ABACUS

A Branch-And-CUt System

Version 2.0

User's Guide and Reference Manual

Stefan Thienel

September 1997

Stefan Thienel
Institut für Informatik
Universität zu Köln
Pohligstraße 1
50969 Köln
Germany E-mail: thienel@informatik.uni-koeln.de

Writing this manual has been partially supported by ESPRIT LTR Project no. 20244 (ALCOM-IT) and H.C.M. Institutional Grant no. ERBCHBGCT940710 (DONET).

Contents

Chapter 1

Introduction

Preface to Release 2.0

During its first year of public availability ABACUS reached a rather active community of users, which is growing slowly but constantly. Many of them contributed to making ABACUS more reliable. I want to thank all of them for their helpful feedback. In particular, I want to mention Max Böhm, who pointed me to several improvement possibilities.

But not only the users worked with ABACUS, also its development continued such that it is now ready for a second release. ABACUS 2.0 offers besides many minor extensions four major new features:

- the interface to the new LP-solver SoPlex
- the support of the Visual C++ compiler
- a generalized strong branching method
- increased safety against name collisions

In particular, I am very happy that the abstract LP-interface proved its usefulness during the integration of the LP-solver SoPlex. Since the adaption of the framework to the Visual C++ compiler could be performed, I am optimistic that also other compilers can be supported in the future.

Users who want to upgrade from version 1.2.x find the new features and the differences to previous versions in Section 3.

Köln, August 1997 *Stefan Thienel*

Preface to Release 1.2

ABACUS is a software system for the implementation of linear-programming based branch-and-bound algorithms, i.e., branch-and-cut algorithms, branch-and-price algorithms, and their combination. It applies the concepts of object oriented programming (programming language C++). An implementation of a problem specific algorithm is obtained by deriving some classes from abstract base classes of ABACUS in order to embed problem specific functions.

While the Chapters 1 to 4 of this manual are a user's guide describing the installation, design, and application of ABACUS the last chapter contains the reference manual. Chapter 2 explains how ABACUS is installed on your computer system and what hardware and software environment is required. In order to simplify the user understanding ABACUS I describe in Chapter 4 the design of the software framework. While I recommend to study in any case the basic concepts outlined in Section 4.1 before beginning with the implementation of an application, it should be sufficient to return to Section 4.2 only

for rather advanced usage. Also Chapter 5 is split into two sections. The first one, Section 5.1, explains the first steps that have to be performed to implement an application. This section should be studied together with the example included in the ABACUS distribution. The second one, Chapter 5.2, shows how default strategies of ABACUS can be modified and outlines some additional features of the system. The reference manual of Chapter 6 is complemented by the index that simplifies finding a certain class or one of its members.

This manual is both available in Postscript and HTML format. The HTML form turns out to be quite useful for finding members of the reference manual.

This user's guide is not intended to teach the concepts of linear-programming based branch-and-bound, but I assume that the reader of this manual and the user of ABACUS is familiar with these algorithms. For an introduction to branch-and-cut I refer to [JRT95], for an introduction to branch-and-price algorithms I recommend to [BJN+97]. Both approaches are described in [Thi95].

Moreover, I also assume that the user of ABACUS is familiar with the concepts of object oriented programming. For the reader who is unexperienced in object oriented programming I refer to [KM90] for a good brief introduction and to [Boo94] for a detailed description. There are many books about the programming language C++. The classical introduction is [Str93]. Very useful reference manuals are [ES92] and the current working paper of the C++ standardization committee [ASC95].

ABACUS originates from the dissertation of its author [Thi95] and has since then been tested, slightly modified and improved. Here, I would like to thank all initial testers, in particular Thomas Christof, Meinrad Funke, and François Margot for their bug reports and helpful comments. I am very grateful to Joachim Kupke for carefully proofreading an earlier version. I also want to thank Denis Naddef, LMC-IMAG, Grenoble, France, for his hospitality while writing the major part of this manual.

Despite these successful tests I consider ABACUS still as an experimental system. Therefore, feedback of the users is appreciated. Some parts of the user's guide were adapted from [Thi95], while the reference manual has been compiled for the first time. Therefore, I also encourage the reader to send me error reports and improvement suggestions for the user's guide and the reference manual.

I am aware that neither the software nor its documentation is perfect, but I think it is time to dare a first public release.

Grenoble, August 1996 *Stefan Thienel*

Chapter 2

Installation

2.1 Obtaining ABACUS

You can download ABACUS from:

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

ABACUS is being further developed. New releases are announced to all users that obtained a license (see Section 2.6). If you wish to receive these news without being already an official user, or if you want to be removed from this list, send a message to:

`abacus@informatik.uni-koeln.de`

2.2 Platforms

ABACUS is currently available for SUN SPARC, IBM RS6000, DEC ALPHA, SILICON GRAPHICS, and HP 9000 workstations. On PCs we support Linux and Windows NT. If you are interested in a version for another platform please contact us directly. The preprocessor flag given in Table 2.1 has to

Architecture	Operating System	Preprocessor-Flag
SUN SPARC	SUN-OS 4.1.3	ABACUS_SYS_SUNOS4
SUN SPARC	SUN-OS 5.4	ABACUS_SYS_SUNOS5
IBM RS6000	AIX 3.2	ABACUS_SYS_AIX
DEC ALPHA	OSF 3.2	ABACUS_SYS_OSF
SILICON GRAPHICS	Irix 5.3	ABACUS_SYS_IRIX
HP 9000	HP-UX 9.05	ABACUS_SYS_HP
HP 9000	HP-UX 10.01	ABACUS_SYS_HP
PC	Linux 2.0.27	ABACUS_SYS_LINUX
PC	Windows NT	ABACUS_SYS_WINNT

Table 2.1: Platforms.

be defined when you compile your own ABACUS program. E.g, for a system running SUN-OS 5.4 use `-DABACUS_SYS_SUNOS5` in the command line of your compiler.

2.3 Compiler

Unfortunately, neither the C++ standardization committee has finished its work nor the various compilers implement the same subset of the programming language. Therefore, we currently only support two

different compilers, but hope to extend ABACUS to other compilers in the near future. If you are interested in ABACUS for another compiler please contact us.

2.3.1 UNIX

On UNIX platforms the GNU-C++ compiler G++version 2.7.1 or 2.7.2 is required.

2.3.2 Windows NT

For Windows NT we support the Visual C++ compiler version 5.0.

2.3.3 Compiler Selection

In the compilation you have to specify the compiler by the preprocessor flag according to Table 2.2.

Compiler	Preprocessor-Flag
gcc 2.7.x	ABACUS_COMPILER_GCC
Visual C++ 4.0	ABACUS_COMPILER_VISUAL_CPP

Table 2.2: Compilers.

2.4 LP-Solver

ABACUS provides a general interface to linear programming solvers. However, the current release supports only the LP-solvers Cplex versions 2.2, 3.0, and 4.0 [Cpl94, Cpl95] and SoPlex 1.0 [Wun97]. There are different ABACUS libraries that can be combined with Cplex, SoPlex, or both LP-solvers.

2.4.1 Cplex

Note, for each version of Cplex you need a special version of ABACUS since Cplex changed function names and arguments. If you are compiling your own ABACUS application you have to specify your Cplex version by a preprocessor flag (Table 2.3). Sometimes we observed with Cplex 3.0 that the wall

Cplex Version	Preprocessor Flag
Cplex 2.2	ABACUS_LP_CPLEX22
Cplex 3.0	ABACUS_LP_CPLEX30
Cplex 4.0	ABACUS_LP_CPLEX40

Table 2.3: Cplex Versions.

clock time of a run is very high compared to the cpu time although the load of the workstation the job was running on was very small. This overhead was resulting from opening, reading, and closing files within the Cplex license checks, which were performed very often. You can verify this by tracing the system calls of your application (e.g., with the UNIX commands `strace` or `truss`). In this case you should ask Cplex for a nicer license. So far, we do not have any experience if this problem has been fixed in Cplex 4.0.

Cplex is a commercial product. Our experience is that it is fast and reliable. You find further information about Cplex at `http://www.cplex.com`.

2.4.2 SoPlex

If you want to use SoPlex you have to set the preprocessor flag `ABACUS_SOPLEX` during the compilation of your application. It is also required to switch to the new include file structure (see Section 3.3) in order to avoid name conflicts.

Note, there are some restrictions of SoPlex in comparison with Cplex:

- SoPlex lacks a function for determining the reason for infeasible LP-relaxations that is required in some cases in a branch-and-cut-and-price algorithm. This feature is provided by the Cplex function `CPXgetijdiv`. Our experience shows that these cases happen rather seldomly. However, if such a case occurs, ABACUS has to stop the optimization. As long as not both cutting planes and columns are generated in the same algorithm this function is not required.
- SoPlex does not provide a barrier method.

SoPlex can be used free charge for academic users and first experiments show that its performance is competitive with commercial solvers. Unfortunately there is currently no support. We observed problems in very rare cases, i.e., SoPlex stopped reporting an internal error. SoPlex can be obtained from `http://www.zib.de/Optimization/Software/Soplex/`.

2.5 Installation of the Files

An ABACUS distribution consists of two files: the platform dependent library and the platform independent files (e.g., include files, documentation, etc.).

The directory `abacus` consists of the subdirectories `include/abacus`, `lib`, `doc`, `example`, and `tools`. The directory `abacus/include/abacus` contains files with the extension `.h`, the headers of all classes, and files with the extension `.inc`, the definition of the member functions of template classes. The configuration file `.abacus` is installed in the directory `abacus`. The documentation in Postscript and HTML format is contained in the subdirectory `abacus/doc`. The complete source code of an example of a simple branch-and-cut algorithm for the traveling salesman problem can be found in the subdirectory `abacus/example`. Helpful Perl scripts for an upgrade from previous ABACUS versions can be found in the subdirectory `abacus/tools`.

2.5.1 UNIX-Platforms

The platform independent files are distributed as a GNU-zipped tar file. Hence, the first step is to unpack the file `abacus.tar.gz`. Copy `abacus.tar.gz` to the directory where you want ABACUS to be installed (e.g. `/usr/local`) and call:

```
gzip -d abacus.tar.gz
tar xf abacus.tar
```

After this installation you should obtain a platform and LP-solver dependent library, e.g., for Linux in combination with Cplex 4.0 the library `libabacus.linux.cplex40.a.gz`. After uncompressing it with

```
gzip -d libabacus.linux.cplex40.a.gz
```

you might like to move it to the subdirectory `abacus/lib` or an other directory.

2.5.2 Windows NT

Both the platform independent files and the ABACUS library are zipped archives. First, obtain the archive `abacus.zip` and move to the directory in which ABACUS should be installed. You can unpack it with the command `pkunzip`. Then, you should obtain the library `libabacus.zip`, move it to the directory `abacus/lib` and unpack also with `pkunzip`.

2.6 The License

In order to use ABACUS you need a license code for every machine on which you want to use ABACUS. This license code depends on the name of the machine and, in addition for UNIX machines, on the ID of the machine. Send this information for every machine on which you like to use ABACUS to:

`abacus@informatik.uni-koeln.de`

Please note also your name and affilation that we can inform you on future updates of ABACUS.
The license code is then returned to you by e-mail. For a single machine with hostname `yourhost` the license code might look like

```
yourhost     112571219315081131
```

This line must be added to a file with the name `.abacusLicense`. You have to add for each machine on which you want to use ABACUS a line with the host name and the license code to the file `.abacusLicense`. The file `.abacusLicense` can be stored in any subdirectory of your file system. It is found with the help of the environment variable `ABACUS_LICENSE_DIR` (see Section 2.7).

2.6.1 UNIX

For UNIX platforms we need for the license generation

- the host name that can usually be obtained by the UNIX commands `hostname`, `uname -n`, or `sysinfo`,
- the host id that is output by the UNIX commands `hostid` or `sysinfo` (the host id is usually a hexadecimal number (e.g., `23ac27fe`) or sometimes a decimal number; the IP address (e.g., 123.45.6.78) is not required for the license),
- the operating system.

2.6.2 Windows NT

For machines running Windows NT we only need the name of the machine in order to return your license code.

2.7 Environment Variables

Two environment variables have to be initialized before ABACUS can be used. The environment variable `ABACUS_LICENSE_DIR` receives the name of the directory in which the file `.abacusLicense` is stored. The environment variable `ABACUS_DIR` is initialized with the directory containing the general configuration file `.abacus`. A master version of this configuration file is available in the abacus package. However, every user might want to make his modifications. Therefore, it is recommendable that every user makes a private copy of the file `.abacus`.

2.7.1 UNIX-Platforms

If the file `.abacus` is stored in the directory `/home/yourhome` and the file `.abacusLicense` is stored in the directory `/usr/local/abacus`, then the environment variables should be initialized in the following way if you are using the C-shell or its relatives:

```
setenv ABACUS_DIR /home/yourhome
setenv ABACUS_LICENSE_DIR /usr/local/abacus
```

If you are using the Bourne-shell you might want to use the commands:

```
export ABACUS_DIR=/home/yourhome
export ABACUS_LICENSE_DIR=/usr/local/abacus
```

Usually it is convenient to add these instructions to your `.login` file.

2.7.2 Windows NT

On Windows NT you have to click on the icon for the setting system parameters. In the window that opens, you have to set the environment variables `ABACUS_DIR` and `ABACUS_LICENSE_DIR`. If the file `.abacus` is stored in the the directory `C:\abacusapplication` and if the license file is stored in `C:\abacus`, the two environment variables should be set in the following way.

```
ABACUS_DIR=C:\abacusapplication
ABACUS_LICENSE_DIR=C:\abacus
```

2.8 Compiling and Linking

For the compilation you have to make sure that the ABACUS include files and the include files of your LP-solver(s) are found. Moreover, the compiler flags for your operating system, compiler, and your LP-solver have to be specified (see Tables 2.1, 2.2, 2.3 and Section 2.4.2).You find a list of all preprocessor flags in Section 6.6.

The object files of your application have to be linked together with the ABACUS library `libabacus.a` and at least one LP-solver (the Cplex callable library or the SoPlex library).

2.8.1 UNIX

For the compilation of your files using ABACUS you should add the `abacus/include` directory either to your include directory path or specify it explicitly with the `-I` compiler option. In the same way add the include file paths of Cplex and/or SoPlex. The compiler flags for your platform and the LP-solver can be defined at compilation time with the `-D` switch of the compiler (e.g., `-DABACUS_SYS_LINUX`).

It might be helpful to consider the `Makefile` of the example included in the ABACUS distribution.

2.8.2 Windows NT

Add the directory `abacus\include` together with its path and the include file paths of your LP-solver(s) to the *Additional include directories* in the project settings. The flags `ABACUS_SYS_WINNT`, `ABACUS_COMPILER_VISUAL_CPP`, and the flags for the used LP-solver(s) should be added to the *Preprocessor definitions* in the project settings.

The project settings of the example, which is part of the UNIX distribution, might be helpful.

2.9 Problems

ABACUS is a rather new software system. Therefore, it is very unlikely that it is completely free of bugs although several applications have been implemented successfully. In order to make ABACUS a stable system, the assistance of the users is required. Report all problems by e-mail to:

`abacus@informatik.uni-koeln.de`

Before reporting a bug, please make sure that it does not come from an incorrect usage of the programming language C++.

Also feedback from the users is highly appreciated. Please report your experiences and make your suggestions. Also comments on this user manual are appreciated. Of course, it is impossible to implement everybody's wish immediately. However, if a missing feature is demanded by several users or might be useful in general, we will try to add it in a future release.

Chapter 3

New Features

This section summarizes all new features that have been introduced since the release of ABACUS 1.2.

3.1 LP-Solver SoPlex

Besides Cplex ABACUS provides now an interface to the LP-Solver SoPlex [Wun97] (see Section 2.4.2).

If SoPlex is used as LP-solver, it might be required to switch to the new include file structure (see Section 3.3) in order to avoid name conflicts. Both SoPlex and ABACUS provide include files with the name `timer.h`.

3.2 Naming Conventions

The previous version did not use any prefix for all globally visible names in order to avoid name collisions with other libraries since the C++ concept of namespaces should make this technique redundant. Unfortunately, it turned out that the GNU C++ compiler does still not support namespaces. The G++-FAQ mentions that even in the next release 2.8 this concept might not be supported.

In order to provide the possibility of avoiding name collisions without namespaces, we added to all globally visible names the prefix `ABA_`. There are two possibilities for reusing your old codes together with the new name concept.

The first method is to include the file `oldnames.h` into every file using ABACUS names without the prefix `ABA_`. In the compilation the preprocessor flag `ABACUS_OLD_NAMES` must be set. With preprocessor definitions the old names are converted to new names. You should be aware that this technique can have dangerous side effects. Therefore, this procedure should **not** be applied if you combine ABACUS with any other library in your application.

The second method is the better method and is not much more work than the first one. In the `tools` subdirectory of the ABACUS distribution you can find the Perl script `old2newnames.pl`. If you apply this script to all source files of your ABACUS application by calling

```
old2newnames.pl <files>
```

a copy of each file given in `<files>` is made in the subdirectory `new-files` and the old names are replaced by the new names.

3.3 Include File Path

Another problem are header files of different libraries with the same name. It can happen that due to the inclusion structure it is not possible to avoid these conflicts by the order of the include file search paths. Therefore, every ABACUS include file (`*.h` and `*.inc`) is included now from the subdirectory `abacus`.

You can continue using the old include file structure by setting the preprocessor flag `ABACUS_OLD_INCLUDE`. Here is an example how an ABACUS file includes other ABACUS files:

```
#ifdef ABACUS_OLD_INCLUDE
#include "array.h"
#else
#include "abacus/array.h"
#endif
```

We strongly recommend the use of the new include file structure. In combination with the LP-solver SoPlex the new include file structure is sometimes required (it depends which ABACUS and which SoPlex files you include). There may be name conflicts since both systems have a file `timer.h`.

Due to this concept also the directory structure of the ABACUS distribution has changed. All include files are now in the subdirectory `include/abacus`.

A conversion can be performed with the help of the Perl script `tools/old2newincludes.pl`. Calling

```
old2newincludes <files>
```

makes a copy of all `<files>` into the subdirectory `new-includes` and adapts them to the new include structure, e.g.,

```
#include "master.h"
```

is replaced by

```
#include "abacus/master.h"
```

in the new files.

3.4 Advanced Control of the Tailing Off Effect

ABACUS automatically controls the tailing off effect according to the parameters `TailOffNLps` and `TailOffPercent` of the configuration file `.abacus`. Solutions of LP-relaxations can be skipped in this control by calling the function `ignoreInTailingOff()` (see Section 5.2.25).

3.5 Problem Specific Fathoming

Problem specific fathoming criteria can be added by the redefinition of the virtual function `ABA_SUB::exceptionFathom()` (see Section 5.2.23).

3.6 Problem Specific Branching

A problem specific branching step can be enforced by the redefinition of the virtual function `ABA_SUB::exceptionBranch()` (see Section 5.2.24).

3.7 Generalized Strong Branching

Generalized strong branching is the possibility of evaluating different branching rules and selecting the best ones. If branching on variables is performed, e.g., the first linear programs of the (potential) sons for various branching variables are solved, in order to find the most promising variable. Together with the built-in branching strategies this feature can be controlled with the new entry `NBranchingVariableCandidates` of the configuration file (Section 5.2.26). Moreover, also other branching strategies can be evaluated as explained in Section 5.2.8.

3.8 Pool without Constraint Duplication

One problem in using ABACUS can be the large number of generated constraints and variables that use a lot of memory. In order to reduce the memory usage we provide a new pool class `ABA_NONDUPLPOOL` that avoids the multiple storage of the same constraint or variable in the same pool. The details are explained in Section 5.2.2.

3.9 Visual C++ Compiler

In addition to the GNU C++ compiler on UNIX operating systems, ABACUS is now also available on Windows NT in combination with the Visual C++ compiler. Further details for using ABACUS in this new environment can be found in Section 2

3.10 Compiler Preprocessor Flag

In the compilation of an ABACUS-application the used compiler must be specified by a preprocessor flag (see Section 2.3).

3.11 LP-Solver Preprocessor Flag

The LP-solvers that are used have to be specified by a preprocessor flag (see Section 2.4). Also the flags for the LP-solver Cplex changed.

3.12 Parameters of Configuration File

Three new parameters have been added to the configuration file `.abacus`.

3.12.1 NBranchingVariableCandidates

The parameter `NBranchingVariableCandidates` can be used to control the number of tested branching variables if our extended strong branching concept is used (see Section 5.2.8).

3.12.2 DefaultLpSolver

An other new parameter is `DefaultLpSolver` allows to choose either `Cplex` or `SoPlex` as default LP-solver for the solution of the LP-relaxations.

3.12.3 SoPlexRepresentation

SoPlex works internally either with column or a row basis. This basis representation can be selected with the parameter `SoPlexRepresentation`. Our tests show that only the row basis works stable in SoPlex 1.0. Further details are explained in Section 5.2.26.

3.13 New Functions

We implemented several new functions. Some of them might be also interesting for the users of ABACUS. For the detailed documentation we refer to the reference manual.

- `ABA_BPRIOQUEUE::getMinKey()`
- `ABA_BHEAP::getMinKey()`

- `bool ABA_GLOBAL::isInteger(double x)`
- In addition to the function

```
void MASTER::initializePools(ABA_BUFFER<ABA_CONSTRAINT*> &constraints,
                             ABA_BUFFER<ABA_VARIABLE*> &Variables,
                             int varPoolSize,
                             int cutPoolSize,
                             bool dynamicCutPool = false);
```

  the function

```
void MASTER::initializePools(ABA_BUFFER<ABA_CONSTRAINT*> &constraints,
                             ABA_BUFFER<ABA_CONSTRAINT*> &cuts,
                             ABA_BUFFER<ABA_VARIABLE*> &Variables,
                             int varPoolSize,
                             int cutPoolSize,
                             bool dynamicCutPool = false);
```

  also allows the insertion of some initial cuts into the cut pool.

- Manipulators for setting the width and the precision of `ABA_OSTREAM` have been added that work like the corresponding manipulators of the class `ostream`.

```
ABA_OSTREAM_MANIP_INT setWidth(int w);
ABA_OSTREAM_MANIP_INT setPrecision(int p);
```

- `ABA_OSTREAM::setFormatFlag(fmtflags)`
- The objective function sense can be changed in the ABA_LP classes with the function

```
void ABA_LP::sense(const ABA_OPTSENSE &newSense).
```

- The != operator is now available for the class `ABA_STRING`.

3.14 Miscellaneous

Besides some bug fixes we made many minor improvements. The most important ones are listed here.

- The output for the output levels `SubProblem` and `LinearProgram` is formatted in a nicer way.
- Besides those Cplex parameters that could be directly controlled by ABACUS functions, it is now possible to get or to modify any Cplex 4.0 parameter with the functions:

```
int CPLEXIF::CPXgetdblparam(int whichParam, double *value);
int CPLEXIF::CPXsetdblparam(int whichParam, double value);
int CPLEXIF::CPXgetintparam(int whichParam, int *value);
int CPLEXIF::CPXsetintparam(int whichParam, int value);
```

- If a linear program is solved with the barrier method, then usually a cross over to an optimal basic solution is performed. The value of a variable in the optimal solution of the barrier method before the cross over can be obtained with the function `double barXVal(int i)`. If this "pre-cross over" solution is available, can be checked with the function `SOLSTAT barXValStatus() const`.
- The minimal required violation of a constraint or variable in a pool separation or pool pricing, respectively, can be specified as a parameter of the functions `ABA_SUB::constraintPoolSeparation` and `ABA_SUB::variablePoolSeparation`. The minimal violation is also a parameter of the function `ABA_POOL::separate` and of redefinitions of this function in derived classes.

Chapter 4

Design

From a user's point of view, who wants to implement a linear-programming based branch-and-bound algorithm, ABACUS provides a small system of base classes from which the application specific classes can be derived. All problem independent parts are "invisible" for the user such that he can concentrate on the problem specific algorithms and data structures.

The basic ideas are pure virtual functions, virtual functions, and virtual dummy functions. A pure virtual function has to be implemented in a class derived by the user of the framework, e.g., the initialization of the branch-and-bound tree with a subproblem associated with the application. In virtual functions we provide default implementations, which are often useful for a big number of applications, but can be redefined if required, e.g., the branching strategy. Finally, under a virtual dummy function we understand a virtual function that does nothing in its default implementation, but can be redefined in a derived class, e.g., the separation of cutting planes. It is not a pure virtual function as its definition is not required for the correctness of the algorithm.

Moreover, an application based on ABACUS can be refined step by step. Only the derivation of a few new classes and the definition of some pure virtual functions is required to get a branch-and-bound algorithm running. Then, this branch-and-bound algorithm can be enhanced by the dynamic generation of constraints and/or variables, primal heuristics, or the implementation of new branching or enumeration strategies.

Default strategies are available for numerous parts of the branch-and-bound algorithm, which can be controlled via a parameter file. If none of the system strategies meets the requirements of the application, the default strategy can simply be replaced by the redefinition of a virtual function in a derived class.

4.1 Basics

The inheritance graph of any set of classes in C++ must be a directed acyclic graph. Very often these inheritance graphs form forests or trees. Also the inheritance graph of ABACUS is designed as a tree with a single exception where we use multiple inheritance.

The following sections and Table 4.1 give a survey of the different classes of ABACUS. The details are outlined in Section 4.2.

ABACUS		
Pure Kernel	**Application Base**	**Auxiliaries**
Linear Program	Master	Basic Data Structures
Pool	Subproblem	Tools
Branch & Bound	Constraints	
	Variables	

Table 4.1: The classes of ABACUS.

Basically the classes of ABACUS can be divided in three different main groups. The application base classes are the most important ones for the user. From these classes the user of the framework has to derive the classes for his applications. The pure kernel classes are usually invisible for the user. To this group belong, e.g., classes for supporting the branch-and-bound algorithm, for the solution of linear programs, and for the management of constraints and variables. Finally, there are the auxiliaries, i.e., classes providing basic data structures and tools, which can optionally be used for the implementation of an application.

4.1.1 Application Base Classes

The following classes are usually involved in the derivation process for the implementation of a new application.

The Master

The class `ABA_MASTER` is one of the central classes of the framework. It controls the optimization process and stores global data structures for the optimization. For each new application a class has to be derived from the class `ABA_MASTER`.

The Subproblem

The class `ABA_SUB` represents a subproblem of the implicit enumeration, i.e., a node of the branch-and-bound tree. The subproblem optimization is performed by the solution of linear programming relaxations. Usually, most running time is spent within the member functions of this class. Also from the class `ABA_SUB` a new class has to be derived for each new application. By redefining virtual functions in the derived class problem specific algorithms as, e.g., cutting plane or column generation, can be embedded.

The Constraints and Variables

ABACUS provides some default concepts for the representation of constraints and variables. However, it still might be necessary that for a new application special classes have to be derived from the classes `ABA_CONSTRAINT` and `ABA_VARIABLE`, which then implement application specific methods and storage formats.

4.1.2 Pure Kernel Classes

This group covers classes that are required for the implementation of the kernel of ABACUS but usually of no direct importance for the user of the framework.

The Root of the Class Tree

All classes of ABACUS have the common base class `ABA_ABACUSROOT`.

The Linear Program

The part of the inheritance graph related to the solution of linear programs contains several classes. There is a general interface to the linear program from which a class for the solution of linear programming relaxations within our branch-and-bound algorithm is derived. Both classes are independent from the used LP-solver, which can be plugged in via a separate class. Currently, we support the LP-solvers Cplex and SoPlex.

The Pool

Constraints and variables are stored in pools. We provide an abstract base class for the representation of pools and derive from this class a standard realization of a pool. Several other classes are required for a safe management of active and inactive constraints and variables.

The Branch-and-Bound Auxiliary Classes

Various classes are required to support the linear-programming based branch-and-bound algorithm, e.g., for the management of the branch-and-bound tree, for the storage of the active and inactive constraints, special buffers for newly generated constraints and variables, for the control of the tailing off effect, and for fixing variables by reduced costs. An important part of the inheritance graph in this context is formed by the various branching rules, which allow a very flexible implementation of branching strategies.

4.1.3 Auxiliaries

We use the following classes for the implementation of other classes within ABACUS, but they might also be useful for the implementation of new applications.

The Basic Data Structures

ABACUS is complemented by a set of basic data structures. Most of them are implemented as generic classes (templates).

The Tools

Finally, we also provide some useful tools, e.g., for generating output, measuring time, and sorting.

4.2 Details

In this section we describe the different subtrees in the class hierarchy and their classes. We give this description not in the form of a manual by describing each member of the class (this is later done partially in Chapter 5 and in detail in the reference manual), but we try to explain the problems, our ideas, why we designed the class hierarchy and the single classes as we did, and discuss also some alternatives.

4.2.1 The Root of the Class-Tree

It is well known that global variables, constants, or functions can cause a lot of problems within a big software system. This is even worse for frameworks such as ABACUS that are used by other programmers and may be linked together with other libraries. Here, name conflicts and undesired side effects are almost inevitable. Since global variables can also make a future parallelization more difficult we have avoided them completely.

We have embedded functions and enumerations that might be used by all other classes in the class `ABA_ABACUSROOT`. We use this class as a base class for all classes within our systems. Since the class `ABA_ABACUSROOT` contains no data members, objects of derived classes are not blown up.

Currently, `ABA_ABACUSROOT` implements only an enumeration with the different exit codes of the framework and implements some public member functions. The most important one of them is the function `exit()`, which calls the system function `exit()`. This construction turns out to be very helpful for debugging purposes.

4.2.2 The Master

In an object oriented implementation of a linear-programming based branch-and-bound algorithm we require one object that controls the optimization, in particular the enumeration and resource limits, and stores data that can be accessed from any other object involved in the optimization of a specific instance. This task is performed by the class `ABA_MASTER`, which is not identical with the root node of the enumeration tree. For each application of ABACUS we have to derive a class from `ABA_MASTER` implementing problem specific "global" data and functions.

Every object, which requires access to this "global" information, stores a pointer to the corresponding object of the class `ABA_MASTER`. This holds for almost all classes of the framework. For example the class

ABA_SUB, implementing a subproblem of the branch-and-bound tree, has as a member a pointer to an object of the class ABA_MASTER (other members of the class ABA_SUB are omitted):

```
class ABA_SUB {
  ABA_MASTER *master_;
};
```

Then, we can access within a member function of the class ABA_SUB, e.g., the global upper bound by calling

```
master_->upperBound();
```

where upperBound() is a member function of the class ABA_MASTER.

Encapsulating this global information in a class is also important, if more than one linear-programming based branch-and-bound is solved within one application. If the pricing problem within a branch-and-price algorithm is again solved with the help of ABACUS, e.g., then separate master objects with different global data are used.

The Base Class Global

Within a specific application there are always some global data members as the output and error streams, zero tolerances, a big number representing "infinity", and some functions related with these data. For the same reasons we discussed already in the description of the class ABA_ABACUSROOT we should avoid storing these data in global variables. It is also not reasonable to add these data to the class ABA_ABACUSROOT, because it would blow up every derived class of ABA_ABACUSROOT and it is neither necessary nor desired to have extra output streams, zero tolerances, etc., for every object.

Instead of implementing this data directly in the class ABA_MASTER we designed an extra class ABA_GLOBAL, from which the class ABA_MASTER is derived. The reason is that there are several classes, especially some basic data structures, which might be useful in programs that are not branch-and-bound algorithms. To simplify their reuse these classes have a pointer to an object of the class ABA_GLOBAL instead of one to an object of the class ABA_MASTER.

Branch-and-Bound Data and Functions

The class ABA_MASTER augments the data inherited from the class ABA_GLOBAL with specific data members and functions for branch-and-bound. It has objects of classes as members that store the list of subproblems which still have to be processed in the implicit enumeration (class ABA_OPENSUB), and that store the variables which might be fixed by reduced cost criteria in later iterations (class ABA_FIXCAND). Moreover, the solution history, timers for parts of the optimization, and a lot of other statistical information is stored within the class ABA_MASTER.

The class ABA_MASTER also provides default implementations of pools for the storage of constraints and variables. We explain the details in Section 4.2.5.

A branch-and-bound framework requires also a flexible way for defining enumeration strategies. The corresponding virtual functions are defined in the class ABA_MASTER, but for a better understanding we explain this concept in Section 4.2.7, when we discuss the data structure for the open subproblems.

Limits on the Optimization Process

The control of limits on the optimization process, e.g., the amounts of CPU time and wall-clock time, and the size of the enumeration tree are performed by members of the class ABA_MASTER during the optimization process. Also the guarantee of the solution is monitored by the class ABA_MASTER.

The Initialization of the Branch-and-Bound Tree

When the optimization is started, the root node of the branch-and-bound tree has to be initialized with an object of the class ABA_SUB. However, the class ABA_SUB is an abstract class, from which a class implementing the problem specific features of the subproblem optimization has to be derived. Therefore, the initialization of the root node is performed by a pure virtual function returning a pointer to a class derived from the class ABA_SUB. This function has to be defined by a problem specific class derived from the class ABA_MASTER.

The Sense of the Optimization

For simplification often programs that can be used for minimization and maximization problems use internally only one sense of the optimization, e.g., maximization. Within a framework this strategy is dangerous, because if we access internal results, e.g., the reduced costs, from an application, we might misinterpret them. Therefore, ABACUS also works internally with the true sense of optimization. The value of the best known feasible solution is denoted *primal bound*, the value of a linear programming relaxation is denoted *dual bound* if all variables price out correctly. The functions `lowerBound()` and `upperBound()` interpret the primal or dual bound, respectively, depending on the sense of the optimization. An equivalent method is also used for the local bounds of the subproblems.

Reading Parameters

Computer programs in a UNIX environment often use configuration files for the control of certain parameters. Usually, these parameters are stored in the home directory of the user or the directory of the program and start with a '.'. We use a similar concept for reading the parameters of ABACUS. These parameters are read from the file `.abacus`.

However, as ABACUS is a framework for the implementation of different algorithms, there are further requirements for the parameter concept. First, there should be a simple way for reading problem specific parameters. An extendable parameter format should relieve the user of opening and reading his own parameter files. Second, a user of our system might have several applications. It should be possible to specify parameters for different applications and to redefine application dependent parameters defined in the file `.abacus`.

Therefore, we provide the following parameter concept. All parameters read from the file `.abacus` are written into a dictionary. Application specific parameters can be specified in extra parameter files following a very simple format. For files using our parameter format we provide already an input function. The parameters read by this input function are also written to the parameter dictionary. Hence, parameters of the file `.abacus` can be easily redefined. Moreover, we also provide simple functions to extract the values of the parameters from the dictionary.

The parameters in `.abacus` include limits on the resources of the optimization process, control of various strategies (e.g., the enumeration strategy, the branching strategy, zero tolerances for various decisions, the amount of output, parameters for the LP-solver). A detailed list of parameters can be found in Section 5.2.26.

4.2.3 The Subproblem

The class ABA_SUB represents a subproblem of the implicit enumeration, i.e., a node of the branch-and-bound tree. The class subproblem is an abstract class, from which a problem specific subproblem has to be derived. In this derivation process problem specific functions can be added, e.g., for the generation of variables or constraints.

The Root Node of the Branch-and-Bound Tree

For the root node of the optimization the constraint and variable sets can be initialized explicitly. As in many applications the initial variable and constraint sets are in a one-to-one correspondence with the items of the initial variable and constraint pools, we provide this default initialization mechanism. By

default, the first linear program is solved with the barrier method followed by a crossover to a basic solution, but we provide a flexible mechanism for the selection of the LP-method (see Section 5.2.11).

The Other Nodes of the Branch-and-Bound Tree

As long as only globally valid constraints and variables are used it would be correct to initialize the constraint and variable system of a subproblem with the system of the previously processed subproblem. However, ABACUS is designed also for locally valid constraints and variables. Therefore, each subproblem inherits the final constraint and variable system of the father node in the enumeration tree. This system might be modified by the applied branching rule. Moreover, this approach avoids also tedious recomputations and makes sure that heuristically generated constraints do not get lost.

If conventional branching strategies, like setting a binary variable, changing the bounds of an integer variable, or even adding a branching constraint are applied, then the basis of the last solved linear program of the father is still dual feasible. As we store the basis status of the variables and slack variables we can avoid phase 1 of the simplex method if we use the dual simplex method.

If due to another branching method, e.g., for branch-and-price algorithms, the dual feasibility of the basis is lost, another LP-method can be used.

Branch-and-Bound

A linear-programming based branch-and-bound algorithm in its simpliest form is obtained if linear programming relaxations in each subproblem are solved that are neither enhanced by the generation of cutting planes nor by the dynamic generation of variables. Such an algorithm requires only two problem specific functions: one to check if a given LP-solution is a feasible solution of the optimization problem, and one for the generation of the sons.

The first function is problem specific, because, if constraints of the integer programming formulation are violated, the condition that all discrete variables have integer values is not sufficient. Therefore, for safety this function is declared pure virtual.

The second required problem specific function is usually only a one-liner, which returns the problem specific subproblem generated by a branching rule.

Hence, the implementation of a pure branch-and-bound algorithm does not require very much effort.

The Optimization of the Subproblem

The core of the class `ABA_SUB` is its optimization by a cutting plane algorithm. As dynamically generated variables are dual cuts we also use the notion cutting plane algorithm for a column generation algorithm. By default, the cutting plane algorithm only solves the LP-relaxation and tries to fix and set variables by reduced costs. Within the cutting plane algorithm four virtual dummy functions for the separation of constraints, for the pricing of variables, for the application of primal heuristics, and for fixing variables by logical implications are called. In a problem specific class derived from the class `ABA_SUB` these virtual functions can be redefined. Motivated by duality theory (see [Thi95]), we handle constraint and variable generation equivalently. If both constraints and variables are generated, then by default constraints are generated. In addition to the mandatory pricing phase before the fathoming of a subproblem, we price out the inactive variables every k iterations. The value of k can be controlled by a parameter. By the redefinition of a virtual function other strategies for the separation/pricing decision can be implemented.

Adding Constraints

Cutting planes may not only be generated in the function `separate()` but also in other functions of the cutting plane phase. For the maximum cut problem, e.g., it is advantageous if the generation of cutting planes is also possible in the function `improve()`, in which usually primally feasible solutions are computed heuristically. If not all constraints of the integer programming formulation are active, then it might be necessary to solve a separation problem also for the feasibility test. Therefore, we allow the generation of cutting planes in every subroutine of the cutting plane algorithm.

Adding Variables

Like for constraints, we also allow the generation of variables during the complete subproblem optimization.

Buffering New Constraints and Variables

New constraints and variables are not immediately added to the subproblem, but stored in buffers and added at the beginning of the next iteration. We present the details of this concept in Section 4.2.7.

Removing Constraints and Variables

In order to avoid corrupting the linear program and the sets of active constraints and variables, and to allow the removal of variables and constraints in any subroutine of the cutting plane phase, we also buffer these variables and constraints. The removal is executed before constraints and variables are added at the beginning of the next iteration of the cutting plane algorithm.

Moreover, we provide default functions for the removal of constraints according to the value or the basis status of the slack variables. Variables can be removed according to the value of the reduced costs. These operations can be controlled by parameters and the corresponding virtual functions can be redefined if other criteria should be applied. We try to remove constraints also before a branching step is performed.

The Active Constraints and Variables

In order to allow a flexible combination of constraint and variable generation, every subproblem has its own set of active constraints and variables, which are represented by the generic class `ABA_ACTIVE`. By default, the variables and the constraints of the last solved linear program of the father of the subproblem are inherited. Therefore, the local constraint and variable sets speed up the optimization. The disadvantage of these local copies is that more memory is allocated per subproblem. However, this local storage of the active constraints and variables will simplify a future parallelization of the framework.

Together with the active constraints and variables we also store in every subproblem the LP-statuses of the variables and slack variables, the upper and lower bounds of the variables, and if a variable is fixed or set.

The Linear Program

As for active constraints and variables also every subproblem has its own linear program, which is only set up for an active subproblem. Of course, the initialization at the beginning and the deletion of the linear program at the end of the subproblem optimization costs some running time compared to a global linear program, which could be stored in the master. However, a local linear program in every subproblem will again simplify the implementation of a parallel version of ABACUS. Our current computational experience shows that this overhead is not too big. However, if in future computational experiments it turns out that these local linear programs slow down the overall running time significantly, the implementation of a special sequential version of the code with one global linear program will not be too difficult, whereas the opposite direction would be harder to realize.

The LP-Method

Currently, three different methods are available in state of the art LP-solvers: the primal simplex method, the dual simplex method, and the barrier method in combination with crossing over techniques for the determination of an optimal basic solution. The choice of the method can be essential for the performance of solution of the linear program. If a primal feasible basis is available, the primal simplex method is often the right choice. If a dual feasible basis is available, the dual simplex method is usually preferred. And finally, if no basis is known, or the linear programs are very large, often the barrier methods yields the best running times.

Therefore, by default a linear program is solved by the barrier method, if it is the first linear program solved in the root node or constraints and variables have been added at the same time, by the primal simplex method, if constraints have been removed or variables have been added, and by the dual simplex method, if constraints have been added, or variables have been removed, or it is the first linear program of a subproblem.

However, it should be possible to add problem specific decision criteria. Here, again a virtual function gives us all flexibility. We keep control when this function is invoked, namely at the point when all decisions concerning addition and removal of constraints and variables have been taken. The function has as arguments the correct numbers of added and removed constraints and variables. If we want to choose the LP-method problem specifically, then we only have to redefine this function in a class derived from the class `ABA_SUB`.

Generation of Non-Liftable Constraints

If constraint and variable generation are combined, then the active constraints must be lifted if a variable is added, i.e., the column of the new variable must be computed. This lifting can not always be done in a straightforward way, it can even require the solution of another optimization problem. Moreover, lifting is not only required when a variable is added, but this problem has to be attacked already during the solution of the pricing problem.

In order to allow the usage of constraints that cannot be lifted or for which the lifting cannot be performed efficiently, we provide a management of non-liftable constraints. Each constraint has a flag if it is liftable. If the pricing routine is called and non-liftable constraints are active, then all non-liftable constraints are removed, the linear programming relaxation is solved again, and we continue with the cutting plane algorithm before we come back to the pricing phase. In order to avoid an infinite repetition of this process we forbid the further generation of non-liftable constraints during the rest of the optimization of this subproblem.

Reoptimization

If the root of the remaining branch-and-bound tree changes, but the new root has been processed earlier, then it can be advantageous to optimize the corresponding subproblem again, in order to get improved conditions for fixing variables by reduced costs. Therefore, we provide the reoptimization of a subproblem. The difference to the ordinary optimization is that no branching is finally performed even if the subproblem is not fathomed. If it turns out during the reoptimization that the subproblem is fathomed, then we can fathom all subproblems contained in the subtree rooted at this subproblem.

Branching

Virtual functions for the flexible definition of branching strategies are implemented in the class `ABA_SUB`. We explain them together with the concept of branching rules in Section 4.2.7.

If constraints are generated heuristically, then the concept of delayed branching can be advantageous. Instead of generating the sons of a subproblem in a branching step, the subproblem is put back into the set of open subproblems. There it stays several rounds dormant, i.e., other subproblems are optimized in the meantime, until the subproblem is processed again. If between two successive optimizations of the subproblem good cutting planes are generated that can be separated from the pool, then this technique can accelerate the optimization. The maximal numbers of optimizations and the minimal number of dormant rounds can be controlled by parameters.

Memory Allocation

Since constraints and variables are added and removed dynamically, we also provide a dynamic memory management system, which requires no user interaction. If there is not enough memory already allocated to add a constraint or variable, memory reallocations are performed automatically. As the reallocation of the local data, in particular of the linear program, can require a lot of CPU time, if it is performed

regularly, we allocate some extra space for the addition of variables and constraints, and for the nonzero entries of the matrix of the LP-solver.

Activation and Deactivation

In order to save memory we set up those data structures that are only required if the subproblem is active, e.g., the linear program, at the beginning of the subproblem optimization and delete the memory again when the subproblem becomes inactive. We observed that the additional CPU time required for these operations is negligible, but the memory savings are significant.

4.2.4 Constraints and Variables

Constraints and variables are central items within linear-programming based branch-and-bound algorithms. As ABACUS is a system for general mixed integer optimization problems and combinatorial optimization problems we require an abstract concept for the representation of constraints and variables. Linear programming duality motivated us to embed common features of constraints and variables in a joint base class.

Constraint/Variable versus Row/Column

Usually, the notions *constraint* and *row*, and the notions *variable* and *column*, respectively, are used equivalently.

Within ABACUS constraints and rows are different items. Constraints are stored in the pool and a subproblem has a set of active constraints. Only if a constraint is added to the linear program, then the corresponding row is computed. More precisely, a row is a representation of a constraint associated with a certain variable set.

The reasons for this differentiation can be explained with the subtour elimination constraints of the traveling salesman problem, which are defined for subsets W of the nodes of a graph as $x(E(W)) \leq |W|-1$. Storing this inequality as it is added to the linear program would require to store all edges (variables) with both endnodes in the set W. Such a format would require $O(|W|^2)$ storage space. However, it would be also sufficient to store the node set W requiring $0(|W|)$ storage space. Given the variable e associated with the edge (t, h), then the coefficient of e in the subtour elimination constraint is 1 if t and h are contained in W, 0 otherwise.

For the solution of the traveling salesman problem we also want to apply sparse graph techniques. Therefore, storing the coefficients of all active and inactive variables of a subtour elimination constraint would waste a lot of memory. If we store only the coefficients of the variables that are active when the constraint is generated, then the computation of the coefficient of an added variable would be difficult or even impossible. However, if we store all nodes defining the constraint, then the coefficients of variables that are later added can be determined easily.

Efficient memory management and dynamic variable generation are the reason why we distinguish between constraints and rows. Each constraint must have a member function that returns the coefficient for a variable such that we can determine the row corresponding to a set of variables.

In these considerations "constraint" can be also replaced by "variable" and "row" by "column". A column is the representation of a variable corresponding to a certain constraint set. Again, we use the traveling salesman problem as example. A variable for the traveling salesman problem corresponds to an edge in a graph. Hence, it can be represented by its end nodes. The column associated with this variable consists of the coefficients of the edge for all active constraints.

We implemented these concepts in the classes `ABA_CONSTRAINT/ABA_VARIABLE`, which are used for the representation of active constraints and variables and for the storage of constraints and variables in the pools, and `ABA_ROW/ABA_COLUMN`, which are used in particular in the interface to the LP-solver.

This differentiation between constraints/variables and rows/columns is not used by any other system for the implementation of linear-programming based branch-and-bound algorithms, because they are usually designed for the solution of general mixed integer optimization problems, which do not necessarily

require this distinction. However, this concept is crucial for a practically efficient and simple application of ABACUS to combinatorial optimization problems.

Common Features of Constraints and Variables

Constraints and variables have several common features, which we consider in a common base class.

A constraint/variable is active if it belongs to the constraint/variable set of an active subproblem. An active constraint/variable must not be removed from its pool. As in a parallel implementation of ABACUS there can be several active subproblems, each constraint/variable has a counter for the number of active subproblems, in which it is active.

Besides being active there can be other reasons why a constraint/variable should not be deleted from its pool, e.g., if the constraint/variable has just been generated, then it is put into a buffer, but is not yet activated (we explain the details in Section 4.2.7). In such a case we want to set a lock on the constraint that it cannot be removed. Again, in a parallel implementation, but also in a sequential one, we may want to set locks at the same time on the same constraint for different reasons. Hence, we count the number of locks of each constraint/variable.

Constraints and variables can be locally or globally valid. Therefore, we provide a flag in the common base class of constraints and variables. The functions to determine if a local constraint or variable is valid for a certain subproblem are associated directly with the classes for constraints and variables, respectively.

It has been stated that the use of locally valid constraints and variables should be avoided as it requires a nasty bookkeeping [PR91]. In order to free the user from this, we have embedded the management of local constraints and variables in ABACUS. The validity of a constraint/variable is automatically checked if it is regenerated from the pool.

We also distinguish between dynamic variables/constraints and static ones. As soon as a static variable/constraint becomes active it cannot be deactivated. An example for static variables are the variables in a general mixed integer optimization problem, examples for static constraints are the constraints of the problem formulation of a general mixed integer optimization problem or the degree constraints of the traveling salesman problem. Dynamic constraints are usually cutting planes. In column generation algorithm variables can be dynamic, too.

A crucial point in the implementation of a special variable or constraint class is the tradeoff between performance and memory usage. We have observed that a memory efficient storage format can be one of the keys to the solutions of larger instances. Such formats are in general not very useful for the computation of the coefficient of a single variable/constraint. Moreover, if the coefficients of a constraint for several variables or the coefficients of a variable for several constraints have to be computed, e.g., when the row/column format of the constraint/variable is generated in order to add it to the LP-solver, then these operations can become a bottleneck. However, given a different format, using more memory, it might be possible to perform these operations more efficiently.

Therefore, we distinguish between the compressed format and the expanded format of a constraint/variable. Before a bigger number of time consuming coefficient computations is performed, we try to generate the expanded format, afterwards the constraint/variable is compressed.

Of course, both expanded and compressed formats are rather constraint/variable specific. But we provide the bookkeeping already in the common base class and try to expand the constraint/variable, e.g., when it is added to the linear program. Afterwards it is compressed again. The implementation of the expansion and compression is optional.

We use again the subtour elimination constraint of the traveling salesman problem as an example for the compressed and expanded format. For an inequality $x(E(W)) \leq |W|-1$ we store the nodes of the set W in the compressed format. The computation of the coefficient of an edge (t,h) requires $O(|W|)$ time and space. As expanded format we use an array `inSubtour` of type `bool` of length n (n is the number of nodes of the graph) and `inSubtour[v]` is `true` if and only if $v \in W$. Now, we can determine the coefficient of an edge (variable) in constant time.

Constraints

ABACUS provides all three different types of constraints: equations, $\leq$-inequalities and $\geq$-inequalities.

The only pure virtual function is the computation of a coefficient of a variable. We use this function to generate the row format of a constraint, to compute the slack of an LP-solution, and to check if an LP-solution violates a constraint. All these functions are declared virtual such that they can be redefined for performance reasons.

We distinguish between locally and globally valid constraints. By default, a locally valid constraint is considered to be valid for the subproblem it was generated and for all subproblems in the tree rooted at this subproblem. This criterion is implemented in a virtual function such that it can be redefined for special constraints.

If variables are generated dynamically, we distinguish between liftable and non-liftable constraints. Non-liftable constraints have to be removed before the pricing problem can be solved (see Section 4.2.3).

ABACUS provides a default non-abstract constraint class with the class `ABA_ROWCON`, where a constraint is represented by its row format, i.e., only the numbers of variables with nonzero coefficients and the corresponding coefficients are stored. This format is useful, e.g., for constraints of general mixed integer optimization problems. From the class `ABA_ROWCON` we derive the class `ABA_SROWOCN`, which implements some member functions more efficiently as it assumes that the variable set is static, i.e., no variables are generated dynamically.

Variables

ABACUS supports continuous, integer, and binary variables in the class `ABA_VARIABLE`. Each variable has a lower and an upper bound, which can be set to plus/minus infinity if the variable is unbounded. We also memorize if a variable is fixed.

The following functions have their dual analogons in the class `ABA_CONSTRAINT`. The only pure virtual function is now the function that returns a coefficient in a constraint. With this function the generation of the column format and the computation of the reduced cost can be performed. We say a variable is violated if it does not price out correctly.

Also variables can be locally or globally valid. A subproblem is by default associated with a locally valid variable. The variable is then valid in all subproblems on the path from this subproblem to the root node. Of course, this virtual function can be redefined for problem specific variables.

We provide already a non-abstract derived variable class. The class `ABA_COLVAR` implements a variable that is represented by the column format, i.e., only the nonzero coefficients together with the numbers of the corresponding rows are stored.

4.2.5 Constraint and Variable Pools

Every constraint and variable either induced by the problem formulation or generated in a separation or pricing step is stored in a pool. A pool is a collection of constraints and variables. We will see later that it is advantageous to keep separate pools for variables and constraints. Then, we will also discuss when it is useful to have also different pools for different types of constraints or variables. But for simplicity we assume now that there is only one variable pool and one constraint pool.

There are two reasons for the usage of pools: saving memory and an additional separation/pricing method.

A constraint or variable usually belongs to the set of active constraints or variables of several subproblems that still have to be processed. Hence, it is advantageous to store in the sets of active constraints or variables only pointers to each constraint or variable, which is stored at some central place, i.e., in a pool that is a member of the corresponding master of the optimization. Our practical experiments show that this memory sensitive storage format is of very high importance, since already this pool format uses a large amount of memory.

Pool Separation/Pricing

From the point of view of a single subproblem a pool may not only contain active but also inactive constraints or variables. The inactive items can be checked in the separation or pricing phase, respectively. We call these techniques pool separation and pool pricing. Again, motivated by duality theory we use the

notion "separation" also for the generation of variables, i.e., for pricing. Pool separation is advantageous in two cases. First, pool separation might be faster than the direct generation of violated constraints or variables. In this case, we usually check the pool for violated constraints or variables, and only if no item is generated, we use the more time consuming direct method. Second, pool separation turns out to be advantageous, if a class of constraints or variables can be separated/priced out only heuristically. In this case, it can happen that the heuristic cannot generate the constraint or variable although it is violated. However, earlier in the optimization process this constraint or variable might have been generated. In this case the constraint or variable can be easily regenerated from the pool. Computational experiments show that this additional separation or pricing method can decrease the running time significantly [JRT94].

During the regeneration of constraints and variables from the pools we also have to take into account that a constraint or variable might be only locally valid.

The pool separation is also one reason for using different pools for variables and constraints. Otherwise, each item would require an additional flag and a lot of unnecessary work would have to be performed during the pool separation.

Pool separation is also one of the reasons why it can be advantageous to provide several constraint or variable pools. Some constraints, e.g., might be more important during the pool separation than other constraints. In this case, we might check this "important" pool first and only if we fail in generating any item we might proceed with other pools or continue immediately with direct separation techniques.

Other classes of constraints or variables might be less important in the sense that they cannot or can only very seldomly be regenerated from the pool (e.g., locally valid constraints or variables). Such items could be kept in a pool that immediately removes all items that do not belong to the active constraint or variable set of any subproblem which still has to be processed. A similar strategy might be required for constraints or variables requiring a big amount of memory.

Finally, there are constraints for which it is advantageous to stay active in any case (e.g., the constraints of the problem formulation in a general mixed integer optimization problem, or the degree constraints for the traveling salesman problem). Also for these constraints separate pools are advantageous.

Garbage Collection

In any case, as soon as a lot of constraints or variables are generated dynamically we can observe that the pools become very, very large. In the worst case this might cause an abnormal termination of the program if it runs out of memory. But already earlier the optimization process might be slowed down since pool separation takes too long. Of course, the second point can be avoided by limited strategies in pool separation, which we will discuss later. But to avoid the first problem we require suitable cleaning up and garbage collection strategies.

The simplest strategy is to remove all items belonging not to any active variable or constraint set of any active or open subproblem in a garbage collection process. The disadvantage of this strategy might be that good items are removed that are accidentally momentarily inactive. A more sophisticated strategy might be counting the number of linear programs or subproblems where this item has been active and removing initially only items with a small counter.

Unfortunately, if the enumeration tree grows very large or if the number of constraints and variables that are active at a single subproblem is high, then even the above brute force technique for the reduction of a pool turns out to be insufficient.

Hence, we have to divide constraints and variables into two groups. On the one hand the items that must not be removed from the pool, e.g., the constraints and variables of the problem formulation of a general mixed integer optimization problem, and on the other hand those items that can either be regenerated in the pricing or separation phase or are not important for the correctness of the algorithm, e.g., cutting planes. If we use the data structures we will describe now, then we can remove safely an item of the second group.

Pool Slots

So far, we have assumed that the sets of active variables or constraints store pointers to variables or constraints, respectively, which are stored in pools. If we would remove the variable or constraint, i.e.,

delete the memory we have allocated for this object, then errors can occur if we access the removed item from a subproblem. These fatal errors could be avoided if a message is sent to every subproblem where the deleted item is currently active. This technique would require additional memory and running time. Therefore, we propose a data structure that can handle this problem very simply and efficiently.

A pool is not a collection of constraints or variables, but a collection of pool slots (class ABA_POOLSLOT). Each slot stores a pointer to a constraint or variable or a 0-pointer if it is void. The sets of active constraints or variables store pointers to the corresponding slots instead of storing pointers to the constraints or variables directly. So, if a constraint or variable has been removed a 0-pointer will be found in the slot and the subproblem recognizes that the constraint or variable must be eliminated since it cannot be regenerated. The disadvantage of this method is that finally our program may run out of memory since there are many useless slots.

In order to avoid this problem we add a version number as data member to each pool slot. Initially the version number is 0 and becomes 1 if a constraint or variable is inserted in the slot. After an item in a slot is deleted a new item can be inserted into the slot. Each time a new item is stored in the slot the version number is incremented. The sets of active constraints and variables do not only store pointers to the corresponding slots but also the version number of the slot when the pointer is initialized. If a member of the active constraints or variables is accessed we compare its original and current version number. If these numbers are not equal we know that this is not the constraint or variable we were originally pointing to and remove it from the active set. We call the data structure storing the pointer to the pool slot and the original version number a reference to a pool slot (class ABA_POOLSLOTREF). Hence, the sets of active constraints and variables are arrays of references to pool slots. We present an example for this pool concept in Figure 4.1. The numbers in the boxes are arbitraryly chosen version numbers.

Standard Pool

The class ABA_POOL is an abstract class, which does not specify the storage format of the collection of pool slots. The simplest implementation is an array of pool slots. The set of free pool slots can be implemented by a linked list. This concept is realized in the class ABA_STANDARDPOOL. Moreover, a ABA_STANDARDPOOL can be static or dynamic. A dynamic ABA_STANDARDPOOL is automatically enlarged, when it is full, an item is inserted, and the cleaning up procedure fails. A static ABA_STANDARDPOOL has a fixed size and no automatic reallocation is performed.

More sophisticated implementations might keep an order of the pool slots such that "important" items are detected earlier in a pool separation such that a limited pool separation might be sufficient. A criterion for this order could be the number of subproblems where this constraint or variable is active or has been active. We will consider such a pool in a future release.

Default Pools

The number of the pools is very problem specific and depends mainly on the separation and pricing methods. Since in many applications a pool for variables, a pool for the constraints of the problem formulation, and a pool for cutting planes are sufficient, we implemented this default concept. If not specified differently, in the initialization of the pools, in the addition of variables and constraints, and in the pool pricing and pool separation these default pools are used. We use a static ABA_STANDARDPOOL for the default constraint and cutting planes pools. The default variable pool is a dynamic ABA_STANDARDPOOL, because the correctness of the algorithm requires that a variable which does not price out correctly can be added in any case, whereas the loss of a cutting plane that cannot be added due to a full pool has no effect on the correctness of the algorithm as long as it does not belong to the integer programming formulation.

If instead of the default pool concept an application specific pool concept is implemented, then the user of the framework must make sure that there is at least one variable pool and one constraint pool and these pools are embedded in a class derived from the class ABA_MASTER.

With this concept we provide a high flexibility: An easy to use default implementation, which can be changed by the redefinition of virtual functions and the application of non-default function arguments.

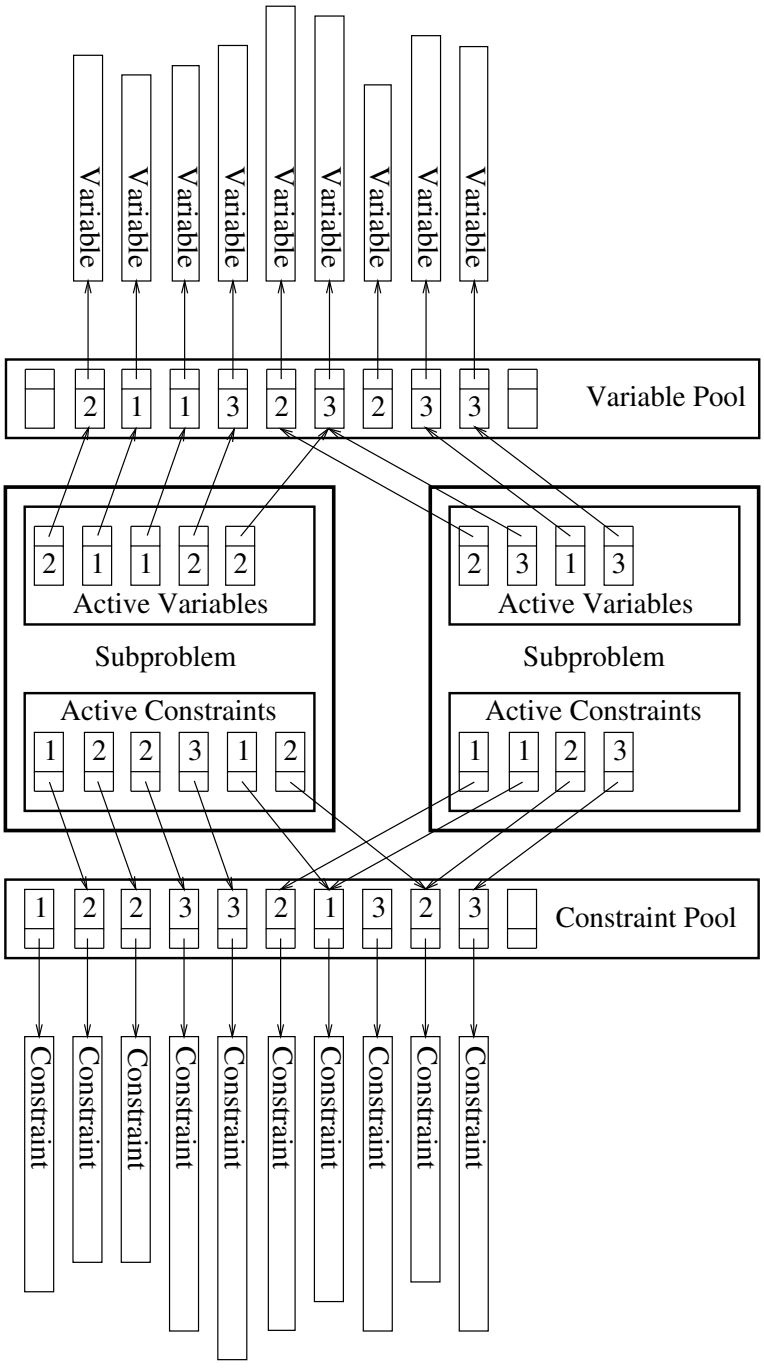

Figure 4.1: The pool concept.

All classes involved in this pool concept are designed as generic classes such that they can be used both for variables and constraints.

4.2.6 Linear Programs

Since ABACUS is a framework for the implementation of linear-programming based branch-and-bound algorithms it is obvious that the solution of linear programs plays a central role, and we require a class concept of the representation of linear programs. Moreover, linear programs might not only be used for the solution of LP-relaxations in the subproblems, but they can also be used for other purposes, e.g., for the separation of lift-and-project cutting planes of zero-one optimization problems [BCC93a, BCC93b] and within heuristics for the determination of good feasible solutions in mixed integer programming [HP93].

Therefore, we would like to provide two basic interfaces for a linear program. The first one should be in a very general form for linear programs defined by a constraint matrix stored in some sparse format. The second one should be designed for the solution of the LP-relaxations in the subproblem. The main differences to the first interface are that the constraint matrix is stored in the abstract `ABA_VARIABLE/ABA_CONSTRAINT` format instead of the `ABA_COLUMN/ABA_ROW` format and that fixed and set variables should be eliminated.

Another important design criterion is that the solution of the linear programs should be independent from the used LP-solver, and plugging in a new LP-solver should be simple.

The Basic Interface

The result of these requirements is the class hierarchy of Figure 4.2. The class `ABA_LP` is an abstract base class providing the public functions that are usually expected: initialization, optimization, addition of rows and columns, deletion of rows and columns, access to the problem data, the solution, the slack variables, the reduced costs, and the dual variables. These functions do some minor bookkeeping and call a pure virtual function having the same name but starting with an underscore (e.g, `optimize()` calls `_optimize`). These functions starting with an underscore are exactly the functions that have to be implemented by an LP-solver.

The LP-Solver Cplex

The class `ABA_CPLEXIF` implements these solver specific functions for the LP-solver Cplex [Cpl94]. If a linear program should be solved with Cplex, an object of the class `ABA_CPLEXIF` is instantiated. As long as only public members that are inherited from the class `ABA_LP` are used except the constructors (which is usually sufficient) using another LP-solver means only replacing the name `ABA_CPLEXIF` by its name in the instantiation after a similar class for this solver as the class `ABA_CPLEXIF` has been implemented.

The advantage of encapsulating the implementation in an extra class instead of using only the private functions is that several LP-solvers can be made available within the framework and changing the solver means only changing the name in the instantiation.

The LP-Solver SoPlex

In an equivalent way the class `ABA_SOPLEXIF` implements an interface to the LP-solver SoPlex.

Linear Programming Relaxations

The most important linear programs being solved within this system are the LP-relaxations solved in the optimization of the subproblems. However, the active constraints and variables of a subproblem are not stored in the format required by the class `ABA_LP`. Therefore, we have to implement a transformation from the `ABA_VARIABLE/ABA_CONSTRAINT` format to the `ABA_COLUMN/ABA_ROW` format.

Two options are available for the realization of this transformation: either it can be implemented in the class `ABA_SUB` or in a new class derived from the class `ABA_LP`. We decided to implement such an

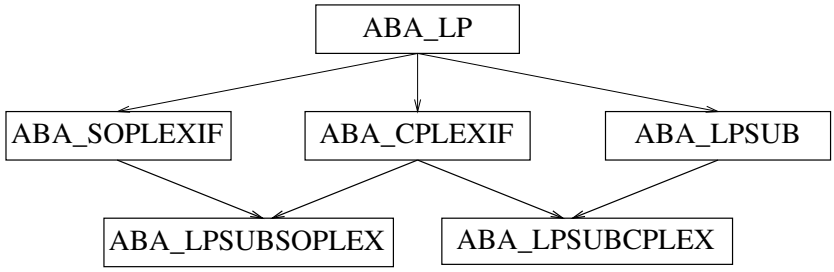

Figure 4.2: The linear programming classes.

interface class, which we call ABA_LPSUB, for the following reasons. First, the interface is better structured. Second, the subproblem optimization becomes more robust for later modifications of the class ABA_LP. Third, we regard the class ABA_LPSUB as a preprocessor for the linear programs solved in the subproblem, because fixed and set variables can be eliminated from the linear program submitted to the solver. It depends on the used solution method if all fixed and set variables should be eliminated. If the simplex method is used and a basis is known, then only nonbasic fixed and set variables should be eliminated. If the barrier method is used we can eliminate all fixed and set variables. The encapsulation of the interface between the subproblem and the class ABA_LP supports a more flexible adaption of the elimination to other LP-solvers in the future and also enables us to use other LP-preprocessing techniques, e.g., constraint elimination, or changing the bounds of variables under certain conditions (see [Sav94]), without modifying the variables and constraints in the subproblem. Preprocessing techniques other than elimination of fixed and set variables are currently not implemented.

Solving Linear Programming Relaxations

The subproblem optimization in the class ABA_SUB uses only the public functions of the class ABA_LPSUB, which is again an abstract class independent from the used LP-solver. Currently the linear programs can be solved either by Cplex or SoPlex.

A linear program solving the relaxations within a subproblem with the LP-solver Cplex is defined by the class ABA_LPSUBCPLEX, which is derived from the classes ABA_LPSUB and ABA_CPLEXIF. The class ABA_LPSUBCPLEX only implements a constructor passing the arguments to the base classes. Using a different LP-solver in this context requires the definition of a class equivalent to the class ABA_LPSUBCPLEX and a redefinition of the virtual function ABA_LPSUB *ABA_SUB::generateLp(), which is a one-line function allocating an object of the class ABA_LPSUBCPLEX and returning a pointer to this object.

This concept is used for implementing interfaces to the LP-solver SoPlex by the class ABA_SOPLEXIF (the equivalent of ABA_CPLEXIF) and the class ABA_LPSUBSOPLEX (the equivalent of ABA_SUBCPLEX).

Therefore, it is easy to use different LP-solvers for different ABACUS applications and it is also possible to use different LP-solvers in a single ABACUS application. For instance, if there is a very fast method for the solution of the linear programs in the root node of the enumeration tree, but all other linear programs should be solved by Cplex, then only a simple modification of ABA_SUB::generateLp() is required.

To avoid multiple instances of the class ABA_LP in objects of the class ABA_LPSUBCPLEX or ABA_LPSUBSOPLEX, the classes ABA_CPLEXIF, ABA_SOPLEXIF, and ABA_LPSUB are virtually derived from the class ABA_LP. In order to save memory we do not make copies of the LP-data in any of the classes of this hierarchy except for the data that is passed to the LP-solvers in the classes ABA_CPLEXIF and ABA_SOPLEXIF.

4.2.7 Auxiliary Classes for Branch-and-Bound

In this section we are going to discuss the design of some important classes that support the linear-programming based branch-and-bound algorithm. These are classes for the management of the open subproblems, for buffering newly generated constraints and variables, for the implementation of branching

rules, for the candidates for fixing variables by reduced costs, for the control of the tailing off effect, and for the storage of a solution history.

The Set of Open Subproblems

During a branch-and-bound algorithm subproblems are dynamically generated in branching steps and later optimized. Therefore, we require a data structure that stores pointers to all unprocessed and dormant subproblems and supports the insertion and the extraction of a subproblem.

One of the important issues in a branch-and-bound algorithm is the enumeration strategy, i.e., which subproblem is extracted from the set of open subproblems for further processing. It would be possible to implement the different classical enumeration strategies, like depth-first search, breadth-first search, or best-first search within this class. But in this case, an application-specific enumeration strategy could not be added in a simple way by a user of ABACUS. Of course, with the help of inheritance and virtual functions a technique similar to the one we implemented for the usage of different LP-solvers for the subproblem optimization could be applied. However, there is a much simpler solution for this problem.

In the class `ABA_MASTER` we define a virtual member function that compares two subproblems according to the selected enumeration strategy and returns 1 if the first subproblem has higher priority, -1 if the second one has higher priority, and 0 if both subproblems have equal priority. Application specific enumeration strategies can be integrated by a redefinition of this virtual function. In order to compare two subproblems within the extraction operation of the class `ABA_OPENSUB` this comparison function of the associated master is called.

The class `ABA_OPENSUB` implements the set of open subproblems as a doubly linked linear list. Each time when another subproblem is required for further processing the complete list is scanned and the best subproblem according to the applied enumeration strategy is extracted. This implementation has the additional advantage, that it is very easy to change the enumeration strategy during the optimization process, e.g., to perform a diving strategy, which uses best-first search but performs a limited depth first search every k iterations.

The drawback of this implementation is the linear running time of the extraction of a subproblem. If the set of open subproblems would be implemented by a heap, then the insertion and the extraction of a subproblem would require logarithmic time, whereas in the current implementation the insertion requires constant, but the extraction requires linear time. But if the enumeration strategy is changed, the heap has to be reinitialized from scratch, which requires linear time.

However, it is typical for a linear-programming based branch-and-bound algorithm that a lot of work is performed in the subproblem optimization, but the total number of subproblems is comparatively small. Also the performance analysis of our current applications shows that the running time spent in the management of the set of open subproblems is negligible. Due to the encapsulation of the management of the set of open subproblem in the private part of the class `ABA_OPENSUB`, it will be no problem to change the implementation, as soon as it is required.

In order to allow the special fathoming technique for fathoming more than one subproblem in case of a contradiction (even though it is currently not implemented), the class `ABA_OPENSUB` supports also the removal of an arbitrary subproblem. This operation cannot be performed in logarithmic time in a heap, but requires linear time. A data structure providing logarithmic running time for the insertion, extraction of the minimal element, and removal of an arbitrary element is, e.g., a red-black tree [Bay72, GS78]. According to our current experience it seems that the implementation effort for these enhanced data structures does not pay.

We provide four rather common enumeration strategies per default: best-first search, breadth-first search, depth-first search, and a simple diving strategy performing depth-first search until the first feasible solution is found and continuing afterwards with best-first search.

If the branching strategy is branching on a binary variable, then these default enumeration strategies are further refined. We can often observe in mixed integer programming that feasible solutions are sparse, i.e., only a small number of variables have a nonzero value. Setting a binary variable to one may induce a subproblem with a smaller number of feasible solutions than for its brother in which the branching variable is set to zero. Therefore, if two subproblems have the same priority in the enumeration, we

prefer that one with the branching variable set to one. This resolution of subproblems having equal priority is performed in a virtual function, such that it can be adapted to each specific application or can be extended to other branching strategies.

Buffering Generated Variables and Constraints

Usually, new constraints are generated in the separation phase. However, it is possible that in some applications violated constraints are also generated in other subroutines of the cutting plane algorithm. In particular, if not all constraints of the integer programming formulation are active in the subproblem a separation routine might have to be called to check the feasibility of the LP-solution. Another example is the maximum cut problem, for which it is rather convenient if new constraints can also be generated while we try to find a better feasible solution after the linear program has been solved. Therefore, it is necessary that constraints can be added by a simple function call from any part of the cutting plane algorithm.

This requirement also holds for variables. For instance, when we perform a special rounding algorithm on a fractional solution during the optimization of the traveling salesman problem, we may detect useful variables that are currently inactive. It should be possible to add such important variables before they may be activated in a later pricing step.

It can happen that too many variables or constraints are generated such that it is not appropriate to add all of them, but only the "best" ones. Measurements for "best" are difficult, for constraints this can be the slack or the distance between the fractional solution and the associated hyperplane, for variables this can be the reduced costs.

Therefore, we have implemented a buffer for generated constraints and variables in the generic class `ABA_CUTBUFFER`, which can be used both for variables and constraints. There is one object of this class for buffering variables, the other one for buffering constraints. Constraints and variables that are added during the subproblem optimization are not added directly to the linear program and the active sets of constraints and variables, but are added to these buffers. The size of the buffers can be controlled by parameters. At the beginning of the next iteration items out of the buffers are added to the active constraint and variable sets and the buffers are emptied. An item added to a buffer can receive an optional rank given by a floating point number. If all items in a buffer have a rank, then the items with maximal rank are added. As the rank is only specified by a floating point number, different measurements for the quality of the constraints or variables can be applied. The number of added constraints and variables can be controlled again by parameters.

If an item is discarded during the selection of the constraints and variables from the buffers, then usually it is also removed from the pool and deleted. However, it may happen that these items should be kept in the pool in order to regenerate them in later iterations. Therefore, it is possible to set an additional flag while adding a constraint or variable to the buffer that prevents it from being removed from the pool if it is not added. Constraints or variables that are regenerated form a pool receive this flag automatically.

Another advantage of this buffering technique is that adding a constraint or variable does not change immediately the current linear program and active sets. The update of these data structures is performed at the beginning of the cutting plane or column generation algorithm before the linear program is solved. Hence, this buffering method together with the buffering of removed constraints and variables relieves us also from some nasty bookkeeping.

Branching

It should be possible that in a framework for linear-programming based branch-and-bound algorithms many different branching strategies can be embedded. Standard branching strategies are branching on a binary variable by setting it to 0 or 1, changing the bounds of an integer variable, or splitting the solution space by a hyperplane such that in one subproblem $a^T x \geq \beta$ and in the other subproblem $a^T x \leq \beta$ must hold. A straightforward generalization is that instead of one variable or one hyperplane we use k variables or k hyperplanes, which results in 2^k-nary enumeration tree instead of a binary enumeration tree.

Another branching strategy is branching on a set of equations ${a_1}^T x = \beta_1, \ldots, {a_l}^T x = \beta_l$. Here, l new subproblems are generated by adding one equation to the constraint system of the father in each case. Of course, as for any branching strategy, the complete set of feasible solutions of the father must be covered by the sets of feasible solutions of the generated subproblems.

For branch-and-price algorithms often different branching rules are applied. Variables not satisfying the branching rule have to be eliminated and it might be necessary to modify the pricing problem. The branching rule of Ryan and Foster [RF81] for set partitioning problems also requires the elimination of a constraint in one of the new subproblems.

So it is obvious that we require on the one hand a rather general concept for branching, which does not only cover all mentioned strategies, but should also be extendable to "unknown" branching methods.

On the other hand it should be simple for a user of the framework to adapt an existing branching strategy like branching on a single variable by adding a new branching variable selection strategy.

Again, an abstract class is the basis for a general branching scheme, and overloading a virtual function provides a simple method to change the branching strategy. We have developed the concept of branching rules. A branching rule defines the modifications of a subproblem for the generation of a son. In a branching step as many rules as new subproblems are instantiated. The constructor of a new subproblem receives a branching rule. When the optimization of a subproblem starts, the subproblem makes a copy of the member data defining its father, i.e., the active constraints and variables, and makes the modifications according to its branching rule.

The abstract base class for different branching rules is the class `ABA_BRANCHRULE`, which declares a pure virtual function modifying the subproblem according to the branching rule. We have to declare this function in the class `ABA_BRANCHRULE` instead of the class `ABA_SUB` because otherwise adding a new branchrule would require a modification of the class `ABA_SUB`.

We derive from the abstract base class `ABA_BRANCHRULE` classes for branching by setting a binary variable (class `ABA_SETBRANCHRULE`), for branching by changing the lower and upper bound of an integer variable (class `ABA_BOUNDBRANCHRULE`), for branching by setting an integer variable to a value (class `ABA_VALBRANCHRULE`), and branching by adding a new constraint (class `ABA_CONBRANCHRULE`).

This concept of branching rules should allow almost every branching scheme. Especially, it is independent of the number of generated sons of a subproblem. Further branching rules can be implemented by deriving new classes from the class `ABA_BRANCHRULE` and defining the pure virtual function for the corresponding modification of the subproblem.

In order to simplify changing the branching strategy we implemented the generation of branching rules in a hierarchy of virtual functions of the class `ABA_SUB`. By default, the branching rules are generated by branching on a single variable. If a different branching strategy should be implemented a virtual function must be redefined in a class derived from the class `ABA_SUB`.

Often in a special branch-and-cut algorithm we only want to modify the branching variable selection strategy. A new branching variable selection strategy can be implemented again by redefining a virtual function.

Candidates for Fixing

Each time when all variables price out correctly during the processing of the root node of the branch-and-bound tree, we store those nonbasic variables that cannot be fixed together with their statuses, reduced costs, and the current dual bound in an object of the class `ABA_FIXCAND`. Later, when the primal bound improves, we can try to fix these variables by reduced cost criteria. This data structure can also be updated if the root node of the remaining branch-and-bound tree changes.

Tailing Off

We implemented the class `ABA_TAILOFF` to memorize the values of the last solved linear programs of a subproblem to control the tailing off effect. An instance of this class is a member of each subproblem.

Solution History

The class ABA_HISTORY stores the solution history, i.e., it memorizes the primal and the dual bound and the current time whenever a new primal or dual bound is found.

4.2.8 Basic Generic Data Structures

We have implemented several basic data structures as templates. We only sketch these classes briefly. For the details on the implementations we refer to text books about algorithms and data structures such as, e.g., [CLR90].

Arrays

Arrays are already supported by C++ as in C. To provide in addition to the subscript operator [] simpler construction, destruction, reallocation, copying, and assignment we have implemented the class ABA_ARRAY.

A Buffer for Objects

Arrays are frequently used for buffering data, i.e., there is an additional counter that is initially 0. Then, objects are inserted in the array at the position of the counter, which is afterwards incremented. In order to simplify such buffering operations we have encapsulated an array together with the counter in the class ABA_BUFFER.

Actually, a buffer is a special array such that the class ABA_BUFFER should be derived from the class ABA_ARRAY. Unfortunately, the version of the GNU-compiler we were using when we developed this part of the system had a bug in the inheritance of templates. In a future release we will derive this class from ABA_ARRAY.

Bounded Stack

A stack stores a set of elements according to the last-in-first-out principle, i.e., only the last inserted element can be accessed or removed. A linked list could implement such a data structure in which an unlimited number of elements (limited only by the available memory) can be inserted. We would have to sacrifice some efficiency for this flexibility. Therefore, we use an array for implementing a stack having a maximal size. If it turns out that the initial estimation on the maximal size is too small, the stack can be reallocated.

Ring

A ring is a collection of elements that has a maximal size. If this maximal size is reached but a new element is inserted, then the oldest element is replaced. No element can be removed explicitly from the ring except that the ring can be emptied in a single step. The class ABA_RING implements such a data structure with an array. We need a ring in the framework to memorize the last k (e.g., $k = 10$) values of the LP-solution in the subproblem optimization, in order to control the tailing off effect. Since this data structure might be useful for other purposes we implemented it as a template.

Linked Lists

The classes ABA_LIST and ABA_DLIST provide implementations of a linked list and a doubly linked list, respectively.

Bounded Heap

A heap is a data structure representing a complete binary tree, where each node satisfies the so called "heap-property". For similar efficiency reasons, we discussed already in the context of the stack, we provide an implementation ABA_BHEAP of a heap with limited size by an array.

Bounded Priority Queue

A priority queue is a data structure storing a set of elements where each element is associated with a key. The priority queue provides the operations inserting an element, finding the element with minimal key, and extracting the element with minimal key. We provide an implementation `ABA_BPRIOQUEUE` of a priority queue of limited size with the help of a heap.

Hash Table

In a hash table a set of elements is stored by computing for each element the address in the table via a hash function applied to the key of the element. As the number of possible values of keys is usually much greater than the number of addresses in the table we require techniques for resolving collisions, i.e., if more than one element is mapped to the same address.

We use direct addressing and collision resolution by chaining in the class `ABA_HASH`. For integer keys we implemented the Fibonacci hash function and for strings a hash function proposed in [Knu93].

Dictionary

A dictionary in our context is a data structure storing elements together with some additional data. Besides the insertion of an element we provide a look up operation returning the data associated with an element. For the implementation of the class `ABA_DICTIONARY` we use a hash table.

4.2.9 Other Basic Data Structures

In this section we shortly outline some other basic data structures, which are not implemented as templates. These are classes for the representation of sparse vectors, for sparse graphs, for strings, and for disjoint sets.

Sparse Vector

Typically, mixed integer optimization problems have a constraint matrix with a very small number of nonzero elements. Storing also the zero elements of constraints would waste a lot of memory and increase the running time. Therefore, we implemented in the class `ABA_SPARVEC`, a data structure which stores only the non-zero elements of a vector together with their coefficients in two arrays. With this implementation the critical operation is the determination of the coefficient of an original component.

In the worst case, i.e., if the coefficient is zero, the complete array must be scanned. However, in a performance analysis of our current applications it turns out that more sophisticated implementations using sorted elements such that binary search can be performed or using hash tables are not necessary.

To simplify the dynamic insertion of elements, which is very common within this software system, an automatic reallocation is performed if the arrays implementing the sparse vector are full. By default, the arrays are increased by ten percent but this value can be changed in the constructor.

We use the class `ABA_SPARVEC` mainly as base class of the classes `ABA_ROW` and `ABA_COLUMN`, which are essential in the interface to the LP-solver and also used for the implementation of special types of constraints and variables.

String

We also implement the class `ABA_STRING` for the representation of character strings. We provide only those member functions which are currently required in our software system. This class still requires extensions in the future.

Disjoint Sets

We provide the classes `ABA_SET` and `ABA_FASTSET` for maintaining disjoint sets represented by integer numbers. The operations for generating a set, union of sets, and finding the representative of the set the element is contained in are effciently supported.

4.2.10 Tools

The following classes are not data structures in a narrow sense, but provide useful tools for the management of the output, for measuring time, and for sorting.

Output Streams

A framework like ABACUS requires different levels of output. A lot of information is required during the development and debugging phase of an application, only some information on the progress of the solution process and the final results are desired in ordinary runs, and finally there should be no output at all if an application of ABACUS is used as a subprogram.

In order to satisfy these requirements usually output statements are either enclosed in

```
#ifdef
...
#endif
```

preprocessor instructions or each output statement is preceded by a statement of the form

```
if (outLevel == ...)
```

We rejected the first method immediately since changing the output level would require a recompilation of the code. The second method has the drawback that all these if-statements before output operations are not very nice and make the source code less readable.

Therefore, we make use of the C++ output streams and derive from the class `ostream` of the i/o-stream library a class `ABA_OSTREAM` implementing a specialized output stream that can be turned on and off. More precisely, we can apply the output operator `<<` as usual, but write to an object of the class `ABA_OSTREAM` instead of `ostream`. If the output should be suppressed, we call a member function to turn it off. If output is desired again later in the program, it can be turned on again. The class `ABA_OSTREAM` is a filter in this context for an output stream of the class `ostream` that can be turned on and off at any time.

The disadvantage of this filter is that if at a certain output level one output statement should pass, the next one should be filtered out, etc., then a lot of code has to be inserted in the program for turning the output on and off, which leads to a less readable code than the classical remedy.

However, we observed that for ABACUS a rather simple structure of output levels and output statements is sufficient. Between the two extreme cases that no output is generated (*Silent*) and a lot of output is produced (*Full*) there are only three levels supported. On each level in addition to all output of the preceding levels some extra information is given. After the level *Silent* follows the level *Statistics* generating only some statistical information at the end of the run. At the level *Subproblem* a short information on the status of the optimization is output at the end of each subproblem optimization. Finally, at the output level *LinearProgram* similar output is generated after every solved linear program.

Therefore, turning the output streams on and off is required very seldomly within ABACUS and its applications such that this concept improves the readability of the code.

Under the operating system UNIX output written to `cout` can be redirected to a file. Unfortunately, in this case no output is visible on the screen. Therefore, we have implemented in the class `ABA_OSTREAM` also the option to generate a log-file. If this option is chosen output is both written on the screen and to the log-file. This effect can also be obtained by using the UNIX command `tee`. However, the output levels for the log-file and the standard output may be different, e.g., one can choose output level *Subproblem* for the standard output stream to monitor the optimization process, but *Full* output on the log-file for a later analysis of the run.

Of course, several instances of the class `ABA_OSTREAM` can be used. The default version of ABACUS uses one for the normal output messages, i.e., as filter for `cout` and one for the warning and error messages, i.e., as filter for `cerr`. In an application it is possible to introduce another output stream for the problem specific output.

Timers

For a simple measurement of the CPU and the wall-clock time of parts of the program we implemented the classes `ABA_TIMER`, `ABA_CPUTIMER`, `ABA_COWTIMER`. The `ABA_TIMER` is the base class of the two other classes and provides the basic functionality of a timer, like starting, stopping, resetting, output, retrieving the time, and checking if the current time exceeds some value. The actual measurement of the time is performed by the pure virtual function `theTime()`. This is the only function (besides the constructors and the destructor) that is defined by the classes `ABA_CPUTIMER` and `ABA_COWTIMER`.

This class hierarchy is a nice, small example where inheritance and late binding save a lot of implementational effort.

Sorting

Sorting an array of elements according to another array of keys is quite frequently required. Usually, sorting functions are good candidates for template functions, but we prefer to embed these functions in a template class. The advantage is that we can provide within a class also member variables for swapping elements of the arrays, which have the same advantages as global variables from point of view of the sorting functions (they do not have to be put on the stack for each function call), but do not have global scope. Within the class `ABA_SORTER` we implemented the quicksort and the heapsort algorithm.

Chapter 5

Using ABACUS

Section 5.1 provides the basic guidelines how a new application can be attacked with the help of ABACUS. While this section describes the first steps a user should follow, we discuss in Section 5.2 advanced features, in particular how default strategies can be modified according to problem specific requirements.

We strongly encourage to study this chapter together with the example of the ABACUS distribution. In this example all concepts of Section 5.1 and several features of Section 5.2 can be found.

In the following sections we also present pieces of C++ code. When we discuss variables that are of the type "pointer to some type", then we usually omit for convenience of presentation the "pointer to" and the operator $*$ if there is no danger of confusion. For instance, given the variable

```
ABA_ARRAY<ABA_CONSTRAINT*> *constraints;
```

we also say "the constraints are stored in the array `constraints`" instead of "the pointer to constraints are stored in the array `*constraints`".

In order to simplify the use ABACUS we are using the following style for the names of classes, functions, variables, and enumerations.

- Names of classes and names of enumerations are written with upper case letters (e.g., `class ABA_COLUMN`).
- Members of enumerations begin with an upper case letter, e.g., `enum STATUS{Fixed, Set}`.
- All other names (functions, objects, variables, function arguments) start with a lower case letter (e.g., `optimize()`).
- We use upper case letters within all names to increase the readability (e.g., `generateSon()`).
- Names of data members of classes end with an underscore such that they can be easily distinguished from local variables of member functions.
- We do not refrain from using a long name if it helps expressing the concepts behind the name.

5.1 Basics

In this section we explain how our framework is used for the implementation of a new application. This section should provide only the guidelines for the first steps of an implementation, for details we refer to Section 5.2 and to the documentation in the reference manual.

If we want to use ABACUS for a new application we have to derive problem specific classes from some base classes. Usually, only four base classes of ABACUS are involved: `ABA_VARIABLE`, `ABA_CONSTRAINT`, `ABA_MASTER`, and `SUBPROBLEM`. For some applications it is even possible that the classes `ABA_VARIABLE` and/or `ABA_CONSTRAINT` are not included in the derivation process if those concepts provided already by ABACUS are sufficient. By the definition of some pure virtual functions of the base classes in the derived

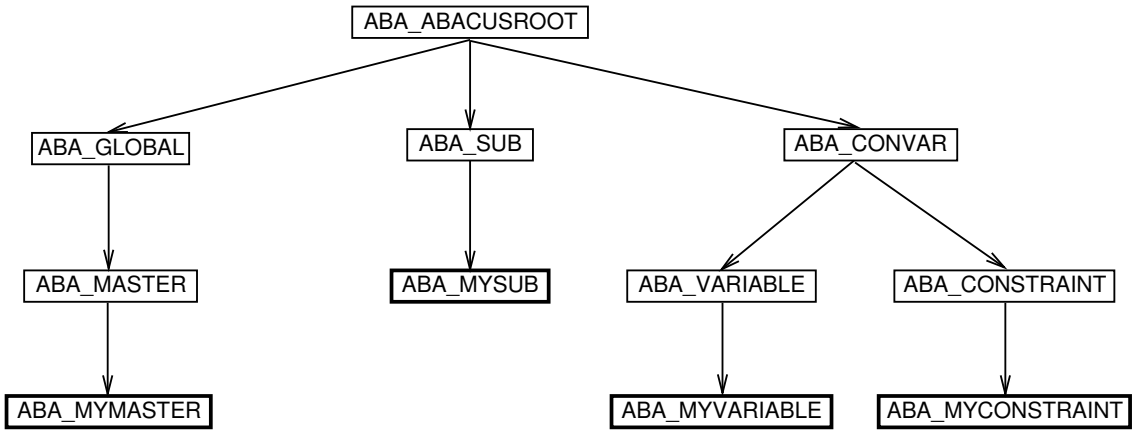

Figure 5.1: Embedding problem specific classes in ABACUS.

classes and the redefinition of some virtual functions a problem specific algorithm can be composed. Figure 5.1 shows how the problem specific classes `MYMASTER`, `MYSUB`, `MYVARIABLE`, and `MYCONSTRAINT` are embedded in the inheritance graph of ABACUS.

Throughout this section we only use the default pool concept of ABACUS, i.e., we have one pool for static constraints, one pool for dynamically generated cutting planes, and one pool for variables. We will outline how an application specific pool concept can be implemented in Section 5.2.1.

5.1.1 Constraints and Variables

The first step in the implementation of a new application is the analysis of its variable and constraint structure. We require at least one constraint class derived from the class `ABA_CONSTRAINT` and at least one variable class derived from the class `ABA_VARIABLE`. The used variable and constraint classes have to match such that a row or a column of the constraint matrix of an LP-relaxation can be generated.

We derive from the class `ABA_VARIABLE` the class `MYVARIABLE` storing the attributes specific to the variables of our application, e.g., its number, or the tail and the head of the associated edge of a graph.

Then we derive the class `MYCONSTRAINT` from the class `ABA_CONSTRAINT`

```
class MYCONSTRAINT : public ABA_CONSTRAINT {
  public:
    virtual double coeff(ABA_VARIABLE *v);
};
```

The function `ABA_CONSTRAINT::coeff(ABA_VARIABLE *v)` is a pure virtual function. Hence, we define it in the class `MYCONSTRAINT`. It returns the coefficient of variable `v` in the constraint. Usually, we need in an implementation of the function `coeff(ABA_VARIABLE *v)` access to the application specific attributes of the variable `v`. Therefore, we have to cast `v` to a pointer to an object of the class `MYVARIABLE` for the computation of the coefficient of `v`. Such that this cast can be performed safely, the variables and constraints used within an application have to be compatible. If run time type information (RTTI) is supported on your system, these casts can be performed safely.

The function `coeff()` is used within the framework when the row format of a constraint is computed, e.g., when the linear program is set up, or a constraint is added to the linear program. When the column associated with a variable is generated, then the virtual member function `coeff()` of the class `ABA_VARIABLE` is used, which is in contrast to the function `coeff()` of the class `ABA_CONSTRAINT` not an abstract function:

```
double ABA_VARIABLE::coeff(ABA_CONSTRAINT *con)
{
```

```
    return con->coeff(this);
  }
```

This method of defining the coefficients of the constraint matrix via the constraints of the matrix originates from cutting plane algorithms. Whereas in a column generation algorithm we usually have a different view on the problem, i.e., the coefficients of the constraint matrix are defined with the help of the variables. In this case, it is appropriate to define the function `MYCONSTRAINT::coeff(ABA_VARIABLE *v)` analogously to the function `ABA_VARIABLE::coeff(ABA_CONSTRAINT *v)` and to define the the function `MYVARIABLE::coeff(ABA_CONSTRAINT *v)`.

ABACUS provides two constraint/variable pairs in its application independent kernel. The most simple one is where each variable is identified by an integer number (class `ABA_NUMVAR`) and each constraint is represented by its nonzero coefficients and the corresponding number of the variables (class `ABA_ROWCON`). We use this constraint/variable pair for general mixed integer optimization problems.

The constraint/variable pair `ABA_NUMCON/ABA_COLVAR` is dual to the previous one. Here the constraints are given by an integer number, but we store the nonzero coefficients and the corresponding row numbers for each variable. Therefore, this constraint/variable pair is useful for column generation algorithms.

ABACUS is not restricted to a single constraint/variable pair within one application. There can be an arbitrary number of constraint and variable classes. It is only required that the coefficients of the constraint matrix can be safely computed for each constraint/variable pair.

5.1.2 The Master

There are two main reasons why we require a problem specific master of the optimization. The first reason is that we have to embed problem specific data members like the problem formulation. The second reason is the initialization of the first subproblem, i.e., the root node of the branch-and-bound tree has to be initialized with a subproblem of the class `MYSUB`. Therefore, a problem specific master has to be derived from the class `ABA_MASTER`:

```
class MYMASTER : public ABA_MASTER {};
```

The Constructor

Usually, the input data is read from a file by the constructor or they are specified by the arguments of the constructor.

From the constructor of the class `MYMASTER` the constructor of the base class `ABA_MASTER` must be called:

```
ABA_MASTER(const char *problemName, bool cutting, bool pricing,
           ABA_OPTSENSE::SENSE optSense = ABA_OPTSENSE::Unknown,
           double eps = 1.0e-4, double machineEps = 1.0e-7,
           double infinity = 1.0e30);
```

Whereas the first three arguments are mandatory, the other ones are optional.

`problemName`	The name of the problem being solved.
`cutting`	If `true`, then cutting planes are generated.
`pricing`	If `true`, then inactive variables are priced out.
`optSense`	The sense of the optimization.
`eps`	A zero-tolerance used within all member functions of objects that have a pointer to this global object.
`machineEps`	Another zero tolerance to compare a value of a floating point variable with 0. This value is usually less than `eps`, because `eps` includes some "safety" tolerance, e.g., to test if a constraint is violated.
`infinity`	All floating point numbers greater than `infinity` are regarded as "infinitely big".

An optional argument of the constructor of the class ABA_MASTER is the sense of the optimization. For some problems (e.g., the binary cutting stock problem or the traveling salesman problem) the sense of the optimization is already known when this constructor is called. For other problems (e.g, the mixed integer optimization problem) the sense of the optimization is determined later when the input data is read in the constructor of the specific application. In this case, the sense of the optimization has to be initialized explicitly before the optimization is started with the function optimize().

The following example of a constructor for the class MYMASTER sets up the master for a branch-and-cut algorithm and initializes the optimization sense explicitly as it is read from the input file.

```
MYMASTER::MYMASTER(const char *problemName) :
  ABA_MASTER(problemName, true, false),
{
  // read the data from the file problemName
  if (/* problemName is a minimization problem*/)
    initializeOptSense(ABA_OPTSENSE::Min);
  else
    initializeOptSense(ABA_OPTSENSE::Max);
}
```

Initialization of the Constraints and Variables

The constraints and variables that are not generated dynamically, e.g., the degree constraints of the traveling salesman problem or the constraints and variables of the problem formulation of a general mixed integer optimization problem, have to be set up and inserted in pools in a member function of the class MYMASTER. These initializations can be also performed in the constructor, but we recommend to use the virtual dummy function initializeOptimization() for this purpose, which is called after the optimization is started with the function optimize().

By default, ABACUS provides three different pools: one for variables and two for constraints. The first constraint pool stores the constraints that are not dynamically generated and with which the first LP-relaxation of the first subproblem is initialized. The second constraint pool is empty at the beginning and is filled up with dynamically generated cutting planes. In general, ABACUS provides a more flexible pool concept to which we will come back later, but for many applications the default pools are sufficient.

After the initial variables and constraints are generated they have to be inserted into the default pools by calling the function

```
virtual void initializePools(ABA_BUFFER<ABA_CONSTRAINT*> &constraints,
                             ABA_BUFFER<ABA_VARIABLE*>   &variables,
                             int                          varPoolSize,
                             int                          cutPoolSize,
                             bool                         dynamicCutPool = false);
```

Here, constraints are the initial constraints, variables are the initial variables, varPoolSize is the initial size of the variable pool, and cutPoolSize is the initial size of the cutting plane pool. The size of the variable pool is always dynamic, i.e., this pool is increased if required. By default, the size of the cutting plane pool is fixed, but it becomes dynamic if the argument dynamicCutPool is true.

There is second version of the function —initializePools()— that allows the insertion of an initial set of cutting planes into the cut pool.

The function initializeOptimization() can be also used to determine a feasible solution by a heuristic such that the primal bound can be initialized.

Hence, the function initializeOptimization() could look as follows under the assumption that the functions nVar() and nCon() are defined in the class MYMASTER and return the number of variables and the number of the constraints, respectively. In the example we initialize the size of the cut pool with 2*nCon(). As the arguments of the constructors of the classes MYVARIABLE and MYCONSTRAINT are problem specific we replace them by "...".

After the pools are set up the primal bound is initialized with the value of a feasible solution returned by the function `myHeuristic()`. While the initialization of the pools is mandatory the initialization of the primal bound is optional.

```
void MYMASTER::initializeOptimization()
{
  ABA_BUFFER<ABA_VARIABLE*> variables(this, nVar());
  for (int i = 0; i < nVar(); i++)
    variables.push(new MYVARIABLE(...));
  ABA_BUFFER<ABA_CONSTRAINT*> constraints(this, nCon());
  for (i = 0; i < nCon(); i++)
    constraints.push(new MYCONSTRAINT(...));
  initializePools(constraints, variables, nVar(), 2*nCon());
  primalBound(myHeuristic());
}
```

The First Subproblem

The root of the branch-and-bound tree has to be initialized with an object of the problem specific subproblem class `MYSUB`, which is derived from the class `ABA_SUB`. This initialization must be performed by a definition of the pure virtual function `firstSub()`, which returns a pointer to the first subproblem. In the following example we assume that the constructor of the class `MYSUB` for the root node of the enumeration tree has only a pointer to the associated master as argument.

```
ABA_SUB *MYMASTER::firstSub()
{
  return new MYSUB(this);
}
```

5.1.3 The Subproblem

Finally, we have to derive a problem specific subproblem from the class `ABA_SUB`:

```
class MYSUB : public ABA_SUB {};
```

Besides the constructors only two pure virtual functions of the base class `ABA_SUB` have to be defined, which check if a solution of the LP-relaxation is a feasible solution of the mixed integer optimization problem, and generate the sons after a branching step, respectively. Moreover, the main functionality of the problem specific subproblem is to enhance the branch-and-bound algorithm with dynamic variable and constraint generation and sophisticated primal heuristics.

The Constructors

The class `ABA_SUB` has two different constructors: one for the root node of the optimization and one for all other subproblems of the optimization. This differentiation is required as the constraint and variable set of the root node can be initialized explicitly, whereas for the other nodes this data is copied from the father node and possibly modified by a branching rule. Therefore, we also have to implement these two constructors for the class `MYSUB`.

The root node constructor for the class `ABA_SUB` must be called from the root node constructor of the class `MYSUB`.

```
ABA_SUB(ABA_MASTER *master,
        double conRes, double varRes, double nnzRes,
        bool relativeRes = true,
        ABA_BUFFER<ABA_POOLSLOT<ABA_CONSTRAINT, ABA_VARIABLE> *> *constraints = 0,
        ABA_BUFFER<ABA_POOLSLOT<ABA_VARIABLE, ABA_CONSTRAINT> *> *variables = 0);
```

`master`	A pointer to the corresponding master of the optimization.
`conRes`	The additional memory allocated for constraints.
`varRes`	The additional memory allocated for variables.
`nnzRes`	The additional memory allocated for nonzero elements of the constraint matrix.
`relativeRes`	If this argument is `true`, then reserve space for variables, constraints, and nonzeros of the previous three arguments is given in percent of the original numbers. Otherwise, the numbers are interpreted as absolute values (casted to integer).
`constraints`	The pool slots of the initial constraints. If the value is 0, then all constraints of the default constraint pool are taken.
`variables`	The pool slots of the initial variables. If the value is 0, then all variables of the default variable pool are taken.

The values of the arguments `conRes`, `varRes`, and `nnzRes` should only be good estimations. An underestimation does not cause a run time error, because space is reallocated internally as required. However, many reallocations decrease the performance. If the "Number of Cplex reinitializations" in the statistics output is high compared to the total number of linear programs, then the estimated values should be corrected. An overestimation only wastes memory.

In the following implementation of a constructor for the root node we do not specify additional memory for variables, because we suppose that no variables are generated dynamically. We accept the default settings of the last three arguments, as this is normally a good choice for many applications.

```
MYSUB::MYSUB(MYMASTER *master) :
  ABA_SUB(master, 50.0, 0.0, 100.0)
{ }
```

While there are some alternatives for the implementation of the root node for the application, the constructor of non-root nodes has usually the same form for all applications, but might be augmented with some problem specific initializations.

```
MYSUB::MYSUB(ABA_MASTER *master, ABA_SUB *father, ABA_BRANCHRULE *branchRule) :
  ABA_SUB(master, father, branchRule)
{}
```

`master`	A pointer to the corresponding master of the optimization.
`father`	A pointer to the father in the enumeration tree.
`branchRule`	The rule defining the subspace of the solution space associated with this node. More information about branching rules can be found in Section 5.2.7. As long as you are using only the default branching on variables you do not have to know anything about the class `ABA_BRANCHRULE`.

The root node constructor for the class `MYSUB` has to be called from the function `firstSub()` of the class `MYMASTER`. The constructor for non-root nodes has to be called in the function `generateSon()` of the class `MYSUB`.

The Feasibility Check

After the LP-relaxation is solved we have to check if its optimum solution is a feasible solution of our optimization problem. Therefore, we have to define the pure virtual function `feasible()` in the class `MYSUB`, which should return `true` if the LP-solution is a feasible solution of the optimization problem, and `false` otherwise:

```
bool MYSUB::feasible()
{}
```

If all constraints of the integer programming formulation are present in the LP-relaxation, then the LP-solution is feasible if all discrete variables have integer values. This check can be performed by calling the member function integerFeasible() of the base class `ABA_SUB`:

```
bool MYSUB::feasible()
{
  return integerFeasible();
}
```

If the LP-solution is feasible and its value is better than the primal bound, then ABACUS automatically updates the primal bound. However, the update of the solution itself is problem specific, i.e., this update has to be performed within the function `feasible()`.

The Generation of the Sons

Like the pure virtual function `firstSub()` of the class `ABA_MASTER`, which generates the root node of the branch-and-bound tree, we need a function generating a son of a subproblem. This function is required as the nodes of the branch-and-bound tree have to be identified with a problem specific subproblem of the class `MYSUB`. This is performed by the pure virtual function `generateSon()`, which calls the constructor for a non-root node of the class `MYSUB` and returns a pointer to the newly generated subproblem. If the constructor for non-root nodes of the class `MYSUB` has the same arguments as the corresponding constructor of the base class `ABA_SUB`, then the function `generateSon()` can have the following form:

```
ABA_SUB *MYSUB::generateSon(ABA_BRANCHRULE *rule)
{
  return new MYSUB(master_, this, rule);
}
```

This function is automatically called during a branching process. If the already built-in branching strategies are used, we do not have to care about the generation of the branching rule `rule`. How other branching strategies can be implemented is presented in Section 5.2.7.

A Branch-and-Bound Algorithm

The two constructors, the function `feasible()`, and the function `generateSon()` must be implemented for the subproblem class of every application. As soon as these functions are available, a branch-and-bound algorithm can be performed. All other functions of the class `MYSUB` that we are going to explain now, are optional in order to improve the performance of the implementation.

The Separation

Problem specific cutting planes can be generated by redefining the virtual dummy function `separate()`. In this case, also the argument `cutting` in the constructor of the class `ABA_MASTER` should receive the value `true`, otherwise the separation is skipped. The first step is the redefinition of the function `separate()` of the base class `ABA_SUB`.

```
int MYSUB::separate()
{ }
```

The function `separate()` returns the number of generated constraints.

We distinguish between the separation from scratch and the separation from a constraint pool. Newly generated constraints have to be added by the function `addCons()` to the buffer of the class `ABA_SUB`, which returns the number of added constraints. Constraints generated in earlier iterations that have been become inactive in the meantime might be still contained in the cut pool. These constraints can be regenerated by calling the function `constraintPoolSeparation()`, which adds the constraints to the buffer without an explicit call of the function `addCons()`.

A very simple separation strategy is implemented in the following example of the function `separate()`. Only if the pool separation fails, we generate new cuts from scratch. The generated constraints are added with the function `addCons()` to the internal buffer, which has a limited size. The number of constraints that can still be added to this buffer is returned by the function `conBufferSpace()`. The

function `mySeparate()` performs here the application specific separation. If more cuts are added with the function `addCons()` than there is space in the internal buffer for cutting planes, then the redundant cuts are discarded. The function `addCons()` returns the number of actually added cuts.

```
int MYSUB::separate()
{
  int nCuts = constraintPoolSeparation();
  if (!nCuts) {
    ABA_BUFFER<ABA_CONSTRAINT*> newCuts(master_, conBufferSpace());
    nCuts = mySeparate(newCuts);
    if (nCuts) nCuts = addCons(newCuts);
  }
  return nCuts;
}
```

Note, ABACUS does not automatically check if the added constraints are really violated. Adding only non-violated constraints, can cause an infinite loop in the cutting plane algorithm, which is only left if the tailing off control is turned on (see Section 5.2.26).

While constraints added with the function `addCons()` are usually allocated by the user, they are deleted by ABACUS. They must **not** be deleted by the user (see Section 5.2.13).

If not all constraints of the integer programming formulation are active, and all discrete variables have integer values, then the solution of a separation problem might be required to check the feasibility of the LP-solution. In order to avoid a redundant call of the same separation algorithm later in the function `separate()`, constraints can be added already here by the function `addCons()`.

In the following example of the function `feasible()` the separation is even performed if there are discrete variables with fractional values such that the separation routine does not have to be called a second time in the function `separate()`.

```
bool MYSUB::feasible()
{
  bool feasible;

  if (integerFeasible()) feasible = true;
  else                   feasible = false;

  ABA_BUFFER<ABA_CONSTRAINT*> newCuts(master_, conBufferSpace());

  int nSep = mySeparate(newCuts);

  if (nSep) {
    feasible = false;
    addCons(newCuts);
  }
  return feasible;
}
```

Pricing out Inactive Variables

The dynamic generation of variables is performed very similarly to the separation of cutting planes. Here, the virtual function `pricing()` has to be redefined and the argument `pricing` in the constructor of the class `ABA_MASTER` should receive the value `true`, otherwise the pricing is skipped.

We illustrate the redefinition of the function `pricing()` by an example that is an analogon to the example given for the function `separate()`.

```
int MYSUB::pricing()
{
  int nNewVars = variablePoolSeparation();
  if (!nNewVars) {
    ABA_BUFFER<ABA_VARIABLE*> newVariables(master_, addVarBufferSpace());
    nNewVars = myPricing(newVariables);
    if (nNewVars) nNewVars = addVars(newVariables);
  }
  return nNewVars;
}
```

While variables added with the function `addVars()` are usually allocated by the user, they are deleted by ABACUS. They must **not** be deleted by the user (see Section 5.2.13).

Primal Heuristics

After the LP-relaxation has been solved in the subproblem optimization the virtual function `improve()` is called. Again, the default implementation does nothing but in a redefinition in the derived class `MYSUB` application specific primal heuristics can be inserted:

```
int MYSUB::improve(double &primalValue)
{ }
```

If a better feasible solution is found its value has to be stored in `primalValue` and the function should return 1, otherwise it should return 0. In this case, the value of the primal bound is updated by ABACUS, whereas the solution itself has to be updated within the function `improve()` as already explained for the function `feasible()`.

It is also possible to update the primal bound already within the function `improve()` if this is more convenient to reduce internal bookkeeping. In the following example we apply the two problem specific heuristics `myFirstHeuristic()` and `mySecondHeuristic()`. After each heuristic we check if the `value` of the solution is better than the best known one with the function call `master_->betterPrimal(value)`. If this function returns `true` we update the value of the best known feasible solution by calling the function `master_->primalBound()`.

```
int MYSUB::improve(double &primalValue)
{
  int status = 0;
  double value;

  myFirstHeuristic(value);
  if (master_->betterPrimal(value)) {
    master_->primalBound(value);
    primalValue = value;
    status = 1;
  }

  mySecondHeuristic(value);
  if (master_->betterPrimal(value)) {
    master_->primalBound(value);
    primalValue = value;
    status = 1;
  }

  return status;
}
```

Accessing Important Data

For a complete description of all members of the class ABA_SUB we refer to the documentation in the reference manual. However, in most applications only a limited number of data is required for the implementation of problem specific functions, like separation or pricing functions. For simplification we want to state some of these members here:

`int nCon() const;`	returns the number of active constraints.
`int nVar() const;`	returns the number of active variables.
`ABA_VARIABLE *variable(int i);`	returns a pointer to the `i`-th active variable.
`ABA_CONSTRAINT *constraint(int i);`	returns a pointer to the `i`-th active constraint.
`double *xVal_;`	an array storing the values of the variables after the linear program is solved.
`double *yVal_`	an array storing the values of the dual variables after the linear program is solved.

5.1.4 Starting the Optimization

After the problem specific classes are defined as discussed in the previous sections, the optimization can be performed with the following main program. We suppose that the master of our new application has as only parameter the name of the input file.

```
#include "mymaster.h"

int main(int argc, char **argv)
{
  MYMASTER master(argv[1]);

  master.optimize();
  return master.status();
}
```

If ABACUS is used on Windows NT together with Cplex, the Cplex DLL must be loaded at the beginning and unloaded at the end of the main program. For details we refer to the Cplex documentation and the example program of the ABACUS distribution.

5.2 Advanced Features

In the previous section we described the first steps for the implementation of a linear-programming based branch-and-bound algorithm with ABACUS. Now, we present several advanced features of ABACUS. We show how various built-in strategies can be used instead of the default strategies and how new problem specific concepts can be integrated.

5.2.1 Using other Pools

By default, ABACUS provides one variable pool, one constraint pool for constraints of the problem formulation, and another constraint pool for cutting planes. For certain applications the implementation of a different pool concept can be advantageous. Suppose we would like to provide two different pools for cutting planes for our application instead of our default cutting plane pool. These pools have to be declared in the class `MYMASTER` and we also provide two public functions returning pointers to these pools.

```
class MYMASTER : public ABA_MASTER {
  public:
    ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE> *myCutPool1()
    {
        return myCutPool1_;
    }
    ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE> *myCutPool2()
    {
      return myCutPool2_;
    }
  private:
    ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE> *myCutPool1_;
    ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE> *myCutPool2_;
};
```

Now, instead of the default cutting plane pool we set up our two problem specific cut pools in the function `initializeOptimization()`. This is done by using 0 as last argument of the function `initializePools()`, which sets the size of the default cut pool to 0. The size of the variable pool is chosen arbitrarily. Then, we allocate the memory for our pools. For simplification, we suppose that the size of each cut pool is 1000.

```
void MYMASTER::initializeOptimization()
{
  // initialize the constraints and variables
  initializePools(constraints, variables, 3*variables.number(), 0);

  myCutPool1_ = new ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE>(this, 1000);
  myCutPool2_ = new ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE>(this, 1000);
}
```

The following redefinition of the function `separate()` shows how constraints can be separated from and added to our pools instead of the default cut pool. If a pointer to a pool is specified as an argument of the function `constraintPoolSeparation()`, then constraints are regenerated from this pool instead of the default cut pool. By specifying a constraint pool as the second argument of the function `addCons()` the constraints are added to this pool instead of the default cut pool. As the member `master_` of the base class `ABA_SUB` is a pointer to an object of the class `ABA_MASTER` we require an explicit cast to call the member functions `myCutPool1()` and `myCutPool2()` of the class `MYMASTER`.

```
int MYSUB::separate()
{
```

```
  ABA_BUFFER<ABA_CONSTRAINT*> newCuts(master_, 100);
  int nCuts = constraintPoolSeparation(0, ((MYMASTER*) master_)->myCutPool1());
  if (!nCuts) {
    nCuts = mySeparate1(newCuts);
    if (nCuts) nCuts = addCons(newCuts, ((MYMASTER*) master_)->myCutPool1());
  }
  if (!nCuts) {
    nCuts = constraintPoolSeparation(0, ((MYMASTER*) master_)->myCutPool2());
    if (!nCuts) {
      nCuts = mySeparate2(newCuts);
      if (nCuts) nCuts = addCons(newCuts, ((MYMASTER*) master_)->myCutPool2());
    }
  }
  return nCuts;
}
```

Using application specific variable pools can be done in an analogous way with the two functions `variablePoolSeparation()` and `addVars()`.

5.2.2 Pool without Multiple Storage of Items

One of the data structures using up very large parts of the memory are the pools for constraints and variables. Limiting the size of the pool has two side effects. First, pool separation or pricing is less powerful with a small pool. Second, the branch-and-bound tree might be processed with reduced speed, since subproblems cannot be initialized with the constraint and variable system of the father node.

On the other hand it can be observed that the same constraint or variable is generated several times in the course of the optimization. This could be avoided by scanning completely the pool before separating or pricing from scratch. But, if direct separation or pricing are fast, such a strategy can be less advantageous.

Therefore ABACUS provides the template class `ABA_NONDUPLPOOL` that avoids storing the same constraint or variable more than once in a pool. More precisely, when an item is inserted in such a pool, the inserted item is compared with the already available items. If it is already present in the pool, the inserted item is deleted and replaced by the already available item.

In order to use this pool, you have to set up your own pool as explained in Section 5.2.1. Instead of a `ABA_STANDARDPOOL` you have to use now an `ABA_NONDUPLPOOL`. For constraints or variables that are inserted in a pool of the template class `ABA_NONDUPLPOOL`, the virtual functions `hashKey`, `name`, and `equal` of the base class `ABA_CONVAR` have to be redefined. These functions are used in the comparison of a new item and the items that are already stored in the pool. For the details of these functions we refer to the reference manual.

5.2.3 Constraints and Variables

We discussed the concept of expanding and compressing constraints and variables already in Section 4.2.4.

This feature can be activated for a specific constraint or variable class if the virtual dummy functions `expand()` and `compress()` are redefined. Here we give an example for constraints, but it can be applied to variables analogously. We discussed the expanded and compressed format of the subtour elimination constraints already in Section 4.2. The nodes defining the subtour elimination constraint are contained in the buffer `nodes_`. When the constraint is expanded each node of the subtour elimination constraint is marked.

```
void SUBTOUR::expand()
{
  if(expanded()) return;
  marked_ = new bool[graph_->nNodes() + 1];
  int nGraph = graph_->nNodes();
```

```
    for (int v = 1; v <= nGraph; v++)
      marked_[v] = false;
    int nNodes = nodes_.size();
    for (int v = 0; v < nNodes; v++)
      marked_[nodes_[v]] = true;
  }
```

For the compression of the constraint only the allocated memory is deleted.

```
  void SUBTOUR::compress()
  {
    if (!expanded()) return;
    delete marked_;
  }
```

Constraints

Often, the definition of constraint specific expanded and compressed formats provides already sufficiently efficient running times for the generation of the row format, the computation of the slack of a given LP-solution, or the check if the constraint is violated.

If nevertheless further tuning is required, then the functions `genRow()` and `slack()` can be redefined. The function

```
virtual int genRow(ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *variables,
                   ABA_ROW &row);
```

stores the row format associated with the variable set `variables` in `row` and returns the number of nonzero coefficients stored in `row`.

The function

```
virtual double slack(ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *variables,
                     double *x);
```

returns the slack of the vector `x` associated with the variable set `variables`. Instead of redefining the function `violated()` due to performance issues, the function `slack()` should be redefined because this function is called from the function `violated()` and uses most of the joint running time.

Variables

The equivalents of the class `ABA_VARIABLE` to the functions `genRow()` and `slack()` of the class `ABA_CONSTRAINT` are the functions `genColumn()` and `redCost()`. Also for these two functions a redefinition due to performance reasons can be considered if the expansion/compression concept is not sufficient or cannot be applied.

The function

```
virtual int genColumn(ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *constraints,
                      ABA_COLUMN &col);
```

stores the column format of the variable associated with the constraint set `constraints` in the argument `col` and returns the number of nonzero coefficients stored in `col`.

The function

```
virtual double redCost(ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *constraints,
                       double *y);
```

returns the reduced cost of the variable corresponding to the dual variables `y` of the active constraints `constraints`. As a redefinition of the virtual member function `slack()` of the class `ABA_CONSTRAINT` might speed up the function `violated()`, also a redefinition of the function `redCost()` can speed up the function `violated()` of the class `ABA_VARIABLE`.

5.2.4 Infeasible Linear Programs

As long as we do not generate variables dynamically, a subproblem can be immediately fathomed if the LP-relaxation is infeasible. However, if not all variables are active we have to check if the addition of an inactive variable can restore the feasibility. An infeasibility can either be detected when the linear program is set up, or later by the LP-solver (see [Thi95]).

If fixed and set variables are eliminated, it might happen when the row format of a constraint is generated in the initialization of the linear program that a constraint has a void left hand side but can never be satisfied due to its right hand side. In this case, the function

```
virtual int initMakeFeas(ABA_BUFFER<ABA_INFEASCON*> &infeasCon,
                         ABA_BUFFER<ABA_VARIABLE*> &newVars,
                         ABA_POOL<ABA_VARIABLE, ABA_CONSTRAINT> **pool);
```

is called. The default implementation always returns 1 to indicate that no variables could be added to restore feasibility. If it might be possible that in our application the addition of variables could restore the feasibility, then this function has to be redefined in a derived class.

The buffer `infeasCon` stores pointers to objects storing the infeasible constraints and the kind of infeasibility. The new variables should be added to the buffer `newVars`, and if the variables should be added to an other pool than the default variable pool, then a pointer to this pool should be assigned to `*pool`. If variables have been added that could restore the feasibility for all infeasible constraints, then the function should return 0, otherwise it should return 1.

If an infeasible linear program is detected by the LP-solver, then the function

```
virtual int makeFeasible();
```

is called. The default implementation of the virtual dummy function does nothing except returning 1 in order to indicate that the feasibility cannot be restored. Otherwise, an iteration of the dual simplex method has to be emulated according to the algorithm outlined in [Thi95]. When the function is called it is guaranteed that the current basis is dual feasible. Exactly one of the member variables `infeasVar_` or `infeasCon_` of the class `ABA_SUB` is nonnegative. If `infeasVar_` is nonnegative, then it holds the number of an infeasible variable, if `infeasCon_` is nonnegative, then it holds the number of an infeasible slack variable. The array `bInvRow_` stores the row of the basis inverse corresponding to the infeasible variable (only basic variables can be infeasible). Then the inactive variables have to be scanned like in the function `pricing()`. Variables that might restore the feasibility have to be added by the function `addCons()`. If no such candidate is found the subproblem can be fathomed.

5.2.5 Other Enumeration Strategies

With the parameter `EnumerationStrategy` in the file `.abacus` the enumeration strategies best-first search, breadth-first search, depth-first search, and a diving strategy can be controlled (see Section 5.2.26). Another problem specific enumeration strategy can be implemented by redefining the virtual function

```
virtual int enumerationStrategy(ABA_SUB *s1, ABA_SUB *s2);
```

which compares the two subproblems `s1` and `s2` and returns 1 if the subproblem `s1` should be processed before `s2`, returns -1 if the subproblem `s2` should be processed before `s1`, and returns 0 if the two subproblems have the same precedence in the enumeration.

We provide again an implementation of the depth-first search strategy as an example for a reimplementation of the function `enumerationStrategy()`.

```
int MYMASTER::enumerationStrategy(ABA_SUB *s1, ABA_SUB *s2)
{
  if(s1->level() > s2->level()) return  1;
  if(s1->level() < s2->level()) return -1;
  return 0;
 }
```

In the default implementation of the depth-first search strategy we do not return 0 immediately if the two subproblems have the same level in the enumeration tree, but we call the virtual function

```
int ABA_MASTER::equalSubCompare(ABA_SUB *s1, ABA_SUB *s2);
```

which return 0 if both subproblems have not been generated by setting a binary variable. Otherwise, that subproblem has higher priority where the branching variable is set to the upper bound, i.e., it returns 1 if the branching variable of `s1` is set to the upper bound, -1 if the branching variable of `s2` is set to the upper bound, and 0 otherwise. Other strategies for resolving equally good subproblems for the built-in enumeration strategies depth-first search and best-first search can be implemented by a redefinition of this virtual function. Moreover, this function can also be generalized for other enumeration strategies.

5.2.6 Selection of the Branching Variable

The default branching variable selection strategy can be changed by the redefinition of the virtual function

```
int ABA_SUB::selectBranchingVariable(int &variable);
```

in a class derived from the class `ABA_SUB`. If a branching variable is found it has to be stored in the argument `variable` and the function should return 0. If the function fails to find a branching variable, it should return 1. Then, the subproblem is automatically fathomed.

Here we present an example where the first fractional variable is chosen as branching variable. In general, this is not a very good strategy.

```
int MYSUB::selectBranchingVariable(int &variable)
{
  for (int i = 0; i < nVar(); i++)
    if (fracPart(xVal_[i]) > master_->machineEps()) {
      variable = i;
      return 0;
    }

  return 1;
}
```

The function `fracPart(double x)` returns the absolute value of the fractional part of `x`.

5.2.7 Using other Branching Strategies

Although branching on a variable is often an adequate strategy for branch-and-cut algorithms, it is in general useless for branch-and-price algorithms. But also for branch-and-cut algorithms other branching strategies, e.g., branching on a constraint can be interesting alternatives.

For the implementation of different branching strategies we have introduced the concept of branching rules in the class `ABA_BRANCHRULE` (see Section 4.2.7). The virtual function

```
int ABA_SUB::generateBranchRules(ABA_BUFFER<ABA_BRANCHRULE*> &rules);
```

returns 0 if it can generate branching rules and stores for each subproblem, that should be generated, a branching rule in the buffer `rules`. If no branching rules can be generated, this function returns 1 and the subproblem is fathomed. The default implementation of the function `generateBranchRules()` generates two rules for two new subproblems by branching on a variable. These rules are represented by the classes `ABA_SETBRANCHRULE` for binary variables and `ABA_BOUNDBRANCHRULE` for integer variables, which are derived from the abstract class `ABA_BRANCHRULE`. Moreover, we provide also rules for branching on constraints (`ABA_CONBRANCHRULE`), and for branching by setting an integer variable to a fixed value (`ABA_VALBRANCHRULE`). Other branching rules have to be derived from the class `ABA_BRANCHRULE`. The default branching strategy can be replaced by the redefinition of the virtual function `generateBranchRules()` in a class derived from the class `ABA_SUB`.

Branching on a Variable

The default branching strategy of ABACUS is branching on a variable. Different branching variable selection strategies can be chosen in the parameter file (see Section 5.2.26). If a problem specific branching variable selections strategy should be implemented it is not required to redefine the function `ABA_SUB::generateBranchRule()`, but a redefinition of the function

```
int ABA_SUB::selectBranchingVariable(int &variable)
```

is sufficient. If a branching variable is found it should be stored in the function argument `variable` and `selectBranchingVariable()` should return 0, otherwise it should return 1.

If no branching variable is found, the subproblem is fathomed.

Branching on a Constraint

As all constraints used in ABACUS, also branching constraints have to be inserted in a pool. The function `ABA_POOL::insert()` returns a pointer to the pool slot the constraint is stored in that is required in the constructor of `ABA_CONBRANCHRULE`. Although the default cut pool can be used for the branching constraints, an extra pool for branching constraints is recommended, because first no redundant work in the pool separation is performed, and second the branching constraint pool should be dynamic such that all branching constraints can be inserted. This pool for the branching constraints should be added to your derived master class. It is sufficient that the `size` of the branching pool is only a rough estimation. If the branching pool is dynamic, it will increase automatically if required.

```
class MYMASTER : ABA_MASTER {
 ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE> *branchingPool_;
}

MYMASTER::MYMASTER(const char *problemName) :
  ABA_MASTER(problemName, true, false)
{
  branchingPool_ = new ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE>(this,
                                                                      size,
                                                                      true);
}

MYMASTER::~MYMASTER()
{
  delete branchingPool_;
}
```

The constraint branching rules have to be generated in the function `MYSUB::generateBranchRules()`. It might be necessary to introduce a new class derived from the class `ABA_CONSTRAINT` for the representation of your branching constraint. For simplification we assume here that your branching constraint is also of type `MYCONSTRAINT`. Each constraint is added to the branching pool.

If the generation of branching constraints failed, you might try to resort to the standard branching on variables.

```
int MYSUB::generateBranchRules(ABA_BUFFER<ABA_BRANCHRULE*> &rules)
{
  if (/* branching constraints can be found */) {
    ABA_POOLSLOT<ABA_CONSTRAINT, ABA_VARIABLE> *poolSlot;

    /* generate the branching rule for the first new subproblem */

    MYCONSTRAINT *constraint1 = new MYCONSTRAINT(...);
```

```
    poolSlot = ((MYMASTER *) master_)->branchingPool_->insert(constraint1);
    rules.push(new ABA_CONBRANCHRULE(master_, poolSlot);

    /* generate the branching rule for the second new subproblem */
    MYCONSTRAINT *constraint2 = new MYCONSTRAINT(...);
    poolSlot = ((MYMASTER *) master_)->branchingPool_->insert(constraint2);
    rules.push(new ABA_CONBRANCHRULE(master_, poolSlot);

    return 0;
  }
  else
    return ABA_SUB::generateBranchRules(rules);  // resort to standard branching
}
```

Moreover, a branching constraint should be locally valid and not dynamic. This has to be specified when calling the constructor of the base class `ABA_CONSTRAINT`. Of course, the subproblem defined by the branching constraint is not available at the time when the branching constraint is generated. However, any locally valid constraint requires an associated subproblem in the constructor. Therefore, the (incorrect) subproblem in which the branching constraint is generated should be used. ABACUS will modify the associated subproblem later in the constructor of the subproblem generated with the constraint branching rule.

When the subproblem generated by the branching constraint is activated at the beginning of its optimization the branching constraint is not immediately added to the linear program and the active constraints, but it is inserted into the buffer for added constraints similarly as cutting planes are added (see Section 5.2.16).

Problem Specific Branching Rules

A problem specific branching rule is introduced by the derivation of a new class `MYBRANCHRULE` from the base class `ABA_BRANCHRULE`. As example we show how a branching rule for setting a variable to its lower or upper bound is implemented. This example has some small differences to the ABACUS class `ABA_SETBRANCHRULE`.

```
class MYBRANCHRULE : public ABA_BRANCHRULE {
  public:
    MYBRANCHRULE(ABA_MASTER *master, int variable, ABA_FSVARSTAT::STATUS status);
    virtual ~MYBRANCHRULE();
    virtual int extract(ABA_SUB *sub);

  private:
    int               variable_;  // the branching variable
    ABA_FSVARSTAT::STATUS status_;    // the status of the branching variable
};
```

The constructor initializes the branching variable and its status (`ABA_FSVARSTAT::SetToLowerBound` or `ABA_FSVARSTAT::SetToUpperBound`).

```
MYBRANCHRULE::MYBRANCHRULE(ABA_MASTER *master,
                           int variable,
                           ABA_FSVARSTAT::STATUS status) :
  ABA_BRANCHRULE(master),
  variable_(variable),
  status_(status)
{ }
```

```
MYBRANCHRULE::~MYBRANCHRULE()
{ }
```

The pure virtual function `extract()` of the base class `ABA_BRANCHRULE` has to be defined in every new branching rule. This function is called when the subproblem is activated at the beginning of its optimization. During the activation of the subproblem a copy of the final constraint and variable system of the father subproblem is made. The function `extract()` should modify this system according to the branching rule.

In our example we first check if setting the branching variable causes a contradiction. In this case we return `1` in order to indicate that the subproblem can be fathomed immediately. Otherwise we set the branching variable and return `0`.

```
int MYBRANCHRULE::extract(ABA_SUB *sub)
{
  if (sub->fsVarStat(variable_)->contradiction(status_))
    return 1;

  sub->fsVarStat(variable_)->status(status_);
  return 0;
}
```

As a second example for the design of a branching rule we show how the constraint branching rule of ABACUS is implemented. After inserted the branching constraint in a pool slot the constraint branching rule can be constructed with this pool slot.

```
class ABA_CONBRANCHRULE : public ABA_BRANCHRULE {
  public:
    ABA_CONBRANCHRULE(ABA_MASTER *master,
                      ABA_POOLSLOT<ABA_CONSTRAINT,
                      ABA_VARIABLE> *poolSlot);
    virtual ~ABA_CONBRANCHRULE();
    virtual int extract(ABA_SUB *sub);

  private:
    ABA_POOLSLOTREF<ABA_CONSTRAINT, ABA_VARIABLE> poolSlotRef_;
};


ABA_CONBRANCHRULE::ABA_CONBRANCHRULE(ABA_MASTER *master,
                           ABA_POOLSLOT<ABA_CONSTRAINT, ABA_VARIABLE> *poolSlot) :
  ABA_BRANCHRULE(master),
  poolSlotRef_(poolSlot)
{ }

ABA_CONBRANCHRULE::~ABA_CONBRANCHRULE()
{ }
```

In the function `extract()` the branching constraint is added to the subproblem. This should always be done with the function `ABA_SUB::addBranchingConstraint()`. Since adding a branching constraint cannot cause a contradiction, we always return 0.

```
int ABA_CONBRANCHRULE::extract(ABA_SUB *sub)
{
  if (sub->addBranchingConstraint(poolSlotRef_.slot())) {
    master_->err() << "ABA_CONBRANCHRULE::extract(): addition of branching  ";
    master_->err() << "constraint to subproblem failed." << endl;
```

```
      exit(Fatal);
    }

    return 0;
  }
```

5.2.8 Strong Branching

In order to reduce the size of the enumeration tree, it is important to select "good" branching rules. We present a framework for measuring the quality of the branching rules. First, we describe the basic idea and explain the details later.

A branching step is performed by generating a set of branching rules, each one defines a son of the current subproblem. We call such a set of branching rules a *sample*. For instance, if we branch on a single binary variable, the corresponding sample consists of two branching rules, one defining the subproblem in which the branching variable is set to the upper bound, the other one the subproblem in which the branching variable is set to the lower bound. Instead of generating a single branching sample, it is now possible to generate a set of branching samples and selecting from this set the "best" sample for generating the sons of the subproblem. In this evaluation process for each branching rule of each branching sample a rank is computed. In the default implementation this rank is given by performing a limited number of iterations of the dual simplex method for the first linear program of the subproblem defined by the branching rule. For maximization problems we select that sample for which the maximal rank of its rules is minimal. For minimization problems we select that sample for which the minimal rank of its rules is maximal.

Both the computation of the ranks and the comparison of the rules can be adapted to problem specific criteria.

Default Strong Branching

Strong branching can be turned on for the built-in branching strategies that are controlled by the parameter `BranchingStrategy` of the configuration file. With the parameter `NBranchingVariableCandidates` the number of tested branching variables can be indicated (see also Section 5.2.26).

Strong Branching with Special Branching Variable Selection

In order to use strong branching in combination with a problem specific branching variable selection strategy, it is only necessary to redefine the virtual function

```
int ABA_SUB::selectBranchingVariableCandidates(ABA_BUFFER<int> &candidates)
```

in the problem specific subproblem class. In the buffer `candidates` the indices of the variables that should be tested as branching variables are collected. If at least one candidate is found, the function should return `1`, otherwise `0`.

ABACUS tests all candidates by solving (partially) the first linear program of all potential sons and selects the branching variable as previously explained.

Ranking Branching Rules

In the default version the rank of a branching rule is computed by the function `lpRankBranchingRule()`. The rank can be determined differently by redefining the virtual function

```
double ABA_SUB::rankBranchingRule(ABA_BRANCHRULE *branchRule)
```

that returns a floating point number associated with the rank of the `branchRule`.

Comparing Branching Samples

After a rank to each rule of each branching sample has been assigned by the function `rankBranching-Rule()` all branching samples are compared and the best one is selected. This comparison is performed by the virtual function

```
int ABA_SUB::compareBranchingSampleRanks(ABA_ARRAY<double> &rank1,
                                         ABA_ARRAY<double> &rank2)
```

that compares the ranks `rank1` of all rules of one branching sample with the ranks `rank2` of the rules of another branching sample. It returns `1` if the ranks stored in `rank1` are better, `0` if both ranks are equal, and `-1` if the ranks stored in `rank2` are better.

For maximization problems in the default version of `compareBranchingSampleRanks()` the array `rank1` is better if its maximal entry is less than the maximal entry of `rank2` (min-max criteria). For minimization problems `rank1` is better if its minimal entry is greater than the minimal entry of `rank2` (max-min criteria).

Problem specific orders of the ranks of branching samples can be implemented by redefining the virtual function `compareBranchingSampleRanks()`.

Selecting Branching Samples

If the redefinition of the function `compareBranchingSample()` is not adequate for a problem specific selection of the branching sample, then the virtual function

```
int ABA_SUB::selectBestBranchingSample(int nSamples,
                                       ABA_BUFFER<ABA_BRANCHRULE*> **samples)
```

can be redefined. The number of branching samples is given by the integer number `nSamples`, the array `samples` stores pointers to buffers storing the branching rules of the samples. The function should return the number of the best branching sample.

Strong Branching with other Branching Rules

As explained in Section 5.2.7 other branching strategies than branching on variables can be chosen by redefining the virtual function

```
int ABA_SUB::generateBranchRules(ABA_BUFFER<ABA_BRANCHRULE*> &rules);
```

in the problem specific subproblem class. Instead of generating immediately a single branching sample and storing it in the buffer `rules` it is possible to generate first a set of samples and selecting the best one by calling the function

```
int ABA_SUB::selectBestBranchingSample(int nSamples,
                                       ABA_BUFFER<ABA_BRANCHRULE*> **samples).
```

For problem specific branching rules that are not already provided by ABACUS, but derived from the base class `ABA_BRANCHRULE`, it is necessary to redefine the virtual function

```
void ABA_BRANCHRULE::extract(ABA_LPSUB *lp)
```

if the ranks of the branching rules are computed by solving the first linear program of the potential sons as ABACUS does in its default version. Similar as the function

```
int ABA_BRANCHRULE::extract(SUB *sub)
```

(see Section 5.2.7) modifies the subproblem according to the branching rule, the virtual function

```
void extract(ABA_LPSUB *lp)
```

should modify the linear programming relaxation in order to evaluate the branching rule.

In addition the virtual function

```
void ABA_BRANCHRULE::unextract(ABA_LPSUB *lp)
```

must also be redefined. It should undo the modifications of the linear programming relaxation performed by `extract(ABA_LPSUB *lp)`.

5.2.9 Activating and Deactivating a Subproblem

Entry points at the beginning and at the end of the subproblem optimization are provided by the functions `activate()` and `deactivate()`.

5.2.10 Calling ABACUS Recursively

The separation or pricing problem in a branch-and-bound algorithm can again be a mixed integer optimization problem. In this case, it might be appropriate to solve this problem again with an application of ABACUS. The pricing problem of a solver for binary cutting stock problems, e.g., is under certain conditions a general mixed integer optimization problem [VBJN94]. The following example shows how this part of the function `pricing()` could look like for the binary cutting stock problem. First, we construct an object of the class `LPFORMAT`, storing the pricing problem formulated as a mixed integer optimization problem, then we initialize the solver of the pricing problem. The class `MIP` is derived from the class `ABA_MASTER` for the solution of general mixed integer optimization problems (the classes `LPFORMAT` and `MIP` are not part of the ABACUS kernel but belong to a not publicly available ABACUS application). After the optimization we retrieve the value of the optimal solution.

```
LPFORMAT knapsackProblem(master_, nOrigVar_, 1 + nSosCons_, &optSense,
                         origObj_, lBound, uBound, varType, constraints);

MIP *knapsackSolver = new MIP(&knapsackProblem, "CSP-Pricer");

knapsackSolver->optimize();

optKnapsackValue = knapsackSolver->primalBound();
```

5.2.11 Selecting the LP-Method

Before the linear programming relaxation is solved, the virtual function

```
ABA_LP::METHOD ABA_SUB::chooseLpMethod(int nVarRemoved, int nConRemoved,
                                       int nVarAdded, int nConAdded)
```

is called in each iteration of the cutting plane algorithm. The parameters of the function refer to the number of removed and added variables and constraints. If a linear programming relaxation should be solved with a strategy different from the default strategy, then this virtual function must be redefined in the class `MYSUB`. According to the criteria of our new application the function `chooseLpMethod()` must return `ABA_LP::Barrier`, `ABA_LP::Primal`, or `ABA_LP::Dual`.

5.2.12 Generating Output

We recommend to use also for problem specific output the built-in output and error streams via the member functions `out()` and `err()` of the class `ABA_GLOBAL`:

```
master_->out() << "This is a message for the output stream." << endl;
master_->err() << "This is a message for the error stream." << endl;
```

For messages output from members of the class ABA_MASTER and its derived classes dereferencing the pointer to the master can be omitted:

```
out() << "This is a message for the output stream from a master class." << endl;
err() << "This is a message for the error stream from a master class." << endl;
```

The functions out() and err() can receive optionally an integer number as argument giving the amount of indentation. One unit of indentation is four blanks.

The amount of output can be controlled by the parameter OutLevel in the file .abacus (see Section 5.2.26). If some output should be generated although it is turned off for a certain output level at this point of the program, then it can be turned temporarily on.

```
int MYSUB::myFunction()
{
  if (master_->outLevel() == ABA_MASTER::LinearProgram) master_->out().on();
  master_->out() << "This output appears only for output level ";
  master_->out() << "'LinearProgram'." << endl;
  if (master_->outLevel() == ABA_MASTER::LinearProgram) master_->out().off();
}
```

5.2.13 Memory Management

The complete memory management of data allocated in member functions of application specific classes has to be performed by the user, i.e., memory allocated in such a function also has to be deallocated in an application specific function. However, there are some exceptions. As soon as a constraint or a variable is added to a pool its memory management is passed to ABACUS. This also holds if the constraint or variable is added to a pool with the functions ABA_SUB::addCons() or ABA_SUB::addVars(). Constraints and variables are allocated in problem specific functions, but deallocated by the framework.

Another exception are branching rules added to a subproblem. But this is only relevant for applications that add a problem specific branching rule. If variables are fixed or set by logical implications, then objects of the class ABA_FSVARSTAT are allocated. Also for these objects the further memory management is performed by the framework.

In order to save memory a part of the data members of a subproblem can be accessed only when the subproblem is currently being optimized. These data members are listed in Table 5.1.

Member	Description
tailOff_	tailing off manager
lp_	linear programming relaxation
addVarBuffer_	buffer for adding variables
addConBuffer_	buffer for adding constraints
removeVarBuffer_	buffer for removing variables
removeConBuffer_	buffer for removing constraints
xVal_	values of the variables in the last solved ABA_LP
yVal_	values of the dual variables in the last solved ABA_LP

Table 5.1: Activated members of ABA_SUB.

5.2.14 Eliminating Constraints

In order to keep the number of active constraints within a moderate size active constraints can be eliminated by setting the built-in parameter ConstraintEliminationMode to Basic or NonBinding (see Section 5.2.26). Other problem specific strategies can be implemented by redefining the virtual function

```
void MYSUB::conEliminate(ABA_BUFFER<int> &remove)
{
  for (int i = 0; i < nCon(); i++)
    if (/* constraint i should be eliminated */)
      remove.push(i);
}
```

within the subproblem of the new application.

The function `conEliminate()` is called within the cutting plane algorithm. Moreover, we provide an even more flexible method for the elimination of constraints by the functions `removeCon()` and `removeCons()`, which can be called from any function within the cutting plane method. The functions

```
void ABA_SUB::removeCon(int i);
void ABA_SUB::removeCons(ABA_BUFFER<int> &remove);
```

which remove the constraint `i` or the constraints stored in the buffer `remove`, respectively.

Both constraints removed by the function `conEliminate()` and by explicitly calling the function `remove()` are not removed immediately from the active constraints and the linear program, but buffered, and the updates are performed at the beginning of the next iteration of the cutting plane method.

5.2.15 Eliminating Variables

Similarly to the constraint elimination, variables can be eliminated either by setting the parameter `VariableEliminationMode` to `ReducedCost` or by redefining the virtual function `varEliminate()` according to the needs of our application.

```
void  ABA_SUB::varEliminate(ABA_BUFFER<int> &remove)
{
  for (int i = 0; i < nVar(); i++)
    if (/* variable i should be eliminated)
      remove.push(i);
}
```

By analogy to the removal of constraints we provide functions to remove variables within any function of the cutting plane algorithm. The functions

```
void ABA_SUB::removeVar(int i);
void ABA_SUB::removeVars(ABA_BUFFER<int> &remove);
```

which remove the variable `i` or the variables stored in the buffer `remove`, respectively.

Like eliminated constraints eliminated variables are buffered and the update is performed at the beginning of the next iteration of the cutting plane algorithm.

5.2.16 Adding Constraints/Variables in General

The functions `separate()` and `pricing()` provide interfaces where constraints/variables are usually generated in the cutting plane or column generation algorithm. Moreover, to provide a high flexibility we allow the addition and removal of constraints and variables within any subroutine of the cutting plane or column generation algorithm as we have already pointed out.

Note, while constraints or variables added with the function `addCons()` or `addVars()` are usually allocated by the user, they are deleted by ABACUS. They must **not** be deleted by the user (see Section 5.2.13).

The sizes of the buffers that store the constraints/variables being added can be controlled by the parameters `MaxConBuffered` and `MaxVarBuffered` in the parameter file `.abacus`. At the start of the next iteration the best `MaxConAdd` constraints and the best `MaxVarAdd` variables are added to the subproblem.

This evaluation of the buffered items is only possible if a rank has been specified for each item in the functions `addCons()` and `addVars()`, respectively.

Moreover, we provide further features for the addition of cutting planes with the function `addCons()`:

```
virtual int addCons(ABA_BUFFER<ABA_CONSTRAINT*>            &constraints,
                    ABA_POOL<ABA_CONSTRAINT, ABA_VARIABLE> *pool = 0,
                    ABA_BUFFER<bool>                       *keepInPool = 0,
                    ABA_BUFFER<double>                     *rank = 0);
```

The buffer `constraints` holds the constraints being added. All other arguments are optional or ignored if they are 0. If the argument `pool` is not 0, then the constraints are added to this pool instead of the default pool. If the flag `(*keepInPool)[i]` is `true` for the `i`-th added constraint, then this constraint will even be stored in the pool if it is not added to the active constraints. In order to define an order of the buffered constraints a `rank` has to be specified for each constraint in the function `addCons()`.

As constraints can be added with the function `addCons()`, the function

```
virtual int addVars(ABA_BUFFER<ABA_VARIABLE*>              &variables,
                    ABA_POOL<ABA_VARIABLE, ABA_CONSTRAINT> *pool = 0,
                    ABA_BUFFER<bool>                       *keepInPool = 0,
                    ABA_BUFFER<double>                     *rank = 0);
```

can be used for a flexible addition of variables to the buffer in a straightforward way.

The function `pricing()` handles non-liftable constraints correctly (see Section 4.2.3). However, if variables are generated within another part of the cutting plane algorithm and non-liftable constraints are present, then run-time errors or wrong results can be produced. If ABACUS is compiled in the safe mode (`-DABACUSSAFE`) this situation is recognized and the program stops with an error message. If in an application both non-liftable constraints are generated and variables are added outside the function `pricing()`, then the user has to remove non-liftable constraints explicitly to avoid errors.

Activation of a Subproblem

After a subproblem becomes active the virtual function `activate()` is called. Its default implementation in the class `ABA_SUB` does nothing but it can be redefined in the derived class `MYSUB`. In this function application specific data structures that are only required for an active subproblem can be set up, e.g., a graph associated with the subproblem:

```
void MYSUB::activate()
{ }
```

Deactivation of a Subproblem

The virtual function `deactivate()` is the counterpart of the function `activate()`. It is called at the end of the optimization of a subproblem and again its default implementation does nothing. In this function, e.g., memory allocations performed in the function `activate()` can be undone:

```
void MYSUB::deactivate()
{ }
```

5.2.17 Fixing and Setting Variables by Logical Implications

Variables can by fixed and set by logical implications by redefining the virtual functions

```
void MYSUB::fixByLogImp(ABA_BUFFER<int> &variables,
                        ABA_BUFFER<ABA_FSVARSTAT*> &status)
{}
```

and

```
void MYSUB::setByLogImp(ABA_BUFFER<int> &variables,
                        ABA_BUFFER<ABA_FSVARSTAT*> &status)
{}
```

The buffers `variables` hold the variables being fixed or set, respectively, and the buffers `status` the statuses they are fixed or set to, respectively. The following piece of code gives a fragment of an implementation of the function `fixByLogImp()`.

```
void MYSUB::fixByLogImp(ABA_BUFFER<int> &variables,
                        ABA_BUFFER<ABA_FSVARSTAT*> &status)
{
  for (int i = 0; i < nVar(); i++)
    if (/* condition for fixing i to lower bound holds */) {
      variables.push(i);
      status.push(new ABA_FSVARSTAT(master_, ABA_FSVARSTAT::FixedToLowerBound));
    }
    else if (/* condition for fixing i to upper bound holds */) {
      variables.push(i);
      status.push(new ABA_FSVARSTAT(master_, ABA_FSVARSTAT::FixedToUpperBound));
    }
}
```

Setting variables by logical implications can be implemented analogously by replacing "`FixedTo`" with "`SetTo`".

5.2.18 Loading an Initial Basis

By default, the barrier method is used for the solution of the first linear program of the subproblem. However, a basis can be also loaded, and then, the LP-method can be accordingly selected with the function `chooseLpMethod()` (see Section 5.2.11). The variable and slack variable statuses can be initialized in the constructor of the root node like in the following example.

```
MYSUB::MYSUB(ABA_MASTER *master) :
ABA_SUB(master, 50.0, 0.0, 100.0)
{
  ABA_LPVARSTAT::STATUS  lStat;
  for (int i = 0; i < nVar(); i++) {
    lStat = /* one of ABA_LPVARSTAT::AtLowerBound, ABA_LPVARSTAT::Basic,
               or ABA_LPVARSTAT::AtUpperBound */;
    lpVarStat(i)->status(lStat);
  }
  ABA_SLACKSTAT::STATUS sStat;
  for (int i = 0; i < nCon(); i++) {
    sStat = /* one of  ABA_SLACKSTAT::Basic or ABA_SLACKSTAT::NonBasicZero */;
    slackStat(i)->status(sStat)
  }
}
```

5.2.19 Integer Objective Functions

If all objective function values of feasible solutions have integer values, then a subproblem can be fathomed earlier because its dual bound can be rounded up for a minimization problem, or down for a maximization problem, respectively. This feature can be controlled by the parameter `ObjInteger` of the parameter file (see Section 5.2.26).

This feature can depend on the specific problem instance. Moreover, if variables are generated dynamically, it is even possible that this attribute depends on the currently active variable set. Therefore, we provide the function

```
void ABA_MASTER::objInteger(bool switchedOn);
```

with which the automatic rounding of the dual bound can be turned on (if `switchedOn` is `true`) or off (if `switchedOn` is `false`).

Helpful for the analysis if all objective function values of all feasible solutions are integer with respect to the currently active variable set of the subproblem might be the function

```
bool ABA_SUB::objAllInteger();
```

that returns `true` if all active variables of the subproblem are discrete and their objective function coefficients are integer, and returns `false` otherwise.

If the set of active variables is static, i.e., no variables are generated dynamically, then the function `objAllInteger()` could be called in the constructor of the root node of the enumeration tree and according to the result the flag of the master can be set:

```
MYSUB::MYSUB(ABA_MASTER *master) :
  ABA_SUB(master, 50.0, 0.0, 100.0)
{
  master_->objInteger(objAllInteger());
}
```

By default, we assume that the objective function values of feasible solutions can also have noninteger values.

5.2.20 An Entry Point at the End of the Optimization

While the virtual function `initializeOptimization()` is called at the beginning of the optimization and can be redefined for the initialization of application specific data (e.g., the variables and constraints), the virtual function `terminateOptimization()` is called at the end of the optimization. Again, the default implementation does nothing and a redefined version can be used, e.g., for visualizing the best feasible solution on the screen.

5.2.21 Output of Statistics

At the end of the optimization a solution history and some general statistics about the optimization are output. Problem specific statistics can be output by redefining the virtual function `output()` of the class `ABA_MASTER` in the class `MYMASTER`. The default implementation of the function `output()` does nothing. Of course, application specific output can be also generated in the function `terminateOptimization()`, but then this output appears before the solution history and some other statistics. If the function `output()` is used, problem specific statistics are output between the general statistics and the value of the optimum solution.

5.2.22 Accessing Internal Data of the LP-Solver

The class `ABA_SUB` has the member function `ABA_LPSUB *lp()` that allows a direct access of the data of the linear program solved within the subproblem. If the member functions of the class `ABA_LPSUB` and its base class `ABA_LP` are not sufficient to retrieve a specific information, a direct access of the data of the LP-Solvers Cplex and SoPlex is possible.

The data retrieved from your LP-solver in this direct way has to be interpreted very carefully. Since variables might be automatically eliminated the actual linear program submitted to the LP-solver might differ from the linear programming relaxation. Only if LP-data is accessed through the member functions of the class `ABA_LPSUB` the "real" linear programming relaxation is obtained.

Warning: Do not modify the data of the LP-solver using the pointers to the internal data structures and the functions of the Cplex or SoPlex callable library. A correct modification of the LP-data is only guaranteed by the member functions of the class `ABA_SUB`.

Accessing Internal Data of Cplex

Internal data of Cplex is retrieved with the functions

```
struct cpxlp *ABA_CPLEXIF::cplexLp();
struct cpxenv *ABA_CPLEXIF::cplexEnv();  // only Cplex 4.0
```

that return pointers to the internal Cplex data structure of the linear programming relaxation.

Since the linear programming relaxation of a subproblem is designed independently from the LP-solver an explicit cast to the class `ABA_LPSUBCPLEX` is required:

```
#ifdef ABACUS_LP_CPLEX40
  currentEnv = ((ABA_LPSUBCPLEX*) lp())->cplexEnv();
#endif

  currentLp = ((ABA_LPSUBCPLEX*) lp())->cplexLp();
```

The class `ABA_LPSUBCPLEX` is derived from the classes `ABA_LPSUB` and `ABA_CPLEXIF`. If your compiler supports on your system already run time type information (RTTI), then the cast can be done in a safer way.

Accessing Internal Data of SoPlex

Internal data of SoPlex is retrieved with the functions

```
SoPlex *ABA_SOPLEXIF::soplex();
```

that returns a pointer to the internal SoPlex data structure of the linear programming relaxation.

Since the linear programming relaxation of a subproblem is designed independently from the LP-solver an explicit cast to the classes `ABA_LPSUBSOPLEX` is required:

```
soplex = ((ABA_LPSUBSOPLEX*) lp())->soplex();
```

The class `ABA_LPSUBSOPLEX` is derived from the class `ABA_LPSUB` and `ABA_SOPLEX`. If your compiler supports on your system already run time type information (RTTI), then the cast can be done in a safer way.

5.2.23 Problem Specific Fathoming Criteria

Sometimes structural problem specific information can be used for fathoming a subproblem. Such criteria can be implemented by redefining the virtual function `ABA_SUB::exceptionFathom()`. This function is called before the separation or pricing is performed. If this function returns `false` (as the default implementation in the base class `ABA_SUB` does), we continue with separation or pricing. Otherwise, if it returns `true`, the subproblem is fathomed.

5.2.24 Enforcing a Branching Step

ABACUS enforces a branching step if a tailing off effect is observed. Other problem specific criteria for branching instead of continuing the cutting plane or column generation algorithm can be specified by redefining the function `ABA_SUB::exceptionBranch()`. This criterion is checked before the separation or pricing is performed. If the function returns `true`, a branching step is performed. Otherwise, we continue with the separation or pricing. The default implementation of the base class `ABA_SUB` always returns `false`.

5.2.25 Advanced Tailing Off Control

ABACUS automatically controls the tailing off effect according to the parameters `TailOffNLps` and `TailOffPercent` of the configuration file `.abacus`. However, sometimes it turns out that certain solutions of the LP-relaxations should be ignored in the tailing off control. The function `ignoreInTailingOff()` can be used to control better the tailing off effect. If this function is called, the next LP-solution is ignored in the tailing-off control. Calling `ignoreInTailingOff()` can, e.g., be considered in the following situation: If only constraints that are required for the integer programming formulation of the optimization problem are added then the next LP-value could be ignored in the tailing-off control. Only "real" cutting planes should be considered in the tailing-off control (this is only an example strategy that might not be practical in many situations, but sometimes turned out to be efficient).

5.2.26 Parameters

The setting of several parameters heavily influences the running time. Good candidates are the modification of the enumeration strategy with the parameter `EnumerationStrategy`, the control of the tailing off effect with the parameters `TailOffNLps` and `TailOffPercent`, an adaption of the skipping method for the cut generation with the parameters `SkipFactor` and `SkipByNode`, and the parameters `CplexPrimalPricing` and `CplexDualPricing` to control the pricing strategies of Cplex.

Here we present a complete list of the parameters that can be modified for the fine tuning of the algorithm in the file `.abacus`. Almost all parameters can be modified with member functions of the class `ABA_MASTER`. Usually, these member functions have the same name as the parameter, but the first letter is a lower case letter.

Warning: The integer numbers used in the parameter files must not exceed the value of `INT_MAX` given in the file `<limits.h>`. The default values are correct for platforms representing the type `int` with 32 bits (usually 2147483647 on machines using the b-complement).

EnumerationStrategy

This parameter controls the enumeration strategy in the branch-and-bound algorithm.

Valid settings:

`Best`	best-first search
`Breadth`	breadth-first search
`Depth`	depth-first search
`Dive`	depth-first search until the first feasible solution is found, then best-first search

Default value: `Best`

Guarantee

The branch-and-bound algorithm stops as soon as a primal bound and a global dual bound are known such that it can be guaranteed that the value of an optimum solution is at most `Guarantee` percent better than the primal bound. The value 0.0 means determination of an optimum solution. If the program terminates with a guarantee greater than 0, then the status of the master is `ABA_MASTER::Guarantee` instead of `ABA_MASTER::Optimal`.

Valid settings:

A nonnegative floating point number.

Default value: `0.0`

MaxLevel

This parameter indicates the maximal level that should be reached in the enumeration tree. Instead of performing a branching operation any subproblem having level `MaxLevel` is fathomed. If the value of `MaxLevel` is 1, then no branching is done, i.e., a pure cutting plane algorithm is performed. If the maximal enumeration level is reached, the master of the optimization receives the status `MaxLevel` in order to indicate that the problem does not necessarily terminate with an optimum solution.

Valid settings:

A positive integer number.

Default value: `999999`

MaxCpuTime

This parameter indicates the maximal CPU time that may be used by the optimization process. If the CPU time exceeds this value, then the master of the optimization receives the status `MaxCpuTime` in order to indicate that the problem does not necessarily terminate with an optimum solution. In this case, the real CPU time can exceed this value since we check the used CPU time only in the main loop of the cutting plane algorithm. Under the operating system UNIX a more exact check can be done with the command `limit`, which kills the process if the maximal CPU time is exceeded, whereas our CPU time control "softly" terminates the run, i.e., the branch-and-bound tree is cleaned, all relevant destructors are called, and the final output is generated.

Valid settings:

A string in the format `h{h}:mm:ss`, where the first number represents the hours, the second one the minutes, and the third one the seconds. Note, internally this string is converted to seconds. Therefore, its value must be less than `INT_MAX` seconds.

Default value: `99999:59:59`

MaxCowTime

This parameter indicates the maximal elapsed time (wall clock time) that may be used by the process. If the elapsed time exceeds this value, then the master of the optimization receives the status `MaxCowTime` in order to indicate that the problem does not necessarily terminate with an optimum solution. In this case, the real elapsed time can exceed this value since we check the elapsed time only in the main loop of the cutting plane algorithm.

Valid settings:

A string in the format `h{h}:mm:ss`, where the first number represents the hours, the second one the minutes, and the third one the seconds. Note, internally this string is converted to seconds. Therefore, its value must be less than `INT_MAX` seconds.

Default value: `99999:59:59`

ObjInteger

If this parameter is `true`, then we assume that all feasible solutions have integer objective function values. In this case, we can fathom a subproblem in the branch-and-bound algorithm already when the gap between the solution of the linear programming relaxation and the primal bound is less than 1.

Valid settings:

`false` or `true`

Default value: `false`

TailOffNLps

This parameter indicates the number of linear programs considered in the tailing off analysis (see parameter `TailOffPercent`).

Valid settings:

An integer number. If this number is nonpositive, then the tailing off control is turned off.

Default value: `0`

TailOffPercent

This parameter indicates the minimal change in percent of the objective function value between the solution of `TailOffNLps` successive linear programming relaxations in the subproblem optimization which is required such that we do not try to stop the cutting plane algorithm and to enforce a branching step.

Valid settings:

A nonnegative floating point number.

Default value: `0.0001`

DelayedBranchingThreshold

This number indicates how often a subproblem should be put back into the set of open subproblems before a branching step is executed. The value 0 means that we branch immediately at the end of the first optimization, if the subproblem is not fathomed. We try to keep the subproblem `MinDormantRounds` untouched, i.e., other subproblems are optimized if possible before we turn back to the optimization of this subproblem.

Valid settings:

A positive integer number.

Default value: `0`

MinDormantRounds

The minimal number of iterations we try to keep a subproblem dormant if delayed branching is applied.

Valid settings:

A positive integer number.

Default value: 1

OutputLevel

We can control the amount of output during the optimization by this parameter.

For the parameter values `Subproblem` and `LinearProgram` a seven column output is generated with the following meaning:

`#sub`	total number of subproblems
`#open`	current number of open subproblems
`current`	the number of the currently optimized subproblem
`#iter`	number of iterations in the cutting plane algorithm
`ABA_LP`	value of the LP-relaxation
`dual`	global dual bound
`primal`	primal bound

Valid settings:

`Silent`	No output.
`Statistics`	Output of the result and some statistics at the end of the optimization.
`Subproblem`	Additional one-line output after the first solved ABA_LP of the root node and at the end of the optimization of each subproblem.
`LinearProgram`	Additional one-line output after the solution of a linear program.
`Full`	Detailed output in all phases of the optimization.

Default value: `Full`

LogLevel

We can control the amount of output written to the log file in the same way as the output to the standard output stream.

Valid settings:

See parameter `OutputLevel`. If the `LogLevel` is not `Silent` two log files are created. While the file with the name of the problem instance and the extension `.log` contains the output written to `ABA_MASTER::out()` (filtered according the `LogLevel`), the all messages written to `ABA_MASTER::err()` are also written to the file with the name of the problem instance and the extension `.error.log`.

Default value: `Silent`

PrimalBoundInitMode

This parameter controls the initialization of the primal bound. The modes `Optimum` and `OptimumOne` are useful for tests.

Valid settings:

`None`	The primal bound is initialized with "infinity" for minimization problems and "minus infinity" for maximization problems, respectively.
`Optimum`	The primal bound is initialized with the value of an optimum solution, if it can be read from the file with the name of the parameter `OptimumFileName`.
`OptimumOne`	The primal bound is initialized with the value of an optimum solution plus one for minimization problems, and the value of an optimum solutions minus one for maximization problems. This is only possible if the value of an optimum solution can be read from the file with the name given by the parameter `OptimumFileName`.

Default value: None

PricingFrequency

This parameter indicates the number of iterations between two additional pricing steps in the cutting plane phase for algorithms performing both constraint and variable generation. If this number is 0, then no additional pricing is performed.

Valid settings:

A nonnegative integer number.

Default value: `0`

SkipFactor

This parameter indicates the frequency of cutting plane and variable generationskipping!factor in the subproblems according to the parameter `SkippingMode`. The value 1 means that cutting planes and variables are generated in every subproblem independent from the skipping mode.

Valid settings:

A positive integer number.

Default value: `1`

SkippingMode

This parameter controls the skipping mode, i.e., if constraints or variables are generated in a subproblem.

Valid settings:

`SkipByNode`	Generate constraints and variables only every `SkipFactor` processed node.
`SkipByLevel`	Generate constraints and variables only every `SkipFactor` level in the branch-and-bound tree.

Default value: `SkipByNode`

FixSetByRedCost

Variables are fixed and set by reduced cost criteria if and only if this parameter is `true`. The default setting is `false`, as fixing or setting variables to 0 can make the pricing problem intractable in branch-and-price algorithms.

Valid settings:

`false` or `true`

Default value: `false`

PrintLP

If this parameter is `true`, then the linear program is output every iteration. This is only useful for debugging.

Valid settings:

`false` or `true`

Default value: `false`

CplexPrimalPricing

This parameter controls the pricing strategy of the primal simplex method of the LP-solver Cplex. The notions are taken from the reference manuals of Cplex 3.0 and Cplex 4.0 [Cpl94, Cpl95].

Valid settings:

`CPX_PPRIIND_PARTIAL`	Reduced cost pricing
`CPX_PPRIIND_AUTO`	Hybrid reduced cost pricing and Devex pricing
`CPX_PPRIIND_DEVEX`	Devex pricing
`CPX_PPRIIND_STEEP`	Steepest edge pricing
`CPX_PPRIIND_STEEPQSTART`	Steepest edge pricing with slack norms
`CPX_PPRIIND_FULL`	Full pricing

Default value: `CPX_PPRIIND_AUTO`

CplexDualPricing

This parameter controls the default pricing strategy of the dual simplex method of the LP-solver Cplex. The notions are taken form the reference manual of Cplex 3.0 and Cplex 4.0 [Cpl94, Cpl95].

Valid settings:

`CPX_DPRIIND_AUTO`	Determined automatically
`CPX_DPRIIND_FULL`	Standard dual pricing
`CPX_DPRIIND_STEEP`	Steepest edge pricing
`CPX_DPRIIND_FULLSTEEP`	Steepest edge pricing in slack space
`CPX_DPRIIND_STEEPQSTART`	Steepest edge pricing, unit initial norms

Default value: `CPX_DPRIIND_STEEP`

CplexOutputLevel

The LP-solver Cplex provides the output of information when a linear program is solved. This parameter controls the amount of this amount for a linear programming relaxation solved by Cplex. The output is directly written to the standard output and is not filtered by our output stream concept.

Valid settings:

`0`	No output of Cplex.
`1`	Output every refactorization of Cplex.
`2`	Output every iteration of Cplex.

Default value: `1`

MaxConAdd

This parameter determines the maximal number of constraints added to the linear programming relaxation per iteration in the cutting plane algorithm.

Valid settings:

A nonnegative integer number.

Default value: `100`

MaxConBuffered

After the cutting plane generation the `MaxConAdd` best constraints are selected from all generated constraints that are kept in a buffer. This parameter indicates the size of this buffer.
Valid settings:

A nonnegative integer number.

Default value: `100`

MaxVarAdd

This parameter determines the maximal number of variables added to the linear programming relaxation per iteration in the cutting plane algorithm.
Valid settings:

A nonnegative integer number.

Default value: `100`

MaxVarBuffered

After the variable generation the `MaxVarAdd` best variables are selected from all generated variables that are kept in a buffer. This parameter indicates the size of this buffer.
Valid settings:

A nonnegative integer number.

Default value: `100`

MaxIterations

The parameter limits the number of iterations of the cutting plane phase of a single subproblem.
Valid settings:

A nonnegative integer number or -1 if unlimited.

Default value: `-1`

EliminateFixedSet

Fixed and set variables are eliminated from the linear program submitted to the LP-solver if this parameter is `true` and the variable is eliminable. By default, a variable is eliminable if it has not been basic in the last solved linear program.
Valid settings:

`false` or `true`

Default value: `false`

NewRootReOptimize

If the root of the remaining branch-and-bound tree changes and this node is not the active subproblem, then we reoptimize this subproblem, if this parameter is `true`. The reoptimization might provide better criteria for fixing variables by reduced costs.
Valid settings:

`false` or `true`

Default value: `false`

OptimumFileName

This parameter indicates the name of a file storing the values of the optimum solutions. Each line of this file consists of a problem name and the value of the corresponding optimum solution. This is the only optional parameter. Having the optimum values of some instances at hand can be very useful in the testing phase.

Valid settings:

A string.

Default value: This parameter is commented out in the file `.abacus`.

ShowAverageCutDistance

If this parameter is `true`, then the average Euclidean distance of the fractional solution from the added cutting planes is output every iteration of the cutting plane phase.

Valid settings:

`false` or `true`

Default value: `false`

ConstraintEliminationMode

The parameter indicates the method how constraints are eliminated in the cutting plane algorithm.

Valid settings:

`None`	No constraints are eliminated.
`NonBinding`	The non-binding dynamic constraints are eliminated.
`Basic`	The dynamic constraints with basic slack variables are eliminated.

Default value: `Basic`

VariableEliminationMode

This parameter indicates the method how variables are eliminated in a column generation algorithm.

Valid settings:

`None`	No variables are eliminated.
`ReducedCost`	Nonbasic dynamic variables that are neither fixed nor set and for which the absolute value of the reduced costs exceeds the value given by the parameter `VarElimEps` are removed.

Default value: `ReducedCost`

ConElimEps

The parameter indicates the tolerance for the elimination of constraints by the method `NonBinding`.

Valid settings:

A nonnegative floating point number.

Default value: `0.001`

VarElimEps

This parameter indicates the tolerance for the elimination of variables by the method `ReducedCost`.
Valid settings:

A nonnegative floating point number.

Default value: `0.001`

VbcLog

This parameter indicates if a log-file of the enumeration tree should be generated, which can be read by the VBC-tool [Lei95]. The VBC-tool is a utility for the visualization of the branch-and-bound tree.
Valid settings:

`None`	No file for the VBC-Tool is generated.
`File`	The output is written to a file with the name `<name>.<pid>.tree`. `<name>` is the problem name as specified in the constructor of the class `ABA_MASTER` and `<pid>` is the process id.
`Pipe`	The control instructions for the VBC-Tool are written to the global output stream. Each control instuction starts with a $ sign. If the standard output of an ABACUS application is piped through the VBC-Tool, lines starting with a $ sign are regarded as control instructions, all other lines written to a text window.

Default value: `None`

NBranchingVariableCandidates

This number indicates how many candidates for branching variables should be tested according to the `BranchingStrategy`. If this number is 1, a single variable is determined (if possible) that is the branching variable. If this number is greater than 1 each candidate is tested and the best branching variable is selected, i.e., for each candidate the two linear programs of potential sons are solved. The variable for which the minimal change of the two objective function values is maximal is selected as branching variable.
Valid settings:

Positive integer number.

Default value: 1

DefaultLpSolver

This parameter determines the LP-solver that should be applied per default for each subproblem.
Valid settings:

`Cplex`
`SoPlex`

Default value: `Cplex`

SoPlexRepresentation

This parameter selects the basis representation of the LP-solver SoPlex. This can be either a row or a column basis. Traditionally, LP-solvers use a column basis. However, if there are more rows than columns in the linear programs, as it sometimes happens in branch-and-cut algorithms, then a row basis might be more efficient. Unfortunately our tests turn out that only the row basis is stable in SoPlex 1.0.

Valid settings:

```
Row
Column
```

Default value: `Row`

5.2.27 Reading a Parameter File

ABACUS provides a concept for the implementation of application parameter files, which is very easy to use. In these files it is both possible to overwrite the values of parameters already defined in the file `.abacus` and to define extra parameters for the new application.

The format for parameter files is very simple. Each line contains the name of a parameter separated by an arbitrary number of whitespaces from its value. Both parameter name and parameter value can be an arbitrary character string. A line may have at most 1024 characters. Empty lines are allowed. All lines starting with a '#' are considered as comments.

The following lines give an example for the parameter file `.myparameters`.

```
#
# First, we overwrite two parameters from the file .abacus.
#
EnumerationStrategy    Depth
OutputLevel            LinearProgram
#
# Here are the parameters of our new application.
#
#
# Our application has two different separation strategies
#     All              calls all separators in each iteration
#     Hierarchical     follows a hierarchy of the separators
#
SeparationStrategy     All
#
# The parameter MaxNodesPerCut limits the number of nodes involved
# in a cutting plane that is defined by a certain subgraph.
#
MaxNodesPerCut         1000
```

Here, we suppose that the class `MYMASTER` has two members that are initialized from the parameter file.

```
class MYMASTER : public ABA_MASTER {
  /* public and protected members */
  private:
    ABA_STRING separationStrategy_;
    int maxNodesPerCut_;
    /* other private members */
 };
```

The parameter file can be read by redefining the virtual function `initializeParameters()`, which does nothing in its default implementation.

```
void MYMASTER::initializeParameters()
{
  readParameters(".myparameters");
  int status;
  status = getParameter("SeparationStrategy", separationStrategy_);
  if (status) {
    err() << "MYMASTER::initializeParameters(): ";
    err() << "parameter SeparationStrategy missing." << endl;
    exit(Fatal);
  }
  status = getParameter("MaxNodesPerCut", maxNodesPerCut_);
  if (status) {
    err() << "MYMASTER::initializeParameters(): ";
    err() << "parameter MaxNodesPerCut missing." << endl;
    exit(Fatal);
  }
}
```

Parameter files having our format can be read by the function `ABA_MASTER::readParameters()`, which inserts all parameters in a table. Then, the parameters can be extracted from the table with the function `ABA_MASTER::getParameter()`, which is overloaded in the following way:

```
int  getParameter(const char *name, int &parameter);
int  getParameter(const char *name, double &parameter);
int  getParameter(const char *name, ABA_STRING &parameter);
int  getParameter(const char *name, bool &parameter);
int  getParameter(const char *name, char &parameter);
```

If a parameter with the name `name` is found in the parameter table then its value is stored in the argument `parameter` and the function `getParameter()` returns 0, otherwise it returns 1.

Parameters of the base class `ABA_MASTER` that are redefined in the file `.myparameters` do not have to be extracted explicitly, but are initialized automatically. Note, the parameters specified in the file `.abacus` are read in the constructor of the class `ABA_MASTER`, but an application specific parameter file is read when the optimization starts (function `ABA_MASTER::optimize()`.

5.3 Using the ABACUS Templates

ABACUS also provides several basic data structures as templates. For several fundamental types and some ABACUS classes the templates are instantiated already in the library `libabacus.a`. However, if you want to use one of the ABACUS templates for one of your classes then you have to instantiate the templates for these classes yourself.

Moreover, in order to keep the library small, we instantiated the templates only for those types which are required in the kernel of the ABACUS system. Therefore, it can happen that the linker complains about undefined symbols. In this case you have to instantiate these templates, too.

For instance, you want to use an `ABA_ARRAY` template for your class `MYCONSTRAINT` and the fundamental type `unsigned int`, for which we have no instantiations in the library `libabacus.a`. Then you can instantiate the corresponding templates in a file `myarray.cc`.

```
//
// This is the file myarray.cc.
//
#include "abacus/array.h" // the header of the class ABA_ARRAY
#include "abacus/array.inc" // the member functions of the class ABA_ARRAY
template class ABA_ARRAY<MYCONSTRAINT>;
template class ABA_ARRAY<unsigned int>;
// end of file myarray.cc
```

The file `myarray.cc` should be compiled and linked together with your files and the library `libabacus.a`. In the file in which you are using the array templates only the file `array.h` should be included.

For more information on templates we refer to the documentation of the templates for the GNU compiler[1]. We prefer the method using the g++ compiler flag `-fno-implicit-templates`.

[1] http://funnelweb.utcc.utk.edu/~harp/gnu/gcc-2.7.0/gcc_98.html#SEC101

Chapter 6

Reference Manual

The reference manual covers only those classes and class members which are relevant for the user. Therefore, the declarations of the classes in this chapter contain only a subset of the actual members, e.g., private members are usually not documented here. For some classes the copy constructor and/or assignment operator have not been defined, but the default copy constructor and/or assignment operator are not correct. In this case we declare this function and/or this operator as a private member of its class such that its invalid usage is detected already at compile time. In this reference manual this is documented by including the copy constructor and/or assignment operator in the private part of a function. Even if there are other private members of the class they are not documented here.

This reference manual is automatically compiled from the source files of ABACUS. The advantage of this method is that we can always provide an up to date version of the reference manual in future releases of the software. The major drawback of this procedure is that the lack of order of the functions in the current source files is reflected in the reference manual. In particular, there is a often a difference of the order of the member functions in the header of a class and in the documentation. We plan to reorder the functions in one of the next releases. Until this is done, we recommend the reader to find the documentation of a function with the help of the index or with the HTML version of the reference manual.

At the end of the reference manual a list of all preprocessor flags is given.

6.1 Application Base Classes

In order to implement an ABACUS application problem specific classes have to be derived from the classes `ABA_MASTER` and `ABA_SUB`. ABACUS provides already some non-abstract classes derived from the classes `ABA_CONSTRAINT` and `ABA_VARIABLE`, but if there is application specific structure to be exploited, classes also have to be derived from `ABA_VARIABLE` and `ABA_CONSTRAINT`.

Some other classes are included in this section because they are base classes of the application base classes `ABA_MASTER`, `ABA_SUB`, `ABA_CONSTRAINT` and `ABA_VARIABLE`. The class `ABA_ABACUSROOT` is a base class of every class of the system. The class `ABA_GLOBAL` is a base class of the class `ABA_MASTER`. Common features of constraints and variables are embedded in the class `ABA_CONVAR`, from which the classes `ABA_CONSTRAINT` and `ABA_VARIABLE` are derived.

6.1.1 ABA_ABACUSROOT

This class is the base class of all other classes of ABACUS. By embedding an enumeration and some useful functions in this class we can avoid a global scope of these names.

For Compilers that to not provide the type `bool` so far, we add this type here.

```
class ABA_ABACUSROOT {
  public:
```

```
    virtual ~ABA_ABACUSROOT();
    enum EXITCODES{Ok, Fatal};
    virtual void exit(enum EXITCODES code) const;
    const char *onOff(bool value);
    double fracPart(double x) const;
};
```

enum EXITCODES

This enumeration defines the codes used be the function `exit()`.

`Ok`

: The program terminates without error.

`Fatal`

: A severe error occurred leading to an immediate termination of the program.

Destructor (`virtual`)

The destructor is only implemented since it should be virtual function.

```
ABA_ABACUSROOT::~ABA_ABACUSROOT()
```

exit (`virtual`)

The function `exit()` terminates the program and returns `code` to the environment from which the program was called.

```
void ABA_ABACUSROOT::exit(enum EXITCODES code) const
```

Arguments:

`code`

: The exit code given to the environment.

onOff

The function onOff() converts a boolean variable to the strings `"on"` and `"off"`.

```
const char *ABA_ABACUSROOT::onOff(bool value)
```

Return Value:

`"on"`

: If `value` is `true`,

`"off"`

: otherwise.

Arguments:

`value`

: The boolean variable being converted.

fracPart

```
double ABA_ABACUSROOT::fracPart(double x) const
```

Return Value:

The absolute value of the fractional part of the value `x`. E.g., it holds `fracPart(2.33) == 0.33` and `fracPart(-1.77) == 0.77`.

Arguments:

The value of which the fractional part is computed.

6.1.2 ABA_GLOBAL

ABA_GLOBAL. This class stores global data (e.g., a zero tolerance, an output stream) und functions operating with this data. For each application there is usually one global object and almost every object in this system has a pointer to an associated global object or a pointer to an object of a class derived from `ABA_GLOBAL` (e.g., `ABA_MASTER`).

Like the class `ABA_ABACUSROOT`, the class `ABA_GLOBAL` helps us to avoid names with global scope.

```
class ABA_GLOBAL : public ABA_ABACUSROOT {
  public:
    ABA_GLOBAL(double eps = 1.0e-4, double machineEps = 1.0e-7,
               double infinity = 1.0e30);
    virtual ~ABA_GLOBAL();
    friend ostream &operator<<(ostream &out, const ABA_GLOBAL &rhs);
    virtual ABA_OSTREAM& out(int nTab = 0);
    virtual ABA_OSTREAM& err(int nTab = 0);
    double eps() const;
    void eps(double e);
    double machineEps() const;
    void machineEps(double e);
    double infinity() const;
    void infinity(double x);
    bool isInfinity(double x) const;
    bool isMinusInfinity(double x) const;
    bool equal(double x, double y) const;
    bool isInteger(double x) const;
    bool isInteger(double x, double eps) const;
    virtual char enter(istream &in);

  private:
    ABA_GLOBAL(const ABA_GLOBAL &rhs);
    const ABA_GLOBAL &operator=(const ABA_GLOBAL &rhs);
};
```

Constructor

The constructor initializes our filtered output and error stream with the standard output stream `cout` and the standard error stream `cerr`.

```
ABA_GLOBAL::ABA_GLOBAL(double eps, double machineEps, double infinity)
```

Arguments:

`eps`

The zero-tolerance used within all member functions of objects which have a pointer to this global object (default value `1.0e-4`).

`machineEps`

The machine dependent zero tolerance (default value `1.0e-7`).

`infinity`

All values greater than `infinity` are regarded as "infinite big", all values less than `-infinity` are regarded as "infinite small" (default value `1.0e30`).

Destructor (`virtual`)

```
ABA_GLOBAL::~ABA_GLOBAL()
```

Output Operator

The output operator writes some of the data members to an ouput stream.

```
ostream &operator<<(ostream &out, const ABA_GLOBAL &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The object being output.

out (`virtual`)

The function out() returns a reference to the output stream associated with this global object after writing `nTab` (default value 0) tabulators on this stream. This tabulator is not the normal tabulator but consists of four blanks.

```
ABA_OSTREAM& ABA_GLOBAL::out(int nTab)
```

Return Value:

A reference to the global output stream.

Arguments:

`nTab`

The number of tabulators which should be written to the global output stream. The default value is 0.

err (virtual)

The function `err()` behaves like the function `out()` except that the global error stream is used instead of the global output stream.

```
ABA_OSTREAM& ABA_GLOBAL::err(int nTab)
```

Return Value:

A reference to the global error stream.

Arguments:

`nTab`

The number of tabulators which should be written to the global error stream. The default value is 0.

eps

```
double ABA_GLOBAL::eps() const
```

Return Value:

The zero tolerance.

eps

This version of the function `eps()` sets the zero tolerance.

```
void ABA_GLOBAL::eps(double e)
```

Arguments:

`e`

The new value of the zero tolerance.

machineEps

The function `machineEps()` provides a machine dependent zero tolerance. The machine dependent zero tolerance is used, e.g., to test if a floating point value is 0. This value is usually less than `eps()`, which provides, e.g., a safety tolerance if a constraint is violated.

```
double ABA_GLOBAL::machineEps() const
```

Return Value:

The machine dependent zero tolerance.

machineEps

This version of the function `machineEps()` sets the machine dependent zero tolerance.

```
void ABA_GLOBAL::machineEps(double e)
```

Arguments:

`e`

The new value of the machine dependent zero tolerance.

infinity

The function `infinity()` provides a floating point value of "infinite" size. Especially, we assume that `-infinity()` is the lower and `infinity()` is the upper bound of an unbounded variable in the linear program.

```
double ABA_GLOBAL::infinity() const
```

Return Value:

A very large floating point number. The default value of `infinity()` is `1.0e30`.

infinity

This version of the function `infinity()` sets the "infinite value".

```
void ABA_GLOBAL::infinity(double x)
```

Arguments:

`x`

The new value representing "infinity".

isInfinity

```
bool ABA_GLOBAL::isInfinity(double x) const
```

Return Value:

`true`

If `x` is regarded as "infinite" large,

`false`

otherwise.

Arguments:

`x`

The value compared with "infinity".

isMinusInfinity

```
bool ABA_GLOBAL::isMinusInfinity(double x) const
```

Return Value:

`true`

If `x` is regarded as infinite small;

`false`

otherwise.

Arguments:

`x`

The value compared with "minus infinity".

equal

```
bool ABA_GLOBAL::equal(double x, double y) const
```

Return Value:

`true`
: If the absolute difference of `x` and `y` is less than the machine dependent zero tolerance,

`false`
: otherwise.

Arguments:

`x`
: The first value being compared.

`y`
: The second value being compared.

isInteger

```
bool ABA_GLOBAL::isInteger(double x) const
```

Return Value:

`true`
: If the value `x` differs at most by the machine dependent zero tolerance from an integer value,

`false`
: otherwise.

isInteger

```
bool ABA_GLOBAL::isInteger(double x, double eps) const
```

Return Value:

`true`
: If the value `x` differs at most by `eps` from an integer value,

`false`
: otherwise.

enter (`virtual`)

The virtual function `enter()` displays the string `ENTER>` on the global output stream and waits for a character on the input stream `in`, e.g., a keystroke if `in == cin`.

```
char ABA_GLOBAL::enter(istream &in)
```

Return Value:

The character read from the input stream.

Arguments:

`in`
: The input stream the character should be read from.

6.1.3 ABA_MASTER

As the name already indicates, the class ABA_MASTER is the central object of the framework. The most important tasks of the class ABA_MASTER is the management of the implicit enumeration. Moreover, it provides already default implementations for constraints, cutting planes, and variables pools. The class ABA_MASTER also stores various parameter settings and compiles statistics about the solution process.

The class ABA_MASTER is an abstract class from which a problem specific master has to be derived.

```
class ABA_MASTER : public ABA_GLOBAL {
  public:
    enum STATUS {Optimal, Error, Unprocessed, Processing,
                 Guaranteed, MaxLevel, MaxCpuTime,
                 MaxCowTime, ExceptionFathom};

    enum OUTLEVEL {Silent, Statistics, Subproblem, LinearProgram, Full};

    enum ENUMSTRAT {BestFirst, BreadthFirst, DepthFirst, DiveAndBest};

    enum BRANCHINGSTRAT {CloseHalf, CloseHalfExpensive};

    enum PRIMALBOUNDMODE {NoPrimalBound, OptimalPrimalBound,
                          OptimalOnePrimalBound};

    enum SKIPPINGMODE{SkipByNode, SkipByLevel};

    enum CONELIMMODE {NoConElim, NonBinding, Basic};

    enum VARELIMMODE {NoVarElim, ReducedCost};

    enum VBCMODE {None, File, Pipe};

    enum LPSOLVER {Cplex, SoPlex};

    ABA_MASTER(const char *problemName, bool cutting, bool pricing,
               ABA_OPTSENSE::SENSE optSense = ABA_OPTSENSE::Unknown,
               double eps = 1.0e-4, double machineEps = 1.0e-7,
               double infinity = 1.0e30);

    virtual ~ABA_MASTER();
    STATUS optimize();
    double lowerBound() const;
    double upperBound() const;
    double primalBound() const;
    double dualBound() const;
    void primalBound(double x);
    void dualBound(double x);
    bool betterDual(double x) const;
    bool primalViolated(double x) const;
    bool betterPrimal(double x) const;
    bool feasibleFound() const;
    virtual int enumerationStrategy(ABA_SUB *s1, ABA_SUB *s2);
    bool guaranteed();
    double guarantee();
    void printGuarantee();
```

```
bool check();
bool knownOptimum(double &optVal);
virtual void output();
bool cutting() const;
bool pricing() const;
ABA_OPTSENSE *optSense();
ABA_HISTORY *history();
ABA_OPENSUB *openSub();
ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE> *conPool();
ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE> *cutPool();
ABA_STANDARDPOOL<ABA_VARIABLE, ABA_CONSTRAINT> *varPool();
ABA_SUB *root();
ABA_SUB *rRoot();
STATUS status() const;
ABA_STRING *problemName();
ABA_COWTIMER *totalCowTime();
ABA_CPUTIMER *totalTime();
ABA_CPUTIMER *lpTime();
ABA_CPUTIMER *lpSolverTime();
ABA_CPUTIMER *separationTime();
ABA_CPUTIMER *improveTime();
ABA_CPUTIMER *pricingTime();
ABA_CPUTIMER *branchingTime();
int nSub() const;
int nLp() const;
int highestLevel() const;
int nNewRoot() const;
int nSubSelected() const;
void printParameters();
ENUMSTRAT enumerationStrategy() const;
void enumerationStrategy(ENUMSTRAT strat);
BRANCHINGSTRAT branchingStrategy() const;
void branchingStrategy(BRANCHINGSTRAT strat);
LPSOLVER defaultLpSolver() const;
void defaultLpSolver(LPSOLVER lpSolver);
bool soPlexRowRep() const;
void soPlexRowRep(bool rep);
int nBranchingVariableCandidates() const;
void nBranchingVariableCandidates(int n);
double requiredGuarantee() const;
void requiredGuarantee(double g);
int maxLevel() const;
void maxLevel(int ml);
const ABA_STRING& maxCpuTime() const;
void maxCpuTime(const ABA_STRING &t);
const ABA_STRING& maxCowTime() const;
void maxCowTime(const ABA_STRING &t);
bool objInteger() const;
void objInteger(bool b);
int tailOffNLp() const;
void tailOffNLp(int n);
double tailOffPercent() const;
void tailOffPercent(double p);
```

```
OUTLEVEL outLevel() const;
void outLevel(OUTLEVEL mode);
OUTLEVEL logLevel() const;
void logLevel(OUTLEVEL mode);
bool delayedBranching(int nOpt_) const;
void dbThreshold(int threshold);
int dbThreshold() const;
int minDormantRounds() const;
void minDormantRounds(int nRounds);
PRIMALBOUNDMODE pbMode() const;
void pbMode(PRIMALBOUNDMODE mode);
int pricingFreq () const;
void pricingFreq(int f);
int skipFactor() const;
void skipFactor(int f);
void skippingMode(SKIPPINGMODE mode);
SKIPPINGMODE skippingMode() const;
CONELIMMODE conElimMode() const;
void conElimMode(CONELIMMODE mode);
VARELIMMODE varElimMode() const;
void varElimMode(VARELIMMODE mode);
double conElimEps() const;
void conElimEps(double eps);
double varElimEps() const;
void varElimEps(double eps);
bool fixSetByRedCost() const;
void fixSetByRedCost(bool on);
bool printLP() const;
void printLP(bool on);
ABA_STRING& cplexPrimalPricing();
void cplexPrimalPricing(const char *method);
ABA_STRING& cplexDualPricing();
void cplexDualPricing(const char *method);
int cplexOutputLevel() const;
void cplexOutputLevel(int level);
int maxConAdd() const;
void maxConAdd(int max);
int maxConBuffered() const;
void maxConBuffered(int max);
int maxVarAdd() const;
void maxVarAdd(int max);
int maxVarBuffered() const;
void maxVarBuffered(int max);
int maxIterations() const;
void maxIterations(int max);
bool eliminateFixedSet() const;
void eliminateFixedSet(bool turnOn);
bool newRootReOptimize() const;
void newRootReOptimize(bool on);
const ABA_STRING &optimumFileName() const;
void optimumFileName(const char *name);
bool showAverageCutDistance() const;
void showAverageCutDistance(bool on);
```

```
    VBCMODE vbcLog() const;
    void vbcLog(VBCMODE mode);
    void readParameters(const char *fileName);
    int  getParameter(const char *name, int &param);
    int  getParameter(const char *name, double &param);
    int  getParameter(const char *name, ABA_STRING &param);
    int  getParameter(const char *name, bool &param);
    int  getParameter(const char *name, char &param);
    int  getParameter(const char *name, unsigned int &param);

  protected:
    virtual void initializePools(ABA_BUFFER<ABA_CONSTRAINT*> &constraints,
                                 ABA_BUFFER<ABA_VARIABLE*> &Variables,
                                 int varPoolSize,
                                 int cutPoolSize,
                                 bool dynamicCutPool = false);
    virtual void initializePools(ABA_BUFFER<ABA_CONSTRAINT*> &constraints,
                                 ABA_BUFFER<ABA_CONSTRAINT*> &cuts,
                                 ABA_BUFFER<ABA_VARIABLE*> &Variables,
                                 int varPoolSize,
                                 int cutPoolSize,
                                 bool dynamicCutPool = false);
    void initializeOptSense(ABA_OPTSENSE::SENSE sense);
    int bestFirstSearch(ABA_SUB* s1, ABA_SUB* s2);
    virtual int equalSubCompare(ABA_SUB *s1, ABA_SUB *s2);
    int depthFirstSearch(ABA_SUB* s1, ABA_SUB* s2);
    int breadthFirstSearch(ABA_SUB* s1, ABA_SUB* s2);
    int diveAndBestFirstSearch(ABA_SUB *s1, ABA_SUB* s2);

  private:
    virtual ABA_SUB *firstSub() = 0;
    virtual void initializeOptimization();
    virtual void terminateOptimization();
    virtual void initializeParameters();
    ABA_MASTER(const ABA_MASTER &rhs);
    const ABA_MASTER &operator=(const ABA_MASTER& rhs);
};
```

enum STATUS

The various statuses of the optimization process.

`Optimal`

: The optimization terminated with an error and without reaching one of the resource limits. If there is a feasible solution then the optimal solution has been computed.

`Error`

: An error occurred during the optimization process.

`Unprocessed`

: The initial status, before the optimization starts.

`Processing`

: The status while the optimization is performed.

`Guaranteed`

If not the optimal solution is determined, but the required guarantee is reached, then the status is `Guaranteed`.

`MaxLevel`

The status, if subproblems are ignored since the maximum enumeration level is exceeded.

`MaxCpuTime`

The status, if the optimization terminates since the maximum cpu time is exceeded.

`MaxCowTime`

The status, if the optimization terminates since the maximum wall-clock time is exceeded.

`ExceptionFathom`

The status, if at least one subproblem has been fathomed according to a problem specific criteria determined in the function —ABA_SUB::exceptionFathom().

enum OUTLEVEL

This enumeration defines the different output levels:

`Silent`

No output at all.

`Statistics`

No output during the optimization, but output of final statistics.

`Subproblem`

In addition to the previous level also a single line of output after every subproblem optimization.

`LinearProgram`

In addition to the previous level also a single line of output after every solved linear program.

`Full`

Tons of output.

enum ENUMSTRAT

The enumeration defining the different enumeration strategies for the branch and bound algorithm.

`BestFirst`

Best-first search, i.e., select the subproblem with best dual bound, i.e., the subproblem having minimal dual bound for a minimization problem, or the subproblem having maximal dual bound for a maximization problem.

`BreadthFirst`

Breadth-first search, i.e., select the subproblem with minimal level in the enumeration tree.

`DepthFirst`

Depth-first search, i.e., select the subproblem with maximal level in the enumeration tree.

`DiveAndBest`

As long as no primal feasible solution is known the next subproblem is selected according to the depth-first search strategy, otherwise the best-first search strategy is applied.

enum BRANCHINGSTRAT

This enumeration defines the two currently implemented branching variable selection strategies.

`CloseHalf`

Selects the variable with fractional part closest to 0.5.

`CloseHalfExpensive`

Selects the variable with fractional part close to 0.5 (within some interval around 0.5) and has highest absolute objective function coefficient.

enum PRIMALBOUNDMODE

This enumeration provides various methods for the initialization of the primal bound. The modes `OptimalPrimalBound` and `OptimalOnePrimalBound` can be useful in the testing phase. For these modes the value of an optimum solution must stored in the file given by the parameter `OptimumFileName` in the parameter file.

`NoPrimalBound`

The primal bound is initialized with $-\infty$ for maximization problems and ∞ for minimization problems, respectively.

`OptimalPrimalBound`

The primal bound is initialized with the value of the optimum solution.

`OptimalOnePrimalBound`

The primal bound is initialized with the value of optimum solution minus 1 for maximization problems and with the value of the optimum solution plus one for minimization problems, respectively.

enum SKIPPINGMODE

The way nodes are skipped for the generation of cuts.

`SkipByNode`

Cuts are only generated in every `SkipFactor` subproblem, where `SkipFactor` can be controlled with the parameter file `.abacus`.

`SkipByLevel`

Cuts are only generated in every `SkipFactor` level of the enumeration tree.

enum CONELIMMODE

This enumeration defines the ways for automatic constraint elimination during the cutting plane phase.

`NoConElim`

No constraints are eliminated.

`NonBinding`

Nonbinding constraints are eliminated.

`Basic`

Constraints with basic slack variable are eliminated.

enum VARELIMMODE

This enumeration defines the ways for automatic variable elimination during the column generation algorithm.

`NoVarElim`

No variables are eliminated.

`ReducedCost`

Variables with high absolute reduced costs are eliminated.

enum VBCMODE

This enumeration defines what kind of output can be generated for the VBCTOOL.

`None`

No output for the tree interface.

`File`

Output for the tree interface is written to a file.

`Pipe`

Output for the tree interface is pipe to the standard output.

enum VBCMODE

This enumeration defines the available LP-solvers.

`Cplex`

The LP-solver Cplex.

`SoPlex`

The LP-solver SoPlex.

firstSub (`virtual`)

```
virtual ABA_SUB *firstSub() = 0
```

Return Value:

The pure virtual function `firstSub()` should return a pointer to the first subproblem of the optimization, i.e., the root node of the enumeration tree. This is a pure virtual function since a pointer to a problem specific subproblem should be returned, which is derived from the class `ABA_SUB`.

Constructor

```
ABA_MASTER::ABA_MASTER(const char *problemName, bool cutting, bool pricing,
                       ABA_OPTSENSE::SENSE optSense,
                       double eps, double machineEps, double infinity)
```

Arguments:

`problemName`

The name of the problem being solved. Must not be a 0-pointer.

`cutting`

If `true`, then cutting planes can be generated if the function `ABA_SUB::separate()` is redefined.

`pricing`

If `true`, then inactive variables are priced in, if the function `ABA_SUB::pricing()` is redefined.

`optSense`

The sense of the optimization. The default value is `ABA_OPTSENSE::Unknown`. If the sense is unknown when this constructor is called, e.g., if it is read from a file in the constructor of the derived class, then it must be initialized in the constructor of the derived class.

`eps`

The zero-tolerance used within all member functions of objects which have a pointer to this master (default value `1.0e-4`).

`machineEps`

The machine dependent zero tolerance (default value `1.0e-7`).

`infinity`

All values greater than `infinity` are regarded as "infinite big", all values less than `-infinity` are regarded as "infinite small" (default value `1.0e30`).

Destructor (`virtual`)

```
ABA_MASTER::~ABA_MASTER()
```

optimize

The function `optimize()` performs the optimization by branch-and-bound.

```
ABA_MASTER::STATUS ABA_MASTER::optimize()
```

Return Value:

The status of the optimization.

initializeOptimization (`virtual`)

The default implementation of `initializeOptimization()` does nothing. This virtual function can be used as an entrance point to perform some initializations after `optimize()` is called.

```
void ABA_MASTER::initializeOptimization()
```

initializePools (`virtual`)

The virtual function `initializePools()` sets up the default pools for variables, constraints, and cutting planes.

```
void ABA_MASTER::initializePools(ABA_BUFFER<ABA_CONSTRAINT*>  &constraints,
                                 ABA_BUFFER<ABA_VARIABLE*>    &variables,
                                 int varPoolSize,
                                 int cutPoolSize,
                                 bool dynamicCutPool)
```

Arguments:

`constraints`

The constraints of the problem formulation are inserted in the constraint pool. The size of the constraint pool equals the number of `constraints`.

`variables`

The variables of the problem formulation are inserted in the variable pool.

`varPoolSize`

The size of the pool for the variables. If more variables are added the variable pool is automatically reallocated.

`cutPoolSize`

The size of the pool for cutting planes.

`dynamicCutPool`

If this argument is true, then the cut is automatically reallocated if more constraints are inserted than `cutPoolSize`. Otherwise, non-active constraints are removed if the pool becomes full. The default value is false.

initializePools (`virtual`)

The virtual function `initializePools()` is overloaded such that also a first set of cutting planes can be inserted into the cutting plane pool.

```
void ABA_MASTER::initializePools(ABA_BUFFER<ABA_CONSTRAINT*> &constraints,
                                 ABA_BUFFER<ABA_CONSTRAINT*> &cuts,
                                 ABA_BUFFER<ABA_VARIABLE*> &variables,
                                 int varPoolSize,
                                 int cutPoolSize,
                                 bool dynamicCutPool)
```

Arguments:

`constraints`

The constraints of the problem formulation are inserted in the constraint pool. The size of the constraint pool equals the number of `constraints`.

`cuts`

The constraints that are inserted in the cutting plane pool. The number of constraints in the buffer must be less or equal than the size of the cutting plane pool `cutPoolSize`.

`variables`

The variables of the problem formulation are inserted in the variable pool.

`varPoolSize`

The size of the pool for the variables. If more variables are added the variable pool is automatically reallocated.

`cutPoolSize`

The size of the pool for cutting planes.

`dynamicCutPool`

If this argument is true, then the cut is automatically reallocated if more constraints are inserted than `cutPoolSize`. Otherwise, non-active constraints are removed if the pool becomes full. The default value is false.

intializeOptSense

The function `initializeOptSense()` can be used to initialize the sense of the optimization in derived classes, if this has not been already performed when the constructor of `ABA_MASTER` has been called.

```
void ABA_MASTER::initializeOptSense(ABA_OPTSENSE::SENSE sense)
```

Arguments:

`sense`

The sense of the optimization (`ABA_OPTSENSE::Min` or `ABA_OPTSENSE::Max`).

terminateOptimization (`virtual`)

The default implementation of `terminateOptimization()` does nothing. This virtual function can be used as an entrance point after the optimization process is finished.

```
void ABA_MASTER::terminateOptimization()
```

enumerationStrategy (`virtual`)

The virtual function `enumerationStrategy()` analyzes the enumeration strategy set in the parameter file `.abacus` and calls the corresponding comparison function for the subproblems `s1` and `s2`. This function should be redefined for application specific enumeration strategies.

```
int ABA_MASTER::enumerationStrategy(ABA_SUB *s1, ABA_SUB *s2)
```

Return Value:

If `s1` has higher priority than `s2` it returns 1, if `s2` has higher priority it returns -1, and if both subproblems have equal priority it returns 0.

Arguments:

`s1`

A pointer to subproblem.

`s2`

A pointer to subproblem.

bestFirstSearch

The function `bestFirstSearch()` implements the best first search enumeration. If the bounds of both subproblems are equal, then the subproblems are compared with the function `equalSubCompare()`.

```
int ABA_MASTER::bestFirstSearch(ABA_SUB *s1, ABA_SUB *s2)
```

Return Value:

`-1`

If subproblem `s1` has a worse dual bound than `s2`, i.e., if it has a smaller dual bound for minimization or a larger dual bound for maximization problems.

`1`

If subproblem `s2` has a worse dual bound than `s1`.

`0`

If both subproblems have the same priority in the enumeration strategy.

Arguments:

`s1`

A subproblem.

`s2`

A subproblem.

equalSubCompare (`virtual`)

The virtual function `equalSubCompare()` is called from the function `bestFirstSearch()` and from the function `depthFirstSearch()` if the subproblems `s1` and `s2` have the same priority. If both subproblems were generated by setting a binary variable, then that subproblem has higher priority of which the branching variable is set to upper bound.

This function can be redefined to resolve equal subproblems according to problem specific criteria.

```
int ABA_MASTER::equalSubCompare(ABA_SUB *s1, ABA_SUB *s2)
```

Return Value:

`0`

If both subproblems were not generated by setting a variable, or the branching variable of both subproblems is set to the same bound.

`1`

If the branching variable of the first subproblem ist set to the upper bound.

`-1`

If the branching variable of the second subproblem ist set to the upper bound.

Arguments:

`s1`

A subproblem.

`s2`

A subproblem.

depthFirstSearch

The function `depthFirstSearch()` implements the depth first search enumeration strategy, i.e., the subproblem with maximum `level` is selected. If the level of both subproblems are equal, then the subproblems are compared with the function `equalSubCompare()`.

```
int ABA_MASTER::depthFirstSearch(ABA_SUB* s1, ABA_SUB* s2)
```

Return Value:

`-1`

If subproblem `s1` has higher priority,

`0`

if both subproblems have equal priority,

`1`

otherwise.

Arguments:

`s1`

The first subproblem.

`s2`

The second subproblem.

breadthFirstSearch

The function `breadthFirstSearch()` implements the breadth first search enumeration strategy, i.e., the subproblem with minimum `level` is selected. If both subproblems have the same `level`, the smaller one is the one which has been generated earlier, i.e., the one with the smaller `id`.

```
int ABA_MASTER::breadthFirstSearch(ABA_SUB* s1, ABA_SUB* s2)
```

Return Value:

`-1`

If subproblem `s1` has higher priority,

`0`

if both subproblems have equal priority,

`1`

otherwise.

Arguments:

`s1`

The first subproblem.

`s2`

The second subproblem.

diveAndBestFirstSearch

The function `diveAndBestFirstSearch()` performs depth-first search until a feasible solution is found, then the search process is continued with best-first search.

```
int ABA_MASTER::diveAndBestFirstSearch(ABA_SUB *s1, ABA_SUB* s2)
```

Return Value:

`-1`

If subproblem `s1` has higher priority,

`0`

if both subproblems have equal priority,

`1`

otherwise.

Arguments:

`s1`

The first subproblem.

`s2`

The second subproblem.

lowerBound

```
double ABA_MASTER::lowerBound() const
```

Return Value:

The value of the global lower bound.

upperBound

```
double ABA_MASTER::upperBound() const
```

Return Value:

The value of the global upper bound.

primalBound

```
double ABA_MASTER::primalBound() const
```

Return Value:

The value of the primal bound, i.e., the `lowerBound()` for a maximization problem and the `upperBound()` for a minimization problem, respectively.

primalBound

This version of the function `primalBound()` sets the primal bound to `x` and makes a new entry in the solution history. It is an error if the primal bound gets worse.

```
void ABA_MASTER::primalBound(double x)
```

Arguments:

`x`

The new value of the primal bound.

dualBound

```
double ABA_MASTER::dualBound() const
```

Return Value:

The value of the dual bound, i.e., the `upperBound()` for a maximization problem and the `lowerBound()` for a minimization problem, respectively.

dualBound

This version of the function `dualBound()` sets the dual bound to `x` and makes a new entry in the solution history. It is an error if the dual bound gets worse.

```
void ABA_MASTER::dualBound(double x)
```

Arguments:

`x`

The new value of the dual bound.

betterDual

```
bool ABA_MASTER::betterDual(double x) const
```

Return Value:

`true`

If `x` is better than the best known dual bound.

`false`

otherwise.

Arguments:

`x`

The value being compared with the best know dual bound.

primalViolated

The function `primalViolated()` can be used to compare a value with the one of the best known primal bound.

```
bool ABA_MASTER::primalViolated(double x) const
```

Return Value:

`true`

If `x` is not better than the best known primal bound,

`false`

otherwise.

Arguments:

`x`

The value being compared with the primal bound.

betterPrimal

The function `betterPrimal()` can be used to check if a value is better than the best know primal bound.

```
bool ABA_MASTER::betterPrimal(double x) const
```

Return Value:

`true`

If `x` is better than the best known primal bound,

`false`

otherwise.

Arguments:

`x`

The value compared with the primal bound.

feasibleFound

```
bool ABA_MASTER::feasibleFound() const
```

Return Value:

`true`
: If a feasible solution of the optimization problem has been found.

`false`
: otherwise.

root

The function `root()` can be used to access the root node of the branch-and-bound tree.

```
ABA_SUB* ABA_MASTER::root()
```

Return Value:

A pointer to the root node of the enumeration tree.

rRoot

```
ABA_SUB* ABA_MASTER::rRoot()
```

Return Value:

A pointer to the root of the remaining branch-and-bound tree, i.e., the subproblem which is an ancestor of all open subproblems and has highest level in the tree.

guaranteed

The function `guaranteed()` can be used to check if the guarantee requirements are fulfilled, i.e., the difference between upper bound and the lower bound in respect to the lowerBound is less than this guarantee value in percent. If the lower bound is zero, but the upper bound is nonzero, we cannot give any guarantee.

Warning: A guarantee for a solution can only be given if the pricing problem is solved exactly or no column generation is performed at all.

```
bool ABA_MASTER::guaranteed()
```

Return Value:

`true`
: If the guarantee requirements are fulfilled,

`false`
: otherwise.

guarantee

The function `guarantee()` can be used to access the guarantee which can be given for the best known feasible solution. It is an error to call this function if the lower bound is zero, but the upper bound is nonzero.

```
double ABA_MASTER::guarantee()
```

Return Value:

The guarantee for best known feasible solution in percent.

printGuarantee

The function `printGuarantee()` writes the guarantee nicely formated on the output stream associated with this object. If no bounds are available, or the lower bound is zero, but the upper bound is nonzero, then we cannot give any guarantee.

```
void ABA_MASTER::printGuarantee()
```

check

The function `check()` can be used to control the correctness of the optimization if the value of the optimum solution has been loaded. This is done, if a file storing the optimum value is specified with the parameter `OptimumFileName` in the configuration file `.abacus`.

```
bool ABA_MASTER::check()
```

Return Value:

`true`

If the optimum solution of the problem is known and equals the primal bound,

`false`

otherwise.

knownOptimum

The function `knownOptimum()` opens the file specified with the parameter `OptimumFileName` in the configuration file `.abacus` and tries to find a line with the name of the problem instance (as specified in the constructor of `ABA_MASTER`) as first string.

```
bool ABA_MASTER::knownOptimum(double &optVal)
```

Return Value:

`true`

If a line with `problemName_` has been found,

`false`

otherwise.

Arguments:

`optVal`

If the return value is `true`, then `optVal` holds the optimum value found in the line with the name of the problem instance as first string. Otherwise, `optVal` is undefined.

output (virtual)

The function `output()` does nothing but can be redefined in derived classes for output before the timing statistics.

```
void ABA_MASTER::output()
```

problemName

```
ABA_STRING *ABA_MASTER::problemName()
```

Return Value:

A pointer to the name of the instance being optimized (as specified in the constructor of this class).

optSense

```
ABA_OPTSENSE *ABA_MASTER::optSense()
```

Return Value:

A pointer to the object holding the optimization sense of the problem.

history

```
ABA_HISTORY *ABA_MASTER::history()
```

Return Value:

A pointer to the object storing the solution history of this branch and cut problem.

openSub

```
ABA_OPENSUB *ABA_MASTER::openSub()
```

Return Value:

A pointer to the set of open subproblems.

conPool

```
ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE> *ABA_MASTER::conPool()
```

Return Value:

A pointer to the default pool storing the constraints of the problem formulation.

cutPool

```
ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE> *ABA_MASTER::cutPool()
```

Return Value:

A pointer to the default pool for the generated cutting planes.

varPool

```
ABA_STANDARDPOOL<ABA_VARIABLE, ABA_CONSTRAINT> *ABA_MASTER::varPool()
```

Return Value:

A pointer to the default pool storing the variables.

cutting

```
bool ABA_MASTER::cutting() const
```

Return Value:

`true`

If `cutting` has been set to `true` in the call of the constructor of the class `ABA_MASTER`, i.e., if cutting planes should be generated in the subproblem optimization.

`false`

otherwise.

pricing

```
bool ABA_MASTER::pricing() const
```

Return Value:

`true`

If `pricing` has been set to true in the call of the constructor of the class `ABA_MASTER`, i.e., if a columns should be generated in the subproblem optimization.

`false`

otherwise.

totalCowTime

```
ABA_COWTIMER *ABA_MASTER::totalCowTime()
```

Return Value:

A pointer to the timer measuring the total wall clock time.

totalTime

```
ABA_CPUTIMER *ABA_MASTER::totalTime()
```

Return Value:

A pointer to the timer measuring the total cpu time for the optimization.

lpTime

```
ABA_CPUTIMER *ABA_MASTER::lpTime()
```

Return Value:

A pointer to the timer measuring the cpu time spent in members of the LP-interface.

lpSolverTime

```
ABA_CPUTIMER *ABA_MASTER::lpSolverTime()
```

Return Value:

A pointer to the timer measuring the cpu time required by Cplex.

separationTime

```
ABA_CPUTIMER *ABA_MASTER::separationTime()
```

Return Value:

A pointer to the timer measuring the cpu time spent in the separation of cutting planes.

improveTime

```
ABA_CPUTIMER *ABA_MASTER::improveTime()
```

Return Value:

A pointer to the timer measuring the cpu time spent in the heuristics for the computation of feasible solutions.

pricingTime

```
ABA_CPUTIMER *ABA_MASTER::pricingTime()
```

Return Value:

A pointer to the timer measuring the cpu time spent in pricing.

branchingTime

```
ABA_CPUTIMER *ABA_MASTER::branchingTime()
```

Return Value:

A pointer to the timer measuring the cpu time spent in finding and selecting the branching rules.

nSub

```
int ABA_MASTER::nSub() const
```

Return Value:

The number of generated subproblems.

nLp

```
int ABA_MASTER::nLp() const
```

Return Value:

The number of optimized linear programs (only LP-relaxations).

highestLevel

```
int ABA_MASTER::highestLevel () const
```

Return Value:

The highest level in the tree which has been reached during the implicit enumeration.

nNewRoot

```
int ABA_MASTER::nNewRoot() const
```

Return Value:

The number of root changes of the remaining branch-and-cut tree.

nSubSelected

```
int ABA_MASTER::nSubSelected() const
```

Return Value:

The number of subproblems which have already been selected from the set of open subproblems.

initializeParameters (virtual)

The virtual function `initializeParameters()` is only a dummy. This function can be used to initialize parameters of derived classes and to overwrite parameters read from the file `.abacus` by the function `_initializeParameters()`.

```
void ABA_MASTER::initializeParameters()
```

readParameters

The function `readParameters()` opens the parameter file `fileName`, reads all parameters, and inserts them in the parameter table. A parameter file may have at most 1024 characters per line.

```
void ABA_MASTER::readParameters(const char *fileName)
```

Arguments:

`fileName`

The name of the parameter file.

getParameter

The function `getParameter()` searches for the parameter `name` in the parameter table. This function is overloaded for different types of the argument `parameter`.

```
int ABA_MASTER::getParameter(const char *name, int &parameter)
```

Return Value:

`0`

If the parameter is found,

`1`

otherwise.

Arguments:

`name`

The name of the parameter.

`parameter`

The variable `parameter` receives the value of the parameter, if the function returns 1, otherwise it is undefined.

getParameter

```
int ABA_MASTER::getParameter(const char *name, unsigned int &parameter)
```

getParameter

```
int ABA_MASTER::getParameter(const char *name, double &parameter)
```

getParameter

```
int ABA_MASTER::getParameter(const char *name, ABA_STRING &parameter)
```

getParameter

```
int ABA_MASTER::getParameter(const char *name, bool &parameter)
```

getParameter

```
int ABA_MASTER::getParameter(const char *name, char &parameter)
```

printParameters

The function `printParameters()` writes all parameters of the class `ABA_MASTER` together with their values to the global output stream.

```
void ABA_MASTER::printParameters()
```

fixSetByRedCost

```
bool ABA_MASTER::fixSetByRedCost() const
```

Return Value:

`true`
: Then variables are fixed and set by reduced cost criteria.

`false`
: Then no variables are fixed or set by reduced cost criteria.

fixSetByRedCost

The function `fixSetByReCost()` turns fixing and setting variables by reduced cost on or off.

```
void ABA_MASTER::fixSetByRedCost(bool on)
```

Arguments:

`on`
: If `true`, then variable fixing and setting by reduced cost is turned on. Otherwise it is turned of.

printLP

```
bool ABA_MASTER::printLP() const
```

Return Value:

`true`
: Then the linear program is output every iteration of the subproblem optimization.

`false`
: The linear program is not output.

printLP

The function `printLP()` turns the output of the linear program in every iteration on or off.

```
void ABA_MASTER::printLP(bool on)
```

Arguments:

`on`

If `true`, then the linear program is output, otherwise it is not output.

cplexPrimalPricing

```
ABA_STRING& ABA_MASTER::cplexPrimalPricing()
```

Return Value:

The primal pricing strategy for Cplex (see your Cplex manual).

cplexPrimalPricing

The function `cplexPrimalPricing()` changes the pricing method of the primal simplex algorithm of the LP-solver Cplex.

Note, this function does not automatically change the pricing strategy of already constructed objects of the class `ABA_CPLEXIF` or the derived class `ABA_LPSUBCPLEX`. Only the pricing strategy of objects constructed after this function call is changed. In order to change the primal pricing strategy of "living" objects of the class `ABA_CPLEXIF`, use the function `ABA_CPLEXIF::setppriind()`.

```
void ABA_MASTER::cplexPrimalPricing(const char *method)
```

Arguments:

`method`

A string with the new primal pricing method. Consult your Cplex manual for valid strategies.

cplexDualPricing

```
ABA_STRING& ABA_MASTER::cplexDualPricing()
```

Return Value:

The dual pricing strategy for Cplex (see your Cplex manual).

cplexDualPricing

The function `cplexDualPricing()` changes the pricing method of the dual simplex algorithm of the LP-solver Cplex.

Note, this function does not automatically change the pricing strategy of already constructed objects of the class `ABA_CPLEXIF` or the derived class `ABA_LPSUBCPLEX`. Only the pricing strategy of objects constructed after this function call is changed. In order to change the dual pricing strategy of "living" objects of the class `ABA_CPLEXIF`, use the function `ABA_CPLEXIF::setdpriind()`.

```
void ABA_MASTER::cplexDualPricing(const char *method)
```

Arguments:

`method`

A string with the new dual pricing method. Consult your Cplex manual for valid strategies.

cplexOutputLevel

```
int ABA_MASTER::cplexOutputLevel() const
```

Return Value:

The output level of Cplex (0: no ouput of Cplex, 1: a line of output every refactorization, 2: A line of output every iteration).

cplexOutputLevel

The function `cplexOutputLevel()` sets the amount of output produced by the LP-Solver Cplex.

Note, this function does not automatically change the amount of output of already constructed objects of the class `ABA_CPLEXIF` or the derived class `ABA_LPSUBCPLEX`. Only the output of objects constructed after this function call is changed. In order to change the amount of output of "living" objects of the class `ABA_CPLEXIF`, use the function `ABA_CPLEXIF::iterationInformation()`.

```
void ABA_MASTER::cplexOutputLevel(int level)
```

Arguments:

`level`

If `0` no output is generated, if `1` output every refactorization is generated, if `2` output every iteration is generated.

maxConAdd

```
int ABA_MASTER::maxConAdd() const
```

Return Value:

The maximal number of constraints which should be added in every iteration of the cutting plane algorithm.

maxConAdd

The function `maxConAdd()` sets the maximal number of constraints that are added in an iteration of the cutting plane algorithm.

```
void ABA_MASTER::maxConAdd(int max)
```

Arguments:

`max`

The maximal number of constraints.

maxConBuffered

```
int ABA_MASTER::maxConBuffered() const
```

Return Value:

The size of the buffer for generated constraints in the cutting plane algorithm.

maxConBuffered

The function `maxConBuffered()` changes the maximal number of constraints that are buffered in an iteration of the cutting plane algorithm.

Note, this function changes only the default value for subproblems that are activated after its call.

```
void ABA_MASTER::maxConBuffered(int max)
```

Arguments:

`max`

The new maximal number of buffered constraints.

maxVarAdd

```
int ABA_MASTER::maxVarAdd() const
```

Return Value:

The maximal number of variables which should be added in the column generation algorithm.

maxVarAdd

The function `maxVarAdd()` changes the maximal number of variables that are added in an iteration of the subproblem optimization.

```
void ABA_MASTER::maxVarAdd(int max)
```

Arguments:

`max`

The new maximal number of added variables.

maxVarBuffered

```
int ABA_MASTER::maxVarBuffered() const
```

Return Value:

The size of the buffer for the variables generated in the column generation algorithm.

maxVarBuffered

The function `maxVarBuffered()` changes the maximal number of variables that are buffered in an iteration of the subproblem optimization.

Note, this function changes only the default value for subproblems that are activated after its call.

```
void ABA_MASTER::maxVarBuffered(int max)
```

Arguments:

`max`

The new maximal number of buffered variables.

maxIterations

```
int ABA_MASTER::maxIterations() const
```

Return Value:

The maximal number of iterations per subproblem optimization (`-1` means no iteration limit).

maxIterations

The function `maxIterations()` changes the default value for the maximal number of iterations of the optimization of a subproblem.

Note, this function changes only this value for subproblems that are constructed after this function call. For already constructed objects the value can be changed with the function `ABA_SUB::maxIterations()`.

```
void ABA_MASTER::maxIterations(int max)
```

Arguments:

`max`

The new maximal number of iterations of the subproblem optimization (`-1` means no limit).

optimumFileName

```
const ABA_STRING &ABA_MASTER::optimumFileName() const
```

Return Value:

The name of the file that stores the optimum solutions.

optimumFileName

The function `optimumFileName()` changes the name of the file in which the value of the optimum solution is searched.

```
void ABA_MASTER::optimumFileName(const char *name)
```

Arguments:

`name`

The new name of the file.

eliminateFixedSet

```
bool ABA_MASTER::eliminateFixedSet() const
```

Return Value:

`true`

Then we try to eliminate fixed and set variables from the linear program.

`false`

Fixed or set variables are not eliminated.

eliminateFixedSet

This version of the function `eliminateFixedSet()` can be used to turn the elimination of fixed and set variables on or off.

```
void ABA_MASTER::eliminateFixedSet(bool turnOn)
```

Arguments:

`turnOn`

The elimination is turned on if `turnOn` is `true`, otherwise it is turned off.

newRootReOptimize

```
bool ABA_MASTER::newRootReOptimize() const
```

Return Value:

`true`

Then a new root of the remaining branch-and-bound tree is reoptimized such that the associated reduced costs can be used for the fixing of variables.

`false`

A new root is not reoptimized.

newRootReOptimize

The function `newRootReOptimize()` turns the reoptimization of new root nodes of the remaining branch and bound tree on or off.

```
void ABA_MASTER::newRootReOptimize(bool on)
```

Arguments:

`on`

If `true`, new root nodes are reoptimized.

showAverageCutDistance

```
bool ABA_MASTER::showAverageCutDistance() const
```

Return Value:

`true`

Then the average distance of the fractional solution from all added cutting planes is output every iteration of the subproblem optimization.

`false`

The average cut distance is not output.

showAverageCutDistance

The function `showAverageCutDistance()` turns the output of the average distance of the added cuts from the fractional solution on or off.

```
void ABA_MASTER::showAverageCutDistance(bool on)
```

Arguments:

`on`

If `true` the output is turned on, otherwise it is turned off.

vbcLog

```
ABA_MASTER::VBCMODE ABA_MASTER::vbcLog() const
```

Return Value:

The mode of output for the Vbc-Tool.

vbcLog

The function `vbcLog()` changes the mode of output for the Vbc-Tool. This function should only be called before the optimization is started with the function `ABA_MASTER::optimize()`.

```
void ABA_MASTER::vbcLog(VBCMODE mode)
```

Arguments:

`mode`

The new mode.

conElimMode

```
ABA_MASTER::CONELIMMODE ABA_MASTER::conElimMode() const
```

Return Value:

The mode for the elimination of constraints.

conElimMode

The function `conElimMode()` changes the constraint elimination mode.

```
void ABA_MASTER::conElimMode(CONELIMMODE mode)
```

Arguments:

`mode`

The new constraint elimination mode.

varElimMode

```
ABA_MASTER::VARELIMMODE ABA_MASTER::varElimMode() const
```

Return Value:

The mode for the elimination of variables.

varElimMode

The function `varElimMode()` changes the variable elimination mode.

```
void ABA_MASTER::varElimMode(VARELIMMODE mode)
```

Arguments:

`mode`

The new variable elimination mode.

conElimEps

```
double ABA_MASTER::conElimEps() const
```

Return Value:

The zero tolerance for the elimination of constraints by the slack criterion.

conElimEps

The function `conElimEps()` chances the tolerance for the elimination of constraints by the slack criterion.

```
void ABA_MASTER::conElimEps(double eps)
```

Arguments:

`eps`

The new tolerance.

varElimEps

```
double ABA_MASTER::varElimEps() const
```

Return Value:

The zero tolerance for the elimination of variables by the reduced cost criterion.

varElimEps

The function `varElimEps()` chances the tolerance for the elimination of variables by the reduced cost criterion.

```
void ABA_MASTER::varElimEps(double eps)
```

Arguments:

`eps`

The new tolerance.

enumerationStrategy

```
ABA_MASTER::ENUMSTRAT ABA_MASTER::enumerationStrategy() const
```

Return Value:

The enumeration strategy.

enumerationStrategy

This version of the function `enumerationStrategy()` changes the enumeration strategy.

```
void ABA_MASTER::enumerationStrategy(ENUMSTRAT strat)
```

Arguments:

`strat`

The new enumeration strategy.

branchingStrategy

```
ABA_MASTER::BRANCHINGSTRAT ABA_MASTER::branchingStrategy() const
```

Return Value:

The branching strategy.

branchingStrategy

This version of the function `branchingStrategy()` changes the branching strategy.

```
void ABA_MASTER::branchingStrategy(BRANCHINGSTRAT strat)
```

Arguments:

`strat`
: The new branching strategy.

defaultLpSolver

```
ABA_MASTER::LPSOLVER ABA_MASTER::defaultLpSolver() const
```

Return Value:

The default LP-Solver.

defaultLpSolver

This version of the function `defaultLpSolver()` changes the default LP-solver.

```
void ABA_MASTER::defaultLpSolver(LPSOLVER lpSolver)
```

Arguments:

`lpSolver`
: The new solver.

soPlexRowRep

```
bool ABA_MASTER::soPlexRowRep() const
```

Return Value:

`true`
: if the default SoPlex basis representation `ROW`.

`false`
: if the default SoPlex basis representation `COLUMN`.

soPlexRowRep

```
void ABA_MASTER::soPlexRowRep(bool rep)
```

Arguments:

`rep`
: If `rep` is `true`, then the default SoPlex basis representation is `ROW`, otherwise it is `COLUMN`.

nBranchingVariableCandidates

```
int ABA_MASTER::nBranchingVariableCandidates() const
```

Return Value:

The number of variables that should be tested for the selection of the branching variable.

nbranchingVariableCandidates

This version of the function `nbranchingVariableCandidates()` sets the number of tested branching variable candidates.

```
void ABA_MASTER::nBranchingVariableCandidates(int n)
```

Arguments:

`n`

The new value of the number of tested variables for becoming branching variable.

requiredGuarantee

```
double ABA_MASTER::requiredGuarantee() const
```

The guarantee specification for the optimization.

requiredGuarantee

This version of the function `requiredGuarantee()` changes the guarantee specification.

```
void ABA_MASTER::requiredGuarantee(double g)
```

Arguments:

`g`

The new guarantee specification. This must be a nonnative value. Note, if the guarantee specification is changed after a single node of the enumeration tree has been fathomed, then the overall guarantee might differ from the new value.

maxLevel

```
int ABA_MASTER::maxLevel() const
```

Return Value:

The maximal depth up to which the enumeration should be performed. By default the maximal enumeration depth is `INT_MAX`.

maxLevel

This version of the function `maxLevel()` changes the maximal enumeration depth. If it is set to 1 the branch-and-cut algorithm becomes a pure cutting plane algorithm.

```
void ABA_MASTER::maxLevel(int max)
```

Arguments:

`max`

The new value of the maximal enumeration level.

maxCpuTime

```
const ABA_STRING& ABA_MASTER::maxCpuTime() const
```

Return Value:

The maximal cpu time which can be used by the optimization.

maxCpuTime

The function `maxCpuTime()` sets the maximal usable cpu time for the optimization.

```
void ABA_MASTER::maxCpuTime(const ABA_STRING &t)
```

Arguments:

`t`

The new value of the maximal cpu time.

maxCowTime

```
const ABA_STRING& ABA_MASTER::maxCowTime() const
```

Return Value:

The maximal wall-clock time for the optimization.

maxCowtime

This version of the function `maxCowTime()` set the maximal wall-clock time for the optimization.

```
void ABA_MASTER::maxCowTime(const ABA_STRING &t)
```

Arguments:

`t`

The new value of the maximal wall-clock time.

objInteger

```
bool ABA_MASTER::objInteger() const
```

Return Value:

`true`

Then we assume that all feasible solutions have integral objective function values,

`false`

otherwise.

objInteger

This version of function `objInteger()` sets the assumption that the objective function values of all feasible solutions are integer.

```
void ABA_MASTER::objInteger(bool b)
```

Arguments:

`b`

The new value of the assumption.

tailOffNLp

```
int ABA_MASTER::tailOffNLp() const
```

Return Value:

The number of linear programs considered in the tailing off analysis.

tailOffNLp

The function `tailOffNLp()` sets the number of linear programs considered in the tailing off analysis. This new value is only relevant for subproblems activated **after** the change of this value.

```
void ABA_MASTER::tailOffNLp(int n)
```

Arguments:

`n`

The new number of LPs for the tailing off analysis.

tailOffPercent

```
double ABA_MASTER::tailOffPercent() const
```

Return Value:

The minimal change of the dual bound for the tailing off analysis in percent.

tailOffPercent

This version of the function `tailOffPercent()` sets the minimal change of the dual bound for the tailing off analysis. This change is only relevant for subproblems activated **after** calling this function.

```
void ABA_MASTER::tailOffPercent(double p)
```

Arguments:

`p`

The new value for the tailing off analysis.

outLevel

```
ABA_MASTER::OUTLEVEL ABA_MASTER::outLevel() const
```

Return Value:

The output mode.

outLevel

The version of the function `outLevel()` sets the output mode.

```
void ABA_MASTER::outLevel(OUTLEVEL mode)
```

Arguments:

`mode`

The new value of the output mode.

logLevel

```
ABA_MASTER::OUTLEVEL ABA_MASTER::logLevel() const
```

Return Value:

The output mode for the log-file.

logLevel

This version of the function `logLevel()` sets the output mode for the log-file.

```
void ABA_MASTER::logLevel(OUTLEVEL mode)
```

Arguments:

`mode`

The new value of the output mode.

delayedBranching

```
bool ABA_MASTER::delayedBranching(int nOpt) const
```

Return Value:

`true`

If the number of optimizations `nOpt` of a subproblem exceeds the delayed branching threshold,

`false`

otherwise.

Arguments:

`nOpt`

The number of optimizations of a subproblem.

dbThreshold

The function `dbThreshold()` sets the number of optimizations of a subproblem until sons are created in `ABA_SUB::branching()`. If this value is 0, then a branching step is performed at the end of the subproblem optimization as usually if the subproblem can be fathomed. Otherwise, if this value is strictly positive, the subproblem is put back for a later optimization. This can be advantageous if in the meantime good cutting planes or primal bounds can be generated. The number of times the subproblem is put back without branching is indicated by this value.

```
void ABA_MASTER::dbThreshold(int threshold)
```

Arguments:

`threshold`

The new value of the delayed branching threshold.

dbThreshold

```
int ABA_MASTER::dbThreshold() const
```

Return Value:

The number of optimizations of a subproblem until sons are created. For further detatails we refer to `dbThreshold(int)`.

minDormantRound

```
int ABA_MASTER::minDormantRounds() const
```

Return Value:

The maximal number of rounds, i.e., number of subproblem optimizations, a subproblem is dormant, i.e., it is not selected from the set of open subproblem if its status is `Dormant`, if possible.

minDormantRounds

The function `minDormantRounds()` sets the number of rounds a subproblem should stay dormant.

```
void ABA_MASTER::minDormantRounds(int nRounds)
```

Arguments:

`nRounds`

The new minimal number of dormant rounds.

pbMode

```
ABA_MASTER::PRIMALBOUNDMODE ABA_MASTER::pbMode() const
```

Return Value:

The mode of the primal bound initialization.

pbMode

This version of the function `pbMode()` sets the mode of the primal bound initialization.

```
void ABA_MASTER::pbMode(PRIMALBOUNDMODE mode)
```

Arguments:

`mode`

The new mode of the primal bound initialization.

pricingFreq

```
int ABA_MASTER::pricingFreq() const
```

Return Value:

The number of linear programs being solved between two additional pricing steps. If no additional pricing steps should be executed this parameter has to be set to 0. The default value of the pricing frequency is 0. This parameter does not influence the execution of pricing steps which are required for the correctness of the algorithm.

pricingFreq

This version of the function `pricingFreq()` sets the number of linear programs being solved between two additional pricing steps.

```
void ABA_MASTER::pricingFreq(int f)
```

Arguments:

`f`

The pricing frequency.

skipFactor

```
int ABA_MASTER::skipFactor() const
```

Return Value:

The frequency of subproblems in which constraints or variables should be generated.

skipFactor

This version of the function `skipFactor()` sets the frequency for constraint and variable generation.

```
void ABA_MASTER::skipFactor(int f)
```

Arguments:

`f`

The new value of the frequency.

skippingMode

```
ABA_MASTER::SKIPPINGMODE ABA_MASTER::skippingMode() const
```

Return Value:

The skipping strategy.

skippingMode

This version of the function `skippingMode()` sets the skipping strategy.

```
void ABA_MASTER::skippingMode(SKIPPINGMODE mode)
```

Arguments:

`mode`

The new skipping strategy.

6.1.4 ABA_SUB

This class implements an abstract base class for a subproblem of the enumeration, i.e., a node of the branch-and-bound tree. The core of this class is the solution of the linear programming relaxation. If a derived class provides methods for the generation of cutting planes and/or variables, then the subproblem is processed by a cutting plane and/or column generation algorithm. Essential is that every subproblem has its own sets of active constraints and variables, which provides a very high flexibility.

```
class ABA_SUB : public ABA_ABACUSROOT {
  public:
    enum STATUS {Unprocessed, Active, Dormant, Processed, Fathomed};
    enum PHASE {Done, Cutting, Branching, Fathoming};
    ABA_SUB(
        ABA_MASTER *master,
        double conRes,
        double varRes,
        double nnzRes,
        bool relativeRes = true,
        ABA_BUFFER<ABA_POOLSLOT<ABA_CONSTRAINT, ABA_VARIABLE> *> *constraints = 0,
        ABA_BUFFER<ABA_POOLSLOT<ABA_VARIABLE, ABA_CONSTRAINT> *> *variables = 0);
```

```
  ABA_SUB(ABA_MASTER *master, ABA_SUB *father, ABA_BRANCHRULE *branchRule);
  virtual ~ABA_SUB();

  int level() const;
  int id() const;
  STATUS status() const;
  int nVar() const;
  int maxVar() const;
  int nCon() const;
  int maxCon() const;
  double lowerBound() const;
  double upperBound() const;
  double dualBound() const;
  ABA_SUB *father();
  ABA_LPSUB *lp();
  void maxIterations(int max);
  ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *actCon();
  ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *actVar();
  ABA_CONSTRAINT *constraint(int i);
  ABA_SLACKSTAT *slackStat(int i);
  ABA_VARIABLE *variable(int i);
  double lBound(int i) const;
  double uBound(int i) const;
  ABA_FSVARSTAT *fsVarStat(int i);
  ABA_LPVARSTAT *lpVarStat(int i);
  double xVal(int i) const;
  double yVal(int i) const;
  bool ancestor(ABA_SUB *sub);
  ABA_MASTER *master();
  void removeVars(ABA_BUFFER<int> &remove);
  void removeVar(int i);
  double nnzReserve() const;
  bool relativeReserve() const;
  ABA_BRANCHRULE *branchRule();
  bool objAllInteger();
  virtual void removeCons(ABA_BUFFER<int> &remove);
  virtual void removeCon(int i);
  int addConBufferSpace() const;
  int addVarBufferSpace() const;
  int nDormantRounds() const;
  void ignoreInTailingOff();
  virtual int addBranchingConstraint(
                        ABA_POOLSLOT<ABA_CONSTRAINT, ABA_VARIABLE> *slot);

protected:
  virtual int addCons(ABA_BUFFER<ABA_CONSTRAINT*> &constraints,
              ABA_POOL<ABA_CONSTRAINT, ABA_VARIABLE> *pool = 0,
              ABA_BUFFER<bool> *keepInPool = 0,
              ABA_BUFFER<double> *rank = 0);
  virtual int addCons(
               ABA_BUFFER<ABA_POOLSLOT<ABA_CONSTRAINT, ABA_VARIABLE>*> &newCons);
  virtual int addVars(ABA_BUFFER<ABA_VARIABLE*> &variables,
                      ABA_POOL<ABA_VARIABLE, ABA_CONSTRAINT> *pool = 0,
```

```
                    ABA_BUFFER<bool> *keepInPool = 0,
                    ABA_BUFFER<double> *rank = 0);
virtual int addVars(
              ABA_BUFFER<ABA_POOLSLOT<ABA_VARIABLE, ABA_CONSTRAINT>*> &newVars);
virtual int variablePoolSeparation(
                        int ranking = 0,
                        ABA_POOL<ABA_VARIABLE, ABA_CONSTRAINT> *pool = 0,
                        double minViolation = 0.001);
virtual int constraintPoolSeparation(
                        int ranking = 0,
                        ABA_POOL<ABA_CONSTRAINT, ABA_VARIABLE> *pool = 0,
                        double minViolation = 0.001);
virtual void activate();
virtual void deactivate ();
virtual int generateBranchRules(ABA_BUFFER<ABA_BRANCHRULE*> &rules);
virtual int branchingOnVariable(ABA_BUFFER<ABA_BRANCHRULE*> &rules);
virtual int selectBranchingVariable(int &variable);
virtual int selectBranchingVariableCandidates(ABA_BUFFER<int> &candidates);
virtual int selectBestBranchingSample(int nSamples,
                                      ABA_BUFFER<ABA_BRANCHRULE*> **samples);
virtual void rankBranchingSample(ABA_BUFFER<ABA_BRANCHRULE*> &sample,
                                 ABA_ARRAY<double> &rank);
virtual double rankBranchingRule(ABA_BRANCHRULE *branchRule);
double lpRankBranchingRule(ABA_BRANCHRULE *branchRule, int iterLimit = -1);
virtual int compareBranchingSampleRanks(ABA_ARRAY<double> &rank1,
                                        ABA_ARRAY<double> &rank2);
int closeHalfExpensive(int &branchVar, ABA_VARTYPE::TYPE branchVarType);
int closeHalfExpensive(ABA_BUFFER<int> &variables,
                       ABA_VARTYPE::TYPE branchVarType);
int closeHalf(int &branchVar, ABA_VARTYPE::TYPE branchVarType);
int closeHalf(ABA_BUFFER<int> &branchVar, ABA_VARTYPE::TYPE branchVarType);
int findNonFixedSet(ABA_BUFFER<int> &branchVar,
                    ABA_VARTYPE::TYPE branchVarType);
int findNonFixedSet(int &branchVar, ABA_VARTYPE::TYPE branchVarType);
virtual int initMakeFeas(ABA_BUFFER<ABA_INFEASCON*> &infeasCon,
                         ABA_BUFFER<ABA_VARIABLE*> &newVars,
                         ABA_POOL<ABA_VARIABLE, ABA_CONSTRAINT> **pool);
virtual int makeFeasible();
virtual bool goodCol(ABA_COLUMN &col, ABA_ARRAY<double> &row,
                     double x, double lb, double ub);
virtual void setByLogImp(ABA_BUFFER<int> &variable,
                         ABA_BUFFER<ABA_FSVARSTAT*> &status);
virtual bool feasible() = 0;
bool integerFeasible();
virtual bool primalSeparation();
virtual int separate();
virtual void conEliminate(ABA_BUFFER<int> &remove);
virtual void nonBindingConEliminate(ABA_BUFFER<int> &remove);
virtual void basicConEliminate(ABA_BUFFER<int> &remove);
virtual void varEliminate(ABA_BUFFER<int> &remove);
void redCostVarEliminate(ABA_BUFFER<int> &remove);
virtual int pricing();
virtual int improve(double &primalValue);
```

```
virtual ABA_SUB *generateSon(ABA_BRANCHRULE *rule) = 0;
bool boundCrash()    const;
virtual bool pausing();
bool infeasible();
virtual void varRealloc(int newSize);
virtual void conRealloc(int newSize);
virtual ABA_LP::METHOD chooseLpMethod(int nVarRemoved, int nConRemoved,
                                      int nVarAdded, int nConAdded);
void dualBound(double x);
virtual bool tailingOff();
bool betterDual(double x) const;
void lBound(int i, double l);
void uBound(int i, double u);
virtual void selectVars();
virtual void selectCons();
virtual int fixByRedCost(bool &newValues, bool saveCand);
virtual void fixByLogImp(ABA_BUFFER<int> &variable,
                         ABA_BUFFER<ABA_FSVARSTAT*> &status);
virtual int fixAndSet(bool &newValues);
virtual int fixing(bool &newValues, bool saveCand = false);
virtual int setting(bool &newValues);
virtual int setByRedCost();
virtual void fathom(bool reoptimize);
virtual bool fixAndSetTime();
virtual int fix(int i, ABA_FSVARSTAT *newStat, bool &newValue);
virtual int set(int i, ABA_FSVARSTAT *newStat, bool &newValue);
 virtual int set(int i, ABA_FSVARSTAT::STATUS newStat, bool &newValue);
virtual int set(int i, ABA_FSVARSTAT::STATUS newStat, double value,
                bool &newValue);
virtual double dualRound(double x);
virtual double guarantee();
virtual bool guaranteed();
virtual bool removeNonLiftableCons();
virtual int prepareBranching(bool &lastIteration);
virtual void fathomTheSubTree();
virtual int optimize();
virtual void reoptimize();
virtual void initializeVars(int maxVar);
virtual void initializeCons(int maxCon);
virtual PHASE branching();
virtual PHASE fathoming();
virtual PHASE cutting();
virtual ABA_LPSUB *generateLp();
virtual int initializeLp();
virtual int solveLp();
virtual bool exceptionFathom();
virtual bool exceptionBranch();
ABA_MASTER *master_;
ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *actCon_;
ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *actVar_;
ABA_SUB *father_;
ABA_LPSUB *lp_;
ABA_ARRAY<ABA_FSVARSTAT*> *fsVarStat_;
```

```
    ABA_ARRAY<ABA_LPVARSTAT*> *lpVarStat_;
    ABA_ARRAY<double> *lBound_;
    ABA_ARRAY<double> *uBound_;
    ABA_ARRAY<ABA_SLACKSTAT*> *slackStat_;
    ABA_TAILOFF *tailOff_;
    double dualBound_;
    int nIter_;
    int lastIterConAdd_;
    int lastIterVarAdd_;
    ABA_BRANCHRULE *branchRule_;
    bool allBranchOnSetVars_;
    ABA_LP::METHOD lpMethod_;
    ABA_CUTBUFFER<ABA_VARIABLE, ABA_CONSTRAINT> *addVarBuffer_;
    ABA_CUTBUFFER<ABA_CONSTRAINT, ABA_VARIABLE> *addConBuffer_;
    ABA_BUFFER<int> *removeVarBuffer_;
    ABA_BUFFER<int> *removeConBuffer_;
    double *xVal_;
    double *yVal_;
    double *bInvRow_;
    int infeasCon_;
    int infeasVar_;
    bool genNonLiftCons_;

  private:
    ABA_SUB(const ABA_SUB &rhs);
    const ABA_SUB &operator=(const ABA_SUB &rhs);
};
```

enum STATUS

A subproblem can have different statuses:

`Unprocessed`

: The status after generation, but before optimization of the subproblem.

`Active`

: The subproblem is currently processed.

`Dormant`

: The subproblem is partially processed and waiting in the set of open subproblems for further optimization.

`Processed`

: The subproblem is completely processed but could not be fathomed.

`Fathomed`

: The subproblem is fathomed.

enum PHASE

The optimization of the subproblem can be in one of the following phases:

`Done`

: The optimization is done.

`Cutting`

The iterative solution of the LP-relaxation and the generation of cutting planes and/or variables is currently performed.

`Branching`

We try to generate further subproblems as sons of this subproblem.

`Fathoming`

The subproblem is currently being fathomed.

master_

```
ABA_MASTER *master_
```

A pointer to the corresponding master of the optimization.

actCon_

```
ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *actCon_
```

The active constraints of the subproblem.

actVar_

```
ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *actVar_
```

The active variables of the subproblem.

father_

```
ABA_SUB *father_
```

A pointer to the father in the branch-and-cut tree.

lp_

```
ABA_LPSUB *lp_
```

A pointer to the corresponding linear program.

fsVarStat_

```
ABA_ARRAY<ABA_FSVARSTAT*> *fsVarStat_
```

A pointer to an array storing the status of fixing and setting of the active variables. Although fixed and set variables are already kept at their value by the adaption of the lower and upper bounds, we store this information, since, e.g., a fixed or set variable should not be removed, but a variable with an upper bound equal to the lower bound can be removed.

lpVarStat_

```
ABA_ARRAY<ABA_LPVARSTAT*> *lpVarStat_
```

A pointer to an array storing the status of each active variable in the linear program.

lBound_

```
ABA_ARRAY<double> *lBound_
```

A pointer to an array with the local lower bound of the active variables.

uBound_

```
ABA_ARRAY<double> *uBound_
```

A pointer to an array with the local upper bounds of the active variables.

slackStat_

```
ABA_ARRAY<ABA_SLACKSTAT*> *slackStat_
```

A pointer to an array storing the statuses of the slack variables of the last solved linear program.

tailOff_

```
ABA_TAILOFF *tailOff_
```

A pointer to the tailing off manager.

dualBound_

```
double dualBound_
```

The dual bound of the subproblem.

nIter_

```
int nIter_
```

The number of iterations in the cutting plane phase.

lastIterConAdd_

```
int lastIterconAdd_
```

The last iteration in which constraints have been added.

lastIterVarAdd_

```
int lastIterVarAdd_
```

The last iteration in which variables have been added.

branchRule_

```
ABA_BRANCHRULE *branchRule_
```

The branching rule for the subproblem.

allBranchOnSetVars_

`bool allBranchOnSetVars_`

If `true`, then the branching rule of the subproblem and of all ancestor on the path to the root node are branching on a binary variable.

lpMethod_

`ABA_LP::METHOD lpMethod_`

The solution method for the next linear program.

addVarBuffer_

`ABA_CUTBUFFER<ABA_VARIABLE, ABA_CONSTRAINT> *addVarBuffer_`

The buffer of the newly generated variables.

addConBuffer_

`ABA_CUTBUFFER<ABA_CONSTRAINT, ABA_VARIABLE> *addConBuffer_`

The buffer of the newly generated constraints.

removeVarBuffer_

`ABA_BUFFER<int> *removeVarBuffer_`

The buffer of the variables which are removed at the beginning of the next iteration.

removeConBuffer_

`ABA_BUFFER<int> *removeConBuffer_`

The buffer of the constraints which are removed at the beginning of the next iteration.

xVal_

`double *xVal_`

The last LP-solution.

yVal_

`double *yVal_`

The dual variables of the last linear program.

bInvRow_

`double *bInvRow_`

A row of the basis inverse associated with the infeasible variable `infeasVar_` or slack variable `infeasCon_`.

infeasCon_

`int infeasCon_`

The number of an infeasible constraint.

infeasVar_

`int infeasVar_`

The number of an infeasible variable.

genNonLiftCons_

`genNonLiftCons_`

If `true`, then the management of non-liftable constraints is performed.

feasible (`virtual`)

The pure virtual function `feasible()` checks for the feasibility of a solution of the LP-relaxation. If the function returns `true` and the value of the primal bound is worse than the value of this feasible solution, the value of the primal bound is updated automatically.

```
virtual bool feasible() = 0
```

Return Value:

`true`

If the LP-solution is feasible,

`false`

otherwise.

generateSon (`virtual`)

```
virtual ABA_SUB *generateSon(ABA_BRANCHRULE *rule) = 0
```

Return Value:

The function `generateSon()` returns a pointer to an object of a problem specific subproblem derived from the class `ABA_SUB`, which is generated from the current subproblem by the branching rule `rule`.

Arguments:

`rule`

The branching rule with which the subproblem is generated.

Constructor

The constructor for the root node of the enumeration tree.

```
ABA_SUB::ABA_SUB(
             ABA_MASTER *master,
             double conRes,
             double varRes,
             double nnzRes,
             bool relativeRes,
             ABA_BUFFER<ABA_POOLSLOT<ABA_CONSTRAINT, ABA_VARIABLE> *> *constraints,
             ABA_BUFFER<ABA_POOLSLOT<ABA_VARIABLE, ABA_CONSTRAINT> *> *variables)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`conRes`

The additional memory allocated for constraints.

`varRes`

The additional memory allocated for variables.

`nnzRes`

The additional memory allocated for nonzero elements of the constraint matrix.

`relativeRes`

If this argument is `true`, then reserve space for variables, constraints, and nonzeros given by the previous three arguments, is given in percent of the original numbers. Otherwise, the numbers are interpreted as absolute values (casted to integer). The default value is `true`.

`constraints`

The pool slots of the initial constraints. If the value is 0, then the constraints of the default constraint pool are taken. The default value is 0.

`variables`

The pool slots of the initial variables. If the value is 0, then the variables of the default variable pool are taken. The default value is 0.

Constructor

The constructor for non-root nodes of the enumeration tree.

```
ABA_SUB::ABA_SUB(ABA_MASTER *master, ABA_SUB *father, ABA_BRANCHRULE *branchRule)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`father`

A pointer to the father in the enumeration tree.

`branchRule`

The rule defining the subspace of the solution space associated with this subproblem.

Destructor (`virtual`)

```
ABA_SUB::~ABA_SUB()
```

optimize (`virtual`)

The function `optimize()` performs the optimization of the subproblem.

```
int ABA_SUB::optimize()
```

Return Value:

`0`

If the optimization has been performed without error,

`1`

otherwise.

activate (`virtual`)

The virtual function `activate()` does nothing but can be used as an entrance point for problem specific activations by v a reimplementation in derived classes.

```
void ABA_SUB::activate()
```

initializeVars (`virtual`)

The function `initializeVars()` initializes the active variable set.

```
void ABA_SUB::initializeVars(int maxVar)
```

Arguments:

`maxVar`

The maximal number of variables of the subproblem.

initializeCons (`virtual`)

The function `initializeCons()` initializes the active constraint set.

```
void ABA_SUB::initializeCons(int maxCon)
```

Arguments:

`maxCon`

The maximal number of constraints of the subproblem.

deactivate (`virtual`)

The virtual function `deactivate()` can be used as entrance point for problem specific deactivations after the subproblem optimization. The default version of this function does nothing. This function is only called if the function `activate()` for the subproblem has been executed.

```
void ABA_SUB::deactivate()
```

setByLogImp (virtual)

The default implementation of `setByLogImp()` does nothing. In derived classes this function can be reimplemented.

```
void ABA_SUB::setByLogImp(ABA_BUFFER<int> &variables,
                          ABA_BUFFER<ABA_FSVARSTAT*> &status)
```

Arguments:

`variable`

The variables which should be set have to be inserted in this buffer.

`status`

The status of the set variables.

cutting (virtual)

The function `cutting()` iteratively solves the LP-relaxation, generates constraints and/or variables. Also generating variables can be regarded as "cutting", namely as generating cuts for the dual problem.

```
ABA_SUB::PHASE ABA_SUB::cutting ()
```

Return Value:

`Fathoming`

If one of the conditions for fathoming the subproblem is satisfied.

`Branching`

If the subproblem should be splitted in further subproblems.

prepareBranching (virtual)

The function `prepareBranching()` is called before a branching step to remove constraints.

```
int ABA_SUB::prepareBranching(bool &lastIteration)
```

Return Value:

`1`

If constraints have been removed,

`0`

otherwise.

Arguments:

`lastIteration`

This argument is always set to `true` in the function call.

solveLp (`virtual`)

The function `solveLp()` solves the LP-relaxation of the subproblem.

```
int ABA_SUB::solveLp ()
```

Return Value:

`0`
: The linear program has an optimimal solution.

`1`
: If the linear program is infeasible.

`2`
: If the linear program is infeasible for the current variable set, but non-liftable constraints have to be removed before a pricing step can be performed.

exceptionFathom (`virtual`)

The function `exceptionFathom()` can be used to specify a problem specific fathoming criterium that is checked before the separation or pricing. The default implementation always returns `false`.

```
bool ABA_SUB::exceptionFathom()
```

Return Value:

`true`
: If the subproblem should be fathomed,

`false`
: otherwise.

exceptionBranch (`virtual`)

The function `exceptionBranch()` can be used to specify a problem specific criteria for enforcing a branching step. This criterium is checked before the separation or pricing. The default implementation always returns `false`.

```
bool ABA_SUB::exceptionBranch()
```

Return Value:

`true`
: If the subproblem should be fathomed,

`false`
: otherwise.

fixAndSetTime (`virtual`)

The virtual function `fixAndSetTime()` controls if variables should be fixed or set when all variables price out corretly. The default implementation always returns `true`.

```
bool ABA_SUB::fixAndSetTime()
```

Return Value:

`true`
: If variables should be fixed and set,

`false`
: otherwise.

makeFeasible (virtual)

The default implementation of `makeFeasible()` does nothing.

If there is an infeasible structural variable then it is stored in `infeasVar_`, otherwise `infeasVar_` is `-1`. If there is an infeasible slack variable, it is stored in `infeasCon_`, otherwise it is `-1`. At most one of the two members `infeasVar_` and `infeasCon_` can be nonnegative. A reimplementation in a derived class should generate variables to restore feasibility or confirm that the subproblem is infeasible.

The strategy for the generation of inactive variables is completely problem and user specific. For testing if a variable might restore again the feasibility the functions `ABA_VARIABLE::useful()` and `ABA_SUB::goodCol()` might be helpful.

```
int ABA_SUB::makeFeasible()
```

Return Value:

`0`

If feasibility can be restored,

`1`

otherwise.

goodCol (virtual)

```
bool ABA_SUB::goodCol(ABA_COLUMN &col,
                      ABA_ARRAY<double> &row,
                      double x,
                      double lb,
                      double ub)
```

Return Value:

`true`

If the column `col` might restore feasibiblity if the variable with value `x` turns out to be infeasible,

`false`

otherwise.

Arguments:

`col`

The column of the variable.

`row`

The row of the basis inverse associated with the infeasible variable.

`x`

The LP-value of the infeasible variable.

`lb`

The lower bound of the infeasible variable.

`ub`

The upper bound of the infeasible variable.

pricing (`virtual`)

The function `pricing()` should generate inactive variables which do not price out correctly. The default implementation does nothing and returns 0.

```
int ABA_SUB::pricing()
```

Return Value:

The number of new variables.

primalSeparation (`virtual`)

The function `primalSeparation()` is a virtual function which controls, if during the cutting plane phase a (primal) separation step or a pricing step (dual separation) should be performed.

Per default a pure cutting plane algorithm performs always a primal separation step, a pure column generation algorithm never performs a primal separation, and a hybrid algorithm generates usually cutting planes but from time to time also inactive variables are priced out depending on the `pricingFrequency()`.

```
bool ABA_SUB::primalSeparation()
```

Return Value:

`true`

Then cutting planes are generated in this iteration.

`false`

Then columns are generated in this iteration.

xVal

```
double ABA_SUB::xVal(int i) const
```

Return Value:

The value of the `i`-th variable in the last solved linear program.

Arguments:

`i`

The number of the variable under consideration.

yVal

```
double ABA_SUB::yVal(int i) const
```

Return Value:

The value of the `i`-th dual variable in the last solved linear program.

Arguments:

`i`

The number of the variable under consideration.

dualRound (`virtual`)

```
double ABA_SUB::dualRound(double x)
```

Return Value:

If all objective function values of feasible solutions are integer the function `dualRound()` returns `x` rounded up to the next integer if this is a minimization problem, or `x` rounded down to the next integer if this is a maximization problem, respectively. Otherwise, the return value is `x`.

Arguments:

`x`

The value that should be rounded if possible.

guaranteed (`virtual`)

```
bool ABA_SUB::guaranteed()
```

Return Value:

`true`

If the lower and the upper bound of the subproblem satisfies the guarantee requirements,

`false`

otherwise.

guarantee (`virtual`)

The function `guarantee()` may not be called if the lower bound is 0 and upper bound not equal to 0.

```
double ABA_SUB::guarantee()
```

Return Value:

The guarantee that can be given for the subproblem.

ancestor

```
bool ABA_SUB::ancestor(ABA_SUB *sub)
```

Return Value:

`true`

If `this` subproblem is an ancestor of the subproblem `sub`. We define that a subproblem is its own ancestor,

`false`

otherwise.

Arguments:

`sub`

A pointer to a subproblem.

removeNonLiftableCons (virtual)

```
bool ABA_SUB::removeNonLiftableCons()
```

Return Value:

`true`
: If all active constraints can be lifted.

`false`
: otherwise.

chooseLpMethod (virtual)

The virtual function `chooseLpMethod()` controls the method used to solve a linear programming relaxation. The default implementation chooses the barrier method for the first linear program of the root node and for all other linear programs it tries to choose a method such that phase 1 of the simplex method is not required.

```
ABA_LP::METHOD ABA_SUB::chooseLpMethod(int nVarRemoved,
                                       int nConRemoved,
                                       int nVarAdded,
                                       int nConAdded)
```

Return Value:

The method the next linear programming relaxation is solved with.

Arguments:

`nVarRemoved`
: The number of removed variables.

`nConRemoved`
: The number of removed constraints.

`nVarAdded`
: The number of added variables.

`nConAdded`
: The number of added constraint.

master

```
ABA_MASTER *ABA_SUB::master()
```

Return Value:

A pointer to the master of the optimization.

removeVars (virtual)

With function `removeVars()` variables can be removed from the set of active variables. The variables are not removed when this function is called, but are buffered and removed at the beginning of the next iteration.

```
void ABA_SUB::removeVars(ABA_BUFFER<int> &remove)
```

Arguments:

`remove`
: The variables which should be removed.

removeVar (`virtual`)

The function `removeVar()` can be used to remove a single variable from the set of active variables. Like in the function `removeVars()` the variable is buffered and removed at the beginning of the next iteration.

```
void ABA_SUB::removeVar(int i)
```

Arguments:

`i`

The variable which should be removed.

selectVars (`virtual`)

The virtual dummy function `selectVars()` is called before variables are selected from the variable buffer. It can be redefined in a derived class e.g., to remove multiply inserted variables from the buffer.

```
void ABA_SUB::selectVars()
```

selectCons (`virtual`)

The virtual dummy function `selectCons()` is called before constraint are selected from the constraint buffer. It can be redefined in a derived class e.g., to remove multiply inserted constraints from the buffer.

```
void ABA_SUB::selectCons()
```

nnzReserve

```
double ABA_SUB::nnzReserve() const
```

Return Value:

The additional space for nonzero elements of the constraint matrix when it is passed to the LP-solver.

relativeReserve

```
bool ABA_SUB::relativeReserve() const
```

Return Value:

`true`

If the reserve space for variables, constraints, and nonzeros is given in percent of the original space, and `false` if its given as absolute value,

`false`

otherwise.

branchRule

```
ABA_BRANCHRULE *ABA_SUB::branchRule()
```

Return Value:

A pointer to the branching rule of the subproblem.

addCons (`virtual`)

The function `addBranchingConstraint()` adds a branching constraint to the constraint buffer such that it is automatically added at the beginning of the cutting plane algorithm. It should be used in definitions of the pure virtual function `BRANCHRULE::extract()`. The function `addCons()` tries to add new constraints to the constraint buffer and a pool. The memory management of added constraints is passed to ABACUS by calling this function.

```
int ABA_SUB::addCons(ABA_BUFFER<ABA_CONSTRAINT*> &constraints,
                     ABA_POOL<ABA_CONSTRAINT, ABA_VARIABLE> *pool,
                     ABA_BUFFER<bool> *keepInPool,
                     ABA_BUFFER<double> *rank)
```

Return Value:

`0`

If the constraint could be added,

`1`

otherwise.

Return Value:

The number of added constraints.

Arguments:

`slot`

A pointer to the pools slot containing the branching constraint.

Arguments:

`constraints`

The new constraints.

`pool`

The pool in which the new constraints are inserted. If the value of this argument is 0, then the cut pool of the master is selected. Its default value is 0.

`keepInPool`

If `(*keepInPool)[i]` is `true`, then the constraint stays in the pool even if it is not activated. The default value is a 0-pointer.

`rank`

If this pointer to a buffer is nonzero, this buffer should store a rank for each constraint. The greater the rank, the better the variable. The default value of `rank` is 0.

addVars (`virtual`)

The function `addVars()` tries to add new variables to the variable buffer and a pool. The memory management of added variables is passed to ABACUS by calling this function.

```
int ABA_SUB::addVars(ABA_BUFFER<ABA_VARIABLE*> &variables,
                     ABA_POOL<ABA_VARIABLE, ABA_CONSTRAINT> *pool,
                     ABA_BUFFER<bool> *keepInPool,
                     ABA_BUFFER<double> *rank)
```

Return Value:

The number of added variables.

Arguments:

`variable`

The new variables.

`pool`

The pool in which the new variables are inserted. If the value of this argument is 0, then the default variable pool is taken. The default value is 0.

`keepInPool`

If `(*keepInPool)[i]` is `true`, then the variable stays in the pool even if it is not activated. The default value is a 0-pointer.

`rank`

If this pointer to a buffer is nonzero, this buffer should store a rank for each variable. The greater the rank, the better the variable. The default value of `rank` is 0.

variablePoolSeparation (`virtual`)

The function `variablePoolSeparation()` tries to generate inactive variables from a pool.

```
int ABA_SUB::variablePoolSeparation(int ranking,
                                    ABA_POOL<ABA_VARIABLE, ABA_CONSTRAINT> *pool,
                                    double minAbsViolation)
```

Return Value:

The number of generated variables.

Arguments:

`ranking`

This parameter indicates how the ranks of geneated variables should be computed (0: no ranking; 1: violation is rank, 2: absolute value of violation is rank). The default value is 0.

`pool`

The pool the variables are generated from. If `pool` is 0, then the default variable pool is used. The default value of `pool` is 0.

`minAbsViolation`

A violated constraint/variable is only added if the absolute value of its violation is at least `minAbsViolation`. The default value is 0.001.

constraintPoolSeparation (`virtual`)

The function `constraintPoolSeparation()` tries to generate inactive constraints from a pool.

```
int ABA_SUB::constraintPoolSeparation(int ranking,
                                      ABA_POOL<ABA_CONSTRAINT, ABA_VARIABLE> *pool,
                                      double minViolation)
```

Return Value:

The number of generated constraints.

Arguments:

`ranking`

This parameter indicates how the ranks of violated constraints should be computed (0: no ranking; 1: violation is rank, 2: absolute value of violation is rank). The default value is 0.

`pool`

The pool the constraints are generated from. If `pool` is 0, then the default constraint pool is used. The default value of `pool` is 0.

`minAbsViolation`

A violated constraint/variable is only added if the absolute value of its violation is at least `minAbsViolation`. The default value is `0.001`.

addConBufferSpace

The function `addConBufferSpace()` can be used to determine the maximal number of the constraints which still can be added to the constraint buffer. A separation algorithm should stop as soon as the number of generated constraints reaches this number because further work is useless.

```
int ABA_SUB::addConBufferSpace() const
```

Return Value:

The number of constraints which still can be inserted into the constraint buffer.

addVarBufferSpace

The function `addVarBufferSpace()` can be used to determine the maximal number of the variables which still can be added to the variable buffer. A pricing algorithm should stop as soon as the number of generated variables reaches this number because further work is useless.

```
int ABA_SUB::addVarBufferSpace() const
```

Return Value:

The number of variables which still can be inserted into the variable buffer.

objAllInteger

If all variables are `Binary` or `Integer` and all objective function coefficients are integral, then all objective function values of feasible solutions are integral. The function `objAllInteger()` tests this condition for the current set of active variables.

Note, the result of this function can only be used to set the global parameter if `actVar` contains all variables of the problem formulation.

```
bool ABA_SUB::objAllInteger()
```

Return Value:

`true`

If this condition is satisfied by the currently active variable set,

`false`

otherwise.

integerFeasible

The function `integerFeasible()` can be used to check if the solution of the LP-relaxation is primally feasible if for feasibility an integral value for all binary and integer variables is sufficient. This function can be called from the function `feasible()` in derived classes.

```
bool ABA_SUB::integerFeasible()
```

Return Value:

`true`

If the LP-value of all binary and integer variables is integral,

`false`

otherwise.

nDormantRounds

```
int ABA_SUB::nDormantRounds() const
```

Return Value:

The number of subproblem optimization the subproblem is already dormant.

ignoreInTailingOff

The function `ignoreInTailingOff()` can be used to control better the tailing-off effect. If this function is called, the next LP-solution is ignored in the tailing-off control. Calling `ignoreInTailingOff()` can e.g. be considered in the following situation: If only constraints that are required for the integer programming formulation of the optimization problem are added then the next LP-value could be ignored in the tailing-off control. Only "real" cutting planes should be considered in the tailing-off control (this is only an example strategy that might not be practical in many situations, but sometimes turned out to be efficient).

```
void ABA_SUB::ignoreInTailingOff()
```

branching (`virtual`)

The function `branching()` is called if the global lower bound of a branch-and-cut node is still strictly less than the local upper bound, but either no violated cutting planes or variables are found, or we abort the cutting phase for some other strategic reason (e.g., observation of a tailing off effect, or branch pausing).

Usually, two new subproblems are generated. However, our implementation of `branching()` is more sophisticated that allows different branching. Moreover, we also check if this node is only paused. If this is the case the node is put back into the list of open branch-and-cut nodes without generating sons of this node.

Finally if none of the previous conditions is satisfied we generate new subproblems.

```
ABA_SUB::PHASE ABA_SUB::branching()
```

Return Value:

`Done`

If sons of the subproblem could be generated,

`Fathoming`

otherwise.

generateBranchRules (`virtual`)

The function `generateBranchRules()` tries to find rules for splitting the current subproblem in further subproblems. Per default we generate rules for branching on variables (`branchingOnVariable()`). But by redefining this function in a derived class any other branching strategy can be implemented.

```
int ABA_SUB::generateBranchRules(ABA_BUFFER<ABA_BRANCHRULE*> &rules)
```

Return Value:

`0`
: If branching rules could be found,

`1`
: otherwise.

Arguments:

`rules`
: If branching rules are found, then they are stored in this buffer.

branchingOnVariable (`virtual`)

The function `branchingOnVariable()` generates branching rules for two new subproblems by selecting a branching variable with the function `selectBranchingVariable()`. If a new branching variable selection strategy should be used the function `selectBranchingVariable()` should be redefined.

```
int ABA_SUB::branchingOnVariable(ABA_BUFFER<ABA_BRANCHRULE*> &rules)
```

Return Value:

`0`
: If branching rules could be found,

`1`
: otherwise

Arguments:

`rules`
: If branching rules are found, then they are stored in this buffer. The length of this buffer is the number of active variables of the subproblem. If more branching rules are generated a reallocation has to be performed.

selectBranchingVariable (`virtual`)

The function `selectBranchingVariable()` chooses a branching variable.

The function `selectBranchingVariableCandidates()` is asked to generate depending in the parameter `NBranchingVariableCandidates` of the file `.abacus` candidates for branching variables. If only one candidate is generate, this one becomes the branching variable. Otherwise, the pairs of branching rules are generated for all candidates and the "best" branching variables is determined with the function `selectBestBranchingSample()`.

```
int ABA_SUB::selectBranchingVariable(int &variable)
```

Return Value:

`0`

If a branching variable is found,

`1`

otherwise.

Arguments:

`variable`

Holds the branching variable if one is found.

selectBranchingVariableCandidates

The function `selectBranchingVariableCandidates()` selects depending on the branching variable strategy given by the parameter `BranchingStrategy` in the file `.abacus` candidates that for branching variables.

Currently two branching variable selection strategies are implemented. The first one (`CloseHalf`) first searches the binary variables with fractional part closest to 0.5. If there is no fractional binary variable it repeats this process with the integer variables.

The second strategy (`CloseHalfExpensive`) first tries to find binary variables with fraction close to 0.5 and high absolute objective function coefficient. If this fails, it tries to find an integer variable with fractional part close to 0.5 and high absolute objective function coefficient.

If neither a binary nor an integer variable with fractional value is found then for both strategies we try to find non-fixed and non-set binary variables. If this fails we repeat this process with the integer variables.

Other branching variable selection strategies can be implemented by redefining this virtual function in a derived class.

```
int ABA_SUB::selectBranchingVariableCandidates(ABA_BUFFER<int> &candidates)
```

Return Value:

`0`

If a candidate is found,

`1`

otherwise.

Arguments:

`candidates`

The candidates for branching variables are stored in this buffer. We try to find as many variables as fit into the buffer.

closeHalf

The function `closeHalf()` searches a branching variable of type `branchVarType`, with fraction as close to 0.5 as possible.

```
int ABA_SUB::closeHalf(int &branchVar, ABA_VARTYPE::TYPE branchVarType)
```

Return Value:

`0`

If a branching variable is found,

`1`

otherwise.

Arguments:

`branchVar`

Holds the branching variable if one is found.

`branchVartype`

The type of the branching variable can be restricted either to `ABA_VARTYPE::Binary` or `ABA_VARTYPE::Integer`.

closeHalf

The function `closeHalf()` searches searches several possible branching variable of type `branchVarType`, with fraction as close to 0.5 as possible.

```
int ABA_SUB::closeHalf(ABA_BUFFER<int> &variables, ABA_VARTYPE::TYPE branchVarType)
```

Return Value:

`0`

If at least one branching variable is found,

`1`

otherwise.

Arguments:

`variables`

Stores the possible branching variables.

`branchVartype`

The type of the branching variable can be restricted either to `ABA_VARTYPE::Binary` or `ABA_VARTYPE::Integer`.

closeHalfExpensive

The function `closeHalfExpensive()` selects a single branching variable of type `branchVarType`, with fractional part close to 0.5 and high absolute objective function coefficient.

```
int ABA_SUB::closeHalfExpensive(int &branchVar, ABA_VARTYPE::TYPE branchVarType)
```

Return Value:

`0`

If a branching variable is found,

`1`

otherwise.

Arguments:

`branchVar`

Holds the number of the branching variable if one is found.

`branchVartype`

The type of the branching variable can be restricted either to `ABA_VARTYPE::Binary` or `ABA_VARTYPE::Integer`.

closeHalfExpensive

This version of the function `closeHalfExpensive()` selects several candidates for branching variables of type `branchVarType`. Thos variables with fractional part close to 0.5 and high absolute objective function coefficient are selected..

```
int ABA_SUB::closeHalfExpensive(ABA_BUFFER<int> &branchVar,
                                ABA_VARTYPE::TYPE branchVarType)
```

Return Value:

`0`

If at least one branching variable is found,

`1`

otherwise.

Arguments:

`branchVar`

Holds the numbers of possible branching variables if at least one is found. We try to find as many candidates as fit into this buffer. We abort the function with a fatal error if the size of the buffer is 0.

`branchVartype`

The type of the branching variable can be restricted either to `ABA_VARTYPE::Binary` or `ABA_VARTYPE::Integer`.

findNonFixedSet

The function `findNonFixedSet()` selects the first variable that is neither fixed nor set.

```
int ABA_SUB::findNonFixedSet(int &branchVar, ABA_VARTYPE::TYPE branchVarType)
```

Return Value:

`0`

If a variable neither fixed nor set is found,

`1`

otherwise.

Arguments:

`branchVar`

Holds the number of the branching variable if one is found.

`branchVarType`

The type of the branching have (`ABA_VARTYPE::Binary` or `ABA_VARTYPE::Integer`).

findNonFixedSet

The function `findNonFixedSet()` selects the first variables that are neither fixed nor set.

```
int ABA_SUB::findNonFixedSet(ABA_BUFFER<int> &branchVar,
                             ABA_VARTYPE::TYPE branchVarType)
```

Return Value:

`0`

If at least one variable neither fixed nor set is found,

`1`

otherwise.

Arguments:

`branchVar`

Holds the number of the possible branching variables if one is found.

`branchVartype`

The type of the branching variable can be restricted either to `ABA_VARTYPE::Binary` or `ABA_VARTYPE::Integer`.

selectBestBranchingSample (virtual)

The function `selectBestBranchingSample()` evaluates branching samples (we denote a branching sample the set of rules defining all sons of a subproblem in the enumeration tree). For each sample the ranks are determined with the function `rankBranchingSample()`. The ranks of the various samples are compared with the function `compareBranchingSample()`.

```
int ABA_SUB::selectBestBranchingSample(int nSamples,
                                       ABA_BUFFER<ABA_BRANCHRULE*> **samples)
```

Return Value:

The number of the best branching sample, or `-1` in case of an internal error.

Arguments:

`nSamples`

The number of branching samples.

`samples`

An array of pointer to buffers storing the branching rules of each sample.

rankBranchingSample (virtual)

The function `rankBranchingSample()` computes for each branching rule of a branching sample a rank with the function `rankBranchingRule()`.

```
void ABA_SUB::rankBranchingSample(ABA_BUFFER<ABA_BRANCHRULE*> &sample,
                                  ABA_ARRAY<double> &rank)
```

Arguments:

`sample`

A branching sample.

`rank`

An array storing the rank for each branching rule in the sample after the function call.

rankBranchRule (virtual)

The function `rankBranchingRule()` computes the rank of a branching rule. This default implementation computes the rank with the function `lpRankBranchingRule()`. By redefining this virtual function the rank for a branching rule can be computed differently.

```
double ABA_SUB::rankBranchingRule(ABA_BRANCHRULE *branchRule)
```

Return Value:

The rank of the branching rule.

Arguments:

`branchRule`

A pointer to a branching rule.

lpRankBranchingRule

The function `lpRankBranchingRule()` computes the rank of a branching rule by modifying the linear programming relaxation of the subproblem according to the branching rule and solving it. This modifiction is undone after the solution of the linear program.

It is useless, but no error, to call this function for branching rules for which the virtual dummy functions `extract(ABA_LPSUB*)` and `unExtract(ABA_LPSUB*)` of the base class `ABA_BRANCHRULE` are not redefined.

```
double ABA_SUB::lpRankBranchingRule(ABA_BRANCHRULE *branchRule, int iterLimit)
```

Arguments:

`branchRule`

A pointer to a branching rule.

`iterLimit`

The maximal number of iterations that should be performed by the simplex method. If this number is negative there is no iteration limit (besides internal limits of the LP-solver). The default value is `-1`.

compareBranchingSampleRanks (virtual)

The function `compareBranchingSampleRanks()` compares the ranks of two branching samples. For maximimization problem that rank is better for which the maximal rank of a rule is minimal, while for minimization problem the rank is better for which the minimal rank of a rule is maximal. If this value equals for both ranks we continue with the secand greatest value, etc.

```
int ABA_SUB::compareBranchingSampleRanks(ABA_ARRAY<double> &rank1,
                                         ABA_ARRAY<double> &rank2)
```

Return Value:

`1`

If `rank1` is better.

`0`

If both ranks are equal.

`-1`

If `rank2` is better.

fathoming (virtual)

The function fathoming() fathoms the node, and if certain conditions are satisfied, also its ancestor.

The third central phase of the optimization of a subproblem is the Fathoming of a subproblem. A subproblem is fathomed if it can be guaranteed that this subproblem cannot contain a better solution than the best known one. This is the case if the global upper bound does not exceed the local lower bound (maximization problem assumed) or the subproblem cannot contain a feasible solution either if there is a fixing/setting contradiction or the LP-relaxation turns out to be infeasible.

Note, use the function ExceptionFathom() for specifying problem specific fathoming criteria.

```
ABA_SUB::PHASE ABA_SUB::fathoming()
```

Return Value:

The function always returns Done.

fathom (virtual)

The function fathom() fathoms a node and recursively tries to fathom its father.

```
void ABA_SUB::fathom(bool reoptimize)
```

Arguments:

reoptimize

If reoptimize is true, then we perform a reoptimization in the new root.

fixAndSet (virtual)

The function fixAndSet() tries to fix and set variables both by logical implications and reduced cost criteria.

```
int ABA_SUB::fixAndSet(bool &newValues)
```

Return Value:

1

If a contradiction is found,

0

otherwise.

Arguments:

newValues

If a variables is set or fixed to a value different from the last LP-solution, newValues is set to true, otherwise it is set to false.

fixing (virtual)

The function fixing() tries to fix variables by reduced cost criteria and logical implications.

```
int ABA_SUB::fixing(bool &newValues, bool saveCand)
```

Return Value:

1

If a contradiction is found,

0

otherwise.

Arguments:

`newValues`

The parameter `newValues` becomes `true` if variables are fixed to other values as in the current LP-solution.

`saveCand`

If the parameter `saveCand` is `true` a new candidate list of variables for fixing is generated. The default value of `saveCand` is false. Candidates should not be saved if fixing is performed after the addition of variables.

setting (`virtual`)

The function `setting()` tries to set variables by reduced cost criteria and logical implications like `fixing()`, but instead of global conditions only locally valid conditions have to be satisfied.

```
int ABA_SUB::setting(bool &newValues)
```

Return Value:

`1`

If a contradiction has been found,

`0`

otherwise.

Arguments:

`newValues`

The parameter `newValues` becomes `true` if variables are fixed to other values as in the current LP-solution (`setByRedCost()` cannot set variables to new values).

fixByRedCost (`virtual`)

The function `fixByRedCost()` tries to fix variables according to the reduced cost criterion.

```
int ABA_SUB::fixByRedCost(bool &newValues, bool saveCand)
```

Return Value:

`1`

If a contradiction is found,

`0`

otherwise.

Arguments:

`newVales`

If variables are fixed to different values as in the last solved linear program, then `newValues` becomes true.

`saveCand`

If `saveCand` is `true`, then a new list of candidates for later calls is compiled. This is only possible when the root of the remaining branch-and-bound is processed.

fixByLogImp (virtual)

The function `fixByLogImp()` should collect the numbers of the variables to be fixed in `variable` and the respective statuses in `status`. The default implementation of `fixByLogImp()` does nothing. This function has to be redefined if variables should be fixed by logical implications in derived classes.

```
void ABA_SUB::fixByLogImp(ABA_BUFFER<int> &variables,
                          ABA_BUFFER<ABA_FSVARSTAT*> &status)
```

Arguments:

`variables`

The variables which should be fixed.

`status`

The statuses these variables should be fixed to.

setByRedCost (virtual)

The function `setByRedCost()` tries to set variables according to the reduced cost criterion.

```
int ABA_SUB::setByRedCost()
```

Return Value:

`1`

If a contradiction is found,

`0`

otherwise.

constraint

```
ABA_CONSTRAINT *ABA_SUB::constraint(int i)
```

Return Value:

A pointer to the `i`-th active constraint.

Arguments:

`i`

The constraint being accessed.

variable

```
ABA_VARIABLE *ABA_SUB::variable(int i)
```

Return Value:

A pointer to the `i`-th active variable.

Arguments:

`i`

The number of the variable being accessed.

lBound

The function `lBound()` can be used to access the lower of an active variable of the subproblem. **Warning:** This is the lower bound of the variable within the current subproblem which can differ from its global lower bound.

```
double ABA_SUB::lBound(int i) const
```

Return Value:

The lower bound of the `i`-th variable.

Arguments:

`i`

The number of the variable.

lBound

This version of the function `lBound()` sets the local lower bound of a variable. It does not change the global lower bound of the variable. The bound of a fixed or set variable should not be changed.

```
void ABA_SUB::lBound(int i, double x)
```

Arguments:

`i`

The number of the variable.

`x`

The new value of the lower bound.

uBound

The function `uBound()` can be used to access the upper of an active variable of the subproblem. **Warning:** This is the upper bound of the variable within the current subproblem which can differ from its global upper bound.

```
double ABA_SUB::uBound(int i) const
```

Return Value:

The upper bound of the `i`-th variable.

Arguments:

`i`

The number of the variable.

uBound

This version of the function `uBound()` sets thef local upper bound of a variable. This does not change the global lower bound of the variable. The bound of a fixed or set variable should not be changed.

```
void ABA_SUB::uBound(int i, double x)
```

Arguments:

`i`

The number of the variable.

`x`

The new value of the upper bound.

fsVarStat

```
ABA_FSVARSTAT *ABA_SUB::fsVarStat(int i)
```

Return Value:

A pointer to the status of fixing/setting of the `i`-th variable. Note, this is the local status of fixing/setting that might differ from the global status of fixing/setting of the variable (`variable(i)->fsVarStat()`).

Arguments:

`i`

The number of the variable.

lpVarStat

```
ABA_LPVARSTAT *ABA_SUB::lpVarStat(int i)
```

Return Value:

A pointer to the status of the variable `i` in the last solved linear program.

Arguments:

`i`

The number of the variable.

slackStat

```
ABA_SLACKSTAT *ABA_SUB::slackStat(int i)
```

Return Value:

A pointer to the status of the slack variable `i` in the last solved linear program.

Arguments:

`i`

The number of the slack variable.

generateLp (`virtual`)

The virtual function `generateLp()` instantiates an `LP` for the solution of the `LP`-relaxation in this subproblem. The generated `LP` is solved with `CPLEX`. By redefining this function in a derived class other `LP`-solvers can be used.

```
ABA_LPSUB *ABA_SUB::generateLp()
```

Return Value:

A pointer to an object of type `ABA_LPSUBCPLEX`.

reoptimize (`virtual`)

The function `reoptimize()` repeats the optimization of an already optimized subproblem. This function is used to determine the reduced costs for fixing variables of a new root of the remaining branch-and-bound tree.

```
void ABA_SUB::reoptimize ()
```

level

```
int ABA_SUB::level() const
```

Return Value:

The level of the subproblem in the branch-and-bound tree.

id

```
int ABA_SUB::id() const
```

Return Value:

The identity number of the subproblem.

lowerBound

```
double ABA_SUB::lowerBound() const
```

Return Value:

A lower bound on the optimal solution of the subproblem.

upperBound

```
double ABA_SUB::upperBound() const
```

Return Value:

An upper bound on the optimal solution of the subproblem.

dualBound

```
double ABA_SUB::dualBound() const
```

Return Value:

A bound which is better than the optimal solution of the subproblem in respect to the sense of the optimization, i.e., an upper for a maximization problem or a lower bound for a minimization problem, respectively.

dualBound

The function `dualBound()` sets the dual bound of the subproblem, and if the subproblem is the root node of the enumeration tree and the new value is better than its dual bound, also the global dual bound is updated. It is an error if the dual bound gets worse.

In normal applications it is not required to call this function explicitly. This is already done by ABACUS during the subproblem optimization.

```
void ABA_SUB::dualBound(double x)
```

Arguments:

`x`

The new value of the dual bound.

betterDual

```
bool ABA_SUB::betterDual(double x) const
```

Return Value:

`true`
: If `x` is better than the best known dual bound of the subproblem,

`false`
: otherwise.

father

```
ABA_SUB *ABA_SUB::father()
```

Return Value:

A pointer to the father of the subproblem in the branch-and-bound tree.

lp

```
ABA_LPSUB *ABA_SUB::lp()
```

Return Value:

A pointer to the linear program of the subproblem.

boundCrash

```
bool ABA_SUB::boundCrash() const
```

Return Value:

`true`
: If the dual bound is worse than the best known primal bound,

`false`
: otherwise.

status

```
ABA_SUB::STATUS ABA_SUB::status() const
```

Return Value:

The status of the subproblem optimization.

maxIterations

The function `maxIterations()` sets the maximal number of iterations in the cutting plane phase. Setting this value to 1 implies that no cuts are generated in the optimization process of the subproblem.

```
void ABA_SUB::maxIterations(int max)
```

Arguments:

`max`
: The maximal number of iterations.

fix (`virtual`)

The function `fix()` fixes a variable.

```
int ABA_SUB::fix(int i, ABA_FSVARSTAT *newStat, bool &newValue)
```

Return Value:

`1`

If a contradiction is found,

`0`

otherwise.

Arguments:

`i`

The number of the variable being fixed.

`newStat`

A pointer to an object storing the new status of the variable.

`newValue`

If the variable is fixed to a value different from the one of the last LP-solution, the argument `newValue` is set to `true`. Otherwise, it is set to `false`.

set (`virtual`)

The function `set()` sets a variable.

```
int ABA_SUB::set(int i, ABA_FSVARSTAT *newStat, bool &newValue)
```

Return Value:

`1`

If a contradiction is found,

`0`

otherwise.

Arguments:

`i`

The number of the variable being set.

`newStat`

A pointer to the object storing the new status of the the variable.

`newValue`

If the variable is set to a value different from the one of the last LP-solution, `newValue` is set to `true`. Otherwise, it is set to `false`.

set (virtual)

The function `set()` sets a variable.

```
int ABA_SUB::set(int i, ABA_FSVARSTAT::STATUS newStat, bool &newValue)
```

Return Value:

`1`

If a contradiction is found,

`0`

otherwise.

Arguments:

`i`

The number of the variable being set.

`newStat`

The new status of the variable.

`newValue`

If the variable is set to a value different from the one of the last LP-solution, `newValue` is set to `true`. Otherwise, it is set to `false`.

set (virtual)

The function `set()` sets a variable.

```
int ABA_SUB::set(int i, ABA_FSVARSTAT::STATUS newStat, double value, bool &newValue)
```

Return Value:

`1`

If a contradiction is found,

`0`

otherwise.

Arguments:

`i`

The number of the variable being set.

`newStat`

The new status of the variable.

`value`

The value the variable is set to.

`newValue`

If the variable is set to a value different from the one of the last LP-solution, `newValue` is set to `true`. Otherwise, it is set to `false`.

pausing (virtual)

Sometimes it is appropriate to put a subproblem back into the list of open subproblems. This is called pausing. In the default implementation the virtual function pausing() always returns false.

It could be useful to enforce pausing a node if a tailing off effect is observed during its first optimization.

```
bool ABA_SUB::pausing()
```

Return Value:

true

The function pausing() should return true if this condition is satisfied,

false

otherwise.

conEliminate (virtual)

The function conEliminate() can be used as an entry point for application specific elimination of constraints by redefinig it in derived classes.

The default implementation of this function calls either the function nonBindingConEliminate() or the function basicConEliminate() depending on the constraint elimination mode of the master that is initialized via the parameter file.

```
void ABA_SUB::conEliminate(ABA_BUFFER<int> &remove)
```

Arguments:

remove

The constraints that should be eliminated must be inserted in this buffer.

nonBindingConEliminate (virtual)

The function nonBindingConEliminate() retrieves the dynamic constraints with slack exceeding the value given by the parameter ConElimEps.

```
void ABA_SUB::nonBindingConEliminate(ABA_BUFFER<int> &remove)
```

Arguments:

remove

Stores the nonbinding constraints.

basicConEliminate (virtual)

The function basicConEliminate() retrieves all dynamic constraints having basic slack variable.

```
void ABA_SUB::basicConEliminate(ABA_BUFFER<int> &remove)
```

Arguments:

remove

Stores the nonbinding constraints.

varEliminate (virtual)

The function varEliminate() provides an entry point for application specific variable elimination that can be implemented by redefining this function in a derived class.

The default implementation selects the variables with the function redCostVarEliminate().

```
void ABA_SUB::varEliminate(ABA_BUFFER<int> &remove)
```

Arguments:

remove

The variables that should be removed have to be stored in this buffer.

redCostVarEliminate

The function redCostVarEliminate() retrieves all variables with "wrong" reduced costs.

```
void ABA_SUB::redCostVarEliminate(ABA_BUFFER<int> &remove)
```

Arguments:

remove

The variables with "wrong" reduced cost are stored in this buffer.

fathomTheSubTree (virtual)

The function fathomTheSubTree() fathoms all nodes in the subtree rooted at this subproblem. Dormant and Unprocessed nodes are also removed from the set of open subproblems.

```
void ABA_SUB::fathomTheSubTree()
```

actCon

```
ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *ABA_SUB::actCon()
```

Return Value:

A pointer to the currently active constraints.

actVar

```
ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *ABA_SUB::actVar()
```

Return Value:

A pointer to the currently active variables.

separate (virtual)

The virtual dummy function separate() must be redefined in derived classes for the generation of cutting planes. The default implementation does nothing.

```
int ABA_SUB::separate()
```

Return Value:

The number of generated cutting planes.

improve (virtual)

The function `improve()` can be redefined in derived classes in order to implement primal heuristics for finding feasible solutions. The default implementation does nothing.

```
int ABA_SUB::improve(double &primalValue)
```

Return Value:

`0`
: If no better solution could be found,

`1`
: otherwise.

Arguments:

`primalValue`
: Should hold the value of the feasible solution, if a better one is found.

infeasible

```
bool ABA_SUB::infeasible()
```

Return Value:

`true`
: If the subproblem does not contain a feasible solution,

`false`
: otherwise.

addVars (virtual)

The function `addVars()` adds both the variables in `newVars` to the set of active variables and to the linear program of the subproblem. If the new number of variables exceeds the maximal number of variables an automatic reallocation is performed.

```
int ABA_SUB::addVars(
                ABA_BUFFER<ABA_POOLSLOT<ABA_VARIABLE, ABA_CONSTRAINT> *> &newVars)
```

Return Value:

The number of added variables.

Arguments:

`newVars`
: A buffer storing the pool slots of the new variables.

addCons (virtual)

The function `addCons()` adds constraints to the active constraints and the linear program.

```
int ABA_SUB::addCons(
                ABA_BUFFER<ABA_POOLSLOT<ABA_CONSTRAINT, ABA_VARIABLE>*> &newCons)
```

Return Value:

The number of added constraints.

Arguments:

`newCons`
: A buffer storing the pool slots of the new constraints.

removeCons (`virtual`)

The function `removeCons()` adds constraints to the buffer of the removed constraints, which will be removed at the beginning of the next iteration of the cutting plane algorithm.

```
void ABA_SUB::removeCons(ABA_BUFFER<int> &remove)
```

Arguments:

`remove`

The constraints which should be removed.

removeCon (`virtual`)

The following version of the function `removeCon()` adds a single constraint to the set of constraints which are removed from the active set at the beginning of the next iteration.

```
void ABA_SUB::removeCon(int i)
```

Arguments:

`i`

The number of the constraint being removed.

varRealloc (`virtual`)

The function `varRealloc()` reallocates memory that at most `newSize` variables can be handled in the subproblem.

```
void ABA_SUB::varRealloc(int newSize)
```

Arguments:

`newSize`

The new maximal number of variables in the subproblem.

conRealloc (`virtual`)

The function `conRealloc()` reallocates memory that at most `newSize` constraints can be handled in the subproblem.

```
void ABA_SUB::conRealloc(int newSize)
```

Arguments:

`newSize`

The new maximal number of constraints of the subproblem.

nVar

```
int ABA_SUB::nVar() const
```

Return Value:

The number of active variables.

nCon

```
int ABA_SUB::nCon() const
```

Return Value:

The number of active constraints.

maxVar

```
int ABA_SUB::maxVar() const
```

Return Value:

The maximum number of variables which can be handled without reallocation.

maxCon

```
int ABA_SUB::maxCon() const
```

Return Value:

The maximum number of constraints which can be handled without reallocation.

initializeLp (`virtual`)

The function `initializeLp()` initializes the linear program.

```
int ABA_SUB::initializeLp()
```

Return Value:

`0`

If the linear program could be initialized successfully.

`1`

If the linear program turns out to be infeasible.

initMakeFeas (`virtual`)

The default implementation of the virtual `initMakeFeas()` does nothing. A reimplementation of this function should generate inactive variables until at least one variable `v` which satisfies the function `ABA_INFEASCON::goodVar(v)` for each infeasible constraint is found.

```
int ABA_SUB::initMakeFeas(ABA_BUFFER<ABA_INFEASCON*> &infeasCons,
                          ABA_BUFFER<ABA_VARIABLE*> &newVars,
                          ABA_POOL<ABA_VARIABLE, ABA_CONSTRAINT> **pool)
```

Return Value:

`0`

If the feasibility might have been restored,

`1`

otherwise.

Arguments:

`infeasCons`

The infeasible constraints.

`newVars`

The variables that might restore feasibility should be added here.

`pool`

A pointer to the pool to which the new variables should be added. If this is a 0-pointer the variables are added to the default variable pool. The default value is 0.

tailingOff (`virtual`)

The function `tailingOff()` is called when a tailing off effect according to the parameters `TailOffPercent` and `TailOffNLps` of the parameter file is observed. This function can be redefined in derived classes in order to perform actions to resolve the tailing off (e.g., switching on an enhanced separation strategy).

```
bool ABA_SUB::tailingOff()
```

Return Value:

`true`

If a branching step should be enforced. But before branching a pricing operation is perfored. The branching step is only performed if no variables are added. Otherwise, the cutting plane algorithm is continued.

`false`

If the cutting plane algorithm should be continued.

6.1.5 ABA_CONVAR

`ABA_CONVAR` is the common base class for constraints and variables, which are implemented in the derived classes `ABA_CONSTRAINT` and `ABA_VARIABLE`, respectively. It might seem a bit strange to implement a common base class for these two objects. Besides several technical reasons, there is linear programming duality which motivates this point of view. E.g., the separation problem for the primal problem is equivalent to the pricing problem for the dual problem.

`ABA_CONVAR` is **not** the base class for constraints and variables as they are used in the interface to the linear programming solver. There are the classes `ABA_ROW` and `ABA_COLUMN` for this purpose. `ABA_CONVAR` is the father of a class hierarchy for abstract constraints and variables which are used in the branch-and-bound algorithm.

```
class ABA_CONVAR : public ABA_ABACUSROOT {
  public:
    ABA_CONVAR (ABA_MASTER *master, ABA_SUB *sub, bool dynamic, bool local);
    virtual ~ABA_CONVAR();
    bool active() const;
    bool local() const;
    bool global() const;
    virtual bool dynamic() const;
    bool expanded() const;
    virtual void print(ostream &out);
    void _expand();
    void _compress();
    ABA_SUB *sub();
    void sub(ABA_SUB *sub);
    virtual unsigned hashKey();
    virtual const char *name();
```

```
    virtual bool equal(ABA_CONVAR *cv);

  protected:
    ABA_MASTER *master_;
    ABA_SUB *sub_;
    bool expanded_;
    int nReferences_;
    bool dynamic_;
    int nActive_;
    int nLocks_;
    bool local_;

  private:
    virtual void expand();
    virtual void compress();
};
```

master_

`ABA_MASTER *master_`

A pointer to the corresponding master of the optimization.

sub_

`ABA_SUB *sub_`

A pointer to the subproblem associated with the constraint/variable. This may be also the 0-pointer.

expanded_

`bool expanded_`

`true`, if expanded version of constraint/variables available.

nReferences_

`int nReferences_`

The number of references to the pool slot the constraint is stored (`ABA_POOLSLOTREF`).

dynamic_

`bool dynamic_`

If this member is `true` then the constraint/variable can be also removed from the active formulation after it is added the first time. For constraints/variables which should be never removed from the active formulation this member should be set to `false`.

nActive_

`int nActive_`

The number of active subproblems of which the constraint/variable belongs to the set of active constraints/variables. This value is always 0 after construction and has to be set and reset during the subproblem optimization. This member is mainly used to accelerate pool separation and to control that the same variable is not multiply included into a set of active variables.

nLocks_

`int nLocks_`

The number of locks which have been set on the constraint/variable.

local_

`bool local_`

This flag is `true` if the constraint/variable is only locally valid, otherwise it is false.

Constructor

```
ABA_CONVAR::ABA_CONVAR(ABA_MASTER *master, ABA_SUB *sub, bool dynamic, bool local)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sub`

A pointer the subproblem the constraint/variable is associated with. If the item is not associated with any subproblem, then this can also be the 0-pointer.

`dynamic`

If this argument is `true`, then the constraint/variable can also be removed again from the set of active constraints/variables after it is added once.

`local`

If `local` is `true`, then the constraint/variable is only locally valid.

Destructor (`virtual`)

```
ABA_CONVAR::~ABA_CONVAR()
```

active

The function `active()` checks if the constraint/variable is active in at least one active subproblem. In a parallel implementation this can be more than one subproblem.

```
bool ABA_CONVAR::active() const
```

Return Value:

`true`

If the constraint/variable is active,

`false`

otherwise.

local

```
bool ABA_CONVAR::local() const
```

Return Value:

`true`
: If the constraint/variable is only locally valid,

`false`
: otherwise.

global

```
bool ABA_CONVAR::global() const
```

Return Value:

`true`
: If the constraint/variable is globally valid,

`false`
: otherwise.

sub

```
ABA_SUB *ABA_CONVAR::sub()
```

Return Value:

A pointer to the subproblem associated with the constraint/variable. Note, this can also be the 0-pointer.

sub

This version of the function `sub()` associates a new subproblem with the constraint/variable.

```
void ABA_CONVAR::sub(ABA_SUB *sub)
```

Arguments:

`sub`
: The new subproblem associated with the constraint/variable.

dynamic (`virtual`)

```
bool ABA_CONVAR::dynamic() const
```

Return Value:

`true`
: If the constraint/variable can be also removed from the set of active constraints/variables after it has been activated,

`false`
: otherwise.

expanded

```
bool ABA_CONVAR::expanded() const
```

Return Value:

`true`

If the expanded format of a constraint/variable is available,

`false`

otherwise.

_expand

The function `_expand()` tries to generate the expanded format of the constraint/variable. This will be only possible if the virtual function `expand()` is redefined for the specific constraint/variable.

```
void ABA_CONVAR::_expand()
```

_compress

The function `_compress()` removes the expanded format of the constraint/variable. This will be only possible if the virtual function `compress()` is redefined for the specific constraint/variable.

```
void ABA_CONVAR::_compress()
```

expand (virtual)

The default implementation of the function `expand()` is void. It should be redefined in derived classes.

```
void ABA_CONVAR::expand()
```

compress (virtual)

Also the default implementation of the function `compress()` is void. It should be redefined in derived classes.

```
void ABA_CONVAR::compress()
```

print (virtual)

The function writes the constraint/variable on the stream `out`. This function is used since the output operator cannot be declared virtual. The default implementation writes `"ABA_CONVAR::print() is only a dummy."` on the stream `out`. We do not declare this function pure virtual since it is not really required, mainly only for debugging. In this case a constraint/variable specific redefinition is strongly recommended.

Normally, the implementation `out << *this` should be sufficient.

```
void ABA_CONVAR::print(ostream &out)
```

Arguments:

`out`

The output stream.

hashKey (`virtual`)

The function `hashKey()` should provide a key for the constraint/variable that can be used to insert it into a hash table. As usual for hashing, it is not required that any two items have different keys.

This function is required if the constraint/variable is stored in a pool of the class `ABA_NONDUPLPOOL`.

The default implementation shows a warning and calls `exit()`. This function is not a pure virtual function because in the default version of ABACUS it is not required.

```
unsigned ABA_CONVAR::hashKey()
```

Return Value:

An integer providing a hash key for the constraint/variable.

name (`virtual`)

The function `name()` should return the name of the constraint/variable. This function is required to emulate a simple run time type information (RTTI) that is still missing in G++. This function will be removed as soon as RTTI is supported sufficiently.

A user must take care that for each redefined version of this function in a derived class a unique name is returned. Otherwise fatal run time errors can occur. Therefore, we recommend to return always the name of the class.

This function is required if the constraint/variable is stored in a pool of the class `ABA_NONDUPLPOOL`.

The default implementation shows a warning and calls `exit()`. This function is not a pure virtual function because in the default version of ABACUS it is not required.

```
const char *ABA_CONVAR::name()
```

Return Value:

The name of the constraint/variable.

equal (`virtual`)

The function `equal()` should compare if the constraint/variable is identical (in a mathematical sense) with the constraint/variable `cv`. Using RTTI or its emulation provided by the function `name()` it is sufficient to implement this functions for constraints/variables of the same type.

This function is required if the constraint/variable is stored in a pool of the class `ABA_NONDUPLPOOL`.

The default implementation shows a warning and calls `exit()`. This function is not a pure virtual function because in the default version of ABACUS it is not required.

```
bool ABA_CONVAR::equal(ABA_CONVAR *cv)
```

Return Value:

`true`

If the constraint/variable represented by this object represents the same item as the constraint/variable `cv`,

`false`

otherwise.

Arguments:

`cv`

The constraint/variable that should be compared with this object.

6.1.6 ABA_CONSTRAINT

Constraints are one of the central items in a linear-programming based branch-and-bound algorithm. This class forms the virtual base class for all possible constraints given in pool format and is derived from the common base class ABA_CONVAR of all constraints and variables.

```
class ABA_CONSTRAINT : public ABA_CONVAR {
  public:
    ABA_CONSTRAINT (ABA_MASTER *master, ABA_SUB *sub,
                    ABA_CSENSE::SENSE sense, double rhs,
                    bool dynamic, bool local, bool liftable);
    ABA_CONSTRAINT (ABA_MASTER *master);
    ABA_CONSTRAINT(const ABA_CONSTRAINT &rhs);
    virtual ~ABA_CONSTRAINT();

    ABA_CSENSE *sense();
    virtual double coeff(ABA_VARIABLE *v) = 0;
    virtual double rhs();
    bool liftable() const;
    virtual bool valid(ABA_SUB *sub);
    virtual int genRow(ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *var,
                       ABA_ROW &row);
    virtual double slack(ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *variables,
                         double *x);
    virtual bool violated(ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *variables,
                          double *x, double *sl = 0);
    virtual bool violated(double slack) const;
    void printRow(ostream &out, ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *var);
    virtual double distance(double *x,
                            ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *actVar);
  protected:
    virtual ABA_INFEASCON::INFEAS voidLhsViolated(double newRhs) const;
    ABA_CSENSE sense_;
    double rhs_;
    ABA_CONCLASS *conClass_;
    bool liftable_;

  private:
    const ABA_CONSTRAINT &operator=(const ABA_CONSTRAINT &rhs);
};
```

sense_

`ABA_CSENSE sense_`

The sense of the constraint.

rhs_

`double rhs_`

The right hand side of the constraint.

conClass_

`ABA_CONCLASS *conClass_`

The constraint classification

liftable_

`bool liftable_`

This member is `true` if also coefficients of variables which have been inactive at generation time can be computed, `false` otherwise.

coeff (`virtual`)

```
virtual double coeff(ABA_VARIABLE *v) = 0
```

Return Value:

The coefficient of the variable `v` in the constraint.

Arguments:

`v`

A pointer to a variable.

Constructor

```
ABA_CONSTRAINT::ABA_CONSTRAINT(ABA_MASTER *master,
                               ABA_SUB *sub,
                               ABA_CSENSE::SENSE sense,
                               double rhs,
                               bool dynamic,
                               bool local,
                               bool liftable)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sub`

A pointer to the subproblem associated with the constraint. This can be also the 0-pointer.

`sense`

The sense of the constraint.

`rhs`

The right hand side of the constraint.

`dynamic`

If this argument is `true`, then the constraint can be removed from the active constraint set during the cutting plane phase of the subproblem optimization.

`local`

If this argument is `true`, then the constraint is considered to be only locally valid. In this case the argument `sub` must not be 0 as each locally valid constraint is associated with a subproblem.

`liftable`

If this argument is `true`, then a lifting procedure must be available, i.e., that the coefficients of variables which have not been active at generation time of the constraint can be computed.

Constructor

The following constructor initializes an empty constraint. This constructor is, e.g., useful if parallel separation is applied. In this case the constraint can be constructed and receive later its data by message passing.

```
ABA_CONSTRAINT::ABA_CONSTRAINT (ABA_MASTER *master)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

Copy Constructor

```
ABA_CONSTRAINT::ABA_CONSTRAINT(const ABA_CONSTRAINT &rhs)
```

Arguments:

`rhs`

The constraint being copied.

Destructor (`virtual`)

```
ABA_CONSTRAINT::~ABA_CONSTRAINT()
```

sense

```
ABA_CSENSE *ABA_CONSTRAINT::sense()
```

Return Value:

A pointer to the sense of the constraint.

rhs (`virtual`)

```
double ABA_CONSTRAINT::rhs()
```

Return Value:

The right hand side of the constraint.

liftable

The function `liftable()` checks if the constraint is liftable, i.e., if the coefficients of variables inactive at generation time of the constraint can be computed later.

```
bool ABA_CONSTRAINT::liftable() const
```

Return Value:

`true`

If the constraint can be lifted,

`false`

otherwise.

valid (virtual)

The function `valid()` checks if the constraint is valid for the subproblem `sub`. Per default, this is the case if the constraint is globally valid, or the subproblem associated with the constraint is an ancestor of the subproblem sub in the enumeration tree.

```
bool ABA_CONSTRAINT::valid(ABA_SUB *sub)
```

Return Value:

`true`

If the constraint is valid for the subproblem `sub`,

`false`

otherwise.

Arguments:

`sub`

The subproblem for which the validity is checked.

genRow (virtual)

The function `genRow()` generates the row format of the constraint associated with the variable set `var`. This function is declared virtual since faster constraint specific implementations might be desirable.

```
int ABA_CONSTRAINT::genRow(ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *var,
                           ABA_ROW &row)
```

Return Value:

The number of nonzero elements in the row format `row`.

Arguments:

`var`

The variable set for which the row format should be computed.

`row`

Stores the row format after calling this function.

slack (`virtual`)

The function `slack()` computes the slack of the vector `x` associated with the variable set `variables`.

```
double ABA_CONSTRAINT::slack(ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *variables,
                             double *x)
```

Return Value:

The slack induced by the vector `x`.

Arguments:

`variables`
: The variable set associated with the vector `x`.

`x`
: The values of the variables.

violated (`virtual`)

The function `violated()` checks if a constraint is violated by a vector `x` associated with a variable set.

```
bool ABA_CONSTRAINT::violated(ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *variables,
                              double *x,
                              double *sl)
```

Return Value:

`true`
: If the constraint is violated,

`false`
: otherwise.

Arguments:

`variables`
: The variables associated with the vector `x`.

`x`
: The vector for which the violation is checked.

`sl`
: If `sl` is nonzero, then `*sl` will store the value of the violation, i.e., the slack.

violated (`virtual`)

This version of function `violated()` checks for the violation given the slack of a vector.

```
bool ABA_CONSTRAINT::violated(double slack) const
```

Return Value:

`true`
: If the constraint is an equation and the `slack` is nonzero, or if the constraint is a $\leq$-inequality and the slack is negative, or the constraint is a $\geq$-inequality and the slack is positive,

`false`

otherwise.

Arguments:

`slack`

The slack of a vector.

voidLhsViolated (virtual)

The function `voidLhsViolated()` can be called if after variable elimination the left hand side of the constraint has become void and the right hand side has been adapted to `newRhs`. Then this function checks if the constraint is violated.

```
ABA_INFEASCON::INFEAS ABA_CONSTRAINT::voidLhsViolated(double newRhs) const
```

Return Value:

If the value `newRhs` violates the sense of the constraint, i.e, it is `</>/!= 0` and the sense of the constraint is `>=/<=/=`, then we return the infeasibility mode (`TooLarge` or `TooSmall`), otherwise we return `Feasible`.

Arguments:

`newRhs`

The right hand side of the constraint after the elimination of the variables.

printRow

The function `printRow()` writes the row format of the constraint associated with the variable set `var` on an output stream.

```
void ABA_CONSTRAINT::printRow(ostream &out,
                              ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *var)
```

Arguments:

`out`

The output stream.

`var`

The variables for which the row format should be written.

distance (virtual)

The function `distance()`.

```
double ABA_CONSTRAINT::distance(double *x,
                                ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *actVar)
```

Return Value:

The Euclidean distance of the vector `x` associated with the variable set `actVar` to the hyperplane induced by the constraint.

Arguments:

`x`

The point for which the distance should be computed.

`actVar`

The variables associated with `x`.

6.1.7 ABA_VARIABLE

Variables are one of the central items in a linear-programming based branch-and-bound algorithm. This class forms the virtual base class for all possible variables given in pool format and is derived from the common base class ABA_CONVAR of all constraints and variables.

```
class ABA_VARIABLE : public ABA_CONVAR {
  public:
    ABA_VARIABLE(ABA_MASTER *master, ABA_SUB *sub, bool dynamic, bool local,
                 double obj, double lBound, double uBound,
                 ABA_VARTYPE::TYPE type);
    virtual ~ABA_VARIABLE();

    ABA_VARTYPE::TYPE varType() const;
    bool discrete();
    bool binary();
    bool integer();
    virtual double obj();
    double uBound() const;
    double lBound() const;
    void uBound(double newValue);
    void lBound(double newValue);
    ABA_FSVARSTAT *fsVarStat();
    virtual bool valid(ABA_SUB *sub);
    virtual int genColumn(ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *actCon,
                          ABA_COLUMN &col);
    virtual double coeff(ABA_CONSTRAINT *con);
    virtual bool violated(double rc) const;
    virtual bool violated(ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *constraints,
                          double *y, double *slack = 0);
    virtual double redCost(ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *actCon,
                           double *y);
    virtual bool useful(ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *actCon,
                        double *y,
                        double lpVal);
    void printCol(ostream &out,
                  ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *constraints);

  protected:
    ABA_FSVARSTAT fsVarStat_;
    double obj_;
    double lBound_;
    double uBound_;
    ABA_VARTYPE type_;
};
```

fsVarStat_

```
ABA_FSVARSTAT fsVarStat_
```

The global status of fixing and setting of the variable.

obj_

```
double obj_
```

The objective function coefficient of the variable.

lBound_

`double lBound_`

The lower bound of the variable.

uBound_

`double uBound_`

The upper bound of the variable.

type_

`ABA_VARTYPE type_`

The type of the variable.

Constructor

```
ABA_VARIABLE::ABA_VARIABLE(ABA_MASTER *master,
                           ABA_SUB *sub,
                           bool dynamic,
                           bool local,
                           double obj,
                           double lBound,
                           double uBound,
                           ABA_VARTYPE::TYPE type)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sub`

A pointer to the subproblem associated with the variable. This can also be the 0-pointer.

`dynamic`

If this argument is `true`, then the variable can also be removed again from the set of active variables after it is added once.

`local`

If this argument is `true`, then the variable is only locally valid, otherwise it is globally valid. As a locally valid variable is always associated with a subproblem, the argument `sub` must not be 0 if `local` is `true`.

`obj`

The objective function coefficient.

`lBound`

The lower bound of the variable.

`uBound`

The upper bound of the variable.

`type`

The type of the variable.

Destructor (`virtual`)

```
ABA_VARIABLE::~ABA_VARIABLE()
```

varType

```
ABA_VARTYPE::TYPE ABA_VARIABLE::varType() const
```

Return Value:

The type of the variable.

discrete

```
bool ABA_VARIABLE::discrete()
```

Return Value:

`true`
: If the type of the variable is `Integer` or `Binary`,

`false`
: otherwise.

binary

```
bool ABA_VARIABLE::binary()
```

Return Value:

`true`
: If the type of the variable is `Binary`,

`false`
: otherwise.

integer

```
bool ABA_VARIABLE::integer()
```

Return Value:

`true`
: If the type of the variable is `Integer`,

`false`
: otherwise.

obj (`virtual`)

```
double ABA_VARIABLE::obj()
```

Return Value:

The objective function coefficient.

lBound

```
double ABA_VARIABLE::lBound() const
```

Return Value:

The lower bound of the variable.

lBound

This version of the function `lBound()` sets the lower bound of the variable.

```
void ABA_VARIABLE::lBound(double newBound)
```

Arguments:

`newBound`

The new value of the lower bound.

uBound

```
double ABA_VARIABLE::uBound() const
```

Return Value:

The upper bound of the variable.

uBound

This version of the function `uBound()` sets the upper bound of the variable.

```
void ABA_VARIABLE::uBound(double newBound)
```

Arguments:

`newBound`

The new value of the upper bound.

fsVarStat

```
ABA_FSVARSTAT *ABA_VARIABLE::fsVarStat()
```

Return Value:

A pointer to the global status of fixing and setting of the variable. Note, this is the global status of fixing/setting that might differ from the local status of fixing/setting a variable returned by the function `ABA_SUB::fsVarStat()`.

valid (`virtual`)

```
bool ABA_VARIABLE::valid(ABA_SUB *sub)
```

Return Value:

`true`

If the variable is globally valid, or the subproblem `sub` is an ancestor in the enumeration tree of the subproblem associated with the variable,

`false`

otherwise.

Arguments:

`sub`

The subproblem for which validity of the variable is checked.

coeff (`virtual`)

The function `coeff()` computes the coefficient of the variable in the constraint `con`. Per default the coefficient of a variable is computed indirectly via the coefficient of a constraint. Problem specific redefinitions might be required.

```
double ABA_VARIABLE::coeff(ABA_CONSTRAINT *con)
```

Return Value:

The coefficient of the variable in the constraint `con`.

Arguments:

`con`

The constraint of which the coefficient should be computed.

genColumn (`virtual`)

The function `genColumn()` computes the column `col` of the variable associated with the active constraints `*actCon`. Note, the upper and lower bound of the column are initialized with the global upper and lower bound of the variable. Therefore, an adaption with the local bounds might be required.

```
int ABA_VARIABLE::genColumn(ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *actCon,
                            ABA_COLUMN &col)
```

Return Value:

The number of nonzero entries in `col`.

Arguments:

`actCon`

The constraints for which the column of the variable should be computed.

`col`

Stores the column when the function terminates.

violated (`virtual`)

The function `violated()` checks, if a variable does not price out correctly, i.e., if the reduced cost `rc` is positive for a maximization problem and negative for a minimization problem, respectively.

```
bool ABA_VARIABLE::violated(double rc) const
```

Return Value:

`true`

If the variable does not price out correctly.

`false`

otherwise.

Arguments:

`rc`

The reduced cost of the variable.

violated (`virtual`)

This version of the function `violated()` checks if the variable does not price out correctly, i.e., if the reduced cost of the variable associated with the constraint set `constraints` and the dual variables `y` are positive for a maximization problem and negative for a minimization problem, respectively.

```
bool ABA_VARIABLE::violated(ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *constraints,
                            double *y,
                            double *r)
```

Return Value:

`true`

If the variable does not price out correctly.

`false`

otherwise.

Arguments:

`constraints`

The constraints associated with the dual variables `y`.

`y`

The dual variables of the constraint.

`r`

If `r` is not the 0-pointer, it will store the reduced cost after the function call. Per default `r` is 0.

redCost (`virtual`)

The function `redCost()` computes the reduced cost of the variable corresponding the constraint set `actCon` and the dual variables `y`.

```
double ABA_VARIABLE::redCost(ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *actCon,
                             double *y)
```

Return Value:

The reduced cost of the variable.

Arguments:

`actCon`

The constraints associated with the dual variables `y`.

`y`

The dual variables of the constraint.

useful (`virtual`)

An (inactive) discrete variable is considered as `useful()` if its activation might not produce only solutions worse than the best known feasible solution. This is the same criterion for fixing inactive variables by reduced cost criteria.

```
bool ABA_VARIABLE::useful(ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *actCon,
                          double *y,
                          double lpVal)
```

Return Value:

`true`

If the variable is considered as useful,

`false`

otherwise.

Arguments:

`actCon`

The active constraints.

`y`

The dual variables of these constraints.

`lpVal`

The value of the linear program.

printcol

The function `printCol()` writes the column of the variable corresponding to the `constraints` on the stream `out`.

```
void ABA_VARIABLE::printCol(ostream &out,
                            ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE> *constraints)
```

Arguments:

`out`

The output stream.

`constraints`

The constraints for which the column should be written.

6.2 System Classes

This section documents (almost) all internal system classes of ABACUS. This classes are usually not involved in the derivation process for the implementation. However for retrieving special information (e.g., about the linear program) or for advanced usage we provide here a detailed documentation.

6.2.1 ABA_OPTSENSE

We can either minimize or maximize the objective function. We encapsulate this information in a class since it is required in various classes, e.g., in the master of the branch-and-bound algorithm and in the linear program.

```
class ABA_OPTSENSE : public ABA_ABACUSROOT {
  public:
    enum SENSE {Min, Max, Unknown};
    ABA_OPTSENSE(SENSE s = Unknown);
    friend ostream &operator<<(ostream& out, const ABA_OPTSENSE &rhs);
    void sense(SENSE s);
    SENSE sense() const;
    bool min() const;
    bool max() const;
    bool unknown() const;
};
```

enum SENSE

The enumeration defining the sense of optimization.

`Min`
: Minimization problem.

`Max`
: Maximization problem.

`Unknown`
: Unknown optimization sense, required to recognize uninitialized object.

Constructor

The constructor initializes the optimization sense.

```
ABA_OPTSENSE::ABA_OPTSENSE(SENSE s)
```

Arguments:

`s`
: The sense of the optimization. The default value is `Unknown`.

Output Operator

The output operator writes the optimization sense on an output stream in the form `maximize`, `minimize`, or `unknown`.

```
ostream &operator<<(ostream &out, const ABA_OPTSENSE &rhs)
```

Return Value:

The output stream.

Arguments:

`out`
: The output stream.

`rhs`
: The sense being output.

sense

```
ABA_OPTSENSE::SENSE ABA_OPTSENSE::sense() const
```

Return Value:

The sense of the optimization.

sense

This version of the function `sense()` sets the optimization sense.

```
void ABA_OPTSENSE::sense(SENSE s)
```

Arguments:

`s`

The new sense of the optimization.

min

```
bool ABA_OPTSENSE::min() const
```

Return Value:

`true`

If it is minimization problem,

`false`

otherwise.

max

```
bool ABA_OPTSENSE::max() const
```

Return Value:

`true`

If it is maximization problem,

`false`

otherwise.

unknown

```
bool ABA_OPTSENSE::unknown() const
```

Return Value:

`true`

If the optimization sense is unknown,

`false`

otherwise.

6.2.2 ABA_CSENSE

The most important objects in a cutting plane algorithm are constraints, which can be equations (`Equal`) or inequalities with the sense $\leq$ (`Less`) or the sense $\geq$ (`Greater`). We implement the sense of optimization as a class since we require it both in the classes `ABA_CONSTRAINT` and `ABA_ROW`.

```
class ABA_CSENSE : public ABA_ABACUSROOT {
  public:
    enum SENSE {Less, Equal, Greater};
    ABA_CSENSE(ABA_GLOBAL *glob);
    ABA_CSENSE(ABA_GLOBAL *glob, SENSE s);
    ABA_CSENSE(ABA_GLOBAL *glob, char s);
    friend ostream& operator<<(ostream &out, const ABA_CSENSE &rhs);
    const ABA_CSENSE &operator=(SENSE rhs);
    SENSE sense() const;
    void sense(SENSE s);
    void sense(char s);
};
```

enum SENSE

`Less`

$\leq$

`Equal`

$=$

`Greater`

$\geq$

Constructor

If the default constructor is used, the sense is undefined.

```
ABA_CSENSE::ABA_CSENSE(ABA_GLOBAL *glob)
```

Arguments:

`glob`

A pointer to the corresponding global object.

Constructor

This constructor initializes the sense.

```
ABA_CSENSE::ABA_CSENSE(ABA_GLOBAL *glob, SENSE s)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`s`

The sense.

Constructor

With this constructor the sense of the constraint can also be initialized with a single letter.

```
ABA_CSENSE::ABA_CSENSE(ABA_GLOBAL *glob, char s)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`s`

A character representing the sense: `E` or `e` stand for `Equal`, `G` and `g` stand for `Greater`, and `L` or `l` stand for `Less`.

Output Operator

The output operator writes the sense on an output stream in the form `<=`, `=`, or `>=`.

```
ostream &operator<<(ostream &out, const ABA_CSENSE &rhs)
```

Return Value:

The output stream.

Arguments:

`out`

The output stream.

`rhs`

The sense being output.

Assignment Operator

The default assignment operator is overloaded such that also the enumeration `SENSE` can be used on the right hand side.

```
const ABA_CSENSE &ABA_CSENSE::operator=(SENSE rhs)
```

Return Value:

A reference to the sense.

Arguments:

`rhs`

The new sense.

sense

```
ABA_CSENSE::SENSE ABA_CSENSE::sense() const
```

Return Value:

The sense of the constraint.

sense

This overloaded version of `sense()` changes the sense of the constraint.

```
void ABA_CSENSE::sense(SENSE s)
```

Arguments:

`s`
: The new sense.

sense

The sense can also be changed by a character as in the constructor `ABA_CSENSE(ABA_GLOBAL *glob, char s)`.

```
void ABA_CSENSE::sense(char s)
```

Arguments:

`s`
: The new sense.

6.2.3 ABA_VARTYPE

Variables can be of three different types: `Continuous`, `Integer` or `Binary`. We distinguish `Integer` and `Binary` variables since some operations are performed differently (e.g., branching).

```
class ABA_VARTYPE : public ABA_ABACUSROOT {
  public:
    enum TYPE {Continuous, Integer, Binary};
    ABA_VARTYPE();
    ABA_VARTYPE(TYPE t);
    friend ostream &operator<<(ostream &out, const ABA_VARTYPE &rhs);
    TYPE type() const;
    void type(TYPE t);
    bool discrete() const;
    bool binary() const;
    bool integer() const;
};
```

enum TYPE

The enumeration with the different variable types.

`Continuous`
: A continuous variable.

`Integer`
: A general integer variable.

`Binary`
: A variable having value 0 or 1.

Constructor

The default constructor lets the type of the variable uninitialized.

```
ABA_VARTYPE::ABA_VARTYPE()
```

Constructor

This constructor initializes the variable type.

```
ABA_VARTYPE::ABA_VARTYPE(TYPE t)
```

Arguments:

`t`
: The variable type.

Output Operator

The output operator writes the variable type to an output stream in the format `Continuous`, `Integer`, or `Binary`.

```
ostream &operator<<(ostream &out, const ABA_VARTYPE &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`
: The output stream.

`rhs`
: The variable type being output.

type

```
ABA_VARTYPE::TYPE ABA_VARTYPE::type() const
```

Return Value:

The type of the variable.

type

This version of the function `type()` sets the variable type.

```
void ABA_VARTYPE::type(TYPE t)
```

Arguments:

`t`
: The new type of the variable.

discrete

```
bool ABA_VARTYPE::discrete() const
```

Return Value:

`true`
: If the type of the variable is `Integer` or `Binary`,

`false`
: otherwise.

binary

```
bool ABA_VARTYPE::binary() const
```

Return Value:

`true`

If the type of the variable `Binary`,

`false`

otherwise.

integer

```
bool ABA_VARTYPE::integer() const
```

Return Value:

`true`

If the type of the variable is `Integer`,

`false`

otherwise.

6.2.4 ABA_FSVARSTAT

If a variable is fixed to a value, then this means that it keeps this value "forever". If it is set, then the variable keeps the value in the subproblem where the setting is performed and in the subproblems of the subtree rooted at this subproblem.

```
class ABA_FSVARSTAT : public ABA_ABACUSROOT {
  public:
    enum STATUS {Free, SetToLowerBound, Set, SetToUpperBound,
                 FixedToLowerBound, Fixed, FixedToUpperBound};

    ABA_FSVARSTAT(ABA_GLOBAL *glob);
    ABA_FSVARSTAT(ABA_GLOBAL *glob, STATUS status);
    ABA_FSVARSTAT(ABA_GLOBAL *glob, STATUS status, double value);
    ABA_FSVARSTAT(ABA_FSVARSTAT *fsVarStat);
    friend ostream &operator<<(ostream& out, const ABA_FSVARSTAT &rhs);
    STATUS status() const;
    void status(STATUS stat);
    void status(STATUS stat, double val);
    void status(const ABA_FSVARSTAT *stat);
    double value() const;
    void value(double val);
    bool fixed() const;
    bool set() const;
    bool fixedOrSet() const;
    bool contradiction(ABA_FSVARSTAT *fsVarStat) const;
    bool contradiction(STATUS status, double value = 0) const;
};
```

enum STATUS

The enumeration defining the different statuses of variables from the point of view of fixing and setting:

`Free`

The variable is neither fixed nor set.

`SetToLowerBound`

The variable is set to its lower bound.

`Set`

The variable is set to a value which can be accessed with the member function `value()`.

`SetToUpperbound`

The variable is set to its upper bound.

`FixedToLowerBound`

The variable is fixed to its lower bound.

`Fixed`

The variable is fixed to a value which can be accessed with the member function `value()`.

`FixedToUpperBound`

The variable is fixed to its upper bound.

Constructor

This constructor initializes the status as `Free`.

```
ABA_FSVARSTAT::ABA_FSVARSTAT(ABA_GLOBAL *glob)
```

Arguments:

`glob`

A pointer to a global object.

Constructor

This constructor initializes the status explicitely.

```
ABA_FSVARSTAT::ABA_FSVARSTAT(ABA_GLOBAL *glob, STATUS status)
```

Arguments:

`glob`

A pointer to a global object.

`status`

The initial status that must neither be `Fixed` nor `Set`. For these two statuses the next constructor has to be used.

Constructor

This constructor initializes the status explicitely to `Fixed` or `Set`.

```
ABA_FSVARSTAT::ABA_FSVARSTAT(ABA_GLOBAL *glob, STATUS status, double value)
```

Arguments:

`glob`

A pointer to a global object.

`status`

The initial status that must be `Fixed` or `Set`.

`value`

The value associated with the status `Fixed` or `Set`.

Constructor

This constructor makes a copy.

```
ABA_FSVARSTAT::ABA_FSVARSTAT(ABA_FSVARSTAT *fsVarStat)
```

Arguments:

`fsVarStat`

The status is initialized with a copy of `*fsVarStat`.

Output Operator

The output operator writes the status and, if the status is `Fixed` or `Set`, also its value on an output stream.

```
ostream &operator<<(ostream& out, const ABA_FSVARSTAT &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The variable status being output.

status

```
ABA_FSVARSTAT::STATUS ABA_FSVARSTAT::status() const
```

Return Value:

The status of fixing or setting.

status

This version of the function `status()` assigns a new status. For specifying also a value in case of the statuses `Fixed` or `Set` the next version of this function can be use.

```
void ABA_FSVARSTAT::status(STATUS stat)
```

Arguments:

`stat`

The new status.

status

This version of the function `status()` can assign a new status also for the statuses `Fixed` and `Set`.

```
void ABA_FSVARSTAT::status(STATUS stat, double val)
```

Arguments:

`stat`

The new status.

`val`

A value associated with the new status.

status

A version of `status()` for assigning the status of an other object of the class `ABA_FSVARSTAT`.

```
void ABA_FSVARSTAT::status(const ABA_FSVARSTAT *stat)
```

Arguments:

`stat`

A pointer to the object that status and value is copied.

value

```
double ABA_FSVARSTAT::value() const
```

Return Value:

The value of fixing or setting if the variable has status `Fixed` or `Set`.

value

This version of `value()` assigns a new value of fixing or setting.

```
void ABA_FSVARSTAT::value(double val)
```

Arguments:

`val`

The new value.

fixed

```
bool ABA_FSVARSTAT::fixed() const
```

Return Value:

`true`
: If the status is `FixedToLowerBound`, `Fixed`, or `FixedToUpperBound`,

`false`
: otherwise.

set

```
bool ABA_FSVARSTAT::set() const
```

Return Value:

`true`
: If the status is `SetToLowerBound`, `Set`, or `SetToUpperBound`,

`false`
: otherwise.

fixedOrSet

```
bool ABA_FSVARSTAT::fixedOrSet() const
```

Return Value:

`false`
: If the status is `Free`,

`true`
: otherwise.

contradiction

We say there is a contradiction between two status if they are fixed/set to different bounds or values. However, two statuses are not contradiction if one of them is "fixed" and the other one is "set", if this fixing/setting refers to the same bound or value.

```
bool ABA_FSVARSTAT::contradiction(ABA_FSVARSTAT *fsVarStat) const
```

Return Value:

`true`
: If there is a contradiction between the status of this object and `fsVarStat`,

`false`
: otherwise.

Arguments:

`fsVarStat`
: A pointer to the status with which contradiction is is tested.

contradiction

Another version of the function `contradiction()`.

```
bool ABA_FSVARSTAT::contradiction(STATUS status, double value) const
```

Return Value:

`true`

If there is a contradiction between the status of this object and (`status`,`value`),

`false`

otherwise.

Arguments:

`status`

The status with which contradiction is checked.

`value`

The value with which contradiction is checked. The default value of `value` is 0.

6.2.5 ABA_LPVARSTAT

After the solution of a linear program by the simplex method each variable receives a status indicating if the variable is contained in the basis of the optimal solution, or is nonbasic and has a value equal to its lower or upper bound, or is a free variable not contained in the basis. We extend this notion since later in the interface from a cutting plane algorithm to the linear program variables might be eliminated.

```
class ABA_LPVARSTAT : public ABA_ABACUSROOT {
  public:
    enum STATUS {AtLowerBound, Basic, AtUpperBound, NonBasicFree,
                 Eliminated, Unknown};

    ABA_LPVARSTAT(ABA_GLOBAL *glob);
    ABA_LPVARSTAT(ABA_GLOBAL *glob, STATUS status);
    ABA_LPVARSTAT(ABA_LPVARSTAT *lpVarStat);
    friend ostream &operator<<(ostream& out, const ABA_LPVARSTAT &rhs);
    STATUS status() const;
    void status(STATUS stat);
    void status(const ABA_LPVARSTAT *stat);
    bool atBound() const;
    bool basic() const;
};
```

enum STATUS

The enumeration of the statuses a variable gets from the linear program solver:

`AtLowerBound`

The variable is at its lower bound, but not in the basis.

`Basic`

The variable is in the basis.

`AtUpperBound`

The variable is at its upper bound , but not in the basis.

`NonBasicFree`

The variable is unbounded and not in the basis.

`Eliminated`

The variable has been removed by our preprocessor in the class `ABA_LPSUB`. So, it is not present in the LP-solver.

`Unknown`

The LP-status of the variable is unknown since no LP has been solved. This status is also assigned to variables which are fixed or set, yet still contained in the `LP` to avoid a wrong setting or fixing by reduced costs.

Constructor

This constructor initializes the status as `Unknown`.

```
ABA_LPVARSTAT::ABA_LPVARSTAT(ABA_GLOBAL *glob)
```

Arguments:

`glob`

A pointer to the corresponding global object.

Constructor

This constructor initializes the `ABA_LPVARSTAT`.

```
ABA_LPVARSTAT::ABA_LPVARSTAT(ABA_GLOBAL *glob, STATUS status)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`status`

The initial status.

Constructor

This constructor make a copy of `*lpVarStat`.

```
ABA_LPVARSTAT::ABA_LPVARSTAT(ABA_LPVARSTAT *lpVarStat)
```

Arguments:

`lpVarStat`

A copy of this object is made.

Output Operator

The output operator writes the `STATUS` to an output stream in the form `AtLowerBound`, `Basic`, `AtUpperBound`, `NonBasicFree`, `Eliminated`, `Unknown`.

```
ostream &operator<<(ostream& out, const ABA_LPVARSTAT &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`
: The output stream.

`rhs`
: The status being output.

status

```
ABA_LPVARSTAT::STATUS ABA_LPVARSTAT::status() const
```

Return Value:

The LP-status.

status

This version of `status()` sets the status.

```
void ABA_LPVARSTAT::status(STATUS stat)
```

Arguments:

`stat`
: The new LP-status.

status

Another version of the function `status()` for setting the status.

```
void ABA_LPVARSTAT::status(const ABA_LPVARSTAT *stat)
```

Arguments:

`stat`
: The new LP-status.

atBound

```
bool ABA_LPVARSTAT::atBound() const
```

Return Value:

`true`
: If the variable status is `AtUpperBound` or `AtLowerBound`,

`false`
: otherwise.

basic

```
bool ABA_LPVARSTAT::basic() const
```

Return Value:

`true`
: If the status is `Basic`,

`false`
: otherwise.

6.2.6 ABA_SLACKSTAT

As for the structural variables the simplex method also assigns a unique status to each slack variable. A slack variable can be a basic or a nonbasic variable. If it is a nonbasic variable, then we distinguish if the slack variable has value zero or nonzero.

```
class ABA_SLACKSTAT : public ABA_ABACUSROOT {
  public:
    enum STATUS {Basic, NonBasicZero, NonBasicNonZero, Unknown};

    ABA_SLACKSTAT(ABA_GLOBAL *glob);
    ABA_SLACKSTAT(ABA_GLOBAL *glob, STATUS status);
    friend ostream  &operator<<(ostream& out, const ABA_SLACKSTAT &rhs);
    STATUS status() const;
    void status(STATUS stat);
    void status(const ABA_SLACKSTAT *stat);
};
```

enum STATUS

The different statuses of a slack variable:

`Basic`
: The slack variable belongs to the basis.

`NonBasicZero`
: The slack variable does not belong to the basis and has value zero.

`NonBasicNonZero`
: The slack variable does not belong to the basis and has a nonzero value.

`Unknown`
: The status of the slack variable is not known since no linear program with the corresponding constraint has been solved.

Constructor

This constructor initializes the status as `Unknown`.

```
ABA_SLACKSTAT::ABA_SLACKSTAT(ABA_GLOBAL *glob)
```

Arguments:

`glob`
: A pointer to the corresponding global object.

Constructor

A constructor with initialization.

```
ABA_SLACKSTAT::ABA_SLACKSTAT(ABA_GLOBAL *glob, STATUS status)
```

Arguments:

`glob`
: A pointer to the corresponding global object.

`status`
: The slack variable receives the status `status`.

Output Stream

The output operator writes the status to an output stream in the format `Basic`, `NonBasicZero`, `NonBasicNonZero`, or `Unknown`.

```
ostream &operator<<(ostream &out, const ABA_SLACKSTAT &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`
: The output stream.

`rhs`
: The status being output.

status

```
ABA_SLACKSTAT::STATUS ABA_SLACKSTAT::status() const
```

Return Value:

The status of the slack variable.

status

This version of the function `status()` sets the status of the slack variable.

```
void ABA_SLACKSTAT::status(STATUS stat)
```

Arguments:

`stat`
: The new status of the slack variable.

status

This version of the function `status()` sets the status to the one of `*stat`.

```
void ABA_SLACKSTAT::status(const ABA_SLACKSTAT *stat)
```

Arguments:

`stat`
: The status of the slack variable is set to `*stat`.

6.2.7 ABA_LP

The following section provides a generic interface class to linear programs, from which we will derive further classes both for the solution of LP-relaxations (ABA_LPSUB) with a linear-programming based branch-and-bound algorithm and for interfaces to LP-solvers (ABA_CPLEXIF).

The framework should be very flexible in the use of different LP-solvers. Therefore, we implement in the class ABA_LP a very general interface to the linear program. All functions of the framework communicate with the linear program only by the public functions of the class ABA_LP. Linear programs cannot only be used for solving the LP-relaxation within the branch-and-cut algorithm. There are also techniques in integer programming where linear programming is used for generating cutting planes and for applying heuristics. Therefore, we design the class ABA_LP that it can be used very generally.

```
class ABA_LP : public ABA_ABACUSROOT {
  public:
    enum OPTSTAT{Optimal, Unoptimized, Error,
                 Feasible, Infeasible, Unbounded};
    enum SOLSTAT{Available, Missing};
    enum METHOD {Primal, Dual, Barrier};
    ABA_LP (ABA_MASTER *master);
    virtual ~ABA_LP ();
    friend ostream &operator<<(ostream& out, const ABA_LP& rhs);
    void initialize(ABA_OPTSENSE sense, int nRow, int maxRow,
                    int nCol, int maxCol,
                    ABA_ARRAY<double> &obj, ABA_ARRAY<double> &lBound,
                    ABA_ARRAY<double> &uBound, ABA_ARRAY<ABA_ROW*> &rows);
    void initialize(ABA_OPTSENSE sense, int nRow, int maxRow,
                    int nCol, int maxCol,
                    ABA_ARRAY<double> &obj, ABA_ARRAY<double> &lBound,
                    ABA_ARRAY<double> &uBound, ABA_ARRAY<ABA_ROW*> &rows,
                    ABA_ARRAY<ABA_LPVARSTAT::STATUS> &lpVarStat,
                    ABA_ARRAY<ABA_SLACKSTAT::STATUS> &slackStat);
    virtual void loadBasis(ABA_ARRAY<ABA_LPVARSTAT::STATUS> &lpVarStat,
                           ABA_ARRAY<ABA_SLACKSTAT::STATUS> &slackStat);
    ABA_OPTSENSE sense() const;
    void sense(const ABA_OPTSENSE &newSense);
    int nRow() const;
    int maxRow() const;
    int nCol() const;
    int maxCol() const;
    int nnz() const;
    double obj(int i) const;
    double lBound(int i) const;
    double uBound(int i) const;
    void row(int i, ABA_ROW &r) const;
    double rhs(int i) const;
    virtual double value() const;
    virtual double xVal(int i);
    virtual double barXVal(int i);
    virtual double reco(int i);
    virtual double yVal(int c);
    virtual double slack(int c);
    SOLSTAT xValStatus() const;
    SOLSTAT barXValStatus() const;
    SOLSTAT yValStatus() const;
```

```
    SOLSTAT recoStatus() const;
    SOLSTAT slackStatus() const;
    SOLSTAT basisStatus() const;
    int nOpt() const;
    virtual bool infeasible() const;
    virtual int getInfeas(int &infeasRow, int &infeasCol, double *bInvRow);
    virtual ABA_LPVARSTAT::STATUS lpVarStat(int i);
    virtual ABA_SLACKSTAT::STATUS slackStat(int i);
    virtual OPTSTAT optimize(METHOD method);
    void remRows(ABA_BUFFER<int> &ind);
    void addRows(ABA_BUFFER<ABA_ROW*> &newRows);
    void remCols(ABA_BUFFER<int> &cols);
    void addCols(ABA_BUFFER<ABA_COLUMN*> &newCols);
    void changeRhs(ABA_ARRAY<double> &newRhs);
    virtual void changeLBound(int i, double newLb);
    virtual void changeUBound(int i, double newUb);
    virtual int pivotSlackVariableIn(ABA_BUFFER<int> &rows);
    void rowRealloc(int newSize);
    void colRealloc(int newSize);
    int writeBasisMatrix(const char *fileName);
    int setSimplexIterationLimit(int limit);
    int getSimplexIterationLimit(int &limit);

  protected:
    void colsNnz(int nRow, ABA_ARRAY<ABA_ROW*> &rows, ABA_ARRAY<int> &nnz);
    void rows2cols(int nRow, ABA_ARRAY<ABA_ROW*> &rows,
                   ABA_ARRAY<ABA_SPARVEC*> &cols);
    void rowRangeCheck(int r) const;
    void colRangeCheck(int i) const;

  private:
    ABA_LP(const ABA_LP &rhs);
    const ABA_LP &operator=(const ABA_LP &rhs);
};
```

enum OPTSTAT

The optimization status of the linear program.

`Unoptimized`

: Optimization is still required, this is also the case for reoptimization.

`Optimized`

: The optimization has been performed, yet only a call to `_getSol()` can give us the status of optimization.

`Error`

: An error has happened during optimization.

`Optimal`

: The optimal solution has been computed.

`Feasible`

: A primal feasible solution for the linear program, but not the optimal solution has been found.

`Infeasible`

The linear program is primal infeasible.

`Unbounded`

The linear program is unbounded.

enum SOLSTAT

This enumeration describes if parts of the solution like x-values, reduced costs, etc. are available.

`Available`

The part of the solution is available.

`Missing`

The part of the solution is missing.

enum METHOD

The solution method for the linear program.

`Primal`

The primal simplex method.

`Dual`

The dual simplex method.

`Barrier`

The barrier method.

Constructor

```
ABA_LP::ABA_LP (ABA_MASTER *master)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

Destructor (`virtual`)

```
ABA_LP::~ABA_LP()
```

initialize

The function `initialize()` loads the linear program defined by its arguments.

```
void ABA_LP::initialize(ABA_OPTSENSE sense,
                        int nRow,
                        int maxRow,
                        int nCol,
                        int maxCol,
                        ABA_ARRAY<double> &obj,
                        ABA_ARRAY<double> &lBound,
                        ABA_ARRAY<double> &uBound,
                        ABA_ARRAY<ABA_ROW*> &rows)
```

Arguments:

`sense`

The sense of the objective function.

`nCol`

The number of columns (variables).

`maxCol`

The maximal number of columns.

`nRow`

The number of rows.

`maxRow`

The maximal number of rows.

`obj`

An array with the objective function coefficients.

`lb`

An array with the lower bounds of the columns.

`ub`

An array with the upper bounds of the columns.

`rows`

An array storing the rows of the problem.

initialize

This version of the function `initialize()` performs like its previous version, but also initializes the basis with the arguments:

```
void ABA_LP::initialize(ABA_OPTSENSE sense,
                        int nRow,
                        int maxRow,
                        int nCol,
                        int maxCol,
                        ABA_ARRAY<double> &obj,
                        ABA_ARRAY<double> &lBound,
                        ABA_ARRAY<double> &uBound,
                        ABA_ARRAY<ABA_ROW*> &rows,
                        ABA_ARRAY<ABA_LPVARSTAT::STATUS> &lpVarStat,
                        ABA_ARRAY<ABA_SLACKSTAT::STATUS> &slackStat)
```

Arguments:

`lpVarStat`

An array storing the status of the columns.

`slackStat`

An array storing the status of the slack variables.

loadBasis (`virtual`)

The function `loadBasis()` loads a new basis for the linear program.

```
void ABA_LP::loadBasis(ABA_ARRAY<ABA_LPVARSTAT::STATUS> &lpVarStat,
                       ABA_ARRAY<ABA_SLACKSTAT::STATUS> &slackStat)
```

Arguments:

`lpVarStat`
: An array storing the status of the columns.

`slackStat`
: An array storing the status of the slack variables.

optimize (`virtual`)

The function `optimize()` performs the optimization of the linear program.

```
ABA_LP::OPTSTAT ABA_LP::optimize(METHOD method)
```

Return Value:

The status of the optimization.

Arguments:

`method`
: The method with which the optimization is performed.

remRows

The function `remRows()` removes rows of the linear program.

```
void ABA_LP::remRows(ABA_BUFFER<int> &ind)
```

Arguments:

`ind`
: The numbers of the rows that should be removed.

addRows

The function `addRows()` adds rows to the linear program. If the new number of rows exceeds the maximal number of rows a reallocation is performed.

```
void ABA_LP::addRows(ABA_BUFFER<ABA_ROW*> &newRows)
```

Arguments:

`newRows`
: The rows that should be added to the linear program.

rowRealloc

The function `rowRealloc()` performs a reallocation of the row space of the linear program.

```
void ABA_LP::rowRealloc(int newSize)
```

Arguments:

`newSize`
: The new maximal number of rows of the linear program.

remCols

The function `remCols()` removes columns from the linear program.

```
void ABA_LP::remCols(ABA_BUFFER <int> &cols)
```

Arguments:

`cols`

The numbers of the columns that should be removed.

addCols

The function `addCols()` adds columns to the linear program. If the new number of columns exceeds the maximal number of columns a reallocation is performed.

```
void ABA_LP::addCols(ABA_BUFFER<ABA_COLUMN*> &newCols)
```

Arguments:

`newCols`

The new columns that are added.

colRealloc

The function `colRealloc()` performs a reallocation of the column space of the linear program.

```
void ABA_LP::colRealloc(int newSize)
```

Arguments:

`newSize`

The new maximal number of columns of the linear program.

changeRhs

The function `changeRhs()` changes the complete right hand side of the linear program.

```
void ABA_LP::changeRhs(ABA_ARRAY<double> &newRhs)
```

Arguments:

`newRhs`

The new right hand side of the rows.

changeLBound (`virtual`)

The function `changeLBound()`changes the lower bound of a single column.

```
void ABA_LP::changeLBound(int i, double newLb)
```

Arguments:

`i`

The column.

`newLb`

The new lower bound of the column.

changeUBound (`virtual`)

The function `changeUBound()` changes the upper bound of a single column.

```
void ABA_LP::changeUBound(int i, double newUb)
```

Arguments:

`i`
: The column.

`newUb`
: The new upper bound of the column.

pivotSlackVariableIn (`virtual`)

The function `pivotSlackVariableIn()` pivots the slack variables stored in the buffer `rows` into the basis.

```
int ABA_LP::pivotSlackVariableIn(ABA_BUFFER<int> &rows)
```

Return Value:

`0`
: All variables could be pivoted in,

`1`
: otherwise.

Arguments:

`rows`
: The numbers of the slack variables that should be pivoted in.

sense

```
ABA_OPTSENSE ABA_LP::sense() const
```

Return Value:

The optimization sense of the linear program.

sense

```
void ABA_LP::sense(const ABA_OPTSENSE &newSense)
```

Arguments:

`newSense`
: The new sense of the optimization.

nRow

```
int ABA_LP::nRow() const
```

Return Value:

The current number of rows.

maxRow

```
int ABA_LP::maxRow() const
```

Return Value:

The maximal number of rows which can currently be stored. However, more rows can be added to the LP since in this case the automatic reallocation is executed.

nCol

```
int ABA_LP::nCol() const
```

Return Value:

The current number of columns.

maxCol

```
int ABA_LP::maxCol() const
```

Return Value:

The maximal number of columns which can currently be stored. However, more columns can be added to the LP since in this case the automatic reallocation is executed or a reallocation can be performed by an explicit call to `colRealloc()`.

nnz

```
int ABA_LP::nnz() const
```

Return Value:

The number of nonzero elements in the constraint matrix. To be more exact, this is the number of elements stored in the sparse representation of the constraint matrix. If a (redundant) coefficient with value zero is stored, this element is also counted.

obj

```
double ABA_LP::obj(int i) const
```

Return Value:

The objective function coefficient of column `i`.

Arguments:

`i`

A column.

lBound

```
double ABA_LP::lBound(int i) const
```

Return Value:

The lower bound of column `i`.

Arguments:

`i`

A column.

uBound

```
double ABA_LP::uBound(int i) const
```

Return Value:

The upper bound of column `i`.

Arguments:

`i`

A column.

row

The function `row()` retrieves a row from the linear program.

```
void ABA_LP::row(int i, ABA_ROW &r) const
```

Arguments:

`i`

The number of the row.

`r`

Stores the row after calling this function.

rhs

```
double ABA_LP::rhs(int i) const
```

Return Value:

The right hand side of the `i`-th row.

Arguments:

`i`

The number of the row.

value (`virtual`)

```
double ABA_LP::value() const
```

Return Value:

The solution of the `ABA_LP` after the optimization has been performed.

xVal (`virtual`)

```
double ABA_LP::xVal(int i)
```

Return Value:

The x-value of column `i` in the solution of the linear program.

Arguments:

`i`

The number of a column.

barXVal (`virtual`)

```
double ABA_LP::barXVal(int i)
```

Return Value:

The x-value of column `i` in the barrier solution before crossing over to a basis solution of the linear program.

Arguments:

`i`

The number of a column.

reco (`virtual`)

```
double ABA_LP::reco(int i)
```

Return Value:

The reduced cost of column `i`.

Arguments:

`i`

The number of a column.

yVal (`virtual`)

```
double ABA_LP::yVal(int c)
```

Return Value:

The dual variable of row `c`.

Arguments:

`c`

The number of a row.

slack (`virtual`)

```
double ABA_LP::slack(int c)
```

Return Value:

The value of the slack variable of row `c`.

Arguments:

`c`

The number of a row.

xValStatus

```
ABA_LP::SOLSTAT ABA_LP::xValStatus() const
```

Return Value:

`Available`

If the x-values can be retrieved,

`Missing`

otherwise.

barXValStatus

```
ABA_LP::SOLSTAT ABA_LP::barXValStatus() const
```

Return Value:

`Available`
: If the x-values of the barrier solution before cross over can be retrieved,

`Missing`
: otherwise.

recoStatus

```
ABA_LP::SOLSTAT ABA_LP::recoStatus() const
```

Return Value:

`Available`
: If the reduced costs can be retrieved,

`Missing`
: otherwise.

yValStatus

```
ABA_LP::SOLSTAT ABA_LP::yValStatus() const
```

Return Value:

`Available`
: If the dual variables can be retrieved,

`Missing`
: otherwise.

slackStatus

```
ABA_LP::SOLSTAT ABA_LP::slackStatus() const
```

Return Value:

`Available`
: If the status of the slack variables can be retrieved,

`Missing`
: otherwise.

basisStatus

```
ABA_LP::SOLSTAT ABA_LP::basisStatus() const
```

Return Value:

`Available`
: If the status of the columns can be retrieved,

`Missing`
: otherwise.

nOpt

```
int ABA_LP::nOpt() const
```

Return Value:

The number of optimizations of the `ABA_LP`.

infeasible (virtual)

```
bool ABA_LP::infeasible() const
```

Return Value:

`true`
: If the optimization status is `Infeasible`,

`false`
: otherwise.

getInfeas (virtual)

The function `getInfeas()` can be called if the last linear program has been solved with the dual simplex method and is infeasible and all inactive variables price out correctly. Then, the basis is dual feasible, but primal infeasible, i.e., some variables or slack variables violate their bounds. In this case the function `getInfeas()` determines an infeasible variable or slack variable.

```
int ABA_LP::getInfeas(int &infeasRow, int &infeasCol, double *bInvRow)
```

Return Value:

`0`
: On success,

`1`
: otherwise.

Arguments:

`infeasRow`
: Holds after the execution the number of an infeasible slack variable, or -1.

`infeasVar`
: Holds after the execution the number of an infeasible column, or -1.

`bInvRow`
: Holds after the execution the row of the basis inverse corresponding to the infeasible column or slack variable, which is always a basic variable.

If `getInfeas()` is successful, then either `infeasRow` or `infeasVar` is -1 and the other argument holds the nonnegative number of the infeasible variable.

lpVarStat (virtual)

```
ABA_LPVARSTAT::STATUS ABA_LP::lpVarStat(int i)
```

Return Value:

The status of column `i` in the linear program.

Arguments:

`i`
: The number of a column.

slackStat (`virtual`)

```
ABA_SLACKSTAT::STATUS ABA_LP::slackStat(int i)
```

Return Value:

The status of the slack variable of row `i`.

Arguments:

`i`

The number of a row.

colsNnz

The function `colsNnz()` computes the number of nonzero elements in each column of a given set of rows.

```
void ABA_LP::colsNnz(int nRow, ABA_ARRAY<ABA_ROW*> &rows, ABA_ARRAY<int> &nnz)
```

Arguments:

`nRow`

The number of rows.

`rows`

The array storing the rows.

`nnz`

An array of length at least the number of columns of the linear program which will hold the number of nonzero elements of each column.

rows2cols

The function `rows2cols()` computes the columnwise representation of the row matrix.

```
void ABA_LP::rows2cols(int nRow,
                       ABA_ARRAY<ABA_ROW*> &rows,
                       ABA_ARRAY<ABA_SPARVEC*> &cols)
```

Arguments:

`nRow`

The number of rows.

`rows`

The array storing the rows.

`cols`

An array holding pointers to sparse vectors which will contain the columnwise representation of the constraint matrix defined by `rows`. The length of this array must be at least the number of columns. The elements of the array must not be 0-pointers. Sparse vectors of sufficient length should be allocated before the function is called. The size of these sparse vectors can be determined with the function `colsNnz()`.

rowRangeCheck

The function `rowRangeCheck()` terminates the program if there is no row with index `r`.

```
void ABA_LP::rowRangeCheck(int r) const
```

Arguments:

`r`
: The number of a row of the linear program.

colRangeCheck

The function `colRangeCheck()` terminates the program if there is no column with index `i`.

```
void ABA_LP::colRangeCheck(int i) const
```

Arguments:

`i`
: The number of a column.

Output Operator

The output operator writes the objective function, followed by the constraints, the bounds on the columns and the solution values (if available) to an output stream.

```
ostream &operator<<(ostream& out, const ABA_LP& rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`
: The output stream.

`rhs`
: The linear program being output.

writeBasisMatrix

The function `writeBasisMatrix()` writes the complete basis of an optimal linear program to a file.

```
int ABA_LP::writeBasisMatrix(const char *fileName)
```

Return Value:

`0`
: If a basis is available and could be written,

`1`
: otherwise.

Arguments:

`fileName`
: The name of the file the basis is written to.

setSimplexIterationLimit

The function `setSimplexIterationLimit()` changes the iteration limit of the Simplex algorithm.

```
int ABA_LP::setSimplexIterationLimit(int limit)
```

Return Value:

`0`
: If the iteration limit could be set,

`1`
: otherwise.

Arguments:

`limit`
: The new value of the iteration limit.

getSimplexIterationLimit

The function `getSimplexIterationLimit()`.

```
int ABA_LP::getSimplexIterationLimit(int &limit)
```

Return Value:

`0`
: If the iteration limit could be get,

`1`
: otherwise.

Arguments:

`limit`
: Stores the iteration limit if the return value is 0.

6.2.8 ABA_CPLEXIF

The class `ABA_CPLEXIF` implements the interface to the LP-solver Cplex and is a derived class from `LP`. `ABA_CPLEXIF` is not an abstract class. Hence, an object of this class can be used for the explicit solution of a linear program.

LP-relaxations within the branch-and-bound algorithms are provided by the class `ABA_LPSUBCPLEX`, which is derived from this class and the class `ABA_LPSUB`.

In the class `ABA_CPLEXIF` we do not follow our naming conventions several times in order to use the same names as in the Cplex manual [Cpl94].

```
class ABA_CPLEXIF : public virtual ABA_LP {
  public:
    ABA_CPLEXIF(ABA_MASTER *master,
                double nnzSurplus = 0.0,
                bool relativeSurplus = true);
    ABA_CPLEXIF(ABA_MASTER *master,
                ABA_OPTSENSE sense,
                int nRow,
                int maxRow,
                int nCol,
```

```
                int maxCol,
                ABA_ARRAY<double> &obj,
                ABA_ARRAY<double> &lb,
                ABA_ARRAY<double> &ub,
                ABA_ARRAY<ABA_ROW*> &rows,
                double nnzSurplus = 0.0,
                bool relativeSurplus = true);
      virtual ~ABA_CPLEXIF();
      void iterationInformation(int level);
      void setppriind(int priind);
      void setdpriind(int priind);
#ifdef ABACUS_LP_CPLEX40
      int CPXgetdblparam(int whichParam, double *value);
      int CPXsetdblparam(int whichParam, double value);
      int CPXgetintparam(int whichParam, int *value);
      int CPXsetintparam(int whichParam, int value);
      struct cpxenv *cplexEnv();
#endif
      struct cpxlp *cplexLp();
      void print();

    private:
      ABA_CPLEXIF(const ABA_CPLEXIF &rhs);
      const ABA_CPLEXIF &operator=(const ABA_CPLEXIF &rhs);
  };
```

Constructor

This constructor does not initialize the problem data of the linear program. It must be loaded later with the function `initialize()`.

```
ABA_CPLEXIF::ABA_CPLEXIF(ABA_MASTER *master,
                         double nnzSurplus,
                         bool relativeSurplus)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`nnzSurplus`

Additional space for nonzero elements in the constraint matrix of Cplex. Its default value is 0.

`relativeSurplus`

If this argument is `true`, then the additional number of nonzeros is relative in percent to the initial number of nonzeros, otherwise it is interpreted as an absolute value. The default value is `true`.

Constructor

A constructor with initialization.

```
ABA_CPLEXIF::ABA_CPLEXIF(ABA_MASTER *master,
                         ABA_OPTSENSE sense,
                         int nRow,
```

```
                        int maxRow,
                        int nCol,
                        int maxCol,
                        ABA_ARRAY<double> &obj,
                        ABA_ARRAY<double> &lb,
                        ABA_ARRAY<double> &ub,
                        ABA_ARRAY<ABA_ROW*> &rows,
                        double nnzSurplus,
                        bool relativeSurplus)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sense`

The sense of the objective function.

`nCol`

The number of columns (variables).

`maxCol`

The maximal number of columns.

`nRow`

The number of rows.

`maxRow`

The maximal number of rows.

`obj`

An array with the objective function coefficients.

`lb`

An array with the lower bounds of the columns.

`ub`

An array with the upper bounds of the columns.

`rows`

An array storing the rows of the problem.

`nnzSurplus`

Additional space for nonzero elements in the constraint matrix of Cplex. Its default value is 0.

`relativeSurplus`

If this argument is `true`, then the additional number of nonzeros is relative in percent to the initial number of nonzeros, otherwise it is interpreted as an absolute value. The default value is `true`.

Destructor (`virtual`)

```
ABA_CPLEXIF::~ABA_CPLEXIF()
```

iterationInformation

The function `iterationInformation()` emulates the Cplex function `setitfoind()` and `setscr_ind(1)`.

Note: The output of Cplex is not compatible with the `ABA_OSTREAM` class of this framework and is always written to `stdout`.

```
void ABA_CPLEXIF::iterationInformation(int level)
```

Arguments:

`level`

If `level` is 0, then no iteration information is output, if `level` is 1 then iteration information is output after every refactorization, and if `level` is 2, then this information is output every iteration.

setppriind

The function `setppriind()` emulates the Cplex function with the same name.

```
void ABA_CPLEXIF::setppriind(int priind)
```

Arguments:

`priind`

The primal pricing strategy. See your Cplex manual for the possible values.

setdpriind

The function `setdpriind()` emulates the Cplex function with the same name.

```
void ABA_CPLEXIF::setdpriind(int priind)
```

Arguments:

`priind`

The dual pricing strategy. See your Cplex manual for the possible values.

CPXgetdblparam

The function `CPXgetdblparam()` emulates the Cplex function with the same name that obtains the current value of a Cplex parameter of type `double`.

This function is only available in Cplex versions 4.0 or newer.

```
int ABA_CPLEXIF::CPXgetdblparam(int whichParam, double *value)
```

Return Value:

If the value of the parameter can be obtained 0 is returned, otherwise a nonzero value.

Arguments:

`whichParam`

The name of the parameter (see your Cplex manual).

`value`

Holds the parameter value in case of a successful call.

CPXsetdblparam

The function `CPXsetdblparam()` emulates the Cplex function with the same name that sets the value of a Cplex parameter of type `double`.

This function is only available in Cplex versions 4.0 or newer.

```
int ABA_CPLEXIF::CPXsetdblparam(int whichParam, double value)
```

Return Value:

If the value of the parameter can be set 0 is returned, otherwise a nonzero value.

Arguments:

`whichParam`

The name of the parameter (see your Cplex manual).

`value`

The new value of the parameter.

CPXgetintparam

The function `CPXgetintparam()` emulates the Cplex function with the same name that obtains the current value of a Cplex parameter of type `int`.

This function is only available in Cplex versions 4.0 or newer.

```
int ABA_CPLEXIF::CPXgetintparam(int whichParam, int *value)
```

Return Value:

If the value of the parameter can be obtained 0 is returned, otherwise a nonzero value.

Arguments:

`whichParam`

The name of the parameter (see your Cplex manual).

`value`

Holds the parameter value in case of a successful call.

CPXsetintparam

The function `CPXsetintparam()` emulates the Cplex function with the same name that sets the value of a Cplex parameter of type `int`.

This function is only available in Cplex versions 4.0 or newer.

```
int ABA_CPLEXIF::CPXsetintparam(int whichParam, int value)
```

Return Value:

If the value of the parameter can be set 0 is returned, otherwise a nonzero value.

Arguments:

`whichParam`

The name of the parameter (see your Cplex manual).

`value`

The new value of the parameter.

cplexLp

The function `cplexLp()` should be used carefully because it breaks the class principle. In particular, it should not be used to to modify the internal data structures of Cplex.

```
struct cpxlp *ABA_CPLEXIF::cplexLp()
```

Return Value:

A pointer to the internal problem representation of Cplex.

cplexEnv

The function `cplexEnv()` is only available in Cplex Version 4.0 and newer. Like the function `cplexLp()` this function should be used very carefully. In particular, it should not be used to to modify the internal data structures of Cplex.

```
struct cpxenv *ABA_CPLEXIF::cplexEnv()
```

Return Value:

A pointer to the cplex environment.

print

The function `print()` writes the program loaded by Cplex to the standard output. This function is especially useful for debugging, but breaks the stream concept of ABACUS.

```
void ABA_CPLEXIF::print()
```

6.2.9 ABA_SOPLEXIF

```
class ABA_SOPLEXIF : public virtual ABA_LP {
  public:
    ABA_SOPLEXIF(ABA_MASTER *master);
    ABA_SOPLEXIF(ABA_MASTER *master,
                 ABA_OPTSENSE sense, int nRow, int maxRow, int nCol, int maxCol,
                 ABA_ARRAY<double> &obj, ABA_ARRAY<double> &lb,
                 ABA_ARRAY<double> &ub, ABA_ARRAY<ABA_ROW*> &rows);
    virtual ~ABA_SOPLEXIF();
    friend ostream &operator<<(ostream& out, const ABA_SOPLEXIF& rhs);

    SoPlex *soplex();

  private:
    ABA_SOPLEXIF(const ABA_SOPLEXIF &rhs);
    const ABA_SOPLEXIF &operator=(const ABA_SOPLEXIF &rhs);
};
```

Constructor

This constructor does not initialize the problem data of the linear program. It must be loaded later with the function `initialize()`.

```
ABA_SOPLEXIF::ABA_SOPLEXIF(ABA_MASTER *master)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

Constructor

A constructor with initialization.

```
ABA_SOPLEXIF::ABA_SOPLEXIF(ABA_MASTER *master,
                           ABA_OPTSENSE sense,
                           int nRow,
                           int maxRow,
                           int nCol,
                           int maxCol,
                           ABA_ARRAY<double> &obj,
                           ABA_ARRAY<double> &lb,
                           ABA_ARRAY<double> &ub,
                           ABA_ARRAY<ABA_ROW*> &rows)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sense`

The sense of the objective function.

`nCol`

The number of columns (variables).

`maxCol`

The maximal number of columns.

`nRow`

The number of rows.

`maxRow`

The maximal number of rows.

`obj`

An array with the objective function coefficients.

`lb`

An array with the lower bounds of the columns.

`ub`

An array with the upper bounds of the columns.

`rows`

An array storing the rows of the problem.

Destructor (`virtual`)

```
ABA_SOPLEXIF::~ABA_SOPLEXIF()
```

Output Operator

The output operator writes the linear program to an output stream in the format of the LP-solver SoPlex.

```
ostream &operator<<(ostream& out, const ABA_SOPLEXIF& rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The linear program being output.

soplex

The function `soplex()` should be used carefully because it breaks the class principle. In particular, it should not be used to to modify the internal data structures of Soplex.

```
SoPlex *ABA_SOPLEXIF::soplex()
```

Return Value:

A pointer to the internal problem representation of Soplex.

6.2.10 ABA_LPSUB

This class is derived from the class `LP` to implement the linear programming relaxations of a subproblem. We require this class as the `ABA_CONSTRAINT/ABA_VARIABLE` format of the constraints/variables has to be transformed to the `ABA_ROW/ABA_COLUMN` format required by the class `LP`. Moreover the class `ABA_LPSUB` is also a preprocessor for the linear programs. Currently we only provide the elimination of (nonbasic) fixed and set variables. Future extensions should be considered.

The class `ABA_LPSUB` is still an abstract class independent of the used LP-solver. Classes for solving LP-relaxation with the LP-solvers Cplex or SoPlex are the classes `ABA_LPSUBCPLEX` or `ABA_LPSUBSOPLEX`, respectively.

```
class ABA_LPSUB : public virtual ABA_LP {
  public:
    ABA_LPSUB (ABA_MASTER *master, ABA_SUB *sub);
    virtual ~ABA_LPSUB();
    int trueNCol() const;
    int trueNnz() const;
    double lBound(int i)  const;
    double uBound(int i)  const;
    virtual double value() const;
    virtual double xVal(int i);
    virtual double barXVal(int i);
    virtual double reco(int i);
    virtual ABA_LPVARSTAT::STATUS lpVarStat(int i);
    virtual int getInfeas(int &infeasCon, int &infeasVar, double *bInvRow);
    virtual bool infeasible() const;
    ABA_BUFFER<ABA_INFEASCON*> *infeasCon();
    virtual void loadBasis(ABA_ARRAY<ABA_LPVARSTAT::STATUS> &lpVarStat,
```

```
                              ABA_ARRAY<ABA_SLACKSTAT::STATUS> &slackStat);
  protected:
    void initialize();

  private:
    ABA_LPSUB(const ABA_LPSUB &rhs);
    const ABA_LPSUB &operator=(const ABA_LPSUB &rhs);
};
```

Constructor

```
ABA_LPSUB::ABA_LPSUB (ABA_MASTER *master, ABA_SUB *sub)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sub`

The subproblem of which the LP-relaxation is solved.

Destructor (`virtual`)

```
ABA_LPSUB::~ABA_LPSUB()
```

initialize

The function `initialize()` has to be called in the constructor of the class derived from this class and from a class implementing an LP-solver. This function will pass the linear program of the associated subproblem to the solver.

```
void ABA_LPSUB::initialize()
```

trueNCol

```
int ABA_LPSUB::trueNCol() const
```

Return Value:

The number of columns which are passed to the LP-solver, i.e., the number of active variables of the subproblem minus the number of eliminated variables.

trueNnz

```
int ABA_LPSUB::trueNnz() const
```

Return Value:

The number of nonzeros which are currently present in the constraint matrix of the LP-solver.

lBound (`virtual`)

```
double ABA_LPSUB::lBound(int i) const
```

Return Value:

The lower bound of variable `i`. If a variable is eliminated, we return the value the eliminated variable is fixed or set to.

Arguments:

`i`

The number of a variable.

uBound (`virtual`)

```
double ABA_LPSUB::uBound(int i) const
```

Return Value:

The upper bound of variable `i`. If a variable is eliminated, we return the value the eliminated variable is fixed or set to.

Arguments:

`i`

The number of a variable.

value (`virtual`)

```
double ABA_LPSUB::value() const
```

Return Value:

The objective function value of the linear program.

xVal (`virtual`)

```
double ABA_LPSUB::xVal(int i)
```

Return Value:

The x-value of variable `i` after the solution of the linear program.

barXVal (`virtual`)

```
double ABA_LPSUB::barXVal(int i)
```

Return Value:

The x-value of variable `i` after the solution of the linear program before crossing over to a basic solution.

reco (`virtual`)

```
double ABA_LPSUB::reco(int i)
```

Return Value:

The reduced cost of variable `i`. We define the reduced costs of eliminated variables as 0.

lpVarStat (virtual)

```
ABA_LPVARSTAT::STATUS ABA_LPSUB::lpVarStat(int i)
```

Return Value:

The status of the variable in the linear program. If the variable `i` is eliminated, then `ABA_LPVARSTAT::Eliminated` is returned.

getInfeas (virtual)

The function `getInfeas()` is called if the last linear program has been solved with the dual simplex method and is infeasible. In this case it computes the infeasible basic variable or constraint and the corresponding row of the basis inverse.

```
int ABA_LPSUB::getInfeas(int &infeasCon, int &infeasVar, double *bInvRow)
```

Return Value:

`0`

If no error occurs,

`1`

otherwise.

Arguments:

`infeasCon`

If nonnegative, this is the number of the infeasible slack variable.

`infeasVar`

If nonnegative, this is the number of the infeasible structural variable. Note, either `infeasCon` or `infeasVar` is nonnegative.

`bInvRow`

An array containing the corresponding row of the basis inverse.

infeasible (virtual)

```
bool ABA_LPSUB::infeasible() const
```

Return Value:

`true`

If the `LP` turned out to be infeasible either if the base class `LP` detected an infeasibility during the solution of the linear program or infeasible constraints have been memorized during the construction of the LP or during the addition of constraints,

`false`

otherwise.

infeasCon

```
ABA_BUFFER<ABA_INFEASCON*> *ABA_LPSUB::infeasCon()
```

Return Value:

A pointer to the buffer holding the infeasible constraints.

loadBasis (`virtual`)

The function `loadBasis()` loads a new basis for the linear program. The function redefines a virtual function of the base class `LP`. Eliminated variables have to be considered when the basis is loaded.

```
void ABA_LPSUB::loadBasis(ABA_ARRAY<ABA_LPVARSTAT::STATUS> &lpVarStat,
                          ABA_ARRAY<ABA_SLACKSTAT::STATUS> &slackStat)
```

Arguments:

`lpVarStat`

An array storing the status of the columns.

`slackStat`

An array storing the status of the slack variables.

6.2.11 ABA_LPSUBCPLEX

This class is derived from the classes `ABA_LPSUB` and `ABA_CPLEX` and implements the solution of the linear programming relaxation of a subproblem using the LP-solver Cplex. The class `LP` is a virtual base class of `ABA_LPSUBCPLEX`.

```
class ABA_LPSUBCPLEX : public ABA_LPSUB, public ABA_CPLEXIF {
  public:
    ABA_LPSUBCPLEX(ABA_MASTER *master, ABA_SUB *sub);
    virtual ~ABA_LPSUBCPLEX();
  private:
    ABA_LPSUBCPLEX(const ABA_LPSUBCPLEX &rhs);
    const ABA_LPSUBCPLEX &operator=(const ABA_LPSUBCPLEX &rhs);
};
```

Constructor

```
ABA_LPSUBCPLEX::ABA_LPSUBCPLEX(ABA_MASTER *master, ABA_SUB *sub)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sub`

The subproblem of which the LP-relaxation is solved.

Destructor (`virtual`)

```
ABA_LPSUBCPLEX::~ABA_LPSUBCPLEX()
```

6.2.12 ABA_LPSUBSOPLEX

This class is derived from the classes `ABA_LPSUB` and `ABA_SOPLEX` and implements the solution of the linear programming relaxation of a subproblem using the LP-solver SoPlex. The class `LP` is a virtual base class of `ABA_LPSUBSOPLEX`.

```
class ABA_LPSUBSOPLEX : public ABA_LPSUB, public ABA_SOPLEXIF {
  public:
    ABA_LPSUBSOPLEX(ABA_MASTER *master, ABA_SUB *sub);
    virtual ~ABA_LPSUBSOPLEX();
  private:
    ABA_LPSUBSOPLEX(const ABA_LPSUBSOPLEX &rhs);
    const ABA_LPSUBSOPLEX &operator=(const ABA_LPSUBSOPLEX &rhs);
};
```

Constructor

```
ABA_LPSUBSOPLEX::ABA_LPSUBSOPLEX(ABA_MASTER *master, ABA_SUB *sub)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sub`

The subproblem of which the LP-relaxation is solved.

Destructor (`virtual`)

```
ABA_LPSUBSOPLEX::~ABA_LPSUBSOPLEX()
```

6.2.13 ABA_BRANCHRULE

Branching should be very flexible within such a framework. Therefore in a branching step each generated subproblem receives an object of the class `ABA_BRANCHRULE`. When the subproblem is activated, it copies the active variables, their bounds and statuses, and the active constraints from the father, and then modifies the subproblem according to the branching rule.

This class is an abstract base class for all branching rules within this framework. We provide by non-abstract derived classes standard branching schemes for setting a binary variable to its lower or upper bound (`ABA_SETBRANCHRULE`), for setting an integer variable to a certain value (`ABA_VALBRANCHRULE`), for changing the bounds of an integer variable (`ABA_CONBRANCHRULE`), and for adding a branching constraint (`ABA_CONBRANCHRULE`).

```
class ABA_BRANCHRULE : public ABA_ABACUSROOT {
  public:
    ABA_BRANCHRULE(ABA_MASTER *master);
    virtual ~ABA_BRANCHRULE();
    virtual int extract(ABA_SUB *sub) = 0;
    virtual void extract(ABA_LPSUB *lp);
    virtual void unExtract(ABA_LPSUB *lp);
    virtual bool branchOnSetVar();
    virtual void initialize(ABA_SUB* sub);
  protected:
    ABA_MASTER *master_;
};
```

master_

```
ABA_MASTER *master_
```

A pointer to the corresponding master of the optimization.

extract (virtual)

The function `extract()` modifies a subproblem by setting the branching variable.

```
virtual int extract(ABA_SUB *sub) = 0
```

Return Value:

`0`

If the subproblem can be modified according to the branching rule.

`1`

If a contradiction occurs.

Arguments:

`sub`

The subproblem being modified.

Constructor

```
ABA_BRANCHRULE::ABA_BRANCHRULE(ABA_MASTER *master)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

Destructor (virtual)

```
ABA_BRANCHRULE::~ABA_BRANCHRULE()
```

branchOnSetVar (virtual)

The virtual function `branchOnSetVar()` should indicate if the branching is performed by setting a binary variable. This is only required as in the current version of the GNU-compiler run time type information is not satisfactorily implemented.

```
bool ABA_BRANCHRULE::branchOnSetVar()
```

Return Value:

The default implementation returns always false. This function must be redefined in the class `ABA_SETBRANCHRULE`, where it has to return `true`.

extract (virtual)

The virtual function `extract()` should modify the linear programming relaxation `lp` in order to determine the quality of the branching rule in a linear programming based branching rule selection. The default implementation does nothing except writing a warning to the error stream. If a derived concrete branching rule should be used in LP-based branching rule selection then this function has to be redefined.

```
void ABA_BRANCHRULE::extract(ABA_LPSUB *lp)
```

Arguments:

`lp`

A pointer to a the linear programming relaxtion of a a subproblem.

unExtract (`virtual`)

The virtual function `unExtract()` should undo the modifictions of the linear programming relaxtion `lp`. This function has to be redefined in a derived class, if also `extract(ABA_LPSUB*)` is redefined there.

```
void ABA_BRANCHRULE::unExtract(ABA_LPSUB *lp)
```

Arguments:

`lp`

A pointer to a the linear programming relaxtion of a a subproblem.

initialize (`virtual`)

The function initialize is a virtual dummy function doing nothing. It is called from the constructor of the subproblem and can be used to perform initializations of the branching rule that can be only done after the generation of the subproblem.

```
void ABA_BRANCHRULE::initialize(ABA_SUB* sub)
```

Arguments:

`sub`

A pointer to the subproblem that should be used for the initialization.

6.2.14 ABA_SETBRANCHRULE

This class implements a branching rule for setting a binary variable to its lower or upper bound.

```
class ABA_SETBRANCHRULE : public ABA_BRANCHRULE {
  public:
    ABA_SETBRANCHRULE(ABA_MASTER *master, int variable,
                      ABA_FSVARSTAT::STATUS status);
    virtual ~ABA_SETBRANCHRULE();
    friend ostream &operator<<(ostream &out, const ABA_SETBRANCHRULE &rhs);
    virtual int extract(ABA_SUB *sub);
    virtual void extract(ABA_LPSUB *lp);
    virtual void unExtract(ABA_LPSUB *lp);
    virtual bool branchOnSetVar();
    bool setToUpperBound() const;
    int variable() const;
};
```

Constructor

```
ABA_SETBRANCHRULE::ABA_SETBRANCHRULE(ABA_MASTER *master,
                                     int variable,
                                     ABA_FSVARSTAT::STATUS status)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`variable`

The branching variable.

`status`

The status the variable is set to (`SetToLowerBound` or `SetToUpperBound`).

Destructor (`virtual`)

```
ABA_SETBRANCHRULE::~ABA_SETBRANCHRULE()
```

Output Operator

The output operator writes the number of the branching variable and its status on an output stream.

```
ostream &operator<<(ostream &out, const ABA_SETBRANCHRULE &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The branching rule being output.

extract (`virtual`)

The function `extract()` modifies a subproblem by setting the branching variable.

```
int ABA_SETBRANCHRULE::extract(ABA_SUB *sub)
```

Return Value:

`0`

If the subproblem can be modified according to the branching rule.

`1`

If a contradiction occurs.

Arguments:

`sub`

The subproblem being modified.

extract (`virtual`)

The function `extract()` is overloaded to modify directly the linear programming relaxation. This required to evaluate the quality of a branching rule with linear programming methods. The changes have to be undone with the function `unextract()` before the next linear program is solved.

```
void ABA_SETBRANCHRULE::extract(ABA_LPSUB *lp)
```

Arguments:

`lp`

A pointer to the linear programming relaxation of a subproblem.

branchOnSetVar (`virtual`)

The function `branchOnSetVar()` redefines the virtual function of the base class `ABA_BRANCHRULE` as this branching rule is setting a binary variable.

```
bool ABA_SETBRANCHRULE::branchOnSetVar()
```

Return Value:

Always `true`.

setToUpperBound

```
bool ABA_SETBRANCHRULE::setToUpperBound() const
```

Return Value:

`true`
: If the branching variable is set to the upper bound,

`false`
: otherwise.

variable

```
int ABA_SETBRANCHRULE::variable() const
```

Return Value:

The number of the branching variable.

6.2.15 ABA_BOUNDBRANCHRULE

This class implements a branching rule for modifying the lower and the upper bound of a variable.

```
class ABA_BOUNDBRANCHRULE : public ABA_BRANCHRULE {
  public:
    ABA_BOUNDBRANCHRULE(ABA_MASTER *master, int variable, double lBound,
                        double uBound);
    virtual ~ABA_BOUNDBRANCHRULE();
    friend ostream &operator<<(ostream &out, const ABA_BOUNDBRANCHRULE &rhs);
    virtual int extract(ABA_SUB *sub);
    virtual void extract(ABA_LPSUB *lp);
    virtual void unExtract(ABA_LPSUB *lp);
    int variable() const;
    double lBound() const;
    double uBound() const;
};
```

Constructor

```
ABA_BOUNDBRANCHRULE::ABA_BOUNDBRANCHRULE(ABA_MASTER *master,
                                         int variable,
                                         double lBound,
                                         double uBound)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`variable`

The branching variable.

`lBound`

The lower bound of the branching variable.

`uBound`

The upper bound of the branching variable.

Destructor (`virtual`)

```
ABA_BOUNDBRANCHRULE::~ABA_BOUNDBRANCHRULE()
```

Output Operator

The output operator writes the branching variable together with its lower and upper bound to an output stream.

```
ostream &operator<<(ostream &out, const ABA_BOUNDBRANCHRULE &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The branch rule being output.

extract (`virtual`)

The function `extract()` modifies a subproblem by changing the lower and the upper bound of the branching variable.

```
int ABA_BOUNDBRANCHRULE::extract(ABA_SUB *sub)
```

Return Value:

`0`

If the subproblem is successfully modified.

`1`

If the modification causes a contradiction.

Arguments:

`sub`

The subproblem being modified.

variable

```
int ABA_BOUNDBRANCHRULE::variable() const
```

Return Value:

The number of the branching variable.

lBound

```
double ABA_BOUNDBRANCHRULE::lBound() const
```

Return Value:

The lower bound of the branching variable.

uBound

```
double ABA_BOUNDBRANCHRULE::uBound() const
```

Return Value:

The upper bound of the branching variable.

6.2.16 ABA_VALBRANCHRULE

This class implements a branching rule for setting a variable to a certain value.

```
class ABA_VALBRANCHRULE : public ABA_BRANCHRULE {
  public:
    ABA_VALBRANCHRULE(ABA_MASTER *master, int variable, double value);
    virtual ~ABA_VALBRANCHRULE();
    friend ostream &operator<<(ostream &out, const ABA_VALBRANCHRULE &rhs);
    virtual int extract(ABA_SUB *sub);
    virtual void extract(ABA_LPSUB *lp);
    virtual void unExtract(ABA_LPSUB *lp);
    int variable() const;
    double value() const;
};
```

Constructor

```
ABA_VALBRANCHRULE::ABA_VALBRANCHRULE(ABA_MASTER *master, int variable, double value)
```

Arguments:

`master`

The corresponding master of the optimization.

`variable`

The branching variable.

`value`

The value the branching variable is set to.

Destructor (`virtual`)

```
ABA_VALBRANCHRULE::~ABA_VALBRANCHRULE()
```

Output Operator

The output operator writes the branching variable together with its value to an output stream.

```
ostream &operator<<(ostream &out, const ABA_VALBRANCHRULE &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The branching rule being output.

extract (`virtual`)

The function `extract()` modifies a subproblem by setting the branching variable.

```
int ABA_VALBRANCHRULE::extract(ABA_SUB *sub)
```

Return Value:

`0`

If the subproblem can be modified according to the branching rule.

`1`

If a contradiction occurs.

Arguments:

`sub`

The subproblem being modified.

variable

```
int ABA_VALBRANCHRULE::variable() const
```

Return Value:

The number of the branching variable.

value

```
double ABA_VALBRANCHRULE::value() const
```

Return Value:

The value of the branching variable.

6.2.17 ABA_CONBRANCHRULE

This class implements the branching by adding a constraint to the set of active constraints.

```
class ABA_CONBRANCHRULE : public ABA_BRANCHRULE {
  public:
    ABA_CONBRANCHRULE(ABA_MASTER *master,
                      ABA_POOLSLOT<ABA_CONSTRAINT, ABA_VARIABLE> *poolSlot);
    virtual ~ABA_CONBRANCHRULE();
    friend ostream &operator<<(ostream &out, const ABA_CONBRANCHRULE &rhs);
    virtual int extract(ABA_SUB *sub);
    virtual void extract(ABA_LPSUB *lp);
    virtual void unExtract(ABA_LPSUB *lp);
    virtual void initialize(ABA_SUB *sub);
    ABA_CONSTRAINT *constraint();
  private:
    const ABA_CONBRANCHRULE &operator=(const ABA_CONBRANCHRULE &rhs);
};
```

Constructor

Note, the subproblem associated with the branching constraint will be modified in the constructor of the subproblem generated with this branching rule such that later the check for local validity of the branching constraint is performed correcly.

```
ABA_CONBRANCHRULE::ABA_CONBRANCHRULE(
                       ABA_MASTER *master,
                       ABA_POOLSLOT<ABA_CONSTRAINT, ABA_VARIABLE> *poolSlot)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`poolSlot`

A pointer to the pool slot of the branching constraint.

Destructor (`virtual`)

```
ABA_CONBRANCHRULE::~ABA_CONBRANCHRULE()
```

Output Operator

The output operator writes the branching constraint on an output stream.

```
ostream &operator<<(ostream &out, const ABA_CONBRANCHRULE &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The branch rule being output.

extract (`virtual`)

The function `extract()` adds the branching constraint to the subproblem.

```
int ABA_CONBRANCHRULE::extract(ABA_SUB *sub)
```

Return Value:

Always 0, since there cannot be a contractiction.

Arguments:

`sub`

The subproblem being modified.

initialize (`virtual`)

The function initialize redefines the virtual function of the base class `ABA_BRANCHRULE` in order to initialize the subproblem associated with the branching constraint.

```
void ABA_CONBRANCHRULE::initialize(ABA_SUB* sub)
```

Arguments:

`sub`

A pointer to the subproblem that is associated with the branching constraint.

constraint (`virtual`)

```
ABA_CONSTRAINT *ABA_CONBRANCHRULE::constraint()
```

Return Value:

A pointer to the branching constraint or a 0-pointer, if this constraint is not available.

6.2.18 ABA_POOL

Every constraint and variable has to to be stored in a pool. This class implements an abstract template class for a pool, which can be used to store objects of the class `ABA_VARIABLE` or of the class `ABA_CONSTRAINT`. A constraint or variable is not directly stored in the pool, but in an `ABA_POOLSLOT`. Hence, a pool is a collection of pool slots.

A pool has two template arguments: the `BaseType` and the `CoType`. Only two scenarios make sense in the current context. For a pool storing constraints the `BaseType` is `ABA_CONSTRAINT` and the `CoType` is `ABA_VARIABLE`. For a pool storing variables the `BaseType` is `ABA_VARIABLE` and the corresponding `CoType` is `ABA_CONSTRAINT`.

The class `ABA_POOL` is an abstract class from which concrete classes have to be derived, implementing the data structures for the storage of pool slots. We provide already in the class `ABA_STANDARDPOOL` a simple but convenient implementation of a pool. We refer to all constraints and variables via the class `ABA_POOLSLOTREF`.

```
template<class BaseType, class CoType>
class ABA_POOL : public ABA_ABACUSROOT {
  public:
    enum RANKING {NO_RANK, RANK, ABS_RANK};

    ABA_POOL(ABA_MASTER *master);
    virtual ~ABA_POOL();
    virtual ABA_POOLSLOT<BaseType, CoType> *insert(BaseType *cv) = 0;
```

```
      void removeConVar(ABA_POOLSLOT<BaseType, CoType> *slot);
      int number() const;
      virtual int separate(double *z,
                           ABA_ACTIVE<CoType, BaseType> *active,
                           ABA_SUB *sub,
                           ABA_CUTBUFFER<BaseType, CoType> *cutBuffer,
                           double minAbsViolation = 0.001,
                           int ranking = 0) = 0;

   protected:
      virtual int softDeleteConVar(ABA_POOLSLOT<BaseType, CoType> *slot);
      virtual void hardDeleteConVar(ABA_POOLSLOT<BaseType, CoType> *slot);
      virtual ABA_POOLSLOT<BaseType, CoType> *getSlot() = 0;
      virtual void putSlot(ABA_POOLSLOT<BaseType, CoType> *slot) = 0;
      ABA_MASTER *master_;
      int number_;
  };
```

enum RANKING

The enumeration `RANKING` indicates how the rank of a constraint/variable in a pool separation is determined.

`NO_RANK`

No rank is computed.

`RANK`

The violation computed by the function `violated()` of the classes `ABA_CONSTRAINT` or `ABA_VARIABLE` is used as rank.

`ABS_RANK`

The absolute value of the violation is taken as rank.

master_

`ABA_MASTER *master_`

A pointer to the corresponding master of the optimization.

number_

`int number_`

The current number of constraints in the pool.

separate (virtual)

The function `separate()` checks if a pair of a vector and an active constraint/variable set violates any item in the pool. If the pool is a constraint pool, then the vector is an LP-solution and the active set is the set of active variables. Otherwise, if the pool is a variable pool, then the vector contains the dual variables and the active set is the set of associated active constraints.

```
virtual int separate(double *z, ABA_ACTIVE<CoType, BaseType> *active,
                     ABA_SUB *sub,
                     ABA_CUTBUFFER<BaseType, CoType> *cutBuffer,
                     double minAbsViolation = 0.001,
                     RANKING ranking = NO_RANK) = 0
```

Return Value:

The number of violated items.

Arguments:

`z`

The vector for which violation is checked.

`active`

The constraint/variable set associated with `z`.

`sub`

The subproblem for which validity of the violated item is required.

`cutBuffer`

The violated constraints/variables are added to this buffer.

`minAbsViolation`

A violated constraint/variable is only added to the `cutBuffer` if the absolute value of its violation is at least `minAbsViolation`. The default value is `0.001`.

`ranking`

If 1, the violation is associated with a rank of item in the buffer, if 2 the absolute violation is used, if 0 no rank is associated with the item.

insert (`virtual`)

The function `insert()` tries to insert a constraint/variable in the pool.

```
virtual ABA_POOLSLOT<BaseType, CoType> *insert(BaseType *cv) = 0
```

Return Value:

A pointer to the pool slot where the item has been inserted, or 0 if the insertion failed.

Arguments:

`cv`

The constraint/variable being inserted.

getSlot (`virtual`)

The function `getSlot()` tries to find a free slot in the pool. This function is protected since it should be only used by `insert()`. The data structure managing the free poolslots can be individually defined for each derived pool class.

```
virtual ABA_POOLSLOT* getSlot() = 0
```

Return Value:

A pointer to a free `ABA_POOLSLOT` where a constraint/variable can be inserted. If no pool slot is available `getSlot()` returns 0.

putSlot (virtual)

The virtual function putSlot() makes an ABA_POOLSLOT again available for later calls of getSlot(). If somebody else refers to this constraint the function should abort with an error message.

```
virtual void putSlot(ABA_POOLSLOT *slot) = 0
```

Arguments:

slot
: The slot made available for further use.

Constructor

The constructor initializes an empty pool.

```
ABA_POOL<BaseType, CoType>::ABA_POOL(ABA_MASTER *master)
```

Arguments:

master
: A pointer to the corresponding master of the optimization.

Destructor (virtual)

```
ABA_POOL<BaseType, CoType>::~ABA_POOL()
```

removeConVar

The function removeConVar() removes the constraint/variable stored in a pool slot and adds the slot to the list of free slots.

```
void ABA_POOL<BaseType, CoType>::removeConVar(ABA_POOLSLOT<BaseType, CoType> *slot)
```

Arguments:

slot
: The pool slot from which the constraint/variable is removed.

softDeleteConVar

The function softDeleteConVar() removes the constraint/variable stored in the pool slot slot from the pool if the constraint/variable can be deleted. If the constraint/variable can be removed the slot is added to the set of free slots.

```
int ABA_POOL<BaseType, CoType>::softDeleteConVar(
                                        ABA_POOLSLOT<BaseType, CoType> *slot)
```

Return Value:

0
: If the constraint/variable could be deleted.

1
: otherwise.

Arguments:

slot
: A pointer to the pool slot from wich the constraint/variable should be deleted.

hardDeleteConVar

The function `hardDeleteConVar()` removes a constraint/variable from the pool and adds the slot to the set of free slots.

```
void ABA_POOL<BaseType, CoType>::hardDeleteConVar(ABA_POOLSLOT<BaseType, CoType>
                                                  *slot)
```

Arguments:

`slot`

A pointer to the pool slot from wich the constraint/variable should be deleted.

number

```
int ABA_POOL<BaseType, CoType>::number() const
```

Return Value:

The current number of items in the pool.

6.2.19 ABA_STANDARDPOOL

This class is derived from the class `ABA_POOL` and provides a very simple implementation of a pool which is sufficient for a large class of applications. The pool slots are stored in an array and the set of free slots is managed by a linear list.

A standard pool can be static or dynamic. A static standard pool has a fixed size, whereas a dynamic standard pool is automatically enlarged by ten percent if it is full and an item is inserted.

```
template<class BaseType, class CoType>
class ABA_STANDARDPOOL : public ABA_POOL<BaseType, CoType> {
  public:
    ABA_STANDARDPOOL(ABA_MASTER *master, int size, bool autoRealloc = false);
    virtual ~ABA_STANDARDPOOL();
    friend ostream &operator<<(ostream &out, const ABA_STANDARDPOOL &rhs);
    virtual ABA_POOLSLOT<BaseType, CoType> *insert(BaseType *cv);
    virtual void increase(int size);
    int cleanup();
    int size() const;
    ABA_POOLSLOT<BaseType, CoType> *slot(int i);
    virtual int separate(double *x,
                         ABA_ACTIVE<CoType, BaseType> *active,
                         ABA_SUB *sub,
                         ABA_CUTBUFFER<BaseType, CoType> *cutBuffer,
                         double minAbsViolation = 0.001,
                         int ranking = 0);
  protected:
    ABA_STANDARDPOOL(const ABA_STANDARDPOOL &rhs);
    const ABA_STANDARDPOOL &operator=(const ABA_STANDARDPOOL &rhs);
};
```

Constructor

The constructor for an empty pool.

```
ABA_STANDARDPOOL<BaseType, CoType>::ABA_STANDARDPOOL(ABA_MASTER *master,
                                                     int size,
                                                     bool autoRealloc)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`size`

The maximal number of items which can be inserted in the pool without reallocation.

`autoRealloc`

If this argument is `true` an automatic reallocation is performed if the pool is full.

Destructor (`virtual`)

The destructor deletes all slots. The destructor of a pool slot deletes then also the respective constraint or variable.

```
ABA_STANDARDPOOL<BaseType, CoType>::~ABA_STANDARDPOOL()
```

Output Operator

The output operator calls the output operator of each item of a non-void pool slot.

```
ostream &operator<<(ostream &out, const ABA_STANDARDPOOL<BaseType, CoType> &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The pool being output.

insert (`virtual`)

The function `insert()` tries to insert a constraint/variable in the pool. If there is no free slot available, we try to generate free slots by removing redundant items, i.e., items which have no reference to them. If this fails, we either perform an automatic reallocation of the pool or remove non-active items.

```
ABA_POOLSLOT<BaseType, CoType> * ABA_STANDARDPOOL<BaseType, CoType>::insert(
                                                          BaseType *cv)
```

Return Value:

A pointer to the pool slot where the item has been inserted, or 0 if the insertion failed.

Arguments:

`cv`

The constraint/variable being inserted.

increase (`virtual`)

The function `increase()` enlarges the pool to store. To avoid fatal errors we do not allow decreasing the size of the pool.

```
void ABA_STANDARDPOOL<BaseType, CoType>::increase(int size)
```

Arguments:

`size`

The new size of the pool.

cleanup

The function `cleanup()` scans the pool, removes all deletable items, i.e., those items without having references, and adds the corresponding slots to the list of free slots.

```
int ABA_STANDARDPOOL<BaseType, CoType>::cleanup()
```

Return Value:

The number of "cleaned" slots.

size

```
int ABA_STANDARDPOOL<BaseType, CoType>::size() const
```

Return Value:

The maximal number of constraints/variables that can be inserted in the pool.

slot

```
ABA_POOLSLOT<BaseType, CoType> *ABA_STANDARDPOOL<BaseType, CoType>::slot(int i)
```

Return Value:

A pointer to the `i`-th slot in the pool.

Arguments:

`i`

The number of the slot being accessed.

separate (`virtual`)

The function `separate()` checks if a pair of a vector and an active constraint/variable set violates any item in the pool. If the pool is a constraint pool, then the vector is an LP-solution and the active set the set of active variables. Otherwise, if the pool is a variable pool, then the vector stores the values of the dual variables and the active set the associated active constraints.

Before a constraint or variable is generated we check if it is valid for the subproblem `sub`.

The function defines the pure virtual function of the base class `ABA_POOL`.

This is a very simple version of the pool separation. Future versions might scan a priority queue of the available constraints until a limited number of constraints is tested or separated.

```
int ABA_STANDARDPOOL<BaseType, CoType>::separate(
                                        double *z,
                                        ABA_ACTIVE<CoType, BaseType> *active,
                                        ABA_SUB *sub,
                                        ABA_CUTBUFFER<BaseType, CoType> *cutBuffer,
                                        double minAbsViolation,
                                        int ranking)
```

Return Value:

The number of violated items.

Arguments:

`z`

The vector for which violation is checked.

`active`

The constraint/variable set associated with `z`.

`sub`

The subproblem for which validity of the violated item is required.

`cutBuffer`

The violated constraints/variables are added to this buffer.

`minAbsViolation`

A violated constraint/variable is only added to the `cutBuffer` if the absolute value of its violation is at least `minAbsViolation`. The default value is `0.001`.

`ranking`

If 1, the violation is associated with a rank of item in the buffer, if 2 the absolute violation is used, if 0 no rank is associated with the item.

6.2.20 ABA_NONDUPLPOOL

The class `ABA_NONDUPLPOOL` provides an `ABA_STANDARDPOOL` with the additional feature that the same constraint is at most stored once in the pool. For constraints and variables inserted in this pool the virtual member functions `name()`, `hashKey()`, and `equal()` of the base class `ABA_CONVAR` have to be defined. Using these three functions, we check at insertation time if a constraint or variable is already stored in the pool.

The implementation is unsafe in the sense that the data structure for registering a constraint is corrupted if a constraint is removed directly from the pool slot without using a function of this class.

```
template<class BaseType, class CoType>
class ABA_NONDUPLPOOL : public ABA_STANDARDPOOL<BaseType, CoType> {
  public:
    ABA_NONDUPLPOOL(ABA_MASTER *master, int size, bool autoRealloc = false);
    virtual ~ABA_NONDUPLPOOL();
    virtual ABA_POOLSLOT<BaseType, CoType> *insert(BaseType *cv);
    virtual ABA_POOLSLOT<BaseType, CoType> *present(BaseType *cv);
    virtual void increase(int size);
    void statistics(int &nDuplications, int &nCollisions) const;
    ABA_NONDUPLPOOL(const ABA_NONDUPLPOOL &rhs);
    const ABA_NONDUPLPOOL &operator=(const ABA_NONDUPLPOOL &rhs);
};
```

Constructor

The constructor for an empty pool.

```
ABA_NONDUPLPOOL<BaseType, CoType>::ABA_NONDUPLPOOL(ABA_MASTER *master,
                                                   int size,
                                                   bool autoRealloc)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`size`

The maximal number of items which can be inserted in the pool without reallocation.

`autoRealloc`

If this argument is `true` an automatic reallocation is performed if the pool is full.

Destructor (`virtual`)

```
ABA_NONDUPLPOOL<BaseType, CoType>::~ABA_NONDUPLPOOL()
```

insert (`virtual`)

Before the function `insert()` tries to insert a constraint/variable in the pool, it checks if the constraint/variable is already contained in the pool. If the constraint/variable `cv` is contained in the pool, it is deleted.

```
ABA_POOLSLOT<BaseType, CoType> * ABA_NONDUPLPOOL<BaseType, CoType>::insert(
                                                              BaseType *cv)
```

Return Value:

A pointer to the pool slot where the item has been inserted, or a pointer to the pool slot if the item is already contained in the pool, or 0 if the insertion failed.

Arguments:

`cv`

The constraint/variable being inserted.

present

The function `present()` checks if a constraint/variables is already contained in the pool.

```
template<class BaseType, class CoType>
ABA_POOLSLOT<BaseType, CoType> *ABA_NONDUPLPOOL<BaseType, CoType>::present(
                                                                 BaseType *cv)
```

Return Value:

A pointer to the pool slot storing a constraint/variable that is equivalent to `cv` according to the function `ABA_CONVAR::equal()`. If there is no such constraint/variable 0 is returned.

Arguments:

`cv`

A pointer to a constraint/variable for which it should be checked if an equivalent item is already contained in the pool.

increase (virtual)

The function `increase()` enlarges the pool to store. To avoid fatal errors we do not allow decreasing the size of the pool.

```
void ABA_NONDUPLPOOL<BaseType, CoType>::increase(int size)
```

Arguments:

`size`

The new size of the pool.

statistics

The function `statistics()` determines the number of constraints that have not been inserted into the pool, because an equivalent was already present. Also the number of collisions in the hash table is computed. If this number is high, it might indicate that your hash function is not chosen very well.

```
void ABA_NONDUPLPOOL<BaseType, CoType>::statistics(int &nDuplications,
                                                   int &nCollisions) const
```

Arguments:

`nDuplications`

The number of constraints that have not been inserted into the pool because an equivalent one was already present.

`nCollisions`

The number of collisions in the hash table.

6.2.21 ABA_POOLSLOT

Constraints or variables are not directly stored in a pool. But are stored in a pool slot, which form again the basic building blocks of a pool. The reason is that in order to save memory it can be necessary that a constraint or variable in the pool has to be deleted although it is still contained in the active formulation of an inactive subproblem. Of course this deletion can be only done with constraints/variables which can be regenerated or which are not required for the correctness of the algorithm (e.g., for a cutting plane, but not for a variable or constraint of the problem formulation of a general mixed integer optimization problem).

Such that the deletion of a variable or constraint cannot cause a run-time error, we store it in a pool slot. Together with the pointer to the constraint/variable we store also its version number. The version number is initially zero and incremented each time when a new item is inserted in the pool slot. When we refer to a constraint/variable, e.g., from the sets of active constraints or variables, then we point to the slot and memorize the version number (class `ABA_POOLSLOTREF`), when this reference has been set up. Later by comparing the version number of `ABA_POOLSLOTREF` and the one of the corresponding `ABA_POOLSLOT` we can check if still the constraint/variable is contained in the slot which is supposed to be there. Usually, if the expected constraint/variable is missing, it is ignored.

WARNING: A `ABA_POOLSLOT` must not be deleted before the termination of the optimization process, except that it can be guaranteed that there is no reference to this slot from any other place of the program.

```
template<class BaseType, class CoType> class ABA_POOLSLOT : public ABA_ABACUSROOT {
  public:
    ABA_POOLSLOT(ABA_MASTER *master,
                 ABA_POOL<BaseType, CoType> *pool,
                 BaseType *convar = 0);
```

```
    ~ABA_POOLSLOT();
    BaseType *conVar() const;

  private:
    ABA_POOLSLOT(const ABA_POOLSLOT<BaseType, CoType> &rhs);
    const ABA_POOLSLOT<BaseType, CoType>
          &operator=(const ABA_POOLSLOT<BaseType, CoType> &rhs);
};
```

Constructor

```
ABA_POOLSLOT<BaseType, CoType>::ABA_POOLSLOT(ABA_MASTER *master,
                                             ABA_POOL<BaseType, CoType> *pool,
                                             BaseType *conVar)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`pool`

The pool this slot belongs to.

`conVar`

The constraint/variable inserted in this slot if not 0. The default value is 0.

Destructor

The destructor for the poolslot must not be called if there are references to its constraint/variable.

```
ABA_POOLSLOT<BaseType, CoType>::~ABA_POOLSLOT()
```

conVar

```
BaseType * ABA_POOLSLOT<BaseType, CoType>::conVar() const
```

Return Value:

A pointer to the constraint/variable in the pool slot.

6.2.22 ABA_POOLSLOTREF

As already explained in the class `ABA_POOLSLOT` we do not refer directly to constraints/variables but store a pointer to a pool slot and memorize the version number of the slot at initialization time of the class `ABA_POOLSLOTREF`.

```
template<class BaseType, class CoType>
class ABA_POOLSLOTREF : public ABA_ABACUSROOT {
  public:
    ABA_POOLSLOTREF(ABA_MASTER *master);
    ABA_POOLSLOTREF(ABA_POOLSLOT<BaseType, CoType> *slot);
    ABA_POOLSLOTREF(const ABA_POOLSLOTREF<BaseType, CoType> &rhs);
    ~ABA_POOLSLOTREF();
    friend ostream &operator<<(ostream &out, const ABA_POOLSLOTREF &rhs);
    BaseType *conVar() const;
    unsigned long version() const;
```

```
    ABA_POOLSLOT<BaseType, CoType> *slot() const;
    void slot(ABA_POOLSLOT<BaseType, CoType> *s);

  private:
    const ABA_POOLSLOTREF<BaseType, CoType>
                    &operator=(const ABA_POOLSLOTREF<BaseType, CoType> &rhs);
};
```

Constructor

This constructor generates an object referencing to no pool slot.

```
ABA_POOLSLOTREF<BaseType, CoType>::ABA_POOLSLOTREF(ABA_MASTER *master)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

Constructor

This constructor initializes the reference to a pool slot with a given slot. Also the constraint/variable contained in this slot receives a message that a new references to it is created.

```
ABA_POOLSLOTREF<BaseType, CoType>::ABA_POOLSLOTREF(
                                        ABA_POOLSLOT<BaseType, CoType> *slot)
```

Arguments:

`slot`

The pool slot that is referenced now.

Copy Constructor

```
ABA_POOLSLOTREF<BaseType, CoType>::ABA_POOLSLOTREF(const ABA_POOLSLOTREF &rhs)
```

Arguments:

`rhs`

The pool slot that is copied in the initialization process.

Destructor

```
ABA_POOLSLOTREF<BaseType, CoType>::~ABA_POOLSLOTREF()
```

Output Operator

The output operator writes the constraint/variable stored in the referenced slot to an output stream.

```
ostream &operator<<(ostream &out, const ABA_POOLSLOTREF<BaseType, CoType> &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The reference to a pool slot being output.

conVar

```
BaseType* ABA_POOLSLOTREF<BaseType, CoType>::conVar() const
```

Return Value:

A pointer to the constraint/variable stored in the referenced slot if the version number of the slot is equal to the version number at construction/initialization time of this slot. Otherwise, it returns 0.

version

```
unsigned long ABA_POOLSLOTREF<BaseType, CoType>::version() const
```

Return Value:

The version number of the constraint/variable stored in the referenced slot at construction time of the reference to this slot.

slot

```
ABA_POOLSLOT<BaseType, CoType>* ABA_POOLSLOTREF<BaseType, CoType>::slot() const
```

Return Value:

A pointer to the referenced slot.

slot

This version of the function `slot()` initializes the referenced pool slot.

```
void ABA_POOLSLOTREF<BaseType, CoType>::slot(ABA_POOLSLOT<BaseType, CoType> *s)
```

Arguments:

`s`

The new slot that is referenced. This must not be a 0-pointer.

6.2.23 ABA_ROW

This class refines its base class `ABA_SPARVEC` for the representation of constraints in the row format. This class plays an essential role in the interface with the LP-solver.

This class should not be confused with the class `ABA_CONSTRAINT`, which is an abstract class for the representation of constraints within the framework. Moreover, the class `ABA_ROWCON` derived from the class `ABA_CONSTRAINT` provides a constraint representation in row format, but there are also other representations of constraints.

```
class ABA_ROW : public ABA_SPARVEC {
  public:
    ABA_ROW(ABA_GLOBAL *glob,
            int nnz,
            const ABA_ARRAY<int> &s,
            const ABA_ARRAY<double> &c,
            const ABA_CSENSE sense, double r);
    ABA_ROW(ABA_GLOBAL *glob,
            int nnz,
            const ABA_ARRAY<int> &s,
            const ABA_ARRAY<double> &c,
```

```
            const ABA_CSENSE::SENSE sense, double r);
    ABA_ROW(ABA_GLOBAL *glob,
            int nnz,
            int *s,
            double *c,
            ABA_CSENSE::SENSE sense,
            double r);
    ABA_ROW(ABA_GLOBAL *glob, int size);
    ~ABA_ROW();
    friend ostream &operator<<(ostream& out, const ABA_ROW &rhs);
    double rhs() const;
    void rhs(double r);
    ABA_CSENSE *sense();
    void sense(ABA_CSENSE &s);
    void sense(ABA_CSENSE::SENSE s);
    void copy(const ABA_ROW &row);
    void delInd(ABA_BUFFER<int> &buf, double rhsDelta);

  protected:
    ABA_CSENSE sense_;
    double rhs_;
};
```

sense_

`ABA_CSENSE sense_`

The sense of the row.

rhs_

`double rhs_`

The right hand side of the row.

Constructor

```
ABA_ROW::ABA_ROW(ABA_GLOBAL *glob,
                 int nnz,
                 const ABA_ARRAY<int> &s,
                 const ABA_ARRAY<double> &c,
                 const ABA_CSENSE sense,
                 double r)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`nnz`

The number of nonzero elements of the row.

`s`

The array storing the nonzero elements.

`c`

The array storing the nonzero coefficients of the elements of `s`.

`sense`

The sense of the row.

`r`

The right hand side of the row.

Constructor

This is an equivalent constructor using `ABA_CSENSE::SENSE` instead of an object of the class `SENSE` to initialize the sense of the constraint.

```
ABA_ROW::ABA_ROW(ABA_GLOBAL *glob,
                 int nnz,
                 const ABA_ARRAY<int> &s,
                 const ABA_ARRAY<double> &c,
                 const ABA_CSENSE::SENSE sense,
                 double r)
```

Constructor

This is also an equivalent constructor except that `s` and `c` are C-style arrays.

```
ABA_ROW::ABA_ROW(ABA_GLOBAL *glob, int nnz,
                 int *s, double *c,
                 ABA_CSENSE::SENSE sense, double r)
```

Constructor

A constructor without initialization of the nonzeros of the row.

```
ABA_ROW::ABA_ROW(ABA_GLOBAL *glob, int size)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`size`

The maximal numbers of nonzeros.

Destructor (`virtual`)

```
ABA_ROW::~ABA_ROW()
```

Output Operator

The output operator writes the row on an output stream in format like `-2.5 x1 + 3 x3 <= 7`. Only variables with nonzero coefficients are output. The output operator does neither output a `'+'` before the first coefficient of a row, if it is positive, nor outputs coefficients with absolute value 1.

```
ostream &operator<<(ostream& out, const ABA_ROW &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The row being output.

rhs

```
double ABA_ROW::rhs() const
```

Return Value:

The right hand side stored in the row format.

rhs

This version of `rhs()` sets the right hand side of the row.

```
void  ABA_ROW::rhs(double r)
```

Arguments:

`r`

The new value of the right hand side.

sense

```
ABA_CSENSE *ABA_ROW::sense()
```

Return Value:

A pointer to the sense of the row.

sense

This version of `sense()` sets the sense of the row.

```
void ABA_ROW::sense(ABA_CSENSE &s)
```

Arguments:

`s`

The new sense of the row.

sense

And another version of `sense()` to set the sense of the row.

```
void ABA_ROW::sense(ABA_CSENSE::SENSE s)
```

Arguments:

`s`

The new sense of the row.

copy

The function `copy()` behaves like an assignment operator, however, the maximal number of the elements of this row only has to be at least the number of nonzeros of `row`.

```
void ABA_ROW::copy(const ABA_ROW &row)
```

Arguments:

`row`

The row that is copied.

delInd

The function `delInd()` removes the indices listed in `buf` from the support of the row and subtracts `rhsDelta` from its right hand side.

```
void ABA_ROW::delInd(ABA_BUFFER<int> &buf, double rhsDelta)
```

Arguments:

`buf`

The components being removed from the row.

`rhsDelta`

The correction of the right hand side of the row.

6.2.24 ABA_COLUMN

In the same way as the class `ABA_ROW` refines the class `ABA_SPARVEC` for the representation of constraints in row format, the class `ABA_COLUMN` refines `ABA_SPARVEC` for the representation of variables in column format. This class should not be confused with the class `ABA_VARIABLE` for the abstract representation of variables within the framework. Again, there is a class `ABA_COLVAR` derived from `ABA_VARIABLE` having a member of type `ABA_COLUMN`, but there are also other classes derived from `ABA_VARIABLE`.

```
class ABA_COLUMN : public ABA_SPARVEC {
  public:
    ABA_COLUMN(ABA_GLOBAL *glob,
               double obj,
               double lb,
               double ub,
               int nnz,
               ABA_ARRAY<int> &s,
               ABA_ARRAY<double> &c);
    ABA_COLUMN(ABA_GLOBAL *glob, int maxNnz);
    ABA_COLUMN(ABA_GLOBAL *glob,
               double obj,
               double lb,
               double ub,
               ABA_SPARVEC &vec);
    ~ABA_COLUMN();
    friend ostream& operator<<(ostream &out, const ABA_COLUMN &rhs);
    double obj() const;
    void obj(double c);
    double lBound()  const;
    void lBound(double l);
```

```
    double uBound()  const;
    void uBound(double u);
    void copy(const ABA_COLUMN &col);
};
```

Constructor

```
ABA_COLUMN::ABA_COLUMN(ABA_GLOBAL *glob,
                       double obj,
                       double lb,
                       double ub,
                       int nnz,
                       ABA_ARRAY<int> &s,
                       ABA_ARRAY<double> &c)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`obj`

The objective function coefficient.

`lb`

The lower bound.

`ub`

The upper bound.

`nnz`

The number of nonzero elements stored in the arrays `s` and `c`.

`s`

An array of the nonzero elements of the column.

`c`

An array of the nonzero coefficients associated with the elements of `s`.

Constructor

Another constructor generating an uninitialized column.

```
ABA_COLUMN::ABA_COLUMN(ABA_GLOBAL *glob, int maxNnz)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`maxNnz`

The maximal number of nonzero elements that can be stored in the row.

Constructor

A constructor using a sparse vector for the initialization.

```
ABA_COLUMN::ABA_COLUMN(ABA_GLOBAL *glob,
                       double obj,
                       double lb,
                       double ub,
                       ABA_SPARVEC &vec)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`obj`

The objective function coefficient.

`lb`

The lower bound.

`ub`

The upper bound.

`vec`

A sparse vector storing the support and the coefficients of the column.

Destructor (`virtual`)

```
ABA_COLUMN::~ABA_COLUMN()
```

Output Operator

```
ostream &operator<<(ostream &out, const ABA_COLUMN &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The column being output.

obj

```
double ABA_COLUMN::obj() const
```

Return Value:

The objective function coefficient of the column.

obj

This version of the function `obj()` sets the objective function coefficient of the column.

```
void ABA_COLUMN::obj(double c)
```

Arguments:

`c`
: The new value of the objective function coefficient.

lBound

```
double ABA_COLUMN::lBound() const
```

Return Value:

The lower bound of the column.

lBound

This version of the function `lBound()` sets the lower bound of the column.

```
void ABA_COLUMN::lBound(double l)
```

Arguments:

`l`
: The new value of the lower bound.

uBound

```
double ABA_COLUMN::uBound() const
```

Return Value:

The upper bound of the column.

uBound

This version of the function `uBound()` sets the upper bound of the column.

```
void ABA_COLUMN::uBound(double u)
```

Arguments:

`u`
: The new value of the upper bound.

copy

The function `copy()` is very similar to the assignment operator, yet the columns do not have to be of equal size. A reallocation is performed if required.

```
void ABA_COLUMN::copy(const ABA_COLUMN &col)
```

Arguments:

`col`
: The column that is copied.

6.2.25 ABA_NUMCON

Like the class ABA_NUMVAR for variables we provide the class ABA_NUMCON for constraints which are uniquely defined by an integer number.

```
class ABA_NUMCON : public ABA_CONSTRAINT {
  public:
    ABA_NUMCON(ABA_MASTER *master,
               ABA_SUB *sub,
               ABA_CSENSE::SENSE sense,
               bool dynamic,
               bool local,
               bool liftable,
               int number,
               double rhs);
    virtual ~ABA_NUMCON();
    friend ostream &operator<<(ostream &out, const ABA_NUMCON &rhs);
    virtual double coeff(ABA_VARIABLE *v);
    virtual void   print(ostream &out);
    int number() const;
};
```

Constructor

```
ABA_NUMCON::ABA_NUMCON(ABA_MASTER *master,
                       ABA_SUB *sub,
                       ABA_CSENSE::SENSE sense,
                       bool dynamic,
                       bool local,
                       bool liftable,
                       int number,
                       double rhs)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sub`

A pointer to the subproblem associated with the constraint. This can be also the 0-pointer.

`sense`

The sense of the constraint.

`dynamic`

If this argument is `true`, then the constraint can be removed from the active constraint set during the cutting plane phase of the subproblem optimization.

`local`

If this argument is `true`, then the constraint is considered to be only locally valid. As a local constraint is associated with a subproblem, `sub` must not be 0 if `local` is `true`.

`liftable`

If this argument is `true`, then a lifting procedure must be available, i.e., that the coefficients of variables which have not been active at generation time of the constraint can be computed.

`number`

The identification number of the constraint.

`rhs`

The right hand side of the constraint.

Destructor (`virtual`)

```
ABA_NUMCON::~ABA_NUMCON()
```

Output Operator

The output operator writes the identification number and the right hand side to an output stream.

```
ostream &operator<<(ostream &out, const ABA_NUMCON &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The variable being output.

coeff (`virtual`)

```
double ABA_NUMCON::coeff(ABA_VARIABLE *v)
```

Return Value:

The coefficient of the variable `v`.

Arguments:

`v`

The variable of which the coefficient is determined. It must point to an object of the class `ABA_COLVAR`.

print (`virtual`)

The function `print()` writes the row format of the constraint on an output stream. It redefines the virtual function `print()` of the base class `ABA_CONVAR`.

```
void ABA_NUMCON::print(ostream &out)
```

Arguments:

`out`

The output stream.

number

```
int ABA_NUMCON::number() const
```

Return Value:

Returns the identification number of the constraint.

6.2.26 ABA_ROWCON

Earlier we explained that we distinguish between the constraint and the row format. We have seen already that a constraint is transformed to the row format when it is added to the linear program. However, for some constraints of certain optimization problems the row format itself is the most suitable representation. Therefore the class ABA_ROWCON implements constraints stored in the class ABA_ROW.

```
class ABA_ROWCON : public ABA_CONSTRAINT {
  public:
    ABA_ROWCON(ABA_MASTER *master,
               ABA_SUB *sub,
               ABA_CSENSE::SENSE sense,
               int nnz,
               const ABA_ARRAY<int> &support,
               const ABA_ARRAY<double> &coeff,
               double rhs,
               bool dynamic,
               bool local,
               bool liftable);
    ABA_ROWCON(ABA_MASTER *master,
               ABA_SUB *sub,
               ABA_CSENSE::SENSE sense,
               int nnz,
               int *support,
               double *coeff,
               double rhs,
               bool dynamic,
               bool local,
               bool liftable);
    virtual ~ABA_ROWCON();
    virtual double coeff(ABA_VARIABLE *v);
    virtual void print(ostream &out);
    ABA_ROW *row();

  protected:
    ABA_ROW row_;
};
```

row_

ABA_ROW row_

The representation of the constraint.

Constructor

```
ABA_ROWCON::ABA_ROWCON(ABA_MASTER *master,
                       ABA_SUB *sub,
                       ABA_CSENSE::SENSE sense,
                       int nnz,
                       const ABA_ARRAY<int> &support,
                       const ABA_ARRAY<double> &coeff,
                       double rhs,
                       bool dynamic,
                       bool local,
                       bool liftable)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sub`

A pointer to the subproblem associated with the constraint. This can also be the 0-pointer.

`sense`

The sense of the constraint.

`nnz`

The number of nonzero elements of the constraint.

`support`

The array storing the variables with nonzero coefficients.

`coeff`

The nonzero coefficients of the variables stored in `support`.

`rhs`

The right hand side of the constraint.

`dynamic`

If this argument is `true`, then the constraint can be removed from the active constraint set during the cutting plane phase of the subproblem optimization.

`local`

If this argument is `true`, then the constraint is considered to be only locally valid. As a locally valid constraint is associated with a subproblem, `sub` must not be 0 if `local` is `true`.

`liftable`

If this argument is `true`, then a lifting procedure must be available, i.e., that the coefficients of variables which have not been active at generation time of the constraint can be computed.

Constructor

This constructor is equivalent to the previous constructor except that it uses C-style arrays for `support` and `coeff`.

```
ABA_ROWCON::ABA_ROWCON(ABA_MASTER *master,
                       ABA_SUB *sub,
                       ABA_CSENSE::SENSE sense,
                       int nnz,
                       int *support,
                       double *coeff,
                       double rhs,
                       bool dynamic,
                       bool local,
                       bool liftable)
```

Destructor (`virtual`)

```
ABA_ROWCON::~ABA_ROWCON()
```

coeff (`virtual`)

The function `coeff()` computes the coefficient of a variable which must be of type `ABA_NUMVAR`. It redefines the virtual function `coeff()` of the base class `ABA_CONSTRAINT`.

Warning: The worst case complexity of the call of this function is the number of nonzero elements of the constraint.

```
double ABA_ROWCON::coeff(ABA_VARIABLE *v)
```

Return Value:

The coefficient of the variable `v`.

Arguments:

`v`

The variable of which the coefficient is determined.

print (`virtual`)

The function `print()` writes the row format of the constraint on an output stream. It redefines the virtual function `print()` of the base class `ABA_CONVAR`.

```
void ABA_ROWCON::print(ostream &out)
```

Arguments:

`out`

The output stream.

row

```
ABA_ROW *ABA_ROWCON::row()
```

Return Value:

A pointer to the object of the class `ABA_ROW` representing the constraint.

6.2.27 ABA_NUMVAR

This class is derived from the class ABA_VARIABLE and implements a variable which is uniquely defined by a number.

```
class ABA_NUMVAR : public ABA_VARIABLE {
  public:
    ABA_NUMVAR(ABA_MASTER *master,
               ABA_SUB *sub,
               int number,
               bool dynamic,
               bool local,
               double obj,
               double lBound,
               double uBound,
               ABA_VARTYPE::TYPE type);
    virtual ~ABA_NUMVAR();
    friend ostream &operator<<(ostream &out, const ABA_NUMVAR &rhs);
    int number() const;
  protected:
    int number_;
};
```

numvar_

`int number_`

The identification number of the variable.

Constructor

```
ABA_NUMVAR::ABA_NUMVAR(ABA_MASTER *master,
                       ABA_SUB *sub,
                       int number,
                       bool dynamic,
                       bool local,
                       double obj,
                       double lBound,
                       double uBound,
                       ABA_VARTYPE::TYPE type)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sub`

A pointer to the subproblem associated with variable. This can also be the 0-pointer.

`number`

The number of the column associated with the variable.

`dynamic`

If this argument is `true`, then the variable can also be removed again from the set of active variables after it is added once.

`local`

If this argument is `true`, then the variable is only locally valid, otherwise it is globally valid. As a locally valid variable is associated with a subproblem, `sub` must not be 0, if `local` is `true`.

`obj`

The objective function coefficient of the variable.

`lBound`

The lower bound of the variable.

`uBound`

The upper Bound of the variable.

`type`

The type of the variable.

Destructor (`virtual`)

```
ABA_NUMVAR::~ABA_NUMVAR()
```

Output Operator

The output operator writes the number of the variable to an output stream.

```
ostream &operator<<(ostream &out, const ABA_NUMVAR &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The variable being output.

number

```
int ABA_NUMVAR::number () const
```

Return Value:

The number of the variable.

6.2.28 ABA_SROWCON

The member functions `genRow()` and `slack()` of the class `ABA_ROWCON` can be significantly improved if the variable set is static, i.e., no variables are added or removed during the optimization. Therefore we implement the class `ABA_SROWCON`.

```
class ABA_SROWCON : public ABA_ROWCON {
  public:
    ABA_SROWCON(ABA_MASTER *master,
                ABA_SUB *sub,
                ABA_CSENSE::SENSE sense,
                int nnz,
                const ABA_ARRAY<int> &support,
                const ABA_ARRAY<double> &coeff,
                double rhs,
                bool dynamic,
                bool local,
                 bool liftable);
    ABA_SROWCON(ABA_MASTER *master,
                ABA_SUB *sub,
                ABA_CSENSE::SENSE sense,
                int nnz,
                int *support,
                double *coeff,
                double rhs,
                bool dynamic,
                bool local,
                bool liftable);
    virtual ~ABA_SROWCON();
    virtual int genRow(ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *var,
                       ABA_ROW &row);
    virtual double slack(ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *variables,
                         double *x);
};
```

Constructor

```
ABA_SROWCON::ABA_SROWCON(ABA_MASTER *master,
                         ABA_SUB *sub,
                         ABA_CSENSE::SENSE sense,
                         int nnz,
                         const ABA_ARRAY<int> &support,
                         const ABA_ARRAY<double> &coeff,
                         double rhs,
                         bool dynamic,
                         bool local,
                         bool liftable)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sub`

A pointer to the subproblem associated with the constraint. This can be also the 0-pointer.

`sense`

The sense of the constraint.

`nnz`

The number of nonzero elements of the constraint.

`support`

The array storing the variables with nonzero coefficients.

`coeff`

The nonzero coefficients of the variables stored in `support`.

`rhs`

The right hand side of the constraint.

`dynamic`

If this argument is `true`, then the constraint can be removed from the active constraint set during the cutting plane phase of the subproblem optimization.

`local`

If this argument is `true`, then the constraint is considered to be only locally valid. As a locally valid constraint is associated with a subproblem, `sub` must not be 0 if `local` is `true`.

`liftable`

If this argument is `true`, then a lifting procedure must be available, i.e., that the coefficients of variables which have not been active at generation time of the constraint can be computed.

Constructor

This constructor is equivalent to the previous constructor except that it uses C-style arrays for `support` and `coeff`.

```
ABA_SROWCON::ABA_SROWCON(ABA_MASTER *master,
                         ABA_SUB *sub,
                         ABA_CSENSE::SENSE sense,
                         int nnz,
                         int *support,
                         double *coeff,
                         double rhs,
                         bool dynamic,
                         bool local,
                         bool liftable)
```

Destructor (`virtual`)

```
ABA_SROWCON::~ABA_SROWCON()
```

genRow (`virtual`)

The function `genRow()` generates the row format of the constraint associated with the variable set `var`. This function redefines a virtual function of the base class `ABA_ROWCON`.

```
int ABA_SROWCON::genRow(ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *var,
                        ABA_ROW &row)
```

Return Value:

It returns the number of nonzero elements in the row format.

Arguments:

`var`

The variable set for which the row format is generated is only a dummy since the the variable set is assumed to be fixed for this constraint class.

`row`

Holds the row format of the constraint after the execution of this function.

slack (`virtual`)

The function `slack()` computes the slack of a vector associated with the variable set `variables`. This function redefines a virtual function of the base class `ABA_ROWCON`.

```
double ABA_SROWCON::slack(ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT> *variables,
                          double *x)
```

Return Value:

The slack of the vector `x`.

Arguments:

`variable`

The variable set for which the row format is generated is only a dummy since the the variable set is assumed to be fixed for this constraint class.

`x`

An array of length equal to the number of variables.

6.2.29 ABA_COLVAR

Some optimization problems, in particular column generation problems, are better described from a variable point of view instead of a constraint point of view. For such context we provide the class `ABA_COLVAR` which similar to the class `ABA_ROWCON` stores the nonzero coefficient explicitly in an object of the class `ABA_COLUMN`.

The constraint class which is associated with this variables class is the class `ABA_NUMCON` which identifies constraints only by a unique integer number. `ABA_NUMCON` is an abstract class.

```
class ABA_COLVAR : public ABA_VARIABLE {
  public:
    ABA_COLVAR(ABA_MASTER *master,
               ABA_SUB *sub,
               bool dynamic,
               bool local,
               double lBound,
               double uBound,
               ABA_VARTYPE::TYPE varType,
               double obj,
               int nnz,
               ABA_ARRAY<int> &support,
               ABA_ARRAY<double> &coeff);
    ABA_COLVAR(ABA_MASTER *master,
               ABA_SUB *sub,
               bool dynamic,
               bool local,
               double lBound,
               double uBound,
```

```
                ABA_VARTYPE::TYPE varType,
                double obj, ABA_SPARVEC &vector);
      virtual ~ABA_COLVAR();
      friend ostream &operator<<(ostream &out, const ABA_COLVAR &rhs);
      virtual void print(class ostream &out);
      virtual double coeff(ABA_CONSTRAINT *con);
      double coeff(int i);
      ABA_COLUMN *column();

    protected:
      ABA_COLUMN   column_;
  };
```

column_

`ABA_COLUMN column_`

The column representing the variable.

Constructor

```
ABA_COLVAR::ABA_COLVAR(ABA_MASTER *master,
                       ABA_SUB *sub,
                       bool dynamic,
                       bool local,
                       double lBound,
                       double uBound,
                       ABA_VARTYPE::TYPE varType,
                       double obj,
                       int nnz,
                       ABA_ARRAY<int> &support,
                       ABA_ARRAY<double> &coeff)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`sub`

A pointer to the subproblem associated with the variable. This can be also the 0-pointer.

`dynamic`

If this argument is `true`, then the variable can be removed from the active variable set during the subproblem optimization.

`local`

If this argument is `true`, then the constraint is considered to be only locally valid. As a local variable is associated with a subproblem, `sub` must not be 0 if local is `true`.

`lBound`

The lower bound of the variable.

`uBound`

The upper bound of the variable.

`varType`

The type of the variable.

`obj`

The objective function coefficient of the variable.

`nnz`

The number of nonzero elements of the variable.

`support`

The array storing the constraints with the nonzero coefficients.

`coeff`

The nonzero coefficients of the constraints stored in `support`.

Constructor

A constructor substituting `nnz`, `support`, and `coeff` of the previous constructor by an object of the class `ABA_SPARVEC`.

```
ABA_COLVAR::ABA_COLVAR(ABA_MASTER *master,
                       ABA_SUB *sub,
                       bool dynamic,
                       bool local,
                       double lBound,
                       double uBound,
                       ABA_VARTYPE::TYPE varType,
                       double obj,
                       ABA_SPARVEC &vector)
```

Destructor (`virtual`)

```
ABA_COLVAR::~ABA_COLVAR()
```

Output Operator

The output operator writes the column representing the variable to an output stream.

```
ostream &operator<<(ostream &out, const ABA_COLVAR &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The variable being output.

print (`virtual`)

The function `print()` writes the column representing the variable to an output stream. It redefines the virtual function `print()` of the base class `ABA_CONVAR`.

```
void ABA_COLVAR::print(class ostream &out)
```

Arguments:

`out`

The output stream.

coeff (`virtual`)

```
double ABA_COLVAR::coeff(ABA_CONSTRAINT *con)
```

Return Value:

The coefficient of the constraint `con`.

Arguments:

`con`

The constraint of which the coefficient is computed. This must be a pointer to the class `ABA_NUMCON`.

coeff

This version of the function `coeff()` computes the coefficient of a constraint with a given number.

```
double ABA_COLVAR::coeff(int i)
```

Return Value:

The coefficient of constraint `i`.

Arguments:

`i`

The number of the constraint.

column

```
ABA_COLUMN *ABA_COLVAR::column()
```

Return Value:

A pointer to the column representing the variable.

6.2.30 ABA_ACTIVE

This template class implements the sets of active constraints and variables which are associated with each subproblem. Note, also an inactive subproblem can have an active set of constraints and variables, e.g., the sets with which its unprocessed sons in the enumeration tree are initialized.

If an active set of constraints is instantiated then the `BaseType` should be `ABA_CONSTRAINT` and the `CoType` should be `ABA_VARIABLE`, for an active set of variables this is vice versa.

```
template <class BaseType, class CoType>
class ABA_ACTIVE : public ABA_ABACUSROOT {
  public:
    ABA_ACTIVE(ABA_MASTER *master, int max);
    ABA_ACTIVE(ABA_MASTER *master, ABA_ACTIVE *a, int max);
    ABA_ACTIVE(const ABA_ACTIVE<BaseType, CoType> &rhs);
    ~ABA_ACTIVE();
    friend ostream &operator<<(ostream &out,
                               const ABA_ACTIVE<BaseType, CoType> &rhs);
    int number() const;
    int max() const;
    BaseType* operator[](int i);
    ABA_POOLSLOTREF<BaseType, CoType>* poolSlotRef(int i);
    void insert(ABA_POOLSLOT<BaseType, CoType> *ps);
    void insert(ABA_BUFFER<ABA_POOLSLOT<BaseType, CoType> *> &ps);
    void remove(ABA_BUFFER<int> &del);
    void realloc(int newSize);
  private:
    const ABA_ACTIVE<BaseType, CoType>
                           &operator=(const ABA_ACTIVE<BaseType, CoType> & rhs);
};
```

Constructor

```
ABA_ACTIVE<BaseType, CoType>::ABA_ACTIVE(ABA_MASTER *master, int max)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`max`

The maximal number of active constraints/variables.

Constructor

In addition to the previous constructor, this constructor initializes the active set.

```
ABA_ACTIVE<BaseType, CoType>::ABA_ACTIVE(ABA_MASTER *master,
                                         ABA_ACTIVE<BaseType, CoType> *a,
                                         int max)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`a`

At most `max` active constraints/variables are taken from this set.

`max`

The maximal number of active constraints/variables.

Copy Constructor

```
ABA_ACTIVE<BaseType, CoType>::ABA_ACTIVE(const ABA_ACTIVE<BaseType, CoType> &rhs)
```

Arguments:

`rhs`
: The active set that is copied.

Destructor

```
ABA_ACTIVE<BaseType, CoType>::~ABA_ACTIVE()
```

Output Operator

The output operator writes all active constraints and variables to an output stream. If an associated pool slot is void, or the item is newer than the one we refer to, then `"void"` is output.

```
ostream &operator<<(ostream &out, const ABA_ACTIVE<BaseType, CoType> &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`
: The output stream.

`rhs`
: The active set being output.

number

```
int ABA_ACTIVE<BaseType, CoType>::number() const
```

Return Value:

The current number of active items.

max

```
int ABA_ACTIVE<BaseType, CoType>::max() const
```

Return Value:

The maximum number of storable active items (without reallocation).

Subscript Operator

```
BaseType* ABA_ACTIVE<BaseType, CoType>::operator[](int i)
```

Return Value:

A pointer to the `i`-th active item or 0 if this item has been removed in the meantime.

Arguments:

`i`
: The number of the active item.

poolSlotRef

```
ABA_POOLSLOTREF<BaseType, CoType> * ABA_ACTIVE<BaseType, CoType>::poolSlotRef(int i)
```

Return Value:

The i-th entry in the ABA_ARRAY active.

Arguments:

i

The number of the active item.

insert

The function insert() adds a constraint/variable to the active items.

```
void ABA_ACTIVE<BaseType, CoType>::insert(ABA_POOLSLOT<BaseType, CoType> *ps)
```

Arguments:

ps

The pool slot storing the constraint/variable being added.

insert

The function insert() is overloaded that also several items can be added at the same time.

```
void ABA_ACTIVE<BaseType, CoType>::insert(
                            ABA_BUFFER<ABA_POOLSLOT<BaseType, CoType> *> &ps)
```

Arguments:

ps

The buffer storing the pool slots of all constraints/variables that are added.

remove

The function remove() removes items from the list of active items.

```
void ABA_ACTIVE<BaseType, CoType>::remove(ABA_BUFFER<int> &del)
```

Arguments:

del

The numbers of the items that should be removed. These numbers must be upward sorted.

realloc

The function realloc() changes the maximum number of active items which can be stored in an object of this class.

```
void ABA_ACTIVE<BaseType, CoType>::realloc(int newSize)
```

Arguments:

newSize

The new maximal number of active items.

6.2.31 ABA_CUTBUFFER

This template class implements a buffer for constraints and variables which are generated during the cutting plane or column generation phase. There are two reasons why constraints/variables are buffered instead of being added immediately. First, the set of active constraints/variables should not be falsified during the cut/variable generation. Second, optionally a rank can be assigned to each buffered item. Then not all, but only the best items according to this rank are actually added.

```
template<class BaseType, class CoType>
class ABA_CUTBUFFER : public ABA_ABACUSROOT {
  public:
    ABA_CUTBUFFER(ABA_MASTER *master, int size);
    ~ABA_CUTBUFFER();
    int size() const;
    int number() const;
    int space() const;
    int insert(ABA_POOLSLOT<BaseType, CoType> *slot, bool keepInPool);
    int insert(ABA_POOLSLOT<BaseType, CoType> *slot, bool keepInPool,
               double rank);
    void remove(ABA_BUFFER<int> &index);
    ABA_POOLSLOT<BaseType, CoType> *slot(int i);

  private:
    ABA_CUTBUFFER(const ABA_CUTBUFFER<BaseType, CoType> &rhs);
    const ABA_CUTBUFFER<BaseType, CoType>
          &operator=(const ABA_CUTBUFFER<BaseType, CoType> &rhs);
};
```

Constructor

```
ABA_CUTBUFFER<BaseType, CoType>::ABA_CUTBUFFER(ABA_MASTER *master, int size)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`size`

The maximal number of constraints/variables which can be buffered.

Destructor

```
ABA_CUTBUFFER<BaseType, CoType>::~ABA_CUTBUFFER()
```

size

```
int ABA_CUTBUFFER<BaseType, CoType>::size() const
```

Return Value:

The maximal number of items that can be buffered.

number

```
int ABA_CUTBUFFER<BaseType, CoType>::number() const
```

Return Value:

The number of buffered items.

space

```
int ABA_CUTBUFFER<BaseType, CoType>::space() const
```

Return Value:

The number of items which can still be inserted into the buffer.

slot

```
ABA_POOLSLOT<BaseType, CoType> * ABA_CUTBUFFER<BaseType, CoType>::slot(int i)
```

Return Value:

A pointer to the `i`-th `ABA_POOLSLOT` that is buffered.

insert

The function `insert()` adds a slot to the buffer.

```
int ABA_CUTBUFFER<BaseType, CoType>::insert(ABA_POOLSLOT<BaseType, CoType> *slot,
                                            bool keepInPool)
```

Return Value:

`0`

If the item can be inserted.

`1`

If the buffer is already full.

Arguments:

`slot`

The inserted pool slot.

`keepInPool`

If the flag `keepInPool` is `true`, then the item stored in the `slot` is not removed from the pool, even if it is discarded in `extract()`. Items regenerated from a pool should always have this flag set to `true`.

insert

In addition to the previous version of the function `insert()` this version also adds a rank to the item such that all buffered items can be sorted with the function `sort()`.

```
int ABA_CUTBUFFER<BaseType, CoType>::insert(ABA_POOLSLOT<BaseType, CoType> *slot,
                                            bool keepInPool,
                                            double rank)
```

Return Value:

`0`

If the item can be inserted.

`1`

If the buffer is already full.

Arguments:

`rank`

A rank associated with the constraint/variable.

remove

The function `remove()` removes the specified elements from the buffer.

```
void ABA_CUTBUFFER<BaseType, CoType>::remove(ABA_BUFFER<int> &index)
```

Arguments:

`index`

The numbers of the elements which should be removed.

6.2.32 ABA_INFEASCON

If a constraint is transformed from its pool to the row format it may turn out that the constraint is infeasible since variables are fixed or set such that all nonzero coefficients of the left hand side are eliminated and the right hand side has to be updated. The enumeration `INFEAS` indicates if the constraint's left hand side, which is implicitly zero, is either `TooLarge`, `Feasible`, or `TooSmall`.

```
class ABA_INFEASCON : public ABA_ABACUSROOT {
  public:
    enum INFEAS {TooSmall = -1, Feasible, TooLarge};

    ABA_INFEASCON(ABA_MASTER *master, ABA_CONSTRAINT *con, INFEAS inf);
    ABA_CONSTRAINT *constraint();
    INFEAS infeas() const;
    bool goodVar(ABA_VARIABLE *v);
};
```

enum INFEAS

The different ways of infeasibility of a constraint.

`TooSmall`

The left hand side is too small for the right hand side.

`Feasible`

The constraint is not infeasible.

`TooLarge`

The left hand side is too large for the right hand side.

Constructor

```
ABA_INFEASCON::ABA_INFEASCON(ABA_MASTER *master, ABA_CONSTRAINT *con, INFEAS inf)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

`con`

The infeasible constraint.

`inf`

The way of infeasibility.

constraint

```
ABA_CONSTRAINT *ABA_INFEASCON::constraint()
```

Return Value:

A pointer to the infeasible constraint.

infeas

```
ABA_INFEASCON::INFEAS ABA_INFEASCON::infeas() const
```

Return Value:

The way of infeasibility of the constraint.

goodVar

```
bool ABA_INFEASCON::goodVar(ABA_VARIABLE *v)
```

Return Value:

`true`

If the variable v might reduce the infeasibility,

`false`

otherwise.

Arguments:

`v`

A variable for which we test if its addition might reduce infeasibility.

6.2.33 ABA_OPENSUB

During a branch-and-bound algorithm a set of open subproblems has to be maintained. New subproblems are inserted in this set after a branching step, or when a subproblem becomes dormant. A subproblem is extracted from this list if it becomes the active subproblem which is optimized.

```
class ABA_OPENSUB : public ABA_ABACUSROOT {
  public:
    ABA_OPENSUB(ABA_MASTER *master);
    int number() const;
    bool empty() const;
    double  dualBound() const;

  private:
    ABA_OPENSUB(const ABA_OPENSUB &rhs);
    const ABA_OPENSUB &operator=(const ABA_OPENSUB &rhs);
};
```

Constructor

The constructor does not initialize the member `dualBound_` since this can only be done if we know the sense of the objective function which is normally unknown when the constructor of the class `ABA_MASTER` is called which again calls this constructor.

```
ABA_OPENSUB::ABA_OPENSUB(ABA_MASTER *master)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

number

```
int ABA_OPENSUB::number() const
```

Return Value:

The current number of open subproblems contained in this set.

emtpy

```
bool ABA_OPENSUB::empty() const
```

Return Value:

`true`

If there is no subproblem in the set of open subproblems,

`false`

otherwise.

dualBound

```
double ABA_OPENSUB::dualBound() const
```

Return Value:

The value of the dual bound of all subproblems in the list.

6.2.34 ABA_FIXCAND

Variables can be only fixed according to the reduced costs and statuses of variables of the root of the remaining branch-and-bound tree. However, if we store these values, we can repeat the fixing process also in any other node of the enumeration tree when we find a better global lower bound.

Possible candidates for fixing are all variables which have the status `AtLowerBound` or `AtUpperBound`. We store all these candidates together with their values in this class.

If we try to fix variables according to reduced cost criteria in nodes which are not the root of the remaining branch-and-cut tree, we always have to take the candidates and values from this class.

```
class ABA_FIXCAND : public ABA_ABACUSROOT {
  public:
    ABA_FIXCAND(ABA_MASTER *master);
    ~ABA_FIXCAND();

  private:
    ABA_FIXCAND(const ABA_FIXCAND &rhs);
    const ABA_FIXCAND &operator=(const ABA_FIXCAND &rhs);
};
```

Constructor

```
ABA_FIXCAND::ABA_FIXCAND(ABA_MASTER *master)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

Destructor

```
ABA_FIXCAND::~ABA_FIXCAND()
```

6.2.35 ABA_TAILOFF

During the cutting plane phase of the optimization of a single subproblem it can be quite often observed that during the first iterations a significant decrease of the optimum value of the LP occurs, yet, this decrease becomes smaller and smaller in later iterations. This effect is called *tailing off* ([PR91]). Experimental results show that it might be better to branch, although violated constraints can still be generated, than to continue the cutting plane phase. This class stores the history of the values of the last LP-solutions and implements all functions to control this tailing-off effect. The parameters are taken from the associated master.

```
class ABA_TAILOFF : public ABA_ABACUSROOT {
  public:
    ABA_TAILOFF(ABA_MASTER *master);
    ~ABA_TAILOFF();
    friend ostream &operator<<(ostream &out, const ABA_TAILOFF &rhs);

    bool tailOff() const;
    int diff(int nLps, double &d) const;
};
```

Constructor

The constructor takes the length of the tailing off history from `ABA_MASTER::tailOffNLp()`.

```
ABA_TAILOFF::ABA_TAILOFF(ABA_MASTER *master)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

Destructor

```
ABA_TAILOFF::~ABA_TAILOFF()
```

Output Operator

The output operator writes the memorized LP-values on an output stream.

```
ostream &operator<<(ostream &out, const ABA_TAILOFF &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The tailing-off manager being output.

tailOff

The function `tailOff()` checks if there is a tailing-off effect. We assume a tailing-off effect if during the last `ABA_MASTER::tailOffNLps()` iterations of the cutting plane algorithms the dual bound changed at most `ABA_MASTER::tailOffPercent()` percent.

```
bool ABA_TAILOFF::tailOff() const
```

Return Value:

`true`

If a tailing off effect is observed,

`false`

otherwise.

diff

The function `diff()` can be used to retrieve the difference between the last and a previous LP-solution in percent.

```
int ABA_TAILOFF::diff(int nLps, double &d) const
```

Return Value:

`0`

If the difference could be computed, i.e., the old LP-value `nLPs` before the last one is store in the history,

`1`

otherwise.

Arguments:

`nLps`

The number of LPs before the last solved linear program with which the last solved LP-value should be compared.

`d`

Contains the absolute difference bewteen the value of the last solved linear program and the value of the linear program solved `nLPs` before in percent relative to the older value.

6.2.36 ABA_HISTORY

This class implements the storage of the solution history. Each time when a better feasible solution or globally valid dual bound is found, it should be memorized in this class.

```
class ABA_HISTORY : public ABA_ABACUSROOT {
  public:
    ABA_HISTORY(ABA_MASTER *master);
    friend ostream& operator<<(ostream& out, const ABA_HISTORY &rhs);
    void update();
};
```

Constructor

The constructor initializes a history table with 100 possible entries. If this number is exceeded an automatic reallocation is performed.

```
ABA_HISTORY::ABA_HISTORY(ABA_MASTER *master)
```

Arguments:

`master`

A pointer to the corresponding master of the optimization.

Output Operator

```
ostream& operator<<(ostream& out, const ABA_HISTORY &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The solution history being output.

update

The function `update()` adds an additional line to the history table, primal bound, dual bound, and the time are taken from the corresponding master object. The history table is automatically reallocated if necessary.

Usually an explicit call to this function from an application class is not required since `update()` is automatically called if a new global primal or dual bound is found.

```
void ABA_HISTORY::update()
```

6.3 Basic Data Structures

This subsection documents various basic data structures which we have used within ABACUS. They can also be used within an application. The templated basic data structures are documented in Section 6.4.

6.3.1 ABA_SPARVEC

If the number of components of a vector having nonzero coefficients is small (sparse), then it is more adequate to store only the number of these components together with the nonzero coefficients.

```
class ABA_SPARVEC : public ABA_ABACUSROOT {
  public:
    ABA_SPARVEC(ABA_GLOBAL *glob,
                int size,
                double reallocFac = 10.0);
    ABA_SPARVEC(ABA_GLOBAL *glob,
                int size,
                const ABA_ARRAY<int> &s,
                const ABA_ARRAY<double> &c,
                double reallocFac = 10.0);
    ABA_SPARVEC(ABA_GLOBAL *glob,
                int size,
                int *s,
                double *c,
                double reallocFac = 10.0);
    ABA_SPARVEC(const ABA_SPARVEC& rhs);
    ~ABA_SPARVEC();
    const ABA_SPARVEC& operator=(const ABA_SPARVEC& rhs);
    friend ostream& operator<<(ostream& out, const ABA_SPARVEC& rhs);
    int support(int i) const;
    double coeff(int i) const;
    double origCoeff(int i) const;
    void insert(int s, double c);
    void leftShift(ABA_BUFFER<int> &del);
    void copy (const ABA_SPARVEC &vec);
    void clear();
    void rename(ABA_ARRAY<int> &newName);
    int size() const;
    int nnz() const;
    double norm();
    void realloc();
    void realloc(int newSize);
  protected:
    void rangeCheck(int i) const;
    ABA_GLOBAL *glob_;
    int size_;
    int nnz_;
    double reallocFac_;
    int *support_;
    double *coeff_;
};
```

glob_

`ABA_GLOBAL *glob_`

A pointer to the corresponding global object.

size_

`int size_`

The maximal number of nonzero coefficients which can be stored without reallocation.

nnz_

`int nnz_`

The number of stored elements ("nonzeros").

reallocFac_

`double reallocFac_`

If a new element is inserted but the sparse vector is full, then its size is increased by `reallocFac_` percent.

support_

`int *support_`

The array storing the nonzero variables.

coeff_

`double *coeff_`

The array storing the corresponding nonzero coefficients.

Constructor

The constructor for an empty sparse vector.

```
ABA_SPARVEC::ABA_SPARVEC(ABA_GLOBAL *glob,
                         int size,
                         double reallocFac)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`size`

The maximal number of nonzeros of the sparse vector (without reallocation).

`reallocFac`

The reallocation factor (in percent of the original size), which is used in a default reallocation if a variable is inserted when the sparse vector is already full. Its default value is 10.

Constructor

A constructor with initialization of the support and coefficients of the sparse vector. The minimum value of `size` and `s.size` is the number of nonzeros of the sparse vector.

```
ABA_SPARVEC::ABA_SPARVEC(ABA_GLOBAL *glob,
                         int size,
                         const ABA_ARRAY<int> &s,
                         const ABA_ARRAY<double> &c,
                         double reallocFac)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`size`

The maximal number of nonzeros (without reallocation).

`s`

An array storing the support of the sparse vector, i.e., the elements for which a (normally nonzero) coefficient is given in `c`.

`c`

An array storing the coefficients of the support elements given in `s`. This array must have at least the length of the minimum of `size` and `s.size()`.

`reallocFac`

The reallocation factor (in percent of the original size), which is used in a default reallocation if a variable is inserted when the sparse vector is already full. Its default value is 10.

Constructor

This constructor is equivalent to the previous one except that it is using C-style arrays for the initialization of the sparse vector.

```
ABA_SPARVEC::ABA_SPARVEC(ABA_GLOBAL *glob,
                         int nnz,
                         int *s,
                         double *c,
                         double reallocFac)
```

Copy Constructor

```
ABA_SPARVEC::ABA_SPARVEC(const ABA_SPARVEC& rhs)
```

Arguments:

`rhs`

The sparse vector that is copied.

Destructor

```
ABA_SPARVEC::~ABA_SPARVEC()
```

Assignment Operator

The assignment operator requires that the left hand and the right hand side have the same length (otherwise use the function `copy()`).

```
const ABA_SPARVEC& ABA_SPARVEC::operator=(const ABA_SPARVEC& rhs)
```

Return Value:

A reference to the left hand side.

Arguments:

`rhs`
: The right hand side of the assignment.

Output Operator

The output operator writes the elements of the support and their coefficients line by line on an output stream.

```
ostream& operator<<(ostream &out, const ABA_SPARVEC &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`
: The output stream.

`rhs`
: The sparse vector being output.

support

```
int ABA_SPARVEC::support(int i) const
```

Return Value:

The support of the `i`-th nonzero element.

Arguments:

`i`
: The number of the nonzero element.

coeff

The function `coeff()`.

```
double ABA_SPARVEC::coeff(int i) const
```

Return Value:

The coefficient of the `i`-th nonzero element.

Arguments:

`i`
: The number of the nonzero element.

origcoeff

The function `origCoeff()`

```
double ABA_SPARVEC::origCoeff(int i) const
```

Return Value:

The coefficient having support `i`.

Arguments:

`i`

The number of the original coefficient.

insert

The function `insert()` adds a new support/coefficient pair to the vector. If necessary a reallocation of the member data is performed automatically.

```
void ABA_SPARVEC::insert(int s, double c)
```

Arguments:

`s`

The new support.

`c`

The new coefficient.

leftShift

The function `leftShift()` deletes the elements listed in a buffer from the sparse vector. The numbers of indices in this buffer must be upward sorted. The elements before the first element in the buffer are unchanged. Then the elements which are not deleted are shifted left in the arrays.

```
void ABA_SPARVEC::leftShift(ABA_BUFFER<int> &del)
```

Arguments:

`del`

The numbers of the elements removed from the sparse vector.

copy

The function `copy()` is very similar to the assignment operator, yet the size of the two vectors need not be equal and only the support, the coefficients, and the number of nonzeros is copied. A reallocation is performed if required.

```
void ABA_SPARVEC::copy(const ABA_SPARVEC &vec)
```

Arguments:

`vec`

The sparse vector that is copied.

clear

The function `clear()` removes all nonzeros from the sparse vector.

```
void ABA_SPARVEC::clear()
```

rename

The function `rename()` replaces the index of the support by new names.

```
void ABA_SPARVEC::rename(ABA_ARRAY<int> &newName)
```

Arguments:

`newName`

The new names (support) of the elements of the sparse vector. The array `newName` must have at least a length equal to the maximal element in the support of the sparse vector.

size

```
int ABA_SPARVEC::size() const
```

Return Value:

The maximal length of the sparse vector.

nnz

```
int ABA_SPARVEC::nnz() const
```

Return Value:

The number of nonzero elements. This is not necessarily the correct number of nonzeros, yet the number of coefficient/support pairs, which are stored. Some of these pairs may have a zero coefficient.

norm

```
double ABA_SPARVEC::norm()
```

Return Value:

The Euclidean norm of the sparse vector.

realloc

The function `realloc()` increases the size of the sparse vector by `reallocFac_` percent of the original size. This function is called if an automatic reallocation takes place.

```
void ABA_SPARVEC::realloc()
```

realloc

This other version of `realloc()` reallocates the sparse vector to a given length. It is an error to decrease size below the current number of nonzeros.

```
void ABA_SPARVEC::realloc(int newSize)
```

Arguments:

`newSize`

The new maximal number of nonzeroes that can be stored in the sparse vector.

rangeCheck

This function terminates the program with an error message if `i` is negative or greater or equal than the number of nonzero elements.

```
void ABA_SPARVEC::rangeCheck(int i) const
```

Arguments:

`i`

An integer that should be checked if it is in the range of the sparse vector.

6.3.2 ABA_SET

This class implements a data structure for collections of dynamic disjoint sets of integers. Each set has a unique representative being one member of the set. We provide the operations generation of a set with one element, union of sets, and the determination of the set some element is currently contained in.

```
class ABA_SET : public ABA_ABACUSROOT {
    public:
      ABA_SET (ABA_GLOBAL *glob, int size);
      void makeSet(int x);
      bool unionSets(int x, int y);
      int findSet(int x);

    protected:
      ABA_GLOBAL *glob_;
      ABA_ARRAY<int>  parent_;
    };
```

glob_

`ABA_GLOBAL *glob_`

A pointer to the corresponding global object.

parent_

`ABA_ARRAY<int> parent_`

The collection of sets is implemented by a collection of trees. `parent_[i]` is the parent of node `i` in the tree representing the set containing `i`. If `i` is the root of a tree then `parent_[i]` is `i` itself.

Constructor

```
ABA_SET::ABA_SET(ABA_GLOBAL *glob, int size)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`size`

Only integers between 0 and `size-1` can be inserted in the set.

makeSet

The function `makeSet()` creates a set storing only one element and adds it to the collection of sets.

```
void ABA_SET::makeSet(int x)
```

Arguments:

`x`

The single element of the new set.

unionSets

The function `unionSets()` unites the two sets which contain `x` and `y`, respectively. This operation may only be performed if both `x` and `y` have earlier been added to the collection of sets by the function `makeSet()`.

We do not use the heuristic attaching the smaller subtree to the bigger one, since we want to guarantee that the representative of `x` is always the representative of the two united sets.

```
bool ABA_SET::unionSets(int x, int y)
```

Return Value:

`true`

If both sets have been disjoint before the function call,

`false`

otherwise.

Arguments:

`x`

An element of the first set of the union operation.

`y`

An element in the second set of the union operation.

findSet

The function `findSet()` finds the representative of the set containing `x`. This operation may be only performed if `x` has been earlier added to the collection of sets by the function `makeSet()`.

```
int ABA_SET::findSet(int x)
```

Return Value:

The representative of the set containing `x`.

Arguments:

`x`

An element of the searched set.

6.3.3 ABA_FASTSET

This class is derived from the class ABA_SET and holds for each set a rank which approximates the logarithm of the tree size representing the set and is also an upper bound for the height of this tree. In a union operation the tree with smaller rank is attached to the tree with larger rank.

```
class ABA_FASTSET : public ABA_SET {
  public:
    ABA_FASTSET (ABA_GLOBAL *glob, int size);
    bool unionSets(int x, int y);
};
```

Constructor

```
ABA_FASTSET::ABA_FASTSET(ABA_GLOBAL *glob, int size)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`size`

Only integers between 0 and `size-1` can be inserted in the set.

unionSets()

The function `unionSets()` unites the sets `x` and `y`. It differs from the function `unionSets()` of the base class ABA_SET such that the tree with smaller rank is attached to the one with larger rank. Therefore, `x` is no more guaranteed to be the representative of the joint set.

```
bool ABA_FASTSET::unionSets(int x, int y)
```

Return Value:

`true`

If both sets have been disjoint before the function call,

`false`

otherwise.

Arguments:

`x`

An element of the first set of the union operation.

`y`

An element in the second set of the union operation.

6.3.4 ABA_STRING

The class ABA_STRING implements are very simple class for the representation of character strings.

```
class ABA_STRING : public ABA_ABACUSROOT {
  public:
    ABA_STRING(ABA_GLOBAL *glob, const char* cString = "");
    ABA_STRING(ABA_GLOBAL *glob, const char* cString, int index);
    ABA_STRING(const ABA_STRING &rhs);
```

```
    ~ABA_STRING();
    const ABA_STRING& operator=(const ABA_STRING &rhs);
    const ABA_STRING& operator=(const char *rhs);
    friend int operator==(const ABA_STRING &lhs, const ABA_STRING &rhs);
    friend int operator==(const ABA_STRING &lhs, const char *rhs);
    friend int operator!=(const ABA_STRING &lhs, const ABA_STRING &rhs);
    friend int operator!=(const ABA_STRING &lhs, const char *rhs);
    friend ostream& operator<<(ostream &out, const ABA_STRING &rhs);
    char& operator[](int i);
    const char& operator[](int i) const;
    int size() const;
    int ascii2int(int i = 0) const;
    unsigned int ascii2unsignedint() const;
    double ascii2double() const;
    bool ascii2bool() const;
    bool ending(const char *end) const;
    char *string();
};
```

Constructor

```
ABA_STRING::ABA_STRING(ABA_GLOBAL *glob, const char *cString)
```

Arguments:

`glob`

: A pointer to the corresponding global object.

`cString`

: The initializing string, by default the empty string.

Constructor

A constructor building a string from a string and an integer. This constructor is especially useful for building variable or constraint names like `con18`.

```
ABA_STRING::ABA_STRING(ABA_GLOBAL *glob, const char *cString, int index)
```

Arguments:

`glob`

: A pointer to the corresponding global object.

`cString`

: The initializing string.

`index`

: The integer value appending to the `cString` (must be less than `MAX_INT`).

Copy Constructor

```
ABA_STRING::ABA_STRING(const ABA_STRING &rhs)
```

Arguments:

`rhs`

: The string that is copied.

Destructor

```
ABA_STRING::~ABA_STRING()
```

Assignment Operator

The assignment operator makes a copy of the right hand side and reallocates memory if required.

```
const ABA_STRING& ABA_STRING::operator=(const ABA_STRING &rhs)
```

Return Value:

A reference to the object.

Arguments:

`rhs`

The right hand side of the assignment.

Assignment Operator

The assignment operator is overloaded for character strings.

```
const ABA_STRING& ABA_STRING::operator=(const char *rhs)
```

Comparison Operator

```
int operator==(const ABA_STRING &lhs, const ABA_STRING &rhs)
```

Return Value:

`0`

If both strings are not equal,

`1`

otherwise.

Arguments:

`lhs`

The left hand side of the comparison.

`rhs`

The right hand side of the comparison.

Comparison Operator

The comparison operator is overloaded for character strings on the right hand side.

```
int operator==(const ABA_STRING &lhs, const char *rhs)
```

Not-Equal Operator

```
int operator!=(const ABA_STRING &lhs, const ABA_STRING &rhs)
```

Return Value:

`0`
: If both strings are equal,

`1`
: otherwise.

Arguments:

`lhs`
: The left hand side of the comparison.

`rhs`
: The right hand side of the comparison.

Not-Equal Operator

The not-equal operator is overloaded for character strings on the right hand side.

```
int operator!=(const ABA_STRING &lhs, const char *rhs)
```

Output Operator

```
ostream& operator<<(ostream &out, const ABA_STRING &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`
: The output stream.

`rhs`
: The string being output.

Subscript Operator

With the subscript operator a single character of the string can be accessed or modified.

```
char& ABA_STRING::operator[](int i)
```

Return Value:

A reference to the `i`-th character of the string.

Arguments:

`i`
: The number of the character that should be accessed or modified. The first character has number 0.

Subscript Operator

The subscript operator is overloaded for constant use.

```
const char& ABA_STRING::operator[](int i) const
```

size

```
int ABA_STRING::size() const
```

Return Value:

The length of the string, not including the `'\0'` terminating the string.

ascii2int

The function `ascii2int()` is very similar to the function `atoi()` from `<string.h>`. It converts the substring starting at component `i` and ending in the first following component with `'\0'` to an integer. `ascii2int(0)` converts the complete string.

```
int ABA_STRING::ascii2int(int i) const
```

Return Value:

The string converted to an integer value.

Arguments:

`i`

The number of the character at which the conversion should start. The default value of `i` is 0.

ascii2double

The function `ascii2double()` emulates the function `atof()` of the standard C library and converts the string to a floating point number.

```
double ABA_STRING::ascii2double() const
```

Return Value:

The string converted to a floating point number.

ascii2unsignedint

The function `ascii2unsignedint()` converts the string to an `unsigned int` value.

```
unsigned int ABA_STRING::ascii2unsignedint() const
```

Return Value:

The string converted to an unsigned integer.

ascii2bool

The function `ascii2bool()` converts the string to a boolean value. This is only possible for the strings `"true"` and `"false"`.

```
bool ABA_STRING::ascii2bool() const
```

Return Value:

The string converted to `true` or `false`.

ending

```
bool ABA_STRING::ending(const char *end) const
```

Return Value:

`true`
: If the string ends with the string `end`,

`false`
: otherwise.

Arguments:

`end`
: The string with which the ending of the string is compared.

string

```
char *ABA_STRING::string()
```

Return Value:

The `char*` representing the string to make it accessible for C-functions.

6.4 Templates

Various basic data structures are available as templates within ABACUS. For the instantiation of templates we refer to Section 5.3.

6.4.1 ABA_ARRAY

One of the basic classes is a template for arrays. It can be used like a "normal" C-style array, yet has some additional nice features, especially we do not have to care for the allocation and deallocation of memory. The first index of an array is 0 as usual in C++.

```
template <class Type> class ABA_ARRAY : public ABA_ABACUSROOT {
  public:
    ABA_ARRAY(ABA_GLOBAL *glob, int size);
    ABA_ARRAY(ABA_GLOBAL *glob, int size, Type init);
    ABA_ARRAY(ABA_GLOBAL *glob, const ABA_BUFFER<Type> &buf);
    ABA_ARRAY(const ABA_ARRAY<Type> &rhs);
    ~ABA_ARRAY();
    const ABA_ARRAY<Type>& operator=(const ABA_ARRAY<Type>& rhs);
    const ABA_ARRAY<Type>& operator=(const ABA_BUFFER<Type>& rhs);
    friend ostream& operator<<(ostream& out, const ABA_ARRAY<Type> &array);
    Type& operator[](int i);
    const Type& operator[](int i) const;
    void copy(const ABA_ARRAY<Type>& rhs);
    void copy(const ABA_ARRAY<Type>& rhs, int l, int r);
    void leftShift(ABA_BUFFER<int> &ind);
    void leftShift(ABA_ARRAY<bool> &remove);
    void set(int l, int r, Type val);
    void set(Type val);
    int  size() const;
    void realloc(int newSize);
    void realloc(int newSize, Type init);
};
```

Constructor

A constructor without initialization.

```
ABA_ARRAY<Type>::ABA_ARRAY(ABA_GLOBAL *glob, int size)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`size`

The length of the array.

Constructor

A constructor with initialization.

```
ABA_ARRAY<Type>::ABA_ARRAY(ABA_GLOBAL *glob, int size, Type init)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`size`

The length of the array.

`init`

The initial value of all elements of the array.

Constructor

```
ABA_ARRAY<Type>::ABA_ARRAY(ABA_GLOBAL *glob, const ABA_BUFFER<Type> &buf)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`buf`

The array receives the length of this buffer and all buffered elements are copied to the array.

Copy Constructor

```
ABA_ARRAY<Type>::ABA_ARRAY(const ABA_ARRAY<Type> &rhs)
```

Arguments:

`rhs`

The array being copied.

Destructor

```
ABA_ARRAY<Type>::~ABA_ARRAY()
```

Assignment Operator

The assignment operator can only be used for arrays with equal length.

```
const ABA_ARRAY<Type>& ABA_ARRAY<Type>::operator=(const ABA_ARRAY<Type>& rhs)
```

Return Value:

A reference to the array on the left hand side.

Arguments:

`rhs`
: The array being assigned.

Assignment Operator

To assign an object of the class `ABA_BUFFER` to an object of the class `ABA_ARRAY` the size of the left hand side must be at least the size of `rhs`. Then all buffered elements of `rhs` are copied.

```
const ABA_ARRAY<Type>& ABA_ARRAY<Type>::operator=(const ABA_BUFFER<Type>& rhs)
```

Return Value:

A reference to the array on the left hand side.

Arguments:

`rhs`
: The buffer being assigned.

Output Operator

The output operator writes first the number of the element and a `':'` followed by the value of the element line by line to the stream `out`.

```
ostream& operator<<(ostream &out, const ABA_ARRAY<Type> &array)
```

Return Value:

A reference to the output stream.

Arguments:

`out`
: The output stream.

`array`
: The array being output.

Subscript Operator

```
Type& ABA_ARRAY<Type>::operator[](int i)
```

Return Value:

The `i`-th element of the array.

Arguments:

`i`
: The element being accessed.

Subscript Operator

The operator `[]` is overloaded for constant use.

```
const Type& ABA_ARRAY<Type>::operator[](int i) const
```

copy

The function `copy()` copies all elements of `rhs`. The difference to the operator `=` is that also copying between arrays of different size is allowed. If necessary the array on the left hand side is reallocated.

```
void ABA_ARRAY<Type>::copy(const ABA_ARRAY<Type> &rhs)
```

Arguments:

`rhs`
: The array being copied.

copy

This version of the function `copy()` copies the elements `rhs[l]`, `rhs[l+1]`,..., `rhs[r]` into the components `0`,...,`r-l` of the array. If the size of the array is smaller than `r-l+1` storage is reallocated.

```
void ABA_ARRAY<Type>::copy(const ABA_ARRAY<Type> &rhs, int l, int r)
```

Arguments:

`rhs`
: The array that is partially copied.

`l`
: The first element being copied.

`r`
: the last element being copied.

leftShift

The function `leftShift()` removes the components listed in `ind` by shifting the remaining components to the left. Memory management of the removed components must be carefully implemented by the user of this function to avoid memory leaks.

```
void ABA_ARRAY<Type>::leftShift(ABA_BUFFER<int> &ind)
```

Arguments:

`ind`
: The compenents being removed from the array.

leftShift

This version of the function `leftShift()` removes all components `i` with `marked[i]==true` from the array by shifting the other components to the left.

```
void ABA_ARRAY<Type>::leftShift(ABA_ARRAY<bool> &remove)
```

Arguments:

`remove`
: The marked components are removed from the array.

set

The function `set()` assigns the same value to a subset of the components of the array.

```
void ABA_ARRAY<Type>::set(int l, int r, Type val)
```

Arguments:

`l`
: The first component the value is assigned.

`r`
: The last component the value is assigned.

`val`
: The new value of these components.

set

This version of the function `set()` initializes all components of the array with the same value.

```
void ABA_ARRAY<Type>::set(Type val)
```

Arguments:

`val`
: The new value of all components.

size

```
int ABA_ARRAY<Type>::size() const
```

Return Value:

The length of the array.

realloc

The length of an array can be changed with the function `realloc()`. If the array is enlarged all elements of the old array are copied and the values of the additional new elements are undefined. If the array is shortened only the first `newSize` elements are copied.

```
void ABA_ARRAY<Type>::realloc(int newSize)
```

Arguments:

`newSize`
: The new length of the array.

realloc

The function `realloc()` is overloaded such that also an initialization with a new value of the elements of the array after reallocation is possible.

```
void ABA_ARRAY<Type>::realloc(int newSize, Type init)
```

Arguments:

`newSize`
: The new length of the array.

`init`
: The new value of all components of the array.

6.4.2 ABA_BUFFER

Often we need a data structure for buffering information. This class implements such a buffer by an array and storing the number of already buffered elements. If the initial size of the buffer turns out to be too small, then the buffer can be reallocated.

```
template <class Type> class ABA_BUFFER : public ABA_ABACUSROOT {
  public:
    ABA_BUFFER(ABA_GLOBAL *glob, int size);
    ABA_BUFFER(const ABA_BUFFER<Type> &rhs);
    ~ABA_BUFFER();
    const ABA_BUFFER<Type>& operator=(const ABA_BUFFER<Type>& rhs);
    friend ostream& operator<<(ostream& out, const ABA_BUFFER<Type> &buffer);
    Type& operator[](int i);
    const Type& operator[](int i) const;
    int size() const;
    int number() const;
    bool full() const;
    bool empty() const;
    void push(Type item);
    Type pop();
    void clear();
    void leftShift(ABA_BUFFER<int> &ind);
    void realloc (int newSize);
};
```

Constructor

The constructor generates an empty buffer.

```
ABA_BUFFER<Type>::ABA_BUFFER(ABA_GLOBAL *glob, int size)
```

Arguments:

`glob`

The corresponding global object.

`size`

The size of the buffer.

Copy Constructor

```
ABA_BUFFER<Type>::ABA_BUFFER(const ABA_BUFFER<Type> &rhs)
```

Arguments:

`rhs`

The buffer being copied.

Destructor

```
ABA_BUFFER<Type>::~ABA_BUFFER()
```

Assignment Operator

The assignment operator is only allowed between buffers having equal size.

```
const ABA_BUFFER<Type>& ABA_BUFFER<Type>::operator=(const ABA_BUFFER<Type>& rhs)
```

Return Value:

A reference to the buffer on the left hand side of the assignment operator.

Arguments:

`rhs`
: The buffer being assigned.

Output Operator

The output operator writes all buffered elements line by line to an output stream in the format *number*: *value*.

```
ostream& operator<<(ostream &out, const ABA_BUFFER<Type> &buffer)
```

Return Value:

A reference to the stream the buffer is written to.

Arguments:

`out`
: The output stream.

`buffer`
: The buffer being output.

Subscript Operator

The operator `[]` can be used to access an element of the buffer. It is only allowed to access buffered elements.

```
Type& ABA_BUFFER<Type>::operator[](int i)
```

Return Value:

The `i`-th element of the buffer.

Arguments:

`i`
: The number of the component which should be returned.

Subscript Operator

The operator `[]` is overloaded that it can be also used to access elements of constant buffers.

```
const Type& ABA_BUFFER<Type>::operator[](int i) const
```

size

```
int ABA_BUFFER<Type>::size() const
```

Return Value:

The maximal number of elements which can be stored in the buffer.

number

```
int ABA_BUFFER<Type>::number() const
```

Return Value:

The number of buffered elements.

full

```
bool ABA_BUFFER<Type>::full() const
```

Return Value:

`true`
: If no more elements can be inserted into the buffer,

`false`
: otherwise.

empty

```
bool ABA_BUFFER<Type>::empty() const
```

Return Value:

`true`
: If no items are buffered,

`false`
: otherwise.

push

The function `push()` inserts an item into the buffer. It is a fatal error to perform this operation if the buffer is full.

```
void ABA_BUFFER<Type>::push(Type item)
```

Arguments:

`item`
: The item that should be inserted into the buffer.

pop

The function `pop()` removes and returns the last inserted item from the buffer. It is a fatal error to perform this operation on an empty buffer.

```
Type ABA_BUFFER<Type>::pop()
```

Return Value:

The last item that has been inserted into the buffer.

clear

The function `clear()` sets the number of buffered items to 0 such that the buffer is empty.

```
void ABA_BUFFER<Type>::clear()
```

leftShift

The function `leftShift()` removes the components listed in the buffer `ind` by shifting the remaining components to the left. The values stored in `ind` have to be upward sorted. Memory management of the removed components must be carefully implemented by the user of this function to avoid memory leaks.

```
void ABA_BUFFER<Type>::leftShift(ABA_BUFFER<int> &ind)
```

Arguments:

`ind`

The numbers of the components being removed.

realloc

The length of a buffer can be changed with the function `realloc()`. If the size of the buffer is increased all buffered elements are copied. If the size is decreased the number of buffered elements is updated if necessary.

```
void ABA_BUFFER<Type>::realloc(int newSize)
```

Arguments:

`newSize`

The new length of the buffer.

6.4.3 ABA_LISTITEM

We call the basic building block of a linked list an *item* that is implemented by the class `ABA_LISTITEM`. A `ABA_LISTITEM` stores a copy of the inserted element and a pointer to its successor.

```
template<class Type> class ABA_LISTITEM : public ABA_ABACUSROOT {
  friend class ABA_LIST<Type>;
  public:
    ABA_LISTITEM (const Type &elem, ABA_LISTITEM<Type> *succ);
    friend ostream& operator<<(ostream &out, const ABA_LISTITEM<Type> &item);
    Type elem() const;
    ABA_LISTITEM<Type> *succ() const;
};
```

Constructor

```
ABA_LISTITEM<Type>::ABA_LISTITEM(const Type &elem, ABA_LISTITEM<Type> *succ)
```

Arguments:

`elem`

A copy of `elem` becomes the element of the list item.

`succ`

A pointer to the successor of the item in the list.

Output Operator

```
ostream& operator<<(ostream &out, const ABA_LISTITEM<Type> &item)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`item`

The list item being output.

elem

```
Type ABA_LISTITEM<Type>::elem() const
```

Return Value:

The element of the item.

succ

```
ABA_LISTITEM<Type> * ABA_LISTITEM<Type>::succ() const
```

Return Value:

The successor of the item in the list.

6.4.4 ABA_LIST

The following sections implement a template for a linked linear list. Two classes are required for the representation of this data structure. The first one `ABA_LISTITEM` forms the basic building block of the list storing an element and a pointer to the next item of the list, the second one is the `ABA_LIST` itself.

```
template<class Type> class ABA_LIST : public ABA_ABACUSROOT {
  friend class ABA_LISTITEM<Type>;
  public:
    ABA_LIST(ABA_GLOBAL *glob);
    ~ABA_LIST();
    friend ostream& operator<<(ostream&, const ABA_LIST<Type> &list);
    void appendHead(const Type &elem);
    void appendTail(const Type &elem);
    int extractHead(Type &elem);
    int firstElem(Type& elem) const;
    bool empty() const;

 private:
    ABA_LIST(const ABA_LIST &rhs);
    const ABA_LIST<Type>& operator=(const ABA_LIST<Type>& rhs);
};
```

forAllListElem

The iterator `forAllListElem` assigns to `Type e` all elements in the list beginning with the first element. Deletions of elements in the list during the application of this iterator can cause an error.

```
#define forAllListElem(L, item, e) \
  for((item = (L).first()) ? (e = (item)->elem()) : (e = e); item !=0; \
    (item = (item)->succ()) ? (e = (item)->elem()):(e = e))
```

Arguments:

`L`

The list that should be iterated (`ABA_LIST<Type>`).

`item`

An auxilliary pointer to a list item (`ABA_LISTITEM<Type> *`).

`e`

The elements in the list are assigned to this variable (`Type`).

Constructor

The constructor initializes the list with the empty list.

```
ABA_LIST<Type>::ABA_LIST(ABA_GLOBAL *glob)
```

Arguments:

`glob`

A pointer to the corresponding global object.

Destructor

The destructor deallocates the memory of all items in the list.

```
ABA_LIST<Type>::~ABA_LIST()
```

Output Operator

The output operator writes all items of the list on an output stream.

```
ostream& operator<<(ostream &out, const ABA_LIST<Type> &list)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`list`

The list being output.

appendHead

The function `appendHead()` adds an element at the front of the list.

```
void ABA_LIST<Type>::appendHead(const Type &elem)
```

Arguments:

`elem`
: The element being appended.

appendTail

The function `appendTail()` adds an element at the end of the list.

```
void ABA_LIST<Type>::appendTail(const Type &elem)
```

Arguments:

`elem`
: The element being appended.

extractHead

The function `extractHead()` assigns to `elem` the first element in the list and removes it from the list.

```
int ABA_LIST<Type>::extractHead(Type &elem)
```

Return Value:

`0`
: If the operation can be be executed successfully.

`1`
: If the list is empty.

Arguments:

`elem`
: If the list is nonemty, the first element is assigned to `elem`.

firstElem

The function `firstElem()` assign `elem` the first element as the function `extractHead()` but does not remove this element from the list.

```
int ABA_LIST<Type>::firstElem(Type &elem) const
```

Return Value:

`0`
: If the operation can be be executed successfully.

`1`
: If the list is empty.

Arguments:

`elem`
: If the list is nonemty, the first element is assigned to `elem`.

empty

```
bool ABA_LIST<Type>::empty() const
```

Return Value:

`true`
: If no element is contained in the list,

`false`
: otherwise.

6.4.5 ABA_DLISTITEM

We call the basic building block of a doubly linked list an *item*, which is implemented by the class `ABA_DLISTITEM`. A `ABA_DLISTITEM` stores a copy of the inserted element and has pointers to its predecessor and its successor.

```
template<class Type> class ABA_DLISTITEM : public ABA_ABACUSROOT {
  public:
    ABA_DLISTITEM (const Type &elem,
                   ABA_DLISTITEM<Type> *pred,
                   ABA_DLISTITEM<Type> *succ);
    friend ostream& operator<<(ostream &out, const ABA_DLISTITEM<Type> &item);
    Type elem() const;
    ABA_DLISTITEM<Type> *succ() const;
    ABA_DLISTITEM<Type> *pred() const;
};
```

Constructor

```
ABA_DLISTITEM<Type>::ABA_DLISTITEM (const Type &elem, ABA_DLISTITEM<Type> *pred,
                                    ABA_DLISTITEM<Type> *succ)
```

Arguments:

`elem`
: The element of the item.

`pred`
: A pointer to the previous item in the list.

`succ`
: A pointer to the next item in the list.

Output Operator

```
ostream& operator<<(ostream &out, const ABA_DLISTITEM<Type> &item)
```

Return Value:

A reference to the output stream.

Arguments:

`out`
: The output stream.

`item`
: The list item being output.

elem

```
Type ABA_DLISTITEM<Type>::elem() const
```

Return Value:

The element stored in the item.

succ

```
ABA_DLISTITEM<Type>* ABA_DLISTITEM<Type>::succ() const
```

Return Value:

A pointer to the successor of the item in the list.

pred

```
ABA_DLISTITEM<Type>* ABA_DLISTITEM<Type>::pred() const
```

Return Value:

A pointer to the predecessor of the item in the list.

6.4.6 ABA_DLIST

The class `ABA_DLIST` implements a doubly linked linear list.

```
template<class Type> class ABA_DLIST : public ABA_ABACUSROOT {
  public:
    ABA_DLIST(ABA_GLOBAL *glob);
    ~ABA_DLIST();
    friend ostream& operator<<(ostream&, const ABA_DLIST<Type> &list);
    void append(const Type &elem);
    int extractHead(Type &elem);
    int removeHead();
    void remove(const Type &elem);
    bool empty() const;
    int firstElem(Type& elem) const;

 private:
    ABA_DLIST(const ABA_DLIST &rhs);
    const ABA_DLIST<Type>& operator=(const ABA_DLIST<Type>& rhs);
};
```

forAllDListElem

The iterator `forAllDListElem` assigns to `Type e` all elements in the list beginning with the first element. The additional parameter `item` has to be of type `ABA_DLISTITEM<Type>*`. Deletions of elements in the list during the application of this iterator can cause an error.

```
#define forAllDListElem(L, item, e) \
  for((item = (L).first()) ? (e = (item)->elem()) : (e = e); item !=0; \
    (item = (item)->succ()) ? (e = (item)->elem()):(e = e))
```

Arguments:

L

The list that should be iterated (`ABA_DLIST<Type>`).

`item`

An auxilliary pointer to a list item (`ABA_DLISTITEM<Type> *`).

`e`

The elements in the list are assigned to this variable (`Type`).

Constructor

The constructor for an empty list.

```
ABA_DLIST<Type>::ABA_DLIST(ABA_GLOBAL *glob)
```

Destructor

The destructor deallocates the memory of all items in the list.

```
ABA_DLIST<Type>::~ABA_DLIST()
```

Output Operator

The output operator writes all elements of the `list` on an output stream.

```
ostream& operator<<(ostream &out, const ABA_DLIST<Type> &list)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`list`

The list being output.

append

The function `append()` adds an element at the end of the list.

```
void ABA_DLIST<Type>::append(const Type &elem)
```

Arguments:

`elem`

The element being appended.

extractHead

The function `extractHead()` assigns to `elem` the first element in the list and removes it from the list.

```
int ABA_DLIST<Type>::extractHead(Type &elem)
```

Return Value:

`0`
: If the operation can be executed successfully.

`1`
: If the list is empty.

Arguments:

`elem`
: If the list is nonemty, the first element is assigned to `elem`.

removeHead

If the list is non-empty, the function `removeHead()` removes the head of the list.

```
int ABA_DLIST<Type>::removeHead()
```

Return Value:

`0`
: If the list is non-empty before the function is called,

`1`
: otherwise.

remove

This version of the function `remove()` scans the list for an item with element `elem` beginning at the first element of the list. The first matching item is removed from the list.

```
void ABA_DLIST<Type>::remove(const Type &elem)
```

Arguments:

`elem`
: The element which should be removed.

empty

```
bool ABA_DLIST<Type>::empty() const
```

Return Value:

`true`
: If no element is contained in the list,

`false`
: otherwise.

firstElem

The function `firstElem()` retrieves the first element of the list.

```
int ABA_DLIST<Type>::firstElem(Type &elem) const
```

Return Value:

`0`
: If the list is not empty,

`1`
: otherwise.

Arguments:

`elem`
: Stores the first element of the list after the function call if the list is not empty.

6.4.7 ABA_RING

The template `ABA_RING` implements a bounded circular list with the property that if the list is full and an element is inserted the oldest element of the ring is removed. With this implementation single elements cannot be removed, but the whole `ABA_RING` can be cleared.

```
template <class Type> class ABA_RING : public ABA_ABACUSROOT {
  public:
    ABA_RING(ABA_GLOBAL *glob, int size);
    friend ostream &operator<<(ostream &out, const ABA_RING<Type> &ring);
    Type& operator[](int i);
    const Type& operator[](int i) const;
    void insert(Type elem);
    void clear();
    int  size() const;
    int  number() const;
    Type oldest() const;
    int  oldestIndex() const;
    Type newest() const;
    int  newestIndex() const;
    int  previous(int i, Type &p) const;
    bool empty() const;
    bool filled() const;
    void realloc(int newSize);
};
```

Constructor

```
ABA_RING<Type>::ABA_RING(ABA_GLOBAL *glob, int size)
```

Arguments:

`glob`
: A pointer to the corresponding global object.

`size`
: The length of the ring.

Output Operator

The output operator writes the elements of the ring to an output stream starting with the oldest element in the ring.

```
ostream &operator<<(ostream &out, const ABA_RING<Type> &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The ring being output.

Subscript Operator

```
Type& ABA_RING<Type>::operator[](int i)
```

Return Value:

The `i`-th element of the ring. The operation is undefined if no element has been inserted in the `i`-th position so far.

Arguments:

`i`

The element being accessed.

Subscript Operator

The operator `[]` is overloaded for constant use.

```
const Type& ABA_RING<Type>::operator[](int i) const
```

insert

The function `insert()` inserts a new element into the ring. If the ring is already full, this operation overwrites the oldest element in the ring.

```
void ABA_RING<Type>::insert(Type elem)
```

Arguments:

`elem`

The element being inserted.

clear

The function `clear()` empties the ring.

```
void ABA_RING<Type>::clear()
```

size

```
int ABA_RING<Type>::size() const
```

Return Value:

The size of the ring.

number

```
int ABA_RING<Type>::number() const
```

Return Value:

The current number of elements in the ring.

oldest

```
Type ABA_RING<Type>::oldest() const
```

Return Value:

The oldest element in the ring. The result is undefined, if the ring is empty.

oldestIndex

```
int ABA_RING<Type>::oldestIndex() const
```

Return Value:

The index of the oldest element in the ring. The result is undefined, if the ring is empty.

newest

```
Type ABA_RING<Type>::newest() const
```

Return Value:

The newest element in the ring. The result is undefined if the ring is empty.

newestIndex

```
int ABA_RING<Type>::newestIndex() const
```

Return Value:

The index of the newest element in the ring. The result is undefined if the ring is empty.

previous

The function `previous()` can be used to access any element between the oldest and newest inserted element.

```
int ABA_RING<Type>::previous(int i, Type &p) const
```

Return Value:

`0`

If there are enough elements in the ring such that the element `i` entries before the newest one could be accessed,

`1`

otherwise.

Arguments:

`i`

The element `i` elements before the newest element is retrieved. If `i` is 0, then the function retrieves the newest element.

`p`

Contains the `i`-th element before the newest one in a successful call.

empty

```
bool ABA_RING<Type>::empty() const
```

Return Value:

`true`

If no element is contained in the ring,

`false`

otherwise.

filled

```
bool ABA_RING<Type>::filled() const
```

Return Value:

`true`

If the `ABA_RING` is completely filled up,

`false`

otherwise.

realloc

The function `realloc()` changes the length of the ring.

```
void ABA_RING<Type>::realloc(int newSize)
```

Arguments:

`newSize`

The new length of the ring. If the ring decreases below the current number of elements in the ring, then the `newSize` newest elements stay in the ring.

6.4.8 ABA_BSTACK

A stack is a data structure storing a set of elements. Following the last-in first-out (LIFO) principle the access to or the deletion of an element is restricted to the most recently inserted element.

In order to provide an efficient implementation this stack is "bounded", i.e., the number of elements which can be inserted is limited. However, a reallocation can be performed if required.

```
template <class Type> class ABA_BSTACK : public ABA_ABACUSROOT {
  public:
    ABA_BSTACK(ABA_GLOBAL *glob, int size);
    friend ostream& operator<<(ostream& out, const ABA_BSTACK<Type> &rhs);
    int size() const;
    int tos() const;
    bool empty() const;
    bool full() const;
    void push(Type item);
    Type top() const;
    Type pop();
    void realloc (int newSize);
  };
```

Constructor

The constructor initializes an empty stack.

`ABA_BSTACK<Type>::ABA_BSTACK(ABA_GLOBAL *glob, int size)`

Arguments:

`glob`

A pointer to the corresponding global object.

`size`

The maximal number of elements the stack can store.

Output Operator

The output operator writes the numbers of all stacked elements and the elements line by line on an output stream.

`ostream& operator<<(ostream &out, const ABA_BSTACK<Type> &rhs)`

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The stack being output.

size

`int ABA_BSTACK<Type>::size() const`

Return Value:

The maximal number of elements which can be inserted into the stack.

tos

```
int ABA_BSTACK<Type>::tos() const
```

Return Value:

The top of the stack, i.e., the number of the next free component of the stack. This is also the number of elements currently contained in the stack since the first element is inserted in position 0.

empty

```
bool ABA_BSTACK<Type>::empty() const
```

Return Value:

`true`

If there is no element in the stack,

`false`

otherwise.

full

```
bool ABA_BSTACK<Type>::full() const
```

Return Value:

`true`

If the maximal number of elements has been inserted in the stack,

`false`

otherwise.

push

The function `push()` adds an element to the stack. It is a fatal error to insert an element if the stack is full.

```
void ABA_BSTACK<Type>::push(Type item)
```

Arguments:

`item`

The element added to the stack.

top

The function `top()` accesses the last element pushed on the stack without removing it. It is an error to perform this operation on an empty stack.

```
Type ABA_BSTACK<Type>::top() const
```

Return Value:

The last element pushed on the stack.

pop

The function `pop()` accesses like `top()` the last element pushed on the stack and removes in addition this item from the stack. It is an error to perform this operation on an empty stack.

`Type ABA_BSTACK<Type>::pop()`

Return Value:

The last element pushed on the stack.

realloc

The function `realloc()` changes the maximal number of elements of the stack.

`void ABA_BSTACK<Type>::realloc(int newSize)`

Arguments:

`newSize`

The new maximal number of elements on the stack. If `newSize` is less than the current number of elements in the stack, then the `newSize` oldest element are contained in the stack after the reallocation.

6.4.9 ABA_BHEAP

A heap is the representation of a binary tree by an array `a`. The root of the tree is associated with `a[0]`. If a node corresponds to `a[i]`, then its left son corresponds to `a[2*i+1]` and its right son to `a[2*i+2]`. This implicit binary tree is completely filled except possibly its highest level. Every item in the heap has to fulfil the heap property, i.e., its key has to be less than or equal than the keys of both sons.

This template class implements a heap with a fixed maximal size, however a reallocation can be performed if required.

The operations `insert()`, `extractMin()` require $O(\log n)$ running time if n elements are contained in the heap. The operation `getMin()` can even be executed in constant time. A heap can also be constructed from an `ABA_BUFFER` of n elements which requires a running time of $O(n)$ only.

The order of the elements in the heap is given by keys which are inserted together with each element of the heap. The class `Key` must be from an ordered type. Given two objects `k1` and `k2` of type `Key` then `k1` has higher priority if the expression `k1 < k2` holds.

```
template<class Type, class Key> class ABA_BHEAP : public ABA_ABACUSROOT {
  public:
    ABA_BHEAP(ABA_GLOBAL *glob, int size);
    ABA_BHEAP(ABA_GLOBAL *glob,
              const ABA_BUFFER<Type> &elems,
              const ABA_BUFFER<Key> &keys);
    friend ostream& operator<<(ostream& out, const ABA_BHEAP<Type, Key>& rhs);
    void insert(Type elem, Key key);
    Type getMin() const;
    Key getMinKey() const;
    Type extractMin();
    void clear();
    int size() const;
    int number() const;
    bool empty() const;
    void realloc(int newSize);
};
```

Constructor

```
ABA_BHEAP<Type, Key>::ABA_BHEAP(ABA_GLOBAL *glob, int size)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`size`

The maximal number of elements which can be stored.

Constructor

```
ABA_BHEAP<Type, Key>::ABA_BHEAP(ABA_GLOBAL *glob,
                                const ABA_BUFFER<Type> &elems,
                                const ABA_BUFFER<Key> &keys)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`elem`

A `ABA_BUFFER` wich contains the elements.

`elem`

A `ABA_BUFFER` wich contains the keys.

Output Operator

The output operator writes the elements of the heap together with their keys on an output stream.

```
ostream& operator<<(ostream& out, const ABA_BHEAP<Type, Key>& rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The heap being output.

insert

The function `insert()` inserts an item with a key into the heap. It is a fatal error to perform this operation if the heap is full.

```
void ABA_BHEAP<Type, Key>::insert(Type elem, Key key)
```

Arguments:

`elem`

The element being inserted into the heap.

`key`

The key of this element.

getMin

```
Type ABA_BHEAP<Type, Key>::getMin() const
```

Return Value:

The minimum element of the heap. This operation must not be performed if the heap is empty.

getMinKey

```
Key ABA_BHEAP<Type, Key>::getMinKey() const
```

Return Value:

The key of the minimum element of the heap. This operation must not be performed if the heap is empty.

extractMin

The function `extractMin()` accesses and removes the minimum element from the heap.

```
Type ABA_BHEAP<Type, Key>::extractMin()
```

Return Value:

The minimum element of the heap.

clear

The function `clear()` empties the heap.

```
void ABA_BHEAP<Type, Key>::clear()
```

size

```
int ABA_BHEAP<Type, Key>::size() const
```

Return Value:

The maximal number of elements which can be stored in the heap.

number

```
int ABA_BHEAP<Type, Key>::number() const
```

Return Value:

The number of elements in the heap.

empty

```
bool ABA_BHEAP<Type, Key>::empty() const
```

Return Value:

`true`
: If there are no elements in the heap,

`false`
: otherwise.

realloc

The function `realloc()` changes the size of the heap.

```
void ABA_BHEAP<Type, Key>::realloc(int newSize)
```

Arguments:

`newSize`

The new maximal number of elements in the heap.

6.4.10 ABA_BPRIOQUEUE

A priority queue is a data structure storing a set of elements. Each element has a key which must be an ordered data type. The most important operations are the insertion of an element, the determination of the element having the minimal key, and the deletion of the element having minimal key.

Since the priority queue is implemented by a heap (class `ABA_BHEAP`) the insertion of a new element and the deletion of the minimal element require $O(\log n)$ time if n elements are stored in the priority queue. The element having minimal key can be determined in constant time.

To provide an efficient implementation the priority queue is bounded, i.e., the maximal number of elements is an argument of the constructor. However, if required, later a reallocation can be performed.

```
template<class Type, class Key>
class ABA_BPRIOQUEUE : public ABA_ABACUSROOT {
  public:
    ABA_BPRIOQUEUE(ABA_GLOBAL *glob, int size);
    void insert(Type elem, Key key);
    int getMin(Type &min) const;
    int getMinKey(Key &minKey) const;
    int extractMin(Type &min);
    void clear();
    int size() const;
    int number() const;
    void realloc(int newSize);
```

Constructor

The constructor of an empty priority queue.

```
ABA_BPRIOQUEUE<Type, Key>::ABA_BPRIOQUEUE(ABA_GLOBAL *glob, int size)
```

Arguments:

`glob`

A pointer to the corresponding object.

`size`

The maximal number of elements the priority queue can hold without reallocation.

insert

The function `insert()` inserts an element in the priority queue.

```
void ABA_BPRIOQUEUE<Type, Key>::insert(Type elem, Key key)
```

Arguments:

`elem`

The element being inserted.

`key`

The key of the element.

getMin

The function `getMin()` retrieves the element with minimal key from the priority queue.

```
int ABA_BPRIOQUEUE<Type, Key>::getMin(Type &min) const
```

Return Value:

`0`

If the priority queue is non-empty,

`1`

otherwise.

Arguments:

`min`

If the priority queue is non-empty the minimal element is assigned to `min`.

getMinKey

The function `getMinKey()` retrieves the key of the minimal element in the priority queue.

```
int ABA_BPRIOQUEUE<Type, Key>::getMinKey(Key &minKey) const
```

Return Value:

`0`

If the priority queue is non-empty,

`1`

otherwise.

Arguments:

`minKey`

Holds after the call the key of the minimal element in the priority queue, if the queue is non-emtpy.

extractMin

The function `extractMin()` extends the function `getMin(min)` in the way that the minimal element is also removed from the priority queue.

```
int ABA_BPRIOQUEUE<Type, Key>::extractMin(Type& min)
```

Return Value:

`0`

If the priority queue is non-empty,

`1`

otherwise.

Arguments:

`min`

If the priority queue is non-empty the minimal element is assigned to `min`.

clear

The function `clear()` makes the priority queue empty.

```
void ABA_BPRIOQUEUE<Type, Key>::clear()
```

size

```
int ABA_BPRIOQUEUE<Type, Key>::size() const
```

Return Value:

The maximal number of elements which can be stored in the priority queue.

number

```
int ABA_BPRIOQUEUE<Type, Key>::number() const
```

Return Value:

The number of elements stored in the priority queue.

realloc

The function `realloc()` increases the size of the priority queue. It is not allowed to decrease the size of the priority queue. In this case an error message is output and the program stops.

```
void ABA_BPRIOQUEUE<Type, Key>::realloc(int newSize)
```

Arguments:

`newSize`

The new size of the priority queue.

6.4.11 ABA_HASH

This data structure stores a set of items and provides as central functions the insertion of a new item, the search for an item, and the deletion of an item.

Each item is associated with a key. The set of all possible keys is called the universe. A hash table has a fixed size n. A hash function assigns to each key of the universe a number in $\{0, \ldots, n-1\}$, which we denote slot. If an item is inserted in the hash table, then it is stored in the component of the array associated with its slot. Usually, `n` is much smaller than the cardinality of the universe. Hence, it can happen that two elements are mapped to the same slot. This is called a collision. In order to resolve collisions, each slot of the hash table does not store an item explicitly, but is the start of a linear list storing all items mapped to this slot.

This template implements a hash table where collisions are resolved by chaining. Currently hash functions for keys of type `int` and `ABA_STRING` are implemented. If you want to use this data structure for other types (e.g., `YOURTYPE`), you should derive a class from the class `ABA_HASH` and define a hash function `int hf(YOURTYPE key)`.

```
template <class KeyType, class ItemType>
class ABA_HASH : public ABA_ABACUSROOT {
  public:
    ABA_HASH(ABA_GLOBAL *glob, int size);
    ~ABA_HASH();
    friend ostream &operator<<(ostream &out,
                               const ABA_HASH<KeyType, ItemType> &hash);
```

```
    void insert(const KeyType &newKey, const ItemType &newItem);
    void overWrite(const KeyType &newKey, const ItemType &newItem);
    ItemType *find(const KeyType &key);
    bool find(const KeyType &key, const ItemType &item);
    ItemType *initializeIteration(const KeyType &key);
    ItemType *next(const KeyType &key);
    int remove(const KeyType &key);
    int remove(const KeyType &key, const ItemType &item);
    int size() const;
    int nCollisions() const;
    void resize(int newSize);

  private:
    ABA_HASH(const ABA_HASH &rhs);
    ABA_HASH &operator=(const ABA_HASH &rhs);
};
```

Constructor

```
ABA_HASH<KeyType, ItemType>::ABA_HASH(ABA_GLOBAL *glob, int size)
```

Arguments:

`glob`

A pointer to the corresponding global object.

`size`

The size of the hash table.

Destructor

```
ABA_HASH<KeyType, ItemType>::~ABA_HASH()
```

Output Operator

The output operator writes row by row all elements stored in the list associated with a slot on an output stream.

```
ostream &operator<<(ostream &out, const ABA_HASH<KeyType, ItemType> &hash)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The hash table being output.

insert

The function `insert()` adds an item to the hash table.

```
void ABA_HASH<KeyType, ItemType>::insert(const KeyType &key,
                                         const ItemType &item)
```

Arguments:

`key`
: The key of the new item.

`item`
: The item being inserted.

overWrite

The function `overWrite()` performs a regular `insert()` if there is no item with the same key in the hash table, otherwise the item is replaced by the new item.

```
void ABA_HASH<KeyType, ItemType>::overWrite(const KeyType &key,
                                            const ItemType &item)
```

Arguments:

`key`
: The key of the new item.

`item`
: The item being inserted.

find

The function `find()` looks for an item in the hash table with a given key.

```
ItemType * ABA_HASH<KeyType, ItemType>::find(const KeyType &key)
```

Return Value:

A pointer to an item with the given key, or a 0-pointer if there is no item with this key in the hash table. If there is more than one item in the hash table with this key, a pointer to the first item found is returned.

Arguments:

`key`
: The key of the searched item.

find

This version of the function `find()` checks if a prespecified item with a prespecified key is contained in the hash table.

```
bool ABA_HASH<KeyType, ItemType>::find (const KeyType &key, const ItemType &item)
```

Return Value:

`true`
: If there is an element `(key, item)` in the hash table,

`false`

otherwise.

Arguments:

`key`

The key of the item.

`item`

The searched item.

initializeIteration

The functions `initializeIteration()` and `next()` can be used to iterate through all items stored in the hash table having the same key. The function `initializeIteration()` retrieves the first item. The function `next()` can be used to go to the next item having this key.

```
template <class KeyType, class ItemType>
ItemType *ABA_HASH<KeyType, ItemType>::initializeIteration(const KeyType &key)
```

Return Value:

A pointer to the first item found in the hash table having key `key`, or 0 if there is no such item.

Arguments:

`key`

The key of the items through which we want to iterate.

next

The function `next()` can be used to go to the next item in the hash table with key `key`. Before the first call of `next()` for a certain can the iteration has to be initialized by calling `initializeItaration()`.

Note, the function `next()` gives you the next item having `key` key but not the next item in the linked list starting in a slot of the hash table.

```
template <class KeyType, class ItemType>
ItemType *ABA_HASH<KeyType, ItemType>::next(const KeyType &key)
```

Return Value:

A pointer to the next item having key `key`, or 0 if there is no more item with this key in the hash table.

Arguments:

`key`

The key of the items through which we want to iterate.

remove

The function `remove()` removes the first item with a given key from the hash table.

```
int ABA_HASH<KeyType, ItemType>::remove(const KeyType &key)
```

Return Value:

`0`
: If an item with the key is found.

`1`
: If there is no item with this key.

Arguments:

`key`
: The key of the item that should be removed.

remove

This version of the function `remove()` removes the first item with a given key and a prespecified element from the hash table.

```
int ABA_HASH<KeyType, ItemType>::remove(const KeyType &key, const ItemType &item)
```

Return Value:

`0`
: If an item with the key is found.

`1`
: If there is no item with this key.

Arguments:

`key`
: The key of the item that should be removed.

`item`
: The item which is searched.

size

```
int ABA_HASH<KeyType, ItemType>::size() const
```

Return Value:

The length of the hash table.

nCollisions

```
int ABA_HASH<KeyType, ItemType>::nCollisions() const
```

Return Value:

The number of collisions which occurred during all previous calls of the functions `insert()` and `overWrite()`.

resize

The function `resize()` can be used to change the size of the hash table.

```
void ABA_HASH<KeyType, ItemType>::resize(int newSize)
```

Arguments:

`newSize`
: The new size of the hash table (must be positive).

6.4.12 ABA_DICTIONARY

The data structure dictionary is a collection of items with keys. It provides the operations to insert pairs of keys and items and to look up an item given some key.

```
template <class KeyType, class ItemType>
class ABA_DICTIONARY : public ABA_ABACUSROOT {
  public:
    ABA_DICTIONARY(ABA_GLOBAL *glob, int size);
    friend ostream &operator<<(ostream &out,
                               const ABA_DICTIONARY<KeyType, ItemType> &rhs);
    void insert(const KeyType &key, const ItemType &item);
    ItemType *lookUp(const KeyType &key);

  private:
    ABA_DICTIONARY(const ABA_DICTIONARY<KeyType, ItemType> &rhs);
    const ABA_DICTIONARY &operator=(const ABA_DICTIONARY<KeyType, ItemType> &rhs);
};
```

Constructor

```
ABA_DICTIONARY<KeyType, ItemType>::ABA_DICTIONARY(ABA_GLOBAL *glob, int size)
```

Arguments:

`glob`
: A pointer to the corresponding global object.

`size`
: The size of the hash table implementing the dictionary.

Output Operator

The output operator writes the hash table implementing the dictionary on an output stream.

```
ostream &operator<<(ostream &out, const ABA_DICTIONARY<KeyType, ItemType> &rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`
: The output stream.

`rhs`
: The hash table being output.

insert

The function `insert()` adds the item together with a key to the dictionary.

```
void ABA_DICTIONARY<KeyType, ItemType>::insert(const KeyType &key,
                                               const ItemType &item)
```

Arguments:

`key`
: The key of the new item.

`item`
: The new item.

lookUp

```
ItemType* ABA_DICTIONARY<KeyType, ItemType>::lookUp(const KeyType &key)
```

Return Value:

A pointer to the item associated with `key` in the `ABA_DICTIONARY`, or 0 if there is no such item.

Arguments:

`key`
: The key of the searched item.

6.5 Tools

This section documents some tools for sorting objects, measuring time, and generating output.

6.5.1 ABA_SORTER

This class implements several functions for sorting arrays according to increasing keys. We encapsulate these functions in order to avoid name conflicts. Moreover, instead of local variables in the sorting functions we can provide within the class variables for swapping in order to speed up the sorting.

The sorting functions do not keep the elements of the array storing the keys in place but resort it in parallel with the array storing the items.

```
template <class ItemType, class KeyType>
class ABA_SORTER : public ABA_ABACUSROOT {
  public:
    ABA_SORTER (ABA_GLOBAL *glob);
    void quickSort(int n, ABA_ARRAY<ItemType> &items, ABA_ARRAY<KeyType> &keys);
    void quickSort(ABA_ARRAY<ItemType> &items, ABA_ARRAY<KeyType> &keys,
                   int left, int right);
    void heapSort(int n, ABA_ARRAY<ItemType> &items, ABA_ARRAY<KeyType> &keys);
```

Constructor

```
ABA_SORTER<ItemType, KeyType>::ABA_SORTER(ABA_GLOBAL *glob)
```

Arguments:

`glob`
: A pointer to the corresponding global object.

quickSort

The function `quickSort()` sorts the elements of an array of `n` items according to their keys. This function is very efficient for many practical applications. Yet, has a worst case running time of $O(n^2)$.

```
void ABA_SORTER<ItemType, KeyType>::quickSort(int n,
                                              ABA_ARRAY<ItemType> &items,
                                              ABA_ARRAY<KeyType> &keys)
```

Arguments:

`n`
: The number of elements being sorted.

`items`
: The items being sorted.

`keys`
: The keys of the sorted items.

quickSort

This version of the function `quickSort()` sorts an partial array.

```
void ABA_SORTER<ItemType, KeyType>::quickSort(ABA_ARRAY<ItemType> &items,
                                              ABA_ARRAY<KeyType> &keys,
                                              int left,
                                              int right)
```

Arguments:

`items`
: The items being sorted.

`keys`
: The keys of the items.

`left`
: The first item in the partial array being sorted.

`right`
: The last item in the partial array being sorted.

heapSort

The function `heapSort()` sorts an array of `n` items according to their keys. In many practical applications this function is inferior to `quickSort()`, although it has the optimal worst case running time of $O(n \log n)$.

```
void ABA_SORTER<ItemType, KeyType>::heapSort(int n,
                                             ABA_ARRAY<ItemType> &items,
                                             ABA_ARRAY<KeyType> &keys)
```

Arguments:

`n`
: The number of items being sorted.

`items`
: The items being sorted.

`keys`
: The keys of the items.

6.5.2 ABA_TIMER

This class implements a base class for timers measuring the CPU time (`class ABA_CPUTIMER`) and the wall-clock time (`class ABA_COWTIMER`).

```
class ABA_TIMER : public ABA_ABACUSROOT {
  public:
    friend ostream& operator<<(ostream& out, const ABA_TIMER& rhs);
    void start();
    void stop();
    void reset();
    bool running() const;
    long centiSeconds() const;
    long seconds() const;
    long minutes() const;
    long hours() const;
    bool exceeds(const ABA_STRING &maxTime) const;

  protected:
    virtual long theTime() const = 0;
    ABA_GLOBAL *glob_;

};
```

glob_

`ABA_GLOBAL *glob_`

A pointer to the corresponding global object.

Output Operator

The output operator writes the time in the format `hours:minutes:seconds.seconds/100` on an output stream.

```
ostream& operator<<(ostream& out, const ABA_TIMER& rhs)
```

Return Value:

A reference to the output stream.

Arguments:

`out`

The output stream.

`rhs`

The timer being output.

start

The timer is started with the function `start()`. For safety starting a running timer is an error.

```
void ABA_TIMER::start()
```

stop

The function `stop()` stops the timer and adds the difference between the current time and the starting time to the total time. Stopping a non-running timer is an error.

```
void ABA_TIMER::stop()
```

reset

The function `reset()` stops the timer and sets the `totalTime` to 0.

```
void ABA_TIMER::reset()
```

running

```
bool ABA_TIMER::running() const
```

Return Value:

`true`
: If the timer is running,

`false`
: otherwise.

centiSeconds

```
long ABA_TIMER::centiSeconds() const
```

Return Value:

The currently spent time in $\frac{1}{100}$-seconds. It is not necessary to stop the timer to get the correct time.

seconds

```
long ABA_TIMER::seconds() const
```

Return Value:

The currently spent time in seconds. It is not necessary to stop the timer to get the correct time. The result is rounded down to the next integer value.

minutes

```
long ABA_TIMER::minutes() const
```

Return Value:

The currently spent time in minutes. It is not necessary to stop the timer to get the correct time. The result is rounded down to the next integer value.

hours

```
long ABA_TIMER::hours() const
```

Return Value:

The currently spent time in hours. It is not necessary to stop the timer to get the correct time. The result is rounded down to the next integer value.

exceeds

```
bool ABA_TIMER::exceeds(const ABA_STRING &maxTime) const
```

Return Value:

`true`

If the currently spent time exceeds `maxTime`,

`false`

otherwise.

Arguments:

`maxTime`

A string of the form `[[h:]m:]s`, where `h` are the hours, `m` the minutes, and `s` the seconds. Hours and minutes are optional. `h` can be an arbitrary nonnegative integer, `s` and `m` have to be integers in $\{0, \ldots, 59\}$. If `m` or `s` are less than 10, then a leading 0 is allowed (e.g. 3:05:09).

theTime

The function `theTime()` is required for measuring the time difference between the time of the call and some base point (e.g., the program start). It is a pure virtual function because in derived classes different implementation for elapsed time and CPU time are required.

```
virtual long theTime() const = 0
```

Return Value:

The time since some base point (e.g., the program start for the cpu time) in $\frac{1}{100}$ seconds.

6.5.3 ABA_CPUTIMER

This class derived from `ABA_TIMER` implements a timer measuring the cpu time of parts of a program.

```
class ABA_CPUTIMER : public ABA_TIMER {
  public:
    ABA_CPUTIMER(ABA_GLOBAL *glob);
    ABA_CPUTIMER(ABA_GLOBAL *glob, long centiSeconds);
    virtual ~ABA_CPUTIMER();
};
```

Constructor

After the application of the constructor the timer is not running, i.e., to measure time it has to be started explicitly.

```
ABA_CPUTIMER::ABA_CPUTIMER(ABA_GLOBAL *glob)
```

Arguments:

`glob`

A pointer to a global object.

Constructor

This constructor initializes the total time of the timer. The timer is not running, too.

```
ABA_CPUTIMER::ABA_CPUTIMER(ABA_GLOBAL *glob, long centiSeconds)
```

Arguments:

`glob`

A pointer to a global object.

`centiSeconds`

The intial value of the total time in $\frac{1}{100}$ seconds.

Destructor (`virtual`)

```
ABA_CPUTIMER::~ABA_CPUTIMER()
```

6.5.4 ABA_COWTIMER

This class derived from `ABA_TIMER` implements a timer measuring the elpased time (clock-of-the-wall time) of parts of the program.

```
class ABA_COWTIMER : public ABA_TIMER {
  public:
    ABA_COWTIMER(ABA_GLOBAL *glob);
    ABA_COWTIMER(ABA_GLOBAL *glob, long secs);
    virtual ~ABA_COWTIMER();
};
```

Constructor

After the application of the constructor the timer is not running, i.e., to measure time it has to be started explicitly.

```
ABA_COWTIMER::ABA_COWTIMER(ABA_GLOBAL *glob)
```

Arguments:

`glob`

A pointer to a global object.

Constructor

This constructor initializes the total time of the timer. The timer is not running, too.

```
ABA_COWTIMER::ABA_COWTIMER(ABA_GLOBAL *glob, long centiSeconds)
```

Arguments:

`glob`

A pointer to a global object.

`centiSeconds`

The initial value of the timer in $\frac{1}{100}$ seconds.

Destructor (`virtual`)

```
ABA_COWTIMER::~ABA_COWTIMER()
```

Constructor

The constructor turns the output on and associates it with a "real" stream.

```
ABA_OSTREAM::ABA_OSTREAM(ostream &out, const char *logStreamName)
```

Arguments:

`out`

The "real" stream (usually `cout` or `cerr`.)

`logStreamName`

If `logStreamName` is not 0, then the output also directed to a log-file with this name. The default value of `logStreamName` is 0.

Destructor

```
ABA_OSTREAM::~ABA_OSTREAM()
```

Output Operators

We reimplement the output operator `<<` for all fundamental types, for `const char *`, and for some other classes listed below. If the output is turned on the operator of the base class `ostream` is called. If also the output to the logfile is turned on, we write the same message also to the log-file.

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(char o)
```

Return Value:

A reference to the output stream.

Arguments:

`o`

The item being output.

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(unsigned char o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(signed char o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(short o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(unsigned short o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(int o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(unsigned int o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(long o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(unsigned long o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(float o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(double o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(const char *o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(const ABA_STRING &o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(const ABA_TIMER &o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(const ABA_HISTORY &o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(const ABA_LP &o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(const ABA_LPVARSTAT &o)
```

Output Operators

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(const ABA_CSENSE &o)
```

Manipulator

A manipulator is a function having as argument a reference to an `ABA_OSTREAM` and returning an `ABA_OSTREAM`. Manipulators are used that we can call, e.g., the function `endl(o)` by just writing its name omitting brackets and the function argument.

```
ABA_OSTREAM& ABA_OSTREAM::operator<<(ABA_OSTREAM_MANIP m)
```

Return Value:

A reference to the output stream.

Arguments:

`m`

An output stream manipulator.

off

The function `off()` turns the output off.

```
void ABA_OSTREAM::off()
```

on

The function `on()` turns the output on.

```
void ABA_OSTREAM::on()
```

logOn

The function `logOn()` turns the output to the logfile on.

```
void ABA_OSTREAM::logOn()
```

logOn

This version of `logOn()` turns the output to the logfile on and sets the log-file to `logStreamName`.

```
void ABA_OSTREAM::logOn(const char *logStreamName)
```

Arguments:

`logStreamName`
: The name of the log-file.

logOff

The function `logOff()` turns the output to the logfile off.

```
void ABA_OSTREAM::logOff()
```

log

```
ofstream* ABA_OSTREAM::log() const
```

Return Value:

A pointer to the stream associated with the log-file.

setFormatFlag

The function `setFormatFlag()` can be used to set the format flags of the output stream and the log file similar to the function `ios::set()` of the iostream library. For a documentation of all possible flags we refer to the documentation of the GNU C++ Iostream Library.

```
#ifdef ABACUS_COMPILER_VISUAL_CPP
  void ABA_OSTREAM::setFormatFlag(long flag)
#else
  void ABA_OSTREAM::setFormatFlag(fmtflags flag)
#endif
```

Arguments:

`flag`
: The flag being set.

isOn

```
bool ABA_OSTREAM::isOn() const
```

Return Value:

`true`
: If the output is turned on,

`false`
: otherwise.

isLogOn

```
bool ABA_OSTREAM::isLogOn() const
```

Return Value:

`true`
: If the output to the logfile is turned on,

`false`
: otherwise.

flush

The function `flush()` flushes the output and the log stream buffers of the stream `o`. This function can be called via the manipulator `o << flush;`.

```
ABA_OSTREAM& flush(ABA_OSTREAM &o)
```

Return Value:

A reference to the output stream.

Arguments:

`o`
: An output stream.

endl

The function `endl()` writes an end of line to the output and log-file of the stream `o` and flushes both stream buffers. This function can be called via the manipulator `o << endl;`.

```
ABA_OSTREAM& endl(ABA_OSTREAM &o)
```

Return Value:

A reference to the output stream.

Arguments:

`o`
: An output stream.

_setWidth

The function `_setWidth()` sets the width of the field for the next output operation on the log and the output stream.

In most cases the manipulator `setWith` is more convenient to use.

```
ABA_OSTREAM& _setWidth(ABA_OSTREAM &o, int w)
```

Return Value:

A reference to the output stream.

Arguments:

`o`

An output stream.

`w`

The width of the field.

_setPrecision

The function `_setPrecision()` sets the precision for the output stream.

In most cases the manipulator `setPrecision` is more convenient to use.

```
ABA_OSTREAM& _setPrecision(ABA_OSTREAM &o, int p)
```

Return Value:

A reference to the output stream.

Arguments:

`o`

An output stream.

`p`

The precision.

setWidth

The function `setWidth` can be used for output streams of the class `ABA_OSTREAM` as the function `setw` for the class `ostream`, e.g.:

```
master_->out() << setw(10) << x << endl;
```

```
ABA_OSTREAM_MANIP_INT setWidth(int p)
```

Return Value:

A manipulator object with the function `_setWidth()`.

Arguments:

`p`

The width of the output field.

setPrecision

The function `setWidth` can be used for output streams of the class `ABA_OSTREAM` in the same way as the function `setprecision` for the class `ostream`, e.g.:

```
master_->out() << setprecision(10) << x << endl;
```

```
ABA_OSTREAM_MANIP_INT setPrecision(int p)
```

Return Value:

A manipulator object with the function `_setPrecision()`.

Arguments:

`p`

The precision for the output stream.

6.6 Preprocessor Flags

Table 6.1 summarizes all preprocessors flags the are relevant for ABACUS-users.

Flag	Description	See Section
ABACUS_LP_CPLEX22	LP-solver Cplex 2.2	2.4.1
ABACUS_LP_CPLEX30	LP-solver Cplex 3.0	2.4.1
ABACUS_LP_CPLEX40	LP-solver Cplex 4.0	2.4.1
ABACUS_OLD_INCLUDE	old include file search path of ABACUS	3.3
ABACUS_OLD_NAMES	old naming conventions of ABACUS	3.2
ABACUS_COMPILER_GCC	GNU C++ compiler	2.8
ABACUS_COMPILER_VISUAL_CPP	Visual C++ compiler	2.8
ABACUS_SOPLEX	LP-solver SoPlex	2.4.2
ABACUS_SYS_AIX	Operating System AIX	2.2
ABACUS_SYS_IRIX	Operating System IRIX	2.2
ABACUS_SYS_LINUX	Operating System Linux	2.2
ABACUS_SYS_OSF	Operating System OSF	2.2
ABACUS_SYS_HP	Operating System HP-UX	2.2
ABACUS_SYS_SUNOS4	Operating System SUNOS 4.*	2.2
ABACUS_SYS_SUNOS5	Operating System SUNOS 5.*	2.2
ABACUS_SYS_WINNT	Operating System Windows NT	2.2

Table 6.1: Preprocessor Flags.

Chapter 7

Warranty and Copyright

7.1 Warranty

All parts of ABACUS, including the software, the example, and the user's guide and reference manual, are distributed without any warranty. The entire risk of ABACUS is with its user.

7.2 Copyright

ABACUS may be used freely for non-commercial applications by universities and public research organizations. However, a valid license code is required. ABACUS may only be used on machines for which a license code has been assigned according to section 2.6. For commercial licenses please contact us directly.

Bibliography

[ASC95] INFORMATION PROCESSING SYSTEM Accredited Standards Committee, X3. *The ISO/ANSI C++ Draft*, 1995. http://www.cygnus.com/misc/wp/.

[Bay72] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[BCC93a] Egon Balas, Sebastian Ceria, and Gerard Cornuejols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58:295–324, 1993.

[BCC93b] Egon Balas, Sebastian Ceria, and Gerard Cornuejols. Solving mixed 0-1 programs by a lift-and-project method. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 232–242, 1993.

[BJN+97] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for huge integer programs. *Operations Research*, 1997. to appear.

[Boo94] G. Booch. *Object-oriented analysis and design with applications.* The Benjamin Cummings Publishing Company, Redwood City, California, 1994.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms.* MIT Press, Cambridge, 1990.

[Cpl94] Cplex. *Using the Cplex Callable Library and Cplex Mixed Integer Library.* Cplex Optimization, Inc, 1994.

[Cpl95] Cplex. *Using the Cplex Callable Library.* Cplex Optimization, Inc, 1995.

[ES92] M.A. Ellis and B. Stroustrup. *The annotated C++ reference manual.* Addison Wesley, Reading, Massachusetts, 1992.

[GS78] L.J. Guibas and R. Sedgewick. A diochromatic framework for balanced trees. In *Proceedings of the 19th annual symposium on foundations of computer science*, pages 8–21. IEEE Computer Society, 1978.

[HP93] Karla Hoffman and Manfred W. Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39:657–682, 1993.

[JRT94] Michael Jünger, Gerhardt Reinelt, and Stefan Thienel. Provably good solutions for the traveling salesman problem. *Zeitschrift für Operations Research*, 40:183–217, 1994.

[JRT95] Michael Jünger, Gerhardt Reinelt, and Stefan Thienel. Practical problem solving with cutting plane algorithms in combinatorial optimization. In Willian Cook, Lázló Lovász, and Paul Seymour, editors, *Combinatorial Optimization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 111–152. American Mathematical Society, 1995.

[KM90] T. Korson and J.D. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9):40–60, 1990.

[Knu93] Donald E. Knuth. *The Stanford GraphBase: a platform for combinatorial computing.* Addison-Wesley, Reading, Massachusetts, 1993.

[Lei95] Sebastian Leipert. Vbctool—a graphical interface for visualization of branch-and-cut algorithms. Technical report, Institut für Informatik, Universität zu Köln, 1995. http://www.informatik.uni-koeln.de/ls_juenger/projects/vbctool.html.

[PR91] Manfred W. Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[RF81] D.M. Ryan and B.A. Foster. An integer programming approach to scheduling. In A. Wren, editor, *Computer scheduling of public transport urban passenger vehicle and crew scheduling*, pages 269–280. North Holland, Amsterdam, 1981.

[Sav94] Martin W.P. Savelsbergh. Preprocessing and probing for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[Str93] B. Stroustrup. *The C++ programming language—2nd edition.* Addison-Wesley, Reading, Massachusetts, 1993.

[Thi95] Stefan Thienel. *ABACUS—A Branch-And-CUt System.* PhD thesis, Universität zu Köln, 1995.

[VBJN94] Pamela H. Vance, Cynthia Barnhart, Ellis J. Johnson, and George L. Nemhauser. Solving binary cutting stock problems by column generation and branch-and-bound. *Computational Optimization and Applications*, 3:111–130, 1994.

[Wun97] Roland Wunderling. Soplex, the sequential object-oriented simplex class library. Technical report, Konrad Zuse Zentrum für Informationstechnik, Berlin, 1997. http://www.zib.de/Optimization/Software/Soplex/.

Index

All names set in typewriter style refer to C++ names, file names, or names in the configuration file. In particular, all names of the reference manual are written in typewriter style. Members of classes are sub entries of their classes.