Table`

computes a coset table for u.

6 ► `IsomorphismFpGroup`

computes a finitely presented group that is isomorphic to u and displays the number of generators and relators of it.

7 ► `Factor Fp Group`

computes the factor group of the whole group by u, if u is normal.

Note that the exact result of all these information displays is stored in the global variable `LastResultOfInfoDisplay` after each operation. So you can access this easily from the GAP command prompt. It is also returned as `last` value.

6 Graphic Sheets - Basic graphic operations

This chapter describes how graphics are accessed in XGAP via the lowest library functions for graphic sheets. These functions are used in all other parts of XGAP and you normally only need to know them if you want to display other things than graphic posets and subgroup lattices.

6.1 Graphic Sheet Objects

To access any graphics in XGAP you first have to create a **graphic sheet** object. Such objects are linked internally to windows on the screen. You do **not** have to think about redrawing, resizing and other organizing stuff. The graphic sheet object is a GAP object in the category `IsGraphicSheet` and should be saved because it is needed later on for all graphic operations.

1 ► `GraphicSheet(` *title*`,` *width*`,` *height* `)` O

creates a graphic sheet with title *title* and dimension *width* by *height*. A graphic sheet is the basic tool to draw something, it is like a piece of paper on which you can put your graphic objects, and to which you can attach your menus. The coordinate $(0, 0)$ is the upper left corner, $(width - 1, height - 1)$ the lower right.

It is possible to change the default behaviour of a graphic sheet by installing methods (or sometimes called callbacks) for the following events. In order to avoid confusion with the GAP term "method" the term "callback" will be used in the following. For example, to install the function `MyLeftPBDownCallback` as callback for the left mouse button down event of a graphic sheet *sheet*, you have to call `InstallCallback` as follows.

```
gap> InstallCallback( sheet, "LeftPBDown", MyLeftPBDownCallback );
```

XGAP stores for each graphic sheet a list of callback keys and a list of callback functions for each key. That means that when a certain callback key is triggered for a graphic sheet then the corresponding list of callback functions is called one function after the other. The following keys have predefined meanings which are explained below: `Close`, `LeftPBDown`, `RightPBDown`, `ShiftLeftPBDown`, `ShiftRightPBDown`, `CtrlLeftPBDown`, `CtrlRightPBDown`. All of these keys are strings. You can install your own callback functions for new keys, however they will not be triggered automatically.

2 ► `Close(` *sheet* `)`

the function will be called as soon as the user selects "close graphic sheet", the installed function gets the graphic sheet *sheet* to close as argument.

3 ► `LeftPBDown(` *sheet*`,` x`,` y `)`

the function will be called as soon as the user presses the left mouse button inside the graphic sheet, the installed function gets the graphic sheet *sheet*, the x coordinate and y coordinate of the pointer as arguments.

4 ► `RightPBDown(` *sheet*`,` x`,` y `)`

same as `LeftPBDown` except that the user has pressed the right mouse button.

5 ► `ShiftLeftPBDown( sheet, x, y )`

same as `LeftPBDown` except that the user has pressed the left mouse button together with the *SHIFT* key on the keyboard.

6 ► `ShiftRightPBDown( sheet, x, y )`

same as `LeftPBDown` except that the user has pressed the right mouse button together with the *SHIFT* key on the keyboard.

7 ► `CtrlLeftPBDown( sheet, x, y )`

same as `LeftPBDown` except that the user has pressed the left mouse button together with the *CTRL* key on the keyboard.

8 ► `CtrlRightPBDown( sheet, x, y )`

same as `LeftPBDown` except that the user has pressed the right mouse button together with the *CTRL* key on the keyboard.

Here is the documentation for the operations to control the callback functions:

9 ► `InstallCallback( sheet, key, func )` O

Installs a new callback function for the sheet *sheet* for the key *key*. Note that the old functions for this key are **not** deleted.

10 ► `RemoveCallback( sheet, func, call )` O

Removes an old callback. Note that you have to specify not only the *key* but also explicitly the *func* which should be removed from the list!

11 ► `Callback( sheet, key, args )` O

Executes all callback functions of the sheet *sheet* that are stored under the key *func* with the argument list *args*.

Every graphic object in XGAP can be *alive* or not. This is controlled by the filter `IsAlive`. Being *alive* means that the object can be used for further operations. If for example the user closes a window by a mouse operation the corresponding graphic sheet object is no longer *alive*.

12 ► `IsAlive( gobj )` F

This filter controls if a graphic object is *alive*, meaning that it can be used for further graphic operations.

The following operations apply to graphic sheets:

13 ► `Close( sheet )` O

The graphic sheet *sheet* is closed which means that the corresponding window is closed and the sheet becomes *not alive*.

14 ► `Resize( sheet, width, height )` O

The *width* and *height* of the sheet *sheet* are changed. That does **not** automatically mean that the window size is changed. It may also happen that only the scrollbars are changed.

15 ► `WindowId( sheet )` A

Every graphic sheet has a unique number, its *window id*. This is mainly used internally.

16 ► `SetTitle( sheet, title )` O

Every graphic sheet has a title which appears somewhere on the window. It is initially set via the call to the constructor `GraphicSheet` and can be changed later with this operation.

17 ▶ `SaveAsPS(` *sheet*`,` *filename* `)` O

Saves the graphics in the sheet *sheet* as postscript into the file *filename*, which is overwritten, if it exists.

18 ▶ `FastUpdate(` *sheet*`,` *flag* `)` O

Switches the `UseFastUpdate` filter for the sheet *sheet* to the boolean value of *flag*. If this filter is set for a sheet, the screen is no longer updated completely if a graphic object is moved or deleted. You should call `FastUpdate(` *sheet*`, true )` before you start large rearrangements of the graphic objects and `FastUpdate(` *sheet*`, false )` at the end.

6.2 Graphic Objects in Sheets

All graphics within graphic sheets are so called graphic objects. They are GAP objects in the category `IsGraphicObject`. These objects are linked internally to the actual graphics within the window. You can modify these objects via certain operations which leads to the corresponding change of the real graphics on the screen. The types of graphic objects supported in XGAP are: boxes, circles, discs, diamonds, rectangles, lines, texts, vertices and connections. Vertices are compound objects consisting of a circle, rectangle oder diamond with a short text inside. They remember their connections to other vertices. That means that if for example the position of a vertex is changed, the line which makes the connection to some other vertex is also changed automatically. For every graphic object there is a constructor which has the same name as the graphic object (e.g. `Box` is the constructor for boxes).

1 ▶ `IsGraphicObject(` *gobj* `)` C

This is the category in which all graphic objects are.

Constructors:

2 ▶ `Box(` *sheet*`,` x`,` y`,` w`,` h `)` O
▶ `Box(` *sheet*`,` x`,` y`,` w`,` h`,` *defaults* `)` O

creates a new graphic object, namely a filled black box, in the graphic sheet *sheet* and returns a GAP record describing this object. The four corners of the box are (x, y), $(x + w, y)$, $(x + w, y + h)$, and $(x, y + h)$.

Note that the box is $w + 1$ pixel wide and $h + 1$ pixels high.

If a record *defaults* is given and contains a component `color` of value *color*, the function works like the first version of `Box`, except that the color of the box will be *color*. See 6.3 for how to select a *color*.

See 6.4 for a list of operations that apply to boxes.

Note that `Reshape` for boxes takes three parameters, namely the box object, the new width, and the new height of the box.

3 ▶ `Circle(` *sheet*`,` x`,` y`,` r `)` O
▶ `Circle(` *sheet*`,` x`,` y`,` r`,` *defaults* `)` O

creates a new graphic object, namely a black circle, in the graphic sheet *sheet* and returns a GAP record describing this object. The center of the circle is (x, y) and the radius is r.

If a record *defaults* is given and contains a component `color` of value *color*, the function works like the first version of `Circle`, except that the color of the circle will be *color*. See 6.3 for how to select a *color*. If the record contains a component `width` of value *width*, the line width of the circle is set accordingly.

See 6.4 for a list of operations that apply to circles.

Note that `Reshape` for circles takes two parameters, namely the circle object, and the new radius of the circle.

4 ▶ `Disc(` *sheet*`,` x`,` y`,` r `)` O
▶ `Disc(` *sheet*`,` x`,` y`,` r`,` *defaults* `)` O

creates a new graphic object, namely a disc (a black filled circle), in the graphic sheet *sheet* and returns a GAP record describing this object. The center of the disc is (x, y) and the radius is r.

If a record *defaults* is given and contains a component `color` of value *color*, the function works like the first version of `Disc`, except that the color of the disc will be *color*. See 6.3 for how to select a *color*.

See 6.4 for a list of operations that apply to discs.

Note that `Reshape` for discs takes two parameters, namely the disc object, and the new radius.

5 ▶ `Diamond(` *sheet*`,` x`,` y`,` w`,` h `)` O
▶ `Diamond(` *sheet*`,` x`,` y`,` w`,` h`,` *defaults* `)` O

creates a new graphic object, namely a black diamond, in the graphic sheet *sheet* and returns a GAP record describing this object. The left corner of the diamond is (x, y), the others are $(x + w, y - h)$, $(x + w, y + h)$, and $(x + 2w, y)$.

If a record *defaults* is given and contains a component `color` of value *color*, the function works like the first version of `Diamond`, except that the color of the diamond will be *color*. See 6.3 for how to select a *color*. If the record contains a component `width` with integer value *width*, the line width is set accordingly.

See 6.4 for a list of operations that apply to diamonds.

Note that `Reshape` for diamonds takes three parameters, namely the diamond object, and the new *width* and *height* values.

6 ▶ `Rectangle(` *sheet*`,` x`,` y`,` w`,` h `)` O
▶ `Rectangle(` *sheet*`,` x`,` y`,` w`,` h`,` *defaults* `)` O

creates a new graphic object, namely a black rectangle, in the graphic sheet *sheet* and returns a GAP record describing this object. The four corners of the box are (x, y), $(x + w, y)$, $(x + w, y + h)$, and $(x, y + h)$.

Note that the rectangle is $w + 1$ pixel wide and $h + 1$ pixels high.

If a record *defaults* is given and contains a component `color` of value *color*, the function works like the first version of `Rectangle`, except that the color of the rectangle will be *color*. See 6.3 for how to select a *color*. If the record contains a component `width` with integer value *width*, the line width is set accordingly.

See 6.4 for a list of operations that apply to rectangles.

Note that `Reshape` for rectangles takes three parameters, namely the rectangle object, and the new *width* and *height* values.

7 ▶ `Line(` *sheet*`,` x`,` y`,` w`,` h `)` O
▶ `Line(` *sheet*`,` x`,` y`,` w`,` h`,` *defaults* `)` O

creates a new graphic object, namely a black line, in the graphic sheet *sheet* and returns a GAP record describing this object. The line has the end points (x, y) and $(x + w, y + h)$.

If a record *defaults* is given and contains a component `color` of value *color*, the function works like the first version of `Line`, except that the color of the line will be *color*. See 6.3 for how to select a *color*. If the record contains a component `width` with integer value *width*, the line width is set accordingly. If the record contains a component `label` with a string value *label*, a text object is attached as a label to the line.

See 6.4 for a list of operations that apply to lines.

Note that `Reshape` for lines takes three parameters, namely the line object, and the new w and h value. `Change` for lines in contrast takes five parameters, namely the line object and all four coordinates like in the original call.

8 ▶ `Text(` *sheet*`,` *font*`,` *x*`,` *y*`,` *str* `)` O
▶ `Text(` *sheet*`,` *font*`,` *x*`,` *y*`,` *str*`,` *defaults* `)` O

creates a new graphic object, namely the string *str* as a black text, in the graphic sheet *sheet* and returns a GAP record describing this object. The text has the baseline of the first character at (x, y).

If a record *defaults* is given and contains a component `color` of value *color*, the function works like the first version of `Text`, except that the color of the text will be *color*. See 6.3 for how to select a *color*.

See 6.4 for a list of operations that apply to texts.

Note that `Reshape` for texts takes two parameters, namely the text object, and the new font. Use `Relabel` to change the string of the text.

Operations for graphic objects:

9 ▶ `Connection(` *vertex*`,` *vertex* `)` O
▶ `Connection(` *vertex*`,` *vertex*`,` *defaults* `)` O

Connects two vertices with a line. The second variation can get a *defaults* record for the actual line. The same entries as in the *defaults* record for lines are allowed.

10 ▶ `Disconnect(` *vertex*`,` *vertex* `)` O

Deletes connection between two vertices.

11 ▶ `Draw(` *object* `)` O

This operation (re-)draws a graphic object on the screen. You normally do not need to call this yourself. But in some rare cases of object overlaps you could find it useful.

12 ▶ `Delete(` *sheet*`,` *object* `)` O
▶ `Delete(` *object* `)` O

Deletes a graphic object. Calls `Destroy` first, so the graphic object is no more *alive* afterwards. The object is deleted from the list of objects in its graphic sheet. There is no way to reactivate such an object afterwards.

13 ▶ `Destroy(` *object* `)` O

Destroys a graphic object. It disappears from the screen and will not be *alive* any more after this call. Note that *object* is **not** deleted from the list of objects in its graphic sheet *sheet*. This makes it possible to `Revive` it again. In order to delete *object* from *sheet*, use `Delete(` *sheet*`,` *obj* `)`, which calls `Destroy` for *obj*.

14 ▶ `Revive(` *object* `)` O

Note that *object* must be in the list of objects in its graphic sheet! So this is only possible for `Destroyed`, not for `Deleted` graphic objects.

15 ▶ `Move(` *object*`,` *x*`,` *y* `)` O

Changes the position of a graphic object absolutely. It must be *alive* and will be moved at once on the screen.

16 ▶ `MoveDelta(` *object*`,` *dx*`,` *dy* `)` O

Changes the position of a graphic object relatively. It must be *alive* and will be moved at once on the screen.

17 ▶ `PSString(` *object* `)` O

Creates a postscript string which describes the graphic object. Normally you do not need to call this because it is only used internally if the user exports the whole graphic sheet to encapsulated postscript.

18 ▶ `PrintInfo( object )` O

This operation prints debugging info about a graphic object.

19 ▶ `Recolor( object, col )` O

Changes the color of a graphic object. See 6.3 for how to select a *color*.

20 ▶ `Reshape( object, ... )` O

Changes the shape of a graphic object. The parameters depend on the type of the object. See the descriptions of the constructors for the actual usage.

21 ▶ `\in`

This infix operation needs a vector of two integers to its left and a graphic object to its right ("vector of two integers" means a list of two integers e.g. `[15,9]`). It determines, if the position given by the two integer coordinates is inside (e.g. for boxes) or on (e.g. for lines) the graphic objects. Returns a boolean value.

22 ▶ `Change( object, ... )` O

Changes the shape of a graphic object. The parameters depend on the type of the object. See the descriptions of the constructors for the actual usage.

23 ▶ `Relabel( object, str )` O

Changes the label of a graphic object. The second argument must always be a string.

24 ▶ `SetWidth( object, w )` O

Changes the line width of the graphic object. The line width w must be a relatively small integer.

25 ▶ `Highlight( vertex )` O
▶ `Highlight( vertex, flag )` O

In the first form this operation switches the highlighting status of a vertex to ON. In the second form the *flag* decides about ON or OFF. Highlighting normally means a thicker line width and a change in color.

6.3 Colors in XGAP

Depending on the type of display you are using, there may be more or fewer colors available. You should write your programs always such that they work even on monochrome displays. In XGAP these differences can be read off from the so called "color model". The global variable `COLORS` contains all available information.

1 ▶ `COLORS` V

The variable `COLORS` contains a list of available colors. If an entry is `false` this color is not available on your screen. Possible colors are: `"black"`, `"white"`, `"lightGrey"`, `"dimGrey"`, `"red"`, `"blue"`, and `"green"`.

The following example opens a new graphic sheet (see 6.1.1), puts a black box (see 6.2.2) onto it and changes its color. Obviously you need a color display for this example.

```
gap> sheet := GraphicSheet( "Nice Sheet", 300, 300 );
<graphic sheet "Nice Sheet">
gap> box := Box( sheet, 10, 10, 290, 290 );
<box>
gap> Recolor( box, COLORS.green );
gap> Recolor( box, COLORS.blue );
gap> Recolor( box, COLORS.red );
gap> Recolor( box, COLORS.lightGrey );
gap> Recolor( box, COLORS.dimGrey );
gap> Close(sheet);
```

The component `model` is always a string. It is `monochrome`, if the display does not support colors. It is `gray` if we only have gray shades and `colorX` if we have colors. The "X" can be either 3 or 5, depending on how many colors are available.

6.4 Operations for Graphic Objects

The following table gives an overview over the supported graphic objects and the functions which are applicable respectively:

Here are the supported graphic object types: `Box`, `Circle`, `Disc`, `Diamond`, `Rectangle`, `Line`, `Text`, `Vertex`.

These functions apply to all graphic object types: `Draw`, `Delete`, `Destroy`, `Revive`, `Move`, `MoveDelta`, `PSString`, `PrintInfo`, `ViewObj`, `Recolor`, `Reshape`, `\in`, `WindowId`

In addition, the operation `Relabel` applies to objects of types `Line`, `Text`, and `Vertex`; the operation `SetWidth` applies to objects of types `Diamond`, `Rectangle`, `Circle`, and `Line`. There is also `Change` for a `Line` and `Highlight` for a `Vertex`.

6.5 Global Information

There are some global data structures which can and should be consulted if certain information is needed. The first (about color handling) was already described in section 6.3. The second is for vertices:

1 ▶ `VERTEX` V

This globally bound record contains the following components:

`circle`
: integer value for the vertex type “circle”

`diamond`
: integer value for the vertex type “diamond”

`rectangle`
: integer value for the vertex type “rectangle”

`radius`
: radius in pixels of a vertex on the screen

`diameter`
: diameter in pixels of a vertex on the screen

The third structure is about the available fonts.

2 ▶ `FONTS` V

This globally bound record has the following components: `tiny`, `small`, `normal`, `large`, `huge` and `fonts`. The first 5 are itself records each for one available font. They have components `name` for the name of the font and `fontInfo`, which is a list of 3 integers. The first is the maximal size of a character above the baseline in pixels, the second is the maximal size of a character below the baseline in pixels, and the third is the width in pixels of **all** characters, because it is always assumed, that the fonts are non-proportional.

3 ▶ `FontInfo(` *font* `)` O

Returns the information about the font *font*. The result is a triple of integers. The first number is the maximal size of a character above the baseline in pixels, the second is the maximal size of a character below the baseline in pixels, and the third is the width in pixels of **all** characters, because it is always assumed, that the fonts are non-proportional. Use this function rather than accessing the component `fontInfo` of a font object directly!

There is another global structure:

4 ▶ `BUTTONS` V

This record contains the following components: `left` and `right` contain a number for the left resp. right mouse button. `shift` and `ctrl` contain codes for the respective keys on the keyboard.

You should always use these global data instead of hardwiring any integers into your code.

7 User Communication

XGAP has two main means to communicate with the user. The first is the normal command processing: The user types commands, they are transmitted to GAP, are executed, and produce output, which is displayed in the command window. The second is the mouse and the other parts of the graphical user interface. This latter part can be divided into menus, mouse events, dialogs, and popups.

Menus
: Most of the windows of XGAP have menus. The user can select entries in them and this information is transformed to a function call in GAP. Menu entries can be checked or not, so menus can also display information.

Mouse Events
: A mouse event is the pressing or releasing of a mouse button, together with the position where the mouse pointer is in the exact moment this happens and the state of certain keyboard keys (mainly shift and control). Such events also trigger GAP function calls and the corresponding functions can react on these events and for example wait for others.

Dialogs
: Dialogs are windows which display information and into which the user can enter information for example in form of text fields.

Popups
: Popups are special dialogs where the user can not type text but can only click on certain buttons. XGAP has so called "text selectors" which are a convenient construct to display textual information and let the user select parts of it.

Most of those graphical objects have corresponding GAP objects, which are created by constructors and can be used later on by operations.

7.1 Menus in Graphic Sheets

1 ▶ `Menu(` *sheet*`,` *title*`,` *ents*`,` *fncs* `)` O
▶ `Menu(` *sheet*`,` *title*`,` *zipped* `)` O

`Menu` returns a pulldown menu. It is attached to the sheet *sheet* under the title *title*. In the first form *ents* is a list of strings consisting of the names of the menu entries. *fncs* is a list of functions. They are called when the corresponding menu entry is selected by the user. The parameters they get are the graphic sheet as first parameter, the menu object as second, and the name of the selected entry as third parameter. In the second form the entry names and functions are all in one list *zipped* in alternating order, meaning first a menu entry, then the corresponding function and so on. Note that you can delete menus but it is not possible to modify them, once they are attached to the sheet. If a name of a menu entry begins with a minus sign or the list entry in *ents* is not bound, a dummy menu entry is generated, which can sort the menu entries within a menu in blocks. The corresponding function does not matter.

2 ▶ `Check(` *menu*`,` *entry*`,` *flag* `)` O

Modifies the "checked" state of a menu entry. This is visualized by a small check mark behind the menu entry. *menu* must be a menu object, *entry* the string exactly as in the definition of the menu, and *flag* a boolean value.

3 ▶ `Enable(` *menu*`,` *entry*`,` *flag* `)` O
▶ `Enable(` *menu*`,` *boollist* `)` O

Modifies the "enabled" state of a menu entries. Only enabled menu entries can be selected by the user. Disabled menu entries are visualized by grey or shaded letters in the menu. *menu* must be a menu object, *entry* the string exactly as in the definition of the menu, and *flag* a boolean value. *entry* can also be a natural number meaning the index of the corresponding menu entry. In the second form *boollist* must be a list where each entry has either a boolean value or the value `fail` The list must be as long as the number of menu entries in the menu *menu*. All menu entries where a boolean value is provided are enabled or disabled according to this value.

See the explanation of `GraphicSheet` (6.1.2) for the "Close" event, which occurs when the user selects the menu entry `close graphic sheet` in the `Sheet` menu.

7.2 Mouse Events

When a mouse event occurs, this is communicated to GAP via a function call which in turn triggers a callback. As described in 6.1.1 to 6.1.8 the following callback keys are predefined as reactions on mouse events: `LeftPBDown`, `RightPBDown`, `ShiftLeftPBDown`, `ShiftRightPBDown`, `CtrlLeftPBDown`, `CtrlRightPBDown`.

Note that when your function gets called, the mouse button will still be pressed. So it can react and for example wait for the release. There is an easy way to find out about the state of the mouse buttons after the event:

1 ▶ `WcQueryPointer(` *id* `)` F

id must be a `WindowId` of an XGAP sheet. This function returns a vector of four integers. The first two are the coordinates of the mouse pointer relative to the XGAP sheet. Values outside the window are represented by -1. The third element is a number where the pressed buttons are coded. If no mouse button is pressed, the value is zero. `BUTTONS.left` is added to the value, if the left mouse button is pressed and `BUTTONS.right` is added, if the right mouse button is pressed. The fourth value codes the state of the shift and control. Here the values `BUTTONS.shift` and `BUTTONS.ctrl` are used.

This function is used in

2 ▶ `Drag(` *sheet*`,` *x*`,` *y*`,` *bt*`,` *func* `)` O

Call this function when a button event has occurred, so the button *bt* is still pressed. It waits until the user releases the mouse button and calls *func* for every change of the mouse position with the new x and y position as two integer parameters. You can implement a dragging procedure in this way as in the following example: (we assume that a `LeftPBDown` event just occurred and x and y contain the current mouse pointer position):

```
storex := x;
storey := y;
box := Rectangle(sheet,x,y,0,0);
if Drag(sheet,x,y,BUTTONS.left,
        function(x,y)
          local bx,by,bw,bh;
          if x < storex then
            bx := x;
```

```
            bw := storex - x;
          else
            bx := storex;
            bw := x - storex;
          fi;
          if y < storey then
            by := y;
            bh := storey - y;
          else
            by := storey;
            bh := y - storey;
          fi;
          if bx <> box!.x or by <> box!.y then
            Move(box,bx,by);
          fi;
          if bw <> box!.w or bh <> box!.h then
            Reshape(box,bw,bh);
          fi;
        end) then
    the box had at one time at least a certain size
    ... work with box ...
  else
    the box was never big enough, we do nothing
  fi;
  Delete(box);
```

7.3 Dialogs

1 ▶ `Dialog(` *type*`,` *text* `)` O

creates a dialog box and returns a GAP object describing it. There are currently two types of dialogs: A file selector dialog (called `Filename`) and a dialog type called `OKcancel`. *text* is a text that appears as a title in the dialog box.

2 ▶ `Query(` *obj* `)` O
▶ `Query(` *obj*`,` *default* `)` O

Puts a dialog on screen. Returns `false` if the user clicks "Cancel" and a string value or filename, if the user clicks "OK", depending on the type of dialog. *default* is an optional initialization value for the string.

7.4 Popups

1 ▶ `PopupMenu(` *name*`,` *labels* `)` O

creates a new popup menu and returns a GAP object describing it. *name* is the title of the menu and *labels* is a list of strings for the entries. Use `Query` to actually put the popup on the screen.

2 ▶ `Query`

actually puts a popup on screen. `Query` returns the string of the selected entry or `false` if the user clicks outside the popup. See also `Query` for dialogs in 7.3.2.

3 ▶ `TextSelector(` *name*`,` *list*`,` *buttons* `)` O

creates a new text selector and returns a GAP object describing it. *name* is a title. *list* is an alternating list of strings and functions. The strings are displayed and can be selected by the user. If this happens the

corresponding function is called with two parameters. The first is the text selector object itself, the second the string that is selected. A selected string is highlighted and all other strings are reset at the same time. Use `Reset` to reset all entries.

buttons is an analogous list for the buttons that are displayed at the bottom of the text selector. The text selector is displayed immediately and stays on screen until it is closed (use the `Close` operation). Buttons can be enabled and disabled by the `Enable` operation and the string of a text can be changed via `Relabel`.

4 ▶ `Enable(` *sel*`,`*bt*`,`*flag* `)`
▶ `Enable(` *sel*`,`*btindex*`,`*flag* `)`

Enables or disables the button *bt* (string value) or *btindex* (integer index) of the text selector *sel*, according to *flag*.

5 ▶ `Relabel(` *sel*`,` *list* `)`
▶ `Relabel(` *sel*`,` *index*`,` *text* `)`

Changes the strings that are displayed in the text selector. In the first form *list* must be a list of strings. The second form only changes the text at index *index*.

6 ▶ `SetName(` *sel*`,` *index*`,` *string* `)`

Every string in a text selector has a name. The names are stored in the list `names` component of the text selector. So `sel!.names` ist a list containing the names. The names are initialized with the strings from the creation of the text selector.

7 ▶ `Reset(` *sel* `)`

Resets all strings of a text selector, such that they are no longer selected.

8 ▶ `Close(` *sel* `)`

Closes a text selector. It vanishes from screen.

Note that you have access to the indices and names of strings and buttons:

9 ▶ `IndexOfSelectedText`

Whenever the user clicks on a text in a text selector, the global variable is set to the index of the text in the text selector.

10 ▶ `IndexOfSelectedButton`

Whenever the user clicks on a button in a text selector, the global variable is set to the index of the button in the text selector.

8 Graphic Posets

This chapter describes the part of XGAP that allows the user to conveniently display posets graphically.

8.1 Introduction

A poset is just a partially ordered set. To display posets reasonably in a generic way we need additional structure. So for XGAP a poset comes in so called levels. At all times in the life of a graphic poset there are only finitely many levels and they are totally ordered, that is for two levels we can always say, which one is "higher". The position within the graphic sheet reflects this ordering.

The levels are parametrized by "level parameters", which can be any GAP object but must be unique within a graphic poset. A level is always accessed by its level parameter and **not** by its number!

The vertices in each level are grouped into classes. For example for graphic subgroup lattices vertices in the same class correspond to conjugate subgroups, vertices in the same level have the same size or index in the whole group. The classes within each level are parametrized by "class parameters", which can be any GAP object but must be unique within a level. A class within a level is always accessed by its class parameter and **not** by its number!

The user must supply a **partial order** for all of his levels. The mechanism to achieve this is the operation `CompareLevels`, which compares two level parameters. The current **total order** of the levels is always a refinement of the partial order. The user can permute levels, if that does not contradict the partial order defined by `CompareLevels`.

A vertex in the poset that is "contained in" another vertex in the poset order (we speak of "inclusion" like in the case of subgroup lattices) must always be in a level that is lower on the screen, because there is only a connecting line representing the inclusion. This is achieved by the fact, that inclusions of vertices are communicated to XGAP just by creating an "edge" between them. This means, that the vertex in the "lower" level lies in the vertex in the "higher" level. There must not be edges between vertices in the same level!

The terminology "vertices" and "edges" comes from the fact, that a graphic poset is just a special case of a graphic graph, where vertices can be placed anywhere in the sheet and edges have nothing to do with inclusion. It is planned that also a graphic graph library is implemented in XGAP but it is not yet operational. However everything which could be done not only for posets but at the same time for graphs is implemented already within the poset package. This explains the usage of "graph" in many places where you would otherwise expect "poset".

What you have to do to use the graphic poset package is create a graphic poset (a special instance of a graphic sheet), create some levels and perhaps classes within them. Then you can create vertices and edges, to encode the ordering. Everything else is done by the library. See the next section for details about the available operations.

Note that we chose a functional approach for certain decision procedures. This means that for example if you create a vertex and do not specify a position, an operation (`ChoosePosition`) is called to determine the actual position. You can use the generic routines or install your own methods for all of those decisions. In this case you just set a new filter for your posets and overload the generic methods by special routines for objects with your new filter set. You can see this approach in the example in 8.3.

8.2 Operations

Constructors:

1 ▶ `GraphicPoset(` *name*`,` *width*`,` *height* `)` O

creates a new graphic poset which is a specialization of a graphic graph mainly because per definition a poset comes in "levels" or "layers". This leads to some algorithms that are more efficient than the general ones for graphs.

2 ▶ `CreateLevel(` *poset*`,` *levelparam* `)` O
▶ `CreateLevel(` *poset*`,` *levelparam*`,` *lptext* `)` O

A level in a graphic poset can be thought of as a horizontal slice of the poset. It has a y coordinate of the top of the level relatively to the graphic sheet and a height. Every class of vertices in a graphic poset is in a level. The levels are totally ordered by their y coordinate. No two vertices which are included in each other are in the same level. A vertex containing another one is always "higher" on the screen, meaning in a "higher" level. Every level has a unique level parameter, which can be any GAP object. The user is responsible for all methods where a level parameter occurs as parameter and is not just an integer. There is NO GAP object representing a level which is visible for the user of posets. All communication about levels goes via the level parameter. `CreateLevel` creates a new level with level parameter *levelparam* in the graphic poset *poset*. It returns `fail` if there is already a level with a level parameter which is considered "equal" to *levelparam* by `CompareLevels` or *levelparam* if everything went well.

The second method allows to specify which text appears for the level at the right edge of the sheet.

3 ▶ `CreateClass(` *poset*`,` *levelparam*`,` *classparam* `)` O

A class in a graphic poset is a collection of vertices within a level which belong together in some sense. Every vertex in a graphic poset is in a class, which in turn belongs to a level. Every class in a level has a unique class parameter, which can be any GAP object. The user is responsible for all methods where a class parameter occurs as parameter and is not just an integer. There is NO GAP object representing a class which is visible to the user of posets. All communication about classes goes via the class parameter. `CreateClass` creates a new class in the level with level parameter *levelparam* in the graphic poset *poset*. It returns `fail` if there is no level with level parameter *levelparam* or there is already a class in this level with class parameter *classparam*. `CreateClass` returns *classparam* otherwise.

4 ▶ `Vertex(` *graph*`,` *data*`[,` *inf*`] )` O

Creates a new vertex. *inf* is a record in which additional info can be supplied for the new vertex. For general graphic graphs only the `label`, `color`, `shape`, `x` and `y` components are applicable, they contain a short label which will be attached to the vertex, the color, the shape (`circle`, `diamond`, or `rectangle`) and the coordinates relative to the graphic sheet respectively. For graphic posets also the components `levelparam` and `classparam` are evaluated. If the component `hints` is bound in *inf* it must be a list of x coordinates which will be delivered to `ChoosePosition` to help placement. Those x coordinates will be the coordinates of other vertices related to the new one. All values of record components which are not specified will be determined by calling some methods for graphic graphs or posets. Those are: `ChooseLabel` for the label, `ChooseColor` for the color, `ChooseShape` for the shape, `ChoosePosition` for the position, `ChooseLevel` for the level parameter, `ChooseClass` for the class parameter, and `ChooseWidth` for the line width of the vertex. `Vertex` returns `fail` if no vertex was created. This happens only, if one of the choose functions return `fail` or no possible value, for example a non-existing level or class parameter. `Vertex` returns a vertex object if everything went well.

5 ▶ `Edge(` *graph*`,` *vertex1*`,` *vertex2* `)` O
▶ `Edge(` *graph*`,` *vertex1*`,` *vertex2*`,` *defaults* `)` O

Adds a new edge from *vertex1* to *vertex2*. For posets this puts one of the vertices into the other as a maximal subvertex. So either *vertex1* must lie in a "higher" level than *vertex2* or the other way round. There must be

no vertex "between" *vertex1* and *vertex2*. If the two vertices are in the same level or one is already indirectly included in the other `fail` is returned, otherwise `true`. That means, that in the case where one of the two vertices is already a maximal subobject of the other, then the method does nothing and returns `true`. The variation with a *defaults* record just hands this over to the lower levels, meaning that the line width and color are modified.

Destructors:

6 ▶ `Delete(` *poset*`,` *vertex1*`,` *vertex2* `)`
▶ `Delete(` *poset*`,` *vertex1*`)`
▶ `Delete(` *poset*`,` *levelparam*`,` *classparam* `)`

These three variants of the `Delete` operation delete an edge, a vertex and a class respectively.

7 ▶ `DeleteLevel(` *poset*`,` *levelparam* `)` O

The following method applies to a level. It returns `fail` if no level with level parameter *levelparam* is in the poset. Otherwise the level is deleted and all classes within it are also deleted! `DeleteLevel` returns `true` if the level is successfully deleted.

Operations to change a poset:

8 ▶ `ResizeLevel(` *poset*`,` *levelparam*`,` *height* `)` O

Changes the height of a level. The y coordinate can only be changed by permuting levels, see below. Attention: This can increase the size of the sheet! Returns `fail` if no level with level parameter *levelparam* exists and `true` otherwise.

9 ▶ `MoveLevel(` *poset*`,` *levelparam*`,` *position* `)` O

Moves a level to another position. *position* is an absolute index in the list of levels. The level with level parameter *levelparam* will be at the position *position* after the operation. This is only allowed if the new ordering is compatible with the partial order given by `CompareLevels` and if there is no connection of a vertex in the moving level with another level with which it is interchanged. So *levelparam* is compared with all level parameters between the old and the new position. If there is a contradiction, nothing happens and the method returns `fail`. If everything works the operation returns `true`.

10 ▶ `Relabel(` *graph*`,` *vertex*`,` *label* `)` O
▶ `Relabel(` *graph*`,` *vertex* `)` O
▶ `Relabel(` *poset*`,` *vertex1*`,` *vertex2*`,` *label* `)` O
▶ `Relabel(` *poset*`,` *vertex1*`,` *vertex2* `)` O

Changes the label of the vertex *vertex* or the edge between *vertex1* and *vertex2*. This must be a short string. In the method where no label is specified the new label is chosen functionally: the operation `ChooseLabel` is called. Returns `fail` if an error occurs and `true` otherwise. This operation already exists in XGAP for graphic objects.

11 ▶ `Move(` *graph*`,` *vertex*`,` *x*`,` *y* `)` O
▶ `Move(` *graph*`,` *vertex* `)` O

Moves vertex *vertex*. For posets coordinates are relative to the level of the vertex. *vertex* must be a vertex object in *graph*. If no coordinates are specified the operation `ChoosePosition` is called. `Move` returns `fail` if an error occurs and `true` otherwise. This operation already exists in XGAP for graphic objects.

12 ▶ `Reshape(` *graph*`,` *vertex* `)` O
▶ `Reshape(` *graph*`,` *vertex*`,` *shape* `)` O

Changes the shape of the vertex *vertex*. *vertex* must be a vertex object in the graph or poset *graph*. For the method where no shape is specified the new shape is chosen functionally: `ChooseShape` is called for the

corresponding data. `Reshape` returns `fail` if an error occurs and `true` otherwise. This operation already exists in XGAP for graphic objects.

13 ▶ `Recolor(` *graph*`,` *vertex* `)` O
▶ `Recolor(` *graph*`,` *vertex*`,` *color* `)` O
▶ `Recolor(` *poset*`,` *vertex1*`,` *vertex2*`,` *color* `)` O
▶ `Recolor(` *poset*`,` *vertex1*`,` *vertex2* `)` O

Changes the color of the vertex *vertex* or the edge between *vertex1* and *vertex2*. *vertex* must be a vertex object in *graph*. For the method where no color is specified the new color is chosen functionally: `ChooseColor` is called for the corresponding data. `Recolor` returns `fail` if an error occurs and `true` otherwise. This operation already exists in XGAP for graphic objects.

14 ▶ `SetWidth(` *graph*`,` *vertex1*`,` *vertex2*`,` *width* `)` O
▶ `SetWidth(` *graph*`,` *vertex1*`,` *vertex2* `)` O

Changes the line width of an edge. *vertex1* and *vertex2* must be vertices in the graph *graph*. For the method where no line width is specified the width is chosen functionally: `ChooseWidth` is called for the corresponding data pair. Returns `fail` if an error occurs and `true` otherwise. This operation already exists in XGAP for graphic objects.

15 ▶ `Highlight(` *graph*`,` *vertex* `)` O
▶ `Highlight(` *graph*`,` *vertex*`,` *flag* `)` O

Changes the highlighting status of the vertex *vertex*. *vertex* must be a vertex object in *graph*. For the method where no flag is specified the new status is chosen functionally: `ChooseHighlight` is called for the corresponding data. Returns `fail` if an error occurs and `true` otherwise. This operation already exists in XGAP for graphic objects.

16 ▶ `Select(` *graph*`,` *vertex*`,` *flag* `)` O
▶ `Select(` *graph*`,` *vertex* `)` O

Changes the selection state of the vertex *vertex*. *vertex* must be a vertex object in *graph*. The flag determines whether the vertex should be selected or deselected. This operation already exists in XGAP for graphic objects. The method without flags assumes `true`.

17 ▶ `DeselectAll(` *graph* `)` O

Deselects all vertices in the graph or poset *graph*.

18 ▶ `Selected(` *graph* `)` O

Returns a (shallow-)copy of the set of all selected vertices.

Operations for decisions:

19 ▶ `ChooseLabel(` *graph*`,` *data* `)` O
▶ `ChooseLabel(` *graph*`,` *data*`,` *data* `)` O

This operation is called during vertex or edge creation, if the caller didn't specify a label for the vertex or edge. It has to return a short string which will be attached to the vertex. If it returns `fail` the new vertex is not generated! The generic method just returns the empty string, so no label is generated. This method is also called in the `Relabel` method without label parameter.

20 ▶ `ChooseLevel(` *poset*`,` *data* `)` O

This operation is called during vertex creation, if the caller didn't specify a level to which the vertex belongs. It has to return a level parameter which exists in the poset. If it returns `fail` the new vertex is not generated!

21 ▶ `ChooseClass( poset, data, levelparam )` O

This operation is called during vertex creation, if the caller didn't specify a class to which the vertex belongs. It has to return a class parameter which exists in the poset in the level with parameter *levelparam*. If it returns `fail` the new vertex is not generated!

22 ▶ `ChooseColor( graph, data )` O
▶ `ChooseColor( graph, data1, data2 )` O

This operation is called during vertex or edge creation. It has to return a color. If it returns `fail` the new vertex is not generated! It is also called in the `Recolor` method without color parameter.

23 ▶ `ChooseHighlight( graph, data )` O

This operation is called during vertex creation. It has to return a flag which indicates, whether the vertex is highlighted or not. If it returns `fail` the new vertex is not generated! It is also called in the `Highlight` method without flag parameter.

24 ▶ `ChoosePosition( poset, data, levelparam, classparam, hints )` O
▶ `ChoosePosition( graph, data )` O

This operation is called during vertex creation. It has to return a list with two integers: the coordinates. For posets those are relative to the level the vertex resides in. If it returns `fail` the new vertex is not generated! The parameters *levelparam* and *classparam* are level and class parameters respectively.

25 ▶ `ChooseShape( graph, data )` O

This operation is called during vertex creation. It has to return a string out of the following list: `circle`, `diamond`, `rectangle`. If it returns `fail` the new vertex is not generated!

26 ▶ `ChooseWidth( graph, data )` O
▶ `ChooseWidth( graph, data1, data2 )` O

This operation is called during vertex or edge creation. It has to return a line width. If it returns `fail` the new vertex or edge is not generated! This is also called by the `SetWidth` operation without width parameter.

27 ▶ `CompareLevels( poset, levelparam1, levelparam2 )` O

Compare two level parameters. -1 means that the level with parameter *levelparam1* is "higher", 1 means that the one with parameter *levelparam2* is "higher", 0 means that they are equal. `fail` means that they are not comparable.

Operations to get information:

28 ▶ `WhichLevel( poset, y )` O

Determines the level in which position y is. `WhichLevel` returns the level parameter or `fail`.

29 ▶ `WhichClass( poset, x, y )` O

Determines a class with a vertex which contains the position (x, y). The first class found is taken. `WhichClass` returns a list with the level parameter as first and the class parameter as second element. `WhichClass` returns `fail` if no such class is found.

30 ▶ `WhichVertex( graph, x, y )` O
▶ `WhichVertex( graph, data )` O
▶ `WhichVertex( graph, data, func )` O

Determines a vertex which contains the position (x, y). `WhichVertex` returns a vertex. In the third form the function *func* must take two parameters *data* and the data entry of a vertex in question. It must return

`true` or `false`, according to the right vertex being found or not. The function can for example consider just one record component of data records. `WhichVertex` returns `fail` in case no vertex is found.

31 ▶ `WhichVertices(` *graph*, *x*, *y* `)` O
▶ `WhichVertices(` *graph*, *data* `)` O
▶ `WhichVertices(` *graph*, *data*, *func* `)` O

Determines the list of vertices which contain the position (x, y). `WhichVertices` returns a list. In the third form the function *func* must take two parameters *data* and the data entry of a vertex in question. It must return `true` or `false`, according to the vertex belonging into the result or not. The function can for example consider just one record component of data records. Returns the empty list in case no vertex is found.

32 ▶ `Levels(` *poset* `)` O

Returns the list of level parameters in descending order meaning highest to lowest.

33 ▶ `Classes(` *poset*, *levelparam* `)` O

Returns the list of class parameters in the level with parameter *levelparam*. `Classes` Returns `fail` if no level with parameter *levelparam* exists.

34 ▶ `Vertices(` *poset*, *levelparam*, *classparam* `)` O

Returns the list of vertices in the class with parameter *classparam* in the level with parameter *levelparam*. Returns `fail` if no level with parameter *levelparam* or no class with parameter *classparam* exists in the level.

35 ▶ `Maximals(` *poset*, *vertex* `)` O

Returns the list of maximal subvertices in *vertex*.

36 ▶ `MaximalIn(` *poset*, *vertex* `)` O

Returns the list of vertices, in which *vertex* is maximal.

37 ▶ `PositionLevel(` *poset*, *levelparam* `)` O

Returns the y position of the level relative to the graphic sheet and the height. Returns `fail` if no level with parameter *levelparam* exists.

Operations for user communication:

38 ▶ `Menu(` *graph*, *title*, *entrylist*, *typelist*, *functionslist* `)` O

This operation already exists in XGAP for graphic sheets. Builds a new menu with title *title* but with information about the type of the menu entry. This information describes the relation between the selection state of the vertices and the parameters supplied to the functions. It is stored in the list *typelist*, which consists of strings. The following types are supported:

`forany`
: always enabled, generic routines don't change anything

`forone`
: enabled iff exactly one vertex is selected

`fortwo`
: enabled iff exactly two vertices are selected

`forthree`
: enabled iff exactly three vertices are selected

`forsubset`
: enabled iff at least one vertex is selected

`foredge`
: enabled iff a connected pair of two vertices is selected

`formin2`
: enabled iff at least two vertices are selected

`formin3`
: enabled iff at least three vertices are selected

entrylist and *functionslist* are like in the original operation for graphic sheets. The `IsMenu` object is returned. It is also stored in the sheet.

39 ▶ `ModifyEnabled(` *graph*, *from*, *to* `)` O

Modifies the "Enabledness" of menu entries according to their type and number of selected vertices. This operation works on all menu entries of some menus: *from* is the first menu to work on and *to* the last one (indices). Only menus with the property `IsAlive` are considered. `ModifyEnabled` returns nothing.

40 ▶ `InstallPopup(` *graph*, *func* `)` O

Installs a function that is called if the user clicks with the right button on a vertex. The function gets as parameters: *poset*,*vertex*,*x*,*y* (click position)

41 ▶ `PosetLeftClick(` *poset*, *x*, *y* `)` O

This operation is called when the user does a left click in the poset *poset*. The current pointer position is supplied in the parameters x and y. The generic method for `PosetLeftClick` lets the user move, select and deselect vertices or edges. An edge is selected as pair of vertices.

42 ▶ `PosetCtrlLeftClick(` *poset*, *x*, *y* `)` O

This operation is called when the user does a left click in a poset *poset* while holding down the control key. The current pointer position is supplied in the parameters x and y. The generic method for `PosetCtrlLeftClick` lets the user move, select and deselect vertices or edges. The difference to the operation without the control key is, that while selecting the old vertices are NOT deselected. Moving does not move the whole class but only one vertex. This allows for permuting the vertices within a class. An edge is selected as pair of vertices.

43 ▶ `PosetRightClick(` *poset*, *x*, *y* `)` O

This operation is called when the user does a right click in the graph *graph*. The generic method just finds the vertex under the mouse pointer and calls the `rightclickfunction` of the poset or graph which is a component in the GAP object. Note that the `rightclickfunction` can be called with `fail` if no vertex is hit.

Operations for user actions:

44 ▶ `UserDeleteVerticesOp(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Delete vertices`. The generic method actually deletes the selected vertices including all their edges.

45 ▶ `UserDeleteEdgeOp(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Delete edge`. The generic method deletes the edge with no further warning!

46 ▶ `UserMergeClassesOp(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Merge Classes`. The generic method walks through all levels and merges all classes that contain a selected vertex. Afterwards `UserRearrangeClasses` is called.

47 ▶ `UserMagnifyLattice(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Magnify Lattice`. The generic method scales everything by 144/100 including the sheet, all heights of levels and positions of vertices.

48 ▶ `UserShrinkLattice(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Shrink Lattice`. The generic method scales everything by 100/144 including the sheet, all heights of levels and positions of vertices.

49 ▶ `UserResizeLattice(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Resize Lattice`. The generic method asks the user for an x and a y factor and scales everything including the sheet, all heights of levels and positions of vertices.

50 ▶ `UserResizeSheet(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Resize Sheet`. The generic method asks the user for an x and a y pixel number and changes the width and height of the sheet. No positions of levels and vertices are changed. If the user asks for trouble he gets it!

51 ▶ `UserMoveLattice(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Move Lattice`. The generic method asks the user for a pixel number and changes the position of all vertices horizontally. No positions of levels are changed.

52 ▶ `UserChangeLabels(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Change Labels`. The user is prompted for every selected vertex, which label it should have.

53 ▶ `UserAverageY(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Average Y Positions`. In all levels the average y coordinate is calculated and all vertices are moved to this y position.

54 ▶ `UserAverageX(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Average X Positions`. The average of all x coordinates of the selected vertices is calculated. Then all classes with a selected vertex are moved such that the first selected vertex in this class has the calculated position as x position.

55 ▶ `UserRearrangeClasses(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Rearrange Classes`. All classes with a selected vertex are rearranged: The vertices are lined up neatly one after the other, sorted according to their current x position.

56 ▶ `UserUseBlackWhite(` *sheet*, *menu*, *entry* `)` O

This is called if the user selects `Use Black and White` in the menu.

57 ▶ `PosetShowLevels(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Show Levels` in the menu. Switches the display of the little boxes for level handling on and off.

58 ▶ `PosetShowLevelparams(` *sheet*, *menu*, *entry* `)` O

This operation is called when the user selects `Show Level Parameters` in the menu. Switches the display of the level parameters at the right of the screen on and off.

59 ▶ `DoRedraw(` *graph* `)` O

Redraws all vertices and connections.

8.3 An Example

This section shows how to use the poset package to display posets. The code presented here is actually part of the XGAP library and makes up the link to the C MeatAxe.

This is the declaration part:

```
############################################################################
##
#W  meataxe.gd                  XGAP library                  Max Neunhoeffer
##
#Y  Copyright 1998,       Max Neunhoeffer,              Aachen,       Germany
##
##  This file contains declarations for MeatAxe posets
##

DeclareFilter("IsMeatAxeLattice");

############################################################################
##
#O  GraphicMeatAxeLattice(<name>, <width>, <height>)  . creates graphic poset
##
##  creates a new graphic MeatAxe lattice which is a specialization of a
##  graphic poset. Those posets have a new filter for method selection.
##
DeclareOperation("GraphicMeatAxeLattice",[IsString, IsInt, IsInt]);
```

The code only declares a new filter and declares a constructor operation for posets that lie in this new filter.

The implementation:

```
############################################################################
##
#W  meataxe.gi                  XGAP library                  Max Neunhoeffer
##
#Y  Copyright 1998,       Max Neunhoeffer,              Aachen,       Germany
##
##  This file contains code for MeatAxe posets
##

############################################################################
##
#M  GraphicMeatAxeLattice(<name>, <width>, <height>)  . creates graphic poset
##
##  creates a new graphic MeatAxe lattice which is a specialization of a
##  graphic poset. Those posets have a new filter for method selection.
##
InstallMethod( GraphicMeatAxeLattice,
    "for a string, and two integers",
    true,
    [ IsString,
      IsInt,
      IsInt ],
    0,
```

```
function( name, width, height )
  local P;

  P := GraphicPoset(name,width,height);
  SetFilterObj(P,IsMeatAxeLattice);
  return P;
end);

#############################################################################
##
#M  CompareLevels(<poset>,<levelparam1>,<levelparam2>)  . . . . . .
##  . . . . . . . . . . . . . . . . . . . . . . . . .  compares two levelparams
##
##  Compare two level parameters. -1 means that <levelparam1> is "higher",
##  1 means that <levelparam2> is "higher", 0 means that they are equal.
##  fail means that they are not comparable. This method is for the case
##  if level parameters are integers and lower values mean lower levels
##  like in the case of MeatAxe lattices of Michael Ringe.
##
InstallMethod( CompareLevels,
    "for a graphic MeatAxe lattice, and two integers",
    true,
    [ IsGraphicPosetRep and IsMeatAxeLattice, IsInt, IsInt ],
    0,
function( poset, l1, l2 )
  if l1 < l2 then
    return 1;
  elif l1 > l2 then
    return -1;
  else
    return 0;
  fi;
end);
```

Besides the new constructor (which only adds a new filter) we only have to supply a new method for comparison of level parameters for such posets. The levels are numbered with integer numbers such that lower numbers are lower in the lattice.

There is a C program in the MeatAxe that exports a poset to a GAP program which generates the lattice in a graphic poset sheet. The user can then interactively move around vertices and shrink or magnify levels. He can then export the resulting lattice to an encapsulated postscript file. Note that you need a full installation of the C MeatAxe apart from GAP to use this feature.

Another nice little example is in the `examples` subdirectory in the XGAP distribution. It was written by Thomas Breuer (Aachen) to demonstrate the features of XGAP. The user gets a small window with a puzzle and can solve it using the mouse. You can test this example by starting XGAP and `Read`ing the file `pkg/xgap/examples/puzzle.g`. You can do this by using

```
gap> ReadPkg("xgap","examples/puzzle.g");
gap> p := Puzzle(4,4);
```

9 Graphic Graphs

In this version of XGAP this is not fully implemented and not yet usable. Only those parts which are exactly the same for graphic posets are already written. Please contact the author if you would like to have code to display graphs in graphic sheets.

10 Differences to XGAP 3

This rather short chapter is intended for the user who knows XGAP 3 well and quickly wants to know what has changed. So it covers mainly those parts, where existing code using XGAP has to be changed. For the totally new features and packages there are only a few references to the other parts of the documentation.

10.1 Concept

There are two main changes in the concept. The first is the migration to GAP4 with all the bells and whistles like object oriented design with operations, methods and method selection via filters. XGAP4 is rewritten nearly totally with these technologies. This should make the reusage of code in the future easier. One can now use big parts of the code of XGAP for own structures by just replacing some methods via overloading.

The second change is that there is no longer any mathematical "knowledge" or algorithm in XGAP. It is now only a front end and a graphical user interface. All code for finitely presented groups resides now in the GAP library. This is a much cleaner concept and should make the management of the source code easier. At the same time XGAP has become a much more generic program. Operations for subgroups are for example no longer hard wired into XGAP case by case but there is generic code which can be adapted just by hacking a few tables.

These generalizations made some sacrifices necessary, because XGAP does no longer know anything about the mathematics it is displaying. It may for example happen that XGAP does no longer adapt its behaviour to the amount of data that is known about some finitely presented groups. The reason for this is, that the generic poset routines cannot know that the vertices stand for groups at all. So sometimes one has to trigger the comparison of subgroups of finitely presented groups manually (see section 5.12.4 for a description how to do this).

In the old GAP3 version of XGAP there were three different programs for the full subgroup lattice of a (finite) group (`GraphicLattice`), the interactive partial subgroup lattice of a finite group (`InteractiveLattice`) and the interactive partial subgroup lattice of a finitely presented group (`InteractiveFpLattice`) respectively. Now there is only one generic program to display subgroup lattices interactively (`GraphicSubgroupLattice`).

XGAP can now handle subgroups of infinite index. They are either placed in a "finite size" level or in an "infinity" level. See 5.3 for details.

A new logging facility allows to automatically produce a protocol of the actions the user performs via mouse clicks. This is convenient because the normal GAP command script contains no useful information about the selected entries in the menus. See 5.2 for details.

There is a new layer to display generic posets that do not have to be subgroup lattices. It can be used to display posets interactively very easily. This is for example used in the new link to the C-MeatAxe written by Michael Ringe. The code for this link is also included in XGAP4.

Code for the display of graphic graphs is planned but not yet completed.

The user of XGAP should not realize much of those changes (except of course the name of the function to display a subgroup lattice). The programmer on the other hand has to get used to the new techniques. It was not in all places possible to achieve total compatibility for existing code. Some changes also were introduced deliberately to make the programmers adapt their programs to the new situation!

10.2 User Interface

Some menu entries have been moved to new places, mainly because of the division of generic poset code and specialized code for graphic subgroup lattices. There are some new features and nearly all old features have made it into the new version.

The handling of the mouse is unchanged. However the introduction of levels gives the user new possibilities.

10.3 Where code has to be changed

All GAP objects corresponding to graphic sheets and graphic objects are no longer records but component objects. This means that the programmer can no longer mess around in the data structures. If you want to add new fields, then you have to use inheritance and define new categories. This means also that the (internal) data structures of sheets has changed massively. Programs that try to access record components of old XGAP structures will no longer work!

The operation `InstallGSMethod` is no longer present. It is replaced by the "callback" mechanism with the operations `InstallCallback`, `RemoveCallback` and `Callback` (see 6.1.1 for details). This means, that mouse events are handled differently. This was changed deliberately because there is a big difference: In XGAP4 you can install more than one function for one type of mouse event. All such callback functions are called one after the other. There was only one graphic sheet method for each event in XGAP3. So you can **not** just change the name of the operation to install the callback. You have to think about this difference!

See the section 6.4 for an overview which operations exist now for which graphic objects. The main difference is the introduction of `Revive`, `ViewObj` and `WindowId` together with the concept of the `IsAlive` filter.

There was a bug in XGAP3 in the creation of menus: If an entry starts with a minus sign, it will become a separating line instead of a real menu entry. This disturbed the numbering of the menu entries, such that `Enable` and `Check` did not work on the correct entry. This bug is fixed in XGAP4 so code which contained a workaround for this bug has to be changed. `Enable` and `Check` behave now like expected and documented in 7.1.3 and 7.1.2.

Bibliography

[Cav86] Alberto Cavicchioli. A countable class of non-homeomorphic homology spheres with Heegaard genus two. *Geom. Dedicata*, 20:345–348, 1986.

[FJ70] K. Ferber and H. Jürgensen. A Programme for Drawing a Lattice. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 83–86, 1970.

[HR91] Derek F. Holt and Sarah Rees. A Graphics System for Displaying Finite Quotients of Finitely Presented Groups. In L. Finkelstein and W. M. Kantor, editors, *Groups and Computation*, pages 113–126, 1991.

[Kei95] Susanne Keitemeier. Graphische Darstellung von Untergruppendiagrammen im Computeralgebraprogramm GAP. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1995.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "`PermutationCharacter`" comes before "permutation group".

A

B

C

D

E

F

G

H

I

L

V

W