ABACUS

A Branch-And-CUt System

Version 3.0

User's Guide and Reference Manual

2007

ABACUS 3.0 Documentation

For use with Version 3.0 of the ABACUS Library

Contents

Chapter 1

Preface

Preface to Release 3.0

Major enhancements in ABACUS 3.0 include the new solver interface to Osi, the ability to solve LPs with the Volume Algorithm and support for state-of-the-art GNU compilers. The documentation system has been changed from cweb to doxygen and the build process has been simplified. ABACUS 3.0 is released under the Gnu Lesser General Public License (LGPL). See section 3.1 for details.
We thank the members of Michael Jünger's group for many stimulating discussions and valuable insights that helped improve the current release. Special thanks go to Christoph Buchheim, Frauke Liers, Thomas Lange (all University of Cologne) and Markus Chimani (University of Dortmund).

Köln, August 2007 *Frank Baumann*, *Mark Sprenger* and *Andrea Wagner*

Preface to Release 2.3

ABACUS 2.3 is the first commercial release of ABACUS distributed by the newly founded company Oreas GmbH. The changes made involve mainly some bug fixes and a new licensing mechanism, which allows to distribute also 30-days evaluation licenses.

Köln, December 1999 *Matthias Elf* and *Carsten Gutwenger*
Oreas GmbH

Preface to Release 2.2

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.

Based on our earlier work on non-object oriented branch-and-cut frameworks, Stefan Thienel developed ABACUS 1.0 in his PhD thesis that was defended in December 1995. Since January 1996 he developed the public releases ABACUS 1.2 to 2.1 with the partial support of ESPRIT LTR Project no. 20244 (ALCOM-IT) and H.C.M. Institutional Grant no. ERBCHBGCT940710 (DONET). Stefan Thienel laid the foundations of ABACUS with great dedication and enthusiasm. We regret that he decided to leave the Universität zu Köln in spring 1998. Very much to our satisfaction, Max Böhm and Thomas Christof immediately took over the responsibility for ABACUS. We are very glad that ABACUS is again in competent hands and future development and maintenance is guaranteed.

Köln, August 1998 *Michael Jünger*
Heidelberg, August 1998 *Gerhard Reinelt*

ABACUS 2.1 was left ready for release in February 1998 by Stefan Thienel. After Stefan Thienel left university and we took over the responsibility for ABACUS, we decided not to release ABACUS 2.1, but to add some new features to the software. The major enhancements of the resulting version 2.2 are the interface to the LP solver Xpress and the compilation of ABACUS with different native compilers. In addition, we introduced some new functions for easier parameter handling and improved the HTML version of the Reference Manual. A complete presentation of all modifications can be found in Section 3.3.

We are very grateful to Stefan Thienel for his efforts involved in the development, documentation and support of ABACUS, and wish him the very best for his future. For the users of ABACUS, we hope that this transition in responsibility will be almost invisible to them.

Köln, August 1998 *Max Böhm*
Athens, GA, August 1998 *Thomas Christof*

Preface to Release 2.1

The main purpose of version 2.1 of ABACUS is the provision of some bug fixes. However, there are also a few new features that are explained in Section 3.4.

Köln, February 1998 *Stefan Thienel*

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

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

ABACUS can be obtained from

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

Please note that ABACUS requires a working installation of the Open Solver Interface (Osi) provided by The Computational Infrastructure for Operations Research (COIN-OR) project. Please see Section 2.3 for details. If you have any questions about ABACUS please send a mail to

`abacus@informatik.uni-koeln.de` .

2.2 Platforms

ABACUS is currently available for `linux`. If you are interested in a version for another platform please contact us directly.

2.3 Building ABACUS

ABACUS can be compiled with the `GNU-C++` compilers `g++ 3.3.5 - 4.1.2`.

ABACUS provides a general interface to linear programming solvers. The current release supports the LP-solvers supported by COIN Osi version 0.96. Not all of them might be useful in combination with ABACUS though. Before compiling ABACUS 3.0 make sure that COIN Osi is installed. For more information on the installation of COIN Osi see See `https://projects.coin-or.org/Osi` for details.

Set the paths at the top of the Makefile to the include directories of COIN Osi and the LP solvers installed on your system.

Settings for different compilers are stored in the directory Make-settings. Which settings file is used is determined by the variable ABACUS_MAKE_SETTINGS. To compile ABACUS with g++-4.1, for example, do:

make abacus ABACUS_MAKE_SETTINGS=linux20-gcc41

To install abacus to a specific location instead of the base directory set the variables ABACUS_INSTALL_LIBDIR and ABACUS_INSTALL_HEADERDIR in the Makefile and run, for example:

make install ABACUS_MAKE_SETTINGS=linux20-gcc41

For information on how to produce the documentation, please run:

make

2.4 Compiling and Linking

For compiling your files using ABACUS add the `abacus/include` directory either to your include directory path or specify it explicitly with the `-I` compiler option. Furthermore, add the include file paths of the LP-solvers you want to use. The flag for the C++ compiler can be defined at compilation time using the `-D` switch of the compiler (e.g., `-DABACUS_COMPILER_GCC41`) or specified in the `Makefile`. See table 2.1 for valid settings. It might be helpful to consult the `Makefile` of the example included in the ABACUS distribution.

compiler preprocessor flags	
Linux g++ 4.1	`ABACUS_COMPILER_GCC41` or `ABACUS_COMPILER_GCC`
Linux g++ 3.4	`ABACUS_COMPILER_GCC34`
Linux g++ 3.3	`ABACUS_COMPILER_GCC33`
SUN C++ 4.2	`ABACUS_COMPILER_SUN`

Table 2.1: compilers

2.5 Environment Variables

The environment variable `ABACUS_DIR` has to be set to the directory containing the general configuration file `.abacus`. A master version of this configuration file is provided in the base directory of the ABACUS distribution. It is recommended that every user makes a private copy of the file `.abacus` and sets `ABACUS_DIR` accordingly.

To set the environment variable to `/home/yourhome`, for example, using the C-shell or its relatives, do:

```
setenv ABACUS_DIR /home/yourhome
```

If the Bourne-shell is used do:

```
export ABACUS_DIR=/home/yourhome
```

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

2.6 Contact

Feedback from the users is highly appreciated. Please report your experiences and make your suggestions. Also comments on this user manual are appreciated. Report all problems and suggestions 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++.

2.7 Mailing List

There is a mailing list available for ABACUS. To subscribe to this service, please register at

`https://lists.uni-koeln.de/mailman/listinfo/abacus-forum`.

Chapter 3

New Features

This section summarizes all new features that have been introduced successively.

3.1 New Features of ABACUS 3.0

3.1.1 Open Solver Interface

ABACUS now supports the `Open Solver Interface (Osi)` developed by the `COIN-OR (COmputational INfrastructure for Operations Research)` project. All interface classes (|CPLEXIF|, |SOPLEXIF|, |XPRESSIF|) have been replaced by the single new class |OSIIF|. This change has the advantage that every solver supported by Osi can be used to solve LP relaxations. For the user only small modifications (if any) to existing code should be necessary. The setting of parameters for specific solvers now has to be implemented by the user in the problem specific derived class using Osi functions. For this we provide a new virtual function |ABA_MASTER::setSolverParameters()| that can be redefined by the user.

3.1.2 Compilers

New supported compilers are the `GNU-C++` compilers `g++ 3.3 - 4.1`. Support for some older compilers has been dropped.

3.1.3 Library creation by the user

As there is now a single interface to all supported solvers, library creation is greatly simplified. Calling `make` and `make install` after adapting the `Makefile` compiles the library `libabacus-osi.a` and installs the header files to the specified location.

3.1.4 Documentation System

The reference manual is now extracted directly from the `c++` source files using the `doxygen` documentation system. `cweave` and `ctangle` are no longer needed to compile the library.

3.1.5 Approximate solver

ABACUS now can use approximate instead of exact methods for solving LP relaxations. Currently, only the Volume Algorithm is supported, as it is the only approximate method provided by Osi. The new parameter |SolveApprox| and the virtual function |ABA_MASTER::solveApprox()| are provided to switch between exact and approximate solvers. See Section 5.2.11, the reference manual and the example included in the ABACUS distribution for details.

3.1.6 Memory management

The allocation and management of memory for the internal represantation of the LP is completely handled by Osi. The corresponding ABACUS functions are kept only for compatibility reasons.

3.1.7 Preprocessor Flags and Include Paths

A lot of preprocessor flags are no longer used. Especially the flag `ABACUS_OLD_INCLUDE` introduced in version 2.0 is obsolete. To include the array header file, for example, do:

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

3.2 New Features of ABACUS 2.3

3.2.1 Version macro

The include file `abacusroot.h` contains now a define `ABACUS_VERSION` with the version number of the ABACUS release. It is set to `230` in this release.

3.2.2 New classes for separation

New classes |ABA_LPSOLUTION| for storing an LP solution and |ABA_SEPARATOR| for implementing a separation procedure facilitate encapsulation of the code. Moreover, the class |ABA_SEPARATOR| provides functions for checking for duplication of generated constraints/variables.

3.2.3 Rank for constraints/variables

A new virtual function |ABA_CONVAR::rank()| allows to associate a rank with a constraint/variable. This rank can be used for ranking the constraints/variables in the functions |ABA_STANDARDPOOL::separate()|, |ABA_SUB::constraintPoolSeparation| and |ABA_SUB::variablePoolSeparation()|.

3.3 New Features of ABACUS 2.2

Version 2.2 includes a new interface to the Lp-Solver Xpress and Cplex 6.0 and it provides enhaced functionality for parameter handling. Moreover, the library is now available for different native compilers. It can be configured for any combination of supported LP-Solvers by the user. ABACUS now intensively uses inline functions to improve performance.

3.3.1 Lp-Solver Xpress

In addition to the LP-Solvers Cplex and Soplex ABACUS now also provides an interface to the LP-Solver Xpress-MP Version 10. The Xpress libraries `libxosl.a` and `libmp-opt.a` have both to be linked.

Xpress-MP is a commercial product by Dash Associates. You find further information about Xpress at `http://www.dash.co.uk/`.

3.3.2 Lp-Solver Cplex

Cplex 6.0 is now supported.

In addition, a new parameter `CplexHoldEnvironment` is introduced. If this parameter is true, then the Cplex environment is held open during the branch-and-cut optimization. This reserves a Cplex license for the complete time of optimization.

3.3.3 Lp-Methods

The solution method for linear programs `LP::Barrier` is replaced by the methods `LP::BarrierAndCrossover` and `LP::BarrierNoCrossover`.

3.3.4 New Compilers

New supported compilers are the `GNU C++ compiler gcc 2.8` and the `Sun WorkShop Compiler C++ 4.2`. We now provide 32 and 64 bit versions of the ABACUS library compiled with the `SGI MIPSpro 7.2 C++ compiler`.

3.3.5 Library Architectures

The ABACUS library is provided for different combinations of `hardware`, `operating systems` and `compilers`. These combinations are identified by an `<arch>` name. Some architectures are shown in table 3.1.

3.3.6 Library Creation by the User

The library archive file `abacus-<version>-<arch>.tar.gz` contains the basic ABACUS library and libraries for each supported `Interface` to an LP-Solver. Currently supported Interfaces are shown in table 3.2.

You can create ABACUS libraries for any combination of supported LP-Solvers by yourself. Downloaded and unpack the library ditribution archive with the right <arch> in the installation root directiry (e.g. /usr/local/abacus). A directory `abacus-<version/lib/<arch>/stuff` is created which contains all required files to build specific ABACUS libraries. Then create LP-Solver specific ABACUS libraries by using the command `make-lib` in the directory `lib/<arch>` for any desired combination of different LP-solvers.

For example if you want to have ABACUS libraries for Solaris compiled with gcc 2.8 download the file `abacus-2.2-solaris-gcc28.tar.gz`.

```
  gunzip abacus-2.2-solaris-gcc28.tar.gz
  tar xvf abacus-2.2-solaris-gcc28.tar
```

To create libraries with interfaces for Cplex 6.0, Soplex, Xpress and all three together type

Hardware	Operating System	Compiler	<arch>
SUN SPARC	SUN-OS 4.1.3	GNU C++ Compiler 2.8.1	sunos-gcc28
SUN SPARC	SUN-OS 5.6	GNU C++ Compiler 2.8.1	solaris-gcc28
SUN SPARC	SUN-OS 5.6	GNU C++ Compiler 2.7.2	solaris-gcc27
SUN SPARC	SUN-OS 5.6	SUN WorkShop Compiler C++ 4.2	solaris-CC
IBM RS6000	AIX 4.1.5	GNU C++ Compiler 2.8.1	aix4-gcc28
IBM RS6000	AIX 4.1.5	GNU C++ Compiler 2.7.2	aix4-gcc27
DEC ALPHA	OSF 3.2	GNU C++ Compiler 2.8.1	osf-gcc28
DEC ALPHA	OSF 3.2	GNU C++ Compiler 2.7.2	osf-gcc27
SILICON GRAPHICS	Irix 6.2	GNU C++ Compiler 2.7.2	irix6-gcc27
SILICON GRAPHICS	Irix 6.2	MIPSpro 7.2 C++ compiler 32 Bit, mips4	irix6-CCn32
SILICON GRAPHICS	Irix 6.2	MIPSpro 7.2 C++ compiler 64 Bit, mips4	irix6-CC64
HP 9000	HP-UX 10.20	GNU C++ Compiler 2.8.1	hpux10-gcc28
PC	Linux 2.0.27	GNU C++ Compiler 2.8.1	linux20-gcc28
PC	Linux 2.0.27	GNU C++ Compiler 2.7.2	linux20-gcc27
PC	Windows NT	MS Visual C++ 5.0	winnt

Table 3.1: Architecture names.

Interface name	LP-Solver
cplex22	Cplex 2.2
cplex30	Cplex 3.0
cplex40	Cplex 4.0
cplex50	Cplex 5.0
cplex60	Cplex 6.0
soplex	Soplex 1.0
xpress	Xpress-MP 10

Table 3.2: Interface names.

```
cd abacus-2.2/lib/solaris-gcc28
make-lib cplex60
make-lib soplex
make-lib xpress
make-lib cplex60-soplex-xpress
```

The `make-lib <interfaces>` command creates the file
`abacus-<version>/lib/solaris-gcc28/libabacus-<interfaces>.a`,

where `<interfaces>` is an interface string or a combination of interface strings concatenated by the character `-`.

3.3.7 New or Changed Preprocessor Flags

A list of preprocessor flags with new or changed meaning follows:

Preprocessor Flag	Meaning
ABACUS_COMPILER_GCC28	GNU C++ compiler 2.8
ABACUS_COMPILER_GCC27	GNU C++ compiler 2.7
ABACUS_COMPILER_GCC	defaults to ABACUS_COMPILER_GCC28
ABACUS_COMPILER_SUN	SUN WorkShop C++ Compiler 4.2
ABACUS_EXPLICIT_TEMPLATES	no longer needed
ABACUS_CPP_MATH	no longer needed
ABACUS_SYS_xxxxxx	no longer needed
ABACUS_LP_SOPLEX	no longer needed
ABACUS_LP_CPLEXxx	needed only if lpmastercplex.h or cplexif.h is included.

See the updated Makefile of the TSP example in `abacus-2.2/example/Makefile` for a description of the valid compiler and linker flags. This file also explains how to link an application with more than one LP solver.

3.3.8 Templates

It is no longer needed to include template definition files (*.inc). These files are now automatically included by the coresponding header files (*.h).

If you are using gcc 2.8 no special flags for template instatiation need to be defined. If you are using gcc 2.7 we recommended to define the compilerflag `-fno-implicit-templates` and to manually instantiate the templates which are needed, but not contained in the ABACUS library as described in section 5.3.

3.3.9 New LP Master Classes

There is a new abstract class `ABA_LPMASTER` and subclasses `ABA_LPMASTERCPLEX`, `ABA_LPMASTERSOPLEX` and `ABA_LPMASTERXPRESS`. These classes handle LP solver specific parameters and global data. As a consequence some LP solver specific functions which were located in ABA_MASTER are now located in one of these classes. If you are using such a function you have to change your code as shown in the example below:

```
master->cplexOutputLevel(level);
```

should be changed to

```
ABA_LPMASTERCPLEX *cplexMaster = master->lpMasterCplex();
lpMasterCplex->cplexOutputLevel(level);
```

or simply

```
master->lpMasterCplex()->cplexOutputLevel(level);
```

The corresponding header files are `abacus/lpmastercplex.h`, `abacus/lpmastersoplex.h`, and `abacus/lpmasterxpress.h`.

3.3.10 HTML Documentation

In the HTML version of the Reference Manual (Section 6), we added links in the declaration part of the class which point to other classes and to the descriptions of the class members.

3.3.11 Parameter Handling

The system parameter table and the functions for handling parameters moved from class `ABA_MASTER` to its base class `ABA_GLOBAL`. Now, it is possible to use the parameter concept of ABACUS even without generating an object of `ABA_MASTER`. (This might be useful when writing some experimental code using the Tools and Templates of ABACUS, but not writting an complete branch-and-cut-application.)

In addition to the overloaded functions `ABA_GLOBAL::getParameter()`, we now provide the overloaded functions `ABA_GLOBAL::assignParameter()` and `ABA_GLOBAL::findParameter()` with enhanched functionality. The new functions test for the existence of a parameter in the table, compare the current setting with feasible settings and allow for termination of the program if a required paramter is not found, or if it is found but if its setting is not feasible.

Moreover, a branch-and-cut-optimization can be started without reading the parameter file `.abacus`.

See section 5.2.28 for further details on using parameters.

3.3.12 Name changings

This version contains some changings of names that seemed reasonable to us. Most changings were guided by the principle that we want to have the feasible values of the ABACUS parameters coinciding with the enumerators of the corresponding enumeration type. (As all enumerators in one class have to be different, an exception to that rule is the parameter value `None` which is feasible for different paramaters.)

In addition, we changed in general in `SoPlex` the upper `P` to a lower one.

Table 3.3 summarizes the changings. We provide Perl scripts for performing the changings on your ABACUS application. For a parameter file, use

```
upd-parameter-2.2 <parameter-file>
```

and apply

```
upd-sources-2.2 <code-filse>
```

to your C++ code files.

Location	Old	New
Parameter `EnumerationStrategy`	`Best`	`BestFirst`
Parameter `EnumerationStrategy`	`Depth`	`DepthFirst`
Parameter `EnumerationStrategy`	`Breadth`	`BreadthFirst`
`ABA_MASTER::PRIMALBOUNDMODE`	`OptimalPrimalBound`	`Optimum`
`ABA_MASTER::PRIMALBOUNDMODE`	`OptimalOnePrimalBound`	`OptimumOne`
`ABA_MASTER::VBCMODE`	`None`	`NoVbcLog`
various	`SoPlex`	`Soplex`
`ABA_LP::METHOD`	`Barrier`	`BarrierAndCrossover`

Table 3.3: Name changings.

3.4 New Features of ABACUS 2.1

In version 2.1 we added a few new features, fixed some bugs, and improved the performance of some functions.

3.4.1 Elimination of Constraints and Variables

So far a constraint or variable was eliminated from the set of active items as soon as the criterion for elimination hold. Now the number of iterations the criterion must hold until the elimination is performed can be specified in the configuration file `.abacus` (see Section 5.2.26).

3.4.2 Cplex 5.0

Cplex 5.0 is now supported by ABACUS.

3.4.3 Templates

In addition to the explicit instantiation of templates, ABACUS now also supports the implicit instantiation (see Section 5.3).

3.4.4 Bug Fixes

3.4.4.1 Constraint and Variable Selection

The selection of constraints and variables with highest rank from the buffers of generated constraints and variables is now performed correctly again.

3.4.4.2 Variable Generation

We have tested the dymanic variable generation of ABACUS more intensively and could fix some so far unknown bugs.

3.5 New Features of ABACUS 2.0

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

3.5.1 LP-Solver Soplex

Besides Cplex ABACUS provides now an interface to the LP-Solver Soplex [Wun97].

If Soplex is used as LP-solver, it might be required to switch to the new include file structure (see Section 3.5.3) in order to avoid name conflicts. Both Soplex and ABACUS provide include files with the name `timer.h`.

3.5.2 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 `upd-names-2.0`. If you apply this script to all source files of your ABACUS application by calling

```
  upd-names-2.0 <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.

3.5.3 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/upd-includes-2.0`. Calling

```
upd-includes-2.0 <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.

3.5.4 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).

3.5.5 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).

3.5.6 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).

3.5.7 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.

3.5.8 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.

3.5.9 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

3.5.10 Compiler Preprocessor Flag

In the compilation of an ABACUS-application the used compiler must be specified by a preprocessor flag (see Section 2.1).

3.5.11 LP-Solver Preprocessor Flag

The LP-solvers that are used have to be specified by a preprocessor flag (see Section 2.3). Also the flags for the LP-solver Cplex changed.

3.5.12 Parameters of Configuration File

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

3.5.12.1 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).

3.5.12.2 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.

3.5.12.3 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.

3.5.13 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`.

3.5.14 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 and 5.0 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.

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.

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.

4.1.1 Application Base Classes

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

4.1.1.1 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`.

4.1.1.2 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.

4.1.1.3 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.

4.1.2.1 The Root of the Class Tree

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

4.1.2.2 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 supported by the Open Solver Interface (Osi). In theory all these solvers can be used to solve the LP relaxations. We have tested ABACUS with CPLEX, Clp and Glpk.

4.1.2.3 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.

4.1.2.4 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.

4.1.3.1 The Basic Data Structures

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

4.1.3.2 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.

4.2.2.1 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.

4.2.2.2 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.

4.2.2.3 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.

4.2.2.4 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.

4.2.2.5 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.

4.2.2.6 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.

4.2.3.1 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).

4.2.3.2 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.

4.2.3.3 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.

4.2.3.4 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.

4.2.3.5 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.

4.2.3.6 Adding Variables

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

4.2.3.7 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.

4.2.3.8 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.

4.2.3.9 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.

4.2.3.10 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.

4.2.3.11 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.

Currently the Open Solver Interface used by ABACUS does not support the barrier method. Nevertheless a barrier method is provided for compatibility to older versions of ABACUS. This method outputs a warning message and calls the primal simplex method. By default a linear program is solved by the primal simplex method, 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.

4.2.3.12 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.

4.2.3.13 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.

4.2.3.14 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.

4.2.3.15 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.

4.2.3.16 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.

4.2.4.1 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.

4.2.4.2 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.

4.2.4.3 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.

4.2.4.4 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.

4.2.5.1 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.

4.2.5.2 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.

4.2.5.3 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.

4.2.5.4 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.

4.2.5.5 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.

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

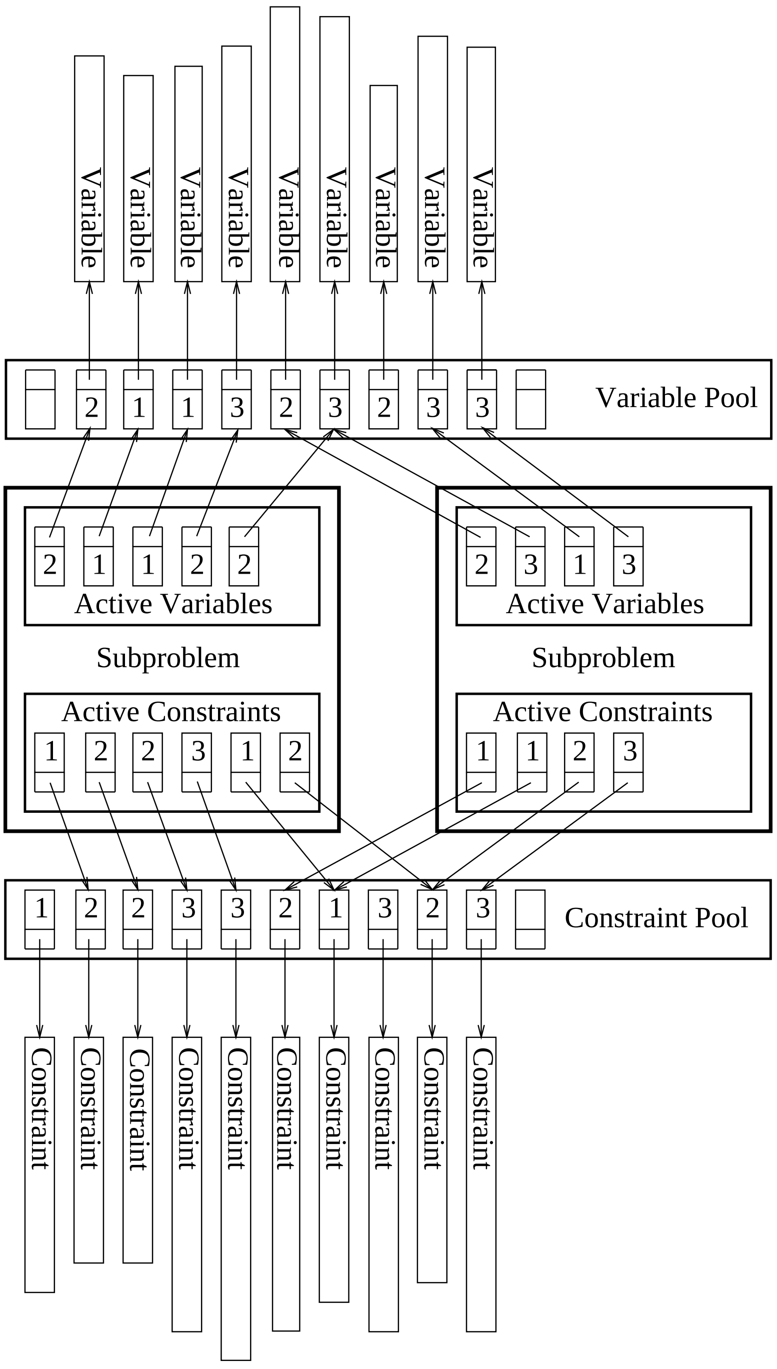


Figure 4.1: The pool concept.

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.

4.2.6.1 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 interface to an LP-solver.

4.2.6.2 The LP-Solver Interface

The class `ABA_OSIIF` implements these solver specific functions. The class `ABA_OSIIF` itself is an interface to `Osi` (Open Solver Interface) which is a uniform API for calling embedded linear and mixed-integer programming solvers. Using this generic API means that the single interface class `ABA_OSIIF` provides access to a whole range of LP-solvers. Another advantage is that any change in the API of a specific LP-solver is handled by th Osi layer. That means that in order to support future versions of LP-solvers no change to the ABACUS code will be necessary. When support for futher solvers is added to Osi only minimal changes to ABACUS will be necessary to make them available for solving linear programs.

4.2.6.3 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 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. 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.

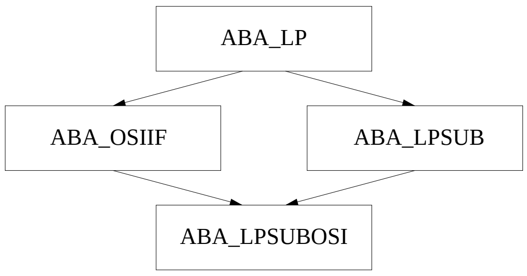


Figure 4.2: The linear programming classes.

4.2.6.4 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.

A linear program solving the relaxations within a subproblem with a LP-solver supported by Osi is defined by the class `ABA_LPSUBOSI`, which is derived from the classes `ABA_LPSUB` and `ABA_OSIIF`. The class `ABA_LPSUBOSI` only implements a constructor passing the arguments to the base classes. Using a LP-solver not suppported by Osi in this context requires the definition of a class equivalent to the class `ABA_LPSUBOSI` 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_LPSUBOSI` and returning a pointer to this object.

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_LPSUBOSI`, the classes `ABA_OSIIF`, 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 class `ABA_OSIIF`.

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.

4.2.7.1 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.

4.2.7.2 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.

4.2.7.3 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.

4.2.7.4 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.

4.2.7.5 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.

4.2.7.6 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].

4.2.8.1 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.

4.2.8.2 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.

4.2.8.3 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.

4.2.8.4 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.

4.2.8.5 Linked Lists

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

4.2.8.6 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.

4.2.8.7 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.

4.2.8.8 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].

4.2.8.9 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.

4.2.9.1 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.

4.2.9.2 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.

4.2.9.3 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.

4.2.10.1 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.

4.2.10.2 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.

4.2.10.3 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 (e.g., `class ABA_COLUMN`), names of enumerations (e.g., `enum VARELIMMODE`), and character strings associated with an enumeration (e.g., `const char* VARELIMMODE_[]`) are written with upper case letters.
- 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.

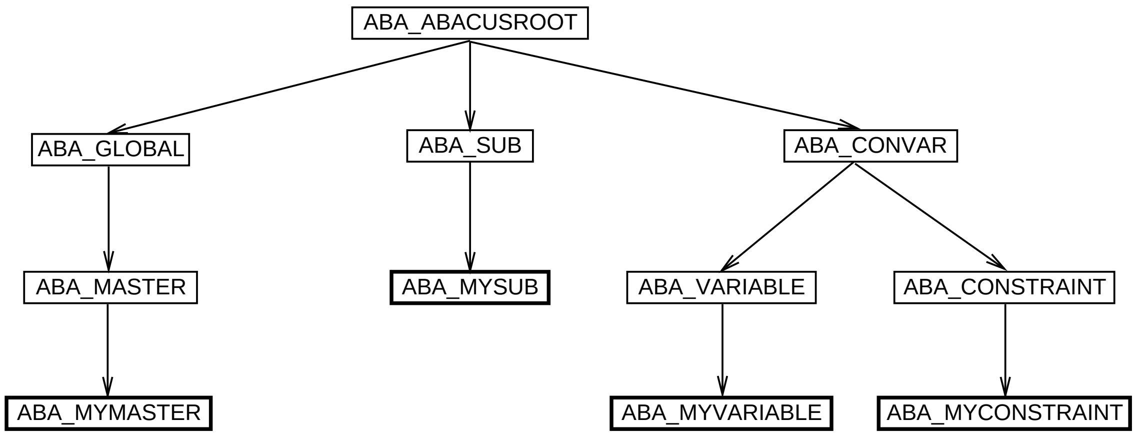


Figure 5.1: Embedding problem specific classes in ABACUS.

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 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 {};
```

5.1.2.1 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 = 1.0e32);
```

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”. Please note that this value might be different from the value the LP-solver uses internally. You should make sure that the value used here is always greater than or equal to the value used by the solver.

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);
}
```

5.1.2.2 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());
}
```

5.1.2.3 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.

5.1.3.1 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. 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`.

5.1.3.2 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()`.

5.1.3.3 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.

5.1.3.4 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.

5.1.3.5 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 `addConBufferSpace()`. 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_, addConBufferSpace());
    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_, addConBufferSpace());

  int nSep = mySeparate(newCuts);

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

5.1.3.6 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).

5.1.3.7 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;
}
```

5.1.3.8 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();
}
```

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_;
}
```

5.2.3.1 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.

5.2.3.2 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 4.2.7.3). 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`.

5.2.7.1 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.

5.2.7.2 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).

5.2.7.3 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.

5.2.8.1 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).

5.2.8.2 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.

5.2.8.3 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`.

5.2.8.4 Comparing Branching Samples

After a rank to each rule of each branching sample has been assigned by the function `rankBranchingRule()` 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()`.

5.2.8.5 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.

5.2.8.6 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 5.2.7.3) 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, if approximate solving is disabled (the default). If the usage of the approximate solver is enabled (by setting the parameter SolveApprox to true in the configuration file `.abacus`), the virtual function `ABA_SUB::solveApproxNow()` is called first. If this function returns true the LP method is set to `ABA_LP::Approximate` (if the current situation in the cutting plane algorithm does not require an exact solution, e.g. to prepare branching).

The parameters of the function `ABA_SUB::chooseLpMethod` 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::BarrierAndCrossover`, `ABA_LP::BarrierNoCrossover`, `ABA_LP::Primal`, or `ABA_LP::Dual`. The LP methods `ABA_LP::BarrierAndCrossover` and `ABA_LP::BarrierNoCrossover` are provided only for compatibility with older versions of ABACUS and custom solver interfaces as the current interface only supports the methods `ABA_LP::Primal` and `ABA_LP::Dual` (and `ABA_LP::Approximate`, see above).

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 4.2.3.12). 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.

5.2.16.1 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()
  { }
```

5.2.16.2 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 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 solver interface. A correct modification of the LP-data is only guaranteed by the member functions of the class `ABA_SUB`.

5.2.22.1 Accessing Internal Data of the LP-solver

Internal data of the solver is retrieved with the function

```
  OsiSolverInterface* ABA_OSIIF::osiLP();
```

that returns a pointer to the OsiSolverInterface object that manages the interaction with the LP-solver.

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

```
  OsiSolverInterface* LpInterface = ((ABA_LPSUBOSI*) lp())->osiLP();
```

The class `ABA_LPSUBOSI` is derived from the classes `ABA_LPSUB` and `ABA_OSIIF`.

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 System 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 specific to the used LP-solver.

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. The parameters specific to the LP-solver can be set by redefining the virtual function ABA_MASTER::setSolverParameters(), see Section 5.2.27 for details.

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).

5.2.26.1 EnumerationStrategy

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

Valid settings:

BestFirst	best-first search
BreadthFirst	breadth-first search
DepthFirst	depth-first search
DiveAndBest	depth-first search until the first feasible solution is found, then best-first search

Default value: BestFirst

5.2.26.2 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`

5.2.26.3 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`

5.2.26.4 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`

5.2.26.5 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`

5.2.26.6 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`

5.2.26.7 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`

5.2.26.8 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`

5.2.26.9 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`

5.2.26.10 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

5.2.26.11 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`

5.2.26.12 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`

5.2.26.13 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

5.2.26.14 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`

5.2.26.15 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`

5.2.26.16 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`

5.2.26.17 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`

5.2.26.18 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`

5.2.26.19 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`

5.2.26.20 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`

5.2.26.21 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`

5.2.26.22 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`

5.2.26.23 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`

5.2.26.24 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`

5.2.26.25 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`

5.2.26.26 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`.

5.2.26.27 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`

5.2.26.28 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`

5.2.26.29 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`

5.2.26.30 ConElimAge

The number of iterations an elimination criterion for a constraint must be satisfied until the constraint is eliminated from the active constraints.

Valid settings:

A nonnegative integer.

Default value: `1`

5.2.26.31 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`

5.2.26.32 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`

5.2.26.33 VarElimAge

The number of iterations an elimination criterion for a variable must be satisfied until the variable is eliminated from the active variables.

Valid settings:

A nonnegative integer.

Default value: `1`

5.2.26.34 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`

5.2.26.35 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

5.2.26.36 DefaultLpSolver

This parameter determines the LP-solver that should be applied per default for each subproblem. Please note that these are the solvers supported by the `Open Solver Interface` and hence by ABACUS˙Nevertheless not all of these solvers may be suitable for solving LP relaxations.

Valid settings:

`Cbc`
`Clp`
`CPLEX`
`DyLP`
`FortMP`
`GLPK`

MOSEK
OSL
SoPlex
SYMPHONY
Vol
XPRESS_MP

Default value: `Clp`

5.2.26.37 SolveApprox

If set to true usage of the Volume Algorithm to solve LP relaxations is enabled. This parameter only enables usage of the approximate solver in general. Whether or not a specific LP relaxation is solved exact or approximate is determined by the function `ABA_MASTER::solveApproxNow()`.

Valid settings:

F

Default value: o r an example reimplementation of this function see the file `tspsub.w` in the `example` directory of the ABACUS source code. `false` or `true` `false`

5.2.27 Solver Parameters

Setting parameters for specific LP-solvers is done by redefining the virtual function `ABA_MASTER::setSolverParameters(OsiSolverInterface* interface, bool solverIsApprox)`. The parameter `interface` is a generic pointer to an object of type `OsiSolverInterface`, it has to be typecast to a pointer to a specific solver interface. Via this pointer all the internals of the solver can be accessed. The parameter `solverIsApprox` is true if the solver for which parameters are set is approximate, i.e. the Volume Algorithm. To set the primal column pivot algorithm for Clpi to "steepest", for example, one would do:

```
bool MYMASTER::setSolverParameters(OsiSolverInterface*interface,bool solverIsApprox)
{
OsiClpSolverInterface* clpIf = dynamic_cast<OsiClpSolverInterface*> (interface);
ClpSimplex* clp_simplex = clpIf->getModelPtr();
ClpPrimalColumnSteepest steepestP;
clp_simplex->setPrimalColumnPivotAlgorithm(steepestP);
return true;
}
```

For a more complex reimplementation of this function see the file `tspmaster.w` in the `example` directory of the ABACUS source code.

5.2.28 Parameter Handling

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 DepthFirst
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:
      enum SEPSTRAT {All,Hierachical};
      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.

Parameter files having our format can be read by the function `ABA_GLOBAL::readParameters()`, which inserts all parameters in a table. Then, the parameters can be extracted from the table with the functions `ABA_GLOABAL::assignParameter()`, `ABA_GLOABAL::findParameter()`, `ABA_GLOABAL::getParameter()` which are overloaded in different ways.

For our application, the code could look like

```
  void MYMASTER::initializeParameters()
  {
    readParameters(".myparameters");
    /* terminate the program if the parameters are not found
       in the table (which was filled by readParamters() ) */

    const char* SeparationStrategy[]={"All","Hierachical"};
    separationStrategy_=(SEPSTRAT)
      findParameter("SeparationStrategy",2, SeparationStrategy);

    /* allow only values between 1 and 5000; */
    assignParameter(maxNodesPerPerCut_, "MaxNodesPerCut", 1, 5000);

  }
```

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()).

A branch-and-cut optimization can be performed even without reading the file .abacus. This can be achieved by setting the 8th parameter of the constructor of ABA_MASTER to false. In this case, ABACUS starts with default settings for the parameters, which can be overwritten by the function ABA_GLOBAL::insertParameter().

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.

ABACUS allows implicit or explicit template instantiation. Implicit template instantiation is the more convenient way. The compiler automatically instantiates a template when required. Its disadvantage is that it increases the compile time and (depending on the compiler) also the size of the generated code. For explicit template instantiation the templates have to be collected manually in file and and this file has to be compiled separately. Note, some compilers do not support explicit template instantiation. Other compilers perform the explicit template instantiation automatically even if the implicit instantiation is selected. Currently, we recommend explicit template instantiation for the GNU-compiler 2.7.x and the SGI compiler and implicit template instantiation for the GNU-compiler 2.8.x, the SUN compiler and the MS Visual C++ compiler.

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 explicitly 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. For this reason we added HTML links in the declaration part of the class which point to other classes and to the descriptions of the class members. For a listing of all functions in lexicographical order we refer to the index.

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`.

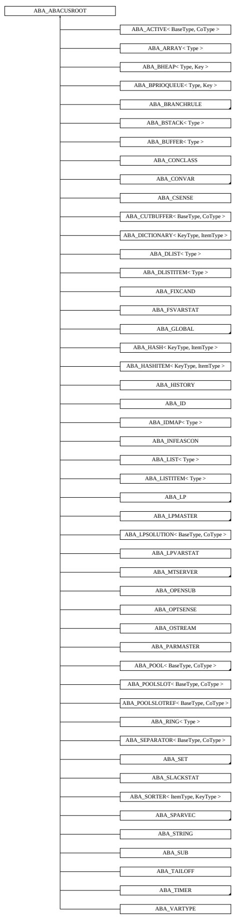
Some other classes are included in this section because they are base classes of the application base classes `ABA_MASTER`, `ABA_SUB`, `ABA_CONSTRAINT` and `ABA_VARIABLE`. The class `ABA_ABACUSROOT` is a base class of every class of the system. The class `ABA_GLOBAL` is a base class of the class `ABA_MASTER`.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

6.2 ABA_ABACUSROOT Class Reference

base class of all other classes of ABACUS

`#include <abacusroot.h>`

Inheritance diagram for ABA_ABACUSROOT::



Public Types

- enum EXITCODES { Ok, Fatal }

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

Public Member Functions

- virtual ∼ABA_ABACUSROOT ()
- virtual void exit (enum EXITCODES code) const

  terminates the program and returns code *to the environment from which the program was called.*

- const char ∗ onOff (bool value)

converts a boolean variable to the strings "on" *and* "off".

- double fracPart (double x) const

6.2.1 Detailed Description

base class of all other classes of ABACUS

Definition at line 81 of file abacusroot.h.

6.2.2 Member Enumeration Documentation

6.2.2.1 enum ABA_ABACUSROOT::EXITCODES

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

Parameters:

Ok The program terminates without error.

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

Enumeration values:

Ok

Fatal

Definition at line 95 of file abacusroot.h.

6.2.3 Constructor & Destructor Documentation

6.2.3.1 virtual ABA_ABACUSROOT::∼ABA_ABACUSROOT () `[virtual]`

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

6.2.4 Member Function Documentation

6.2.4.1 virtual void ABA_ABACUSROOT::exit (enum EXITCODES *code*) const `[virtual]`

terminates the program and returns *code* to the environment from which the program was called.

> We overload the function *exit()* that in a debugger a break point can be easily set within this function in order to investigate the error. We also observed that for some reason it can be impossible to set a break point within a template. Here this function *exit()* was quite helpful during the debugging process.

Exception handling could substitute many calls to the function *exit()*. However, in version 2.6.3 of the GNU / compiler only a prototypical implementation of exception handling is available. As soon as a the GNU compiler provides a stable implementation of exception handling we will use this technique in future releases of this software.

Parameters:

code The exit code given to the environment.

6.2.4.2 double ABA_ABACUSROOT::fracPart (double *x*) const

Returns:

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.

Parameters:

x The value of which the fractional part is computed.

6.2.4.3 const char∗ ABA_ABACUSROOT::onOff (bool *value*)

converts a boolean variable to the strings *"on"* and *"off"*.

Returns:

"on" if *value* is *true*
"off" otherwise

Parameters:

value The boolean variable being converted.

The documentation for this class was generated from the following file:

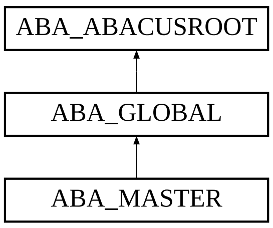
- Include/abacus/abacusroot.h

6.3 ABA_GLOBAL Class Reference

class stores global data (e.g., a zero tolerance, an output stream, a table with system parameters) und functions operating with this data.

`#include <global.h>`

Inheritance diagram for ABA_GLOBAL::



Public Member Functions

- ABA_GLOBAL (double eps=1.0e-4, double machineEps=1.0e-7, double infinity=1.0e32)

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

- virtual ∼ABA_GLOBAL ()

  The destructor.

- virtual ABA_OSTREAM & out (int nTab=0)

  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.*

- virtual ABA_OSTREAM & err (int nTab=0)

  Behaves like the function out() *except that the global error stream is used instead of the global output stream.*

- double eps () const
- void eps (double e)

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

- double machineEps () const
- void machineEps (double e)

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

- double infinity () const

  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.*

- void infinity (double x)

  This version of the function infinity() *sets the "infinite value". Please note that this value might be different from the value the LP-solver uses internally. You should make sure that the value used here is always greater than or equal to the value used by the solver.*

- 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)

  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.*

- void readParameters (const char ∗fileName)

  Opens the parameter file fileName*, reads all parameters, and inserts them in the parameter table.*

- void insertParameter (const char ∗name, const char ∗value)
- int getParameter (const char ∗name, int &param)
- int getParameter (const char ∗name, unsigned 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)
- void assignParameter (int &param, const char ∗name, int minVal, int maxVal)
- void assignParameter (unsigned &param, const char ∗name, unsigned minVal, unsigned maxVal)

  See ABA_GLOBAL::assignParameter} for description.

- void assignParameter (double &param, const char ∗name, double minVal, double maxVal)

  See ABA_GLOBAL::assignParameter} for description.

- void assignParameter (bool &param, const char ∗name)

  See ABA_GLOBAL::assignParameter} for description.

- void assignParameter (ABA_STRING &param, const char ∗name, unsigned nFeasible=0, const char ∗feasible[ ]=0)
- void assignParameter (char &param, const char ∗name, const char ∗feasible=0)
- void assignParameter (int &param, const char ∗name, int minVal, int maxVal, int defVal)
- void assignParameter (unsigned &param, const char ∗name, unsigned minVal, unsigned maxVal, unsigned defVal)

  See ABA_GLOBAL::assignParameter} for description.

- void assignParameter (double &param, const char ∗name, double minVal, double maxVal, double defVal)

  See ABA_GLOBAL::assignParameter} for description.

- void assignParameter (bool &param, const char ∗name, bool defVal)

  See ABA_GLOBAL::assignParameter} for description.

- void assignParameter (ABA_STRING &param, const char ∗name, unsigned nFeasible, const char ∗feasible[ ], const char ∗defVal)
- void assignParameter (char &param, const char ∗name, const char ∗feasible, char defVal)
- int findParameter (const char ∗name, unsigned nFeasible, const int ∗feasible)
- int findParameter (const char ∗name, unsigned nFeasible, const char ∗feasible[ ])

  See ABA_GLOBAL::findParameter} for description.

- int findParameter (const char ∗name, const char ∗feasible)

  See ABA_GLOBAL::findParameter} for description.

Private Member Functions

- ABA_GLOBAL (const ABA_GLOBAL &rhs)
- const ABA_GLOBAL & operator= (const ABA_GLOBAL &rhs)

Private Attributes

- ABA_OSTREAM out_
- ABA_OSTREAM err_
- double eps_
- double machineEps_

  The machine dependent zero tolerance, which is used to , e.g., to test if a floating point value is 0.
- double infinity_
- char ∗ tab_
- ABA_HASH< ABA_STRING, ABA_STRING > paramTable_

Friends

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

6.3.1 Detailed Description

class stores global data (e.g., a zero tolerance, an output stream, a table with system parameters) und functions operating with this data.

Definition at line 58 of file global.h.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 ABA_GLOBAL::ABA_GLOBAL (double *eps* = `1.0e-4`, double *machineEps* = `1.0e-7`, double *infinity* = `1.0e32`)

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

Parameters:

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*.0e32). Please note that this value might be different from the value the LP-solver uses internally. You should make sure that the value used here is always greater than or equal to the value used by the solver.

6.3.2.2 virtual ABA_GLOBAL::∼ABA_GLOBAL () `[virtual]`

The destructor.

6.3.2.3 ABA_GLOBAL::ABA_GLOBAL (const ABA_GLOBAL & *rhs*) `[private]`

6.3.3 Member Function Documentation

6.3.3.1 void ABA_GLOBAL::assignParameter (char & *param*, const char ∗ *name*, const char ∗ *feasible*, char *defVal*)

See ABA_GLOBAL::assignParameter} for description.

Parameters:

param The variable *param* receives the value of the parameter.

name The name of the parameter.

feasible A string containing all feasible settings. If *feasible* is zero, then all settings are allowed.

defVal The default value that is used when the paramter is not found in the parameter table.

6.3.3.2 void ABA_GLOBAL::assignParameter (ABA_STRING & *param*, const char ∗ *name*, unsigned *nFeasible*, const char ∗ *feasible*[ ], const char ∗ *defVal*)

See ABA_GLOBAL::assignParameter} for description.

Parameters:

parameter The variable *parameter* receives the value of the parameter.

name The name of the parameter.

nFeasible The number of feasible settings. If *nFeasible* is equal to zero, then all settings are allowed.

feasible The settings for the parameter to be considered as feasible. Must be an array of *nFeasible* strings.

defVal The default value that is used when the paramter is not found in the parameter table.

6.3.3.3 void ABA_GLOBAL::assignParameter (bool & *param*, const char ∗ *name*, bool *defVal*)

See ABA_GLOBAL::assignParameter} for description.

6.3.3.4 void ABA_GLOBAL::assignParameter (double & *param*, const char ∗ *name*, double *minVal*, double *maxVal*, double *defVal*)

See ABA_GLOBAL::assignParameter} for description.

6.3.3.5 void ABA_GLOBAL::assignParameter (unsigned & *param*, const char ∗ *name*, unsigned *minVal*, unsigned *maxVal*, unsigned *defVal*)

See ABA_GLOBAL::assignParameter} for description.

6.3.3.6 void ABA_GLOBAL::assignParameter (int & *param*, const char ∗ *name*, int *minVal*, int *maxVal*, int *defVal*)

See ABA_GLOBAL::assignParameter} for description.

Parameters:

parameter The variable *parameter* receives the value of the parameter.

name The name of the parameter.

minVal The value of the parameter is considered as infeasible if it is less than *minVal*.

maxVal The value of the parameter is considered as infeasible if it is larger than *maxVal*.

defVal The default value that is used when the paramter is not found in the parameter table.

6.3.3.7 void ABA_GLOBAL::assignParameter (char & *param*, const char ∗ *name*, const char ∗ *feasible* = `0`)

See ABA_GLOBAL::assignParameter} for description.

Parameters:

param The variable *param* receives the value of the parameter.

name The name of the parameter.

feasible A string consisting of all feasible characters. If *feasible* is zero, then all characters are allowed.

6.3.3.8 void ABA_GLOBAL::assignParameter (ABA_STRING & *param*, const char ∗ *name*, unsigned *nFeasible* = `0`, const char ∗ *feasible*[ ] = `0`)

See ABA_GLOBAL::assignParameter} for description.

Parameters:

param The variable *parameter* receives the value of the parameter.

name The name of the parameter.

nFeasible The number of feasible settings. If *nFeasible* is equal to zero, then all values are allowed. 0 is the default value.

feasible If *nFeasible* is greater zero, the this are the settings for the parameter to be considered as feasible. Must be an array of *nFeasible* strings.

6.3.3.9 void ABA_GLOBAL::assignParameter (bool & *param*, const char ∗ *name*)

See ABA_GLOBAL::assignParameter} for description.

6.3.3.10 void ABA_GLOBAL::assignParameter (double & *param*, const char ∗ *name*, double *minVal*, double *maxVal*)

See ABA_GLOBAL::assignParameter} for description.

6.3.3.11 void ABA_GLOBAL::assignParameter (unsigned & *param*, const char ∗ *name*, unsigned *minVal*, unsigned *maxVal*)

See ABA_GLOBAL::assignParameter} for description.

6.3.3.12 void ABA_GLOBAL::assignParameter (int & *param*, const char ∗ *name*, int *minVal*, int *maxVal*)

Searches for the parameter *name* in the parameter table.

If no parameter *name* is found and no default value of the parameter is given, the program terminates with an error messages. The program terminates also with an error message if the value of a parameter is not within a specified feasible region. Depending on the type of the parameter, a feasible region can be an interval (specified by *minVal* and *maxVal*) or can be given by a set of feasible settings (given by a number *nFeasible* and a pointer *feasible* to the feasible values.

This function is overloaded in two ways. First, this function is defined for different types of the argument *parameter*, second, for each such type we have both versions, with and without a default value of the parameter.

Parameters:

param The variable *parameter* receives the value of the parameter.

name The name of the parameter.

minVal The value of the parameter is considered as infeasible if it is less than *minVal*.

maxVal The value of the parameter is considered as infeasible if it is larger than *maxVal*.

6.3.3.13 virtual char ABA_GLOBAL::enter (istream & *in*) `[virtual]`

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.

Returns:

The character read from the input stream.

Parameters:

in The input stream the character should be read from.

6.3.3.14 void ABA_GLOBAL::eps (double *e*) `[inline]`

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

Parameters:

e The new value of the zero tolerance.

Definition at line 439 of file global.h.

6.3.3.15 double ABA_GLOBAL::eps () const `[inline]`

Returns:

The zero tolerance.

Definition at line 434 of file global.h.

6.3.3.16 bool ABA_GLOBAL::equal (double *x*, double *y*) const `[inline]`

Returns:

true If the absolute difference of *x* and *y* is less than the machine dependent zero tolerance, false otherwise.

Parameters:

x The first value being compared.

y The second value being compared.

Definition at line 484 of file global.h.

6.3.3.17 virtual ABA_OSTREAM& ABA_GLOBAL::err (int *nTab* = `0`) `[virtual]`

Behaves like the function *out()* except that the global error stream is used instead of the global output stream.

Returns:

A reference to the global error stream.

Parameters:

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

6.3.3.18 int ABA_GLOBAL::findParameter (const char ∗ *name*, const char ∗ *feasible*)

See ABA_GLOBAL::findParameter} for description.

6.3.3.19 int ABA_GLOBAL::findParameter (const char ∗ *name*, unsigned *nFeasible*, const char ∗ *feasible*[ ])

See ABA_GLOBAL::findParameter} for description.

6.3.3.20 int ABA_GLOBAL::findParameter (const char ∗ *name*, unsigned *nFeasible*, const int ∗ *feasible*)

Searches for the parameter *name* in the parameter table.

If no parameter *name* is found the program terminates with an error messages. The program terminates also with an error message if the value of a parameter is not within a given list of feasible settings. This function is overloaded and can be used for different types of parameters such as integer valued, char valued and string parameters.

Returns:

The index of the matched feasible setting.

Parameters:

name The name of the parameter.

nFeasible The number of feasible settings.

feasible The settings for the parameter to be considered as feasible. Must be an array of *nFeasible* strings.

6.3.3.21 int ABA_GLOBAL::getParameter (const char ∗ *name*, char & *param*)

6.3.3.22 int ABA_GLOBAL::getParameter (const char ∗ *name*, bool & *param*)

6.3.3.23 int ABA_GLOBAL::getParameter (const char ∗ *name*, ABA_STRING & *param*)

6.3.3.24 int ABA_GLOBAL::getParameter (const char ∗ *name*, double & *param*)

6.3.3.25 int ABA_GLOBAL::getParameter (const char ∗ *name*, unsigned int & *param*)

6.3.3.26 int ABA_GLOBAL::getParameter (const char ∗ *name*, int & *param*)

Searches for the parameter *name* in the parameter table.

This function is overloaded for different types of the argument *parameter*. See also the functions *assignParameter* and *findParameter* with enhanced functionality.

Returns:

0 If the parameter is found,
1 otherwise.

Parameters:

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.

6.3.3.27 void ABA_GLOBAL::infinity (double *x*) `[inline]`

This version of the function *infinity()* sets the “infinite value”. Please note that this value might be different from the value the LP-solver uses internally. You should make sure that the value used here is always greater than or equal to the value used by the solver.

Parameters:

x The new value representing “infinity”.

Definition at line 459 of file global.h.

6.3.3.28 double ABA_GLOBAL::infinity () const `[inline]`

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.

Returns:

A very large floating point number. The default value of *infinity()* is *1*.0e32.

Definition at line 454 of file global.h.

6.3.3.29 void ABA_GLOBAL::insertParameter (const char ∗ *name*, const char ∗ *value*)

Inserts a parameter in the parameter table.

If the parameter already is in the table, the value is overwritten.

Parameters:

name The name of the parameter.

value The literal value of the parameter.

6.3.3.30 bool ABA_GLOBAL::isInfinity (double *x*) const `[inline]`

Returns:

true If *x* is regarded as “infinite” large,
false otherwise.

Parameters:

x The value compared with “infinity”.

Definition at line 464 of file global.h.

6.3.3.31 bool ABA_GLOBAL::isInteger (double *x*, double *eps*) const

Returns:

true If the value *x* differs at most by *eps* from an integer value,
false otherwise.

6.3.3.32 bool ABA_GLOBAL::isInteger (double *x*) const `[inline]`

Returns:

true If the value *x* differs at most by the machine dependent zero tolerance from an integer value,
false otherwise.

Definition at line 490 of file global.h.

6.3.3.33 bool ABA_GLOBAL::isMinusInfinity (double *x*) const `[inline]`

Returns:

true If *x* is regarded as infinite small;}
false otherwise.

Parameters:

x The value compared with “minus infinity”.

Definition at line 474 of file global.h.

6.3.3.34 void ABA_GLOBAL::machineEps (double *e*) `[inline]`

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

Parameters:

e The new value of the machine dependent zero tolerance.

Definition at line 449 of file global.h.

6.3.3.35 double ABA_GLOBAL::machineEps () const `[inline]`

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.

Returns:

The machine dependent zero tolerance.

Definition at line 444 of file global.h.

6.3.3.36 const ABA_GLOBAL& ABA_GLOBAL::operator= (const ABA_GLOBAL & *rhs*) `[private]`

6.3.3.37 virtual ABA_OSTREAM& ABA_GLOBAL::out (int *nTab* = `0`) `[virtual]`

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.

Returns:
A reference to the global output stream.

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

6.3.3.38 void ABA_GLOBAL::readParameters (const char ∗ *fileName*)

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.

Parameters:
fileName The name of the parameter file.

6.3.4 Friends And Related Function Documentation

6.3.4.1 ostream& operator<< (ostream & *out*, const ABA_GLOBAL & *rhs*) `[friend]`

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

Returns:
A reference to the output stream.

Parameters:
out The output stream.

rhs The object being output.

6.3.5 Member Data Documentation

6.3.5.1 double ABA_GLOBAL::eps_ `[private]`

A zero tolerance.

Definition at line 411 of file global.h.

6.3.5.2 ABA_OSTREAM ABA_GLOBAL::err_ `[private]`

The global error stream.

Definition at line 407 of file global.h.

6.3.5.3 double ABA_GLOBAL::infinity_ `[private]`

An “infinite” big number.

Definition at line 423 of file global.h.

6.3.5.4 double ABA_GLOBAL::machineEps_ `[private]`

The machine dependent zero tolerance, which is used to , e.g., to test if a floating point value is 0.

This value is usually less than *eps_*, which represents, e.g., a safety tolerance if a constraint is violated.

Definition at line 419 of file global.h.

6.3.5.5 ABA_OSTREAM ABA_GLOBAL::out_ `[private]`

The global output stream.

Definition at line 403 of file global.h.

6.3.5.6 ABA_HASH<ABA_STRING, ABA_STRING> ABA_GLOBAL::paramTable_ `[private]`

Definition at line 428 of file global.h.

6.3.5.7 char∗ ABA_GLOBAL::tab_ `[private]`

A string used as tabulator in the functions *out()* and *err()*.

Definition at line 427 of file global.h.

The documentation for this class was generated from the following file:

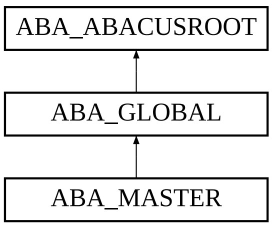
- Include/abacus/global.h

6.4 ABA_MASTER Class Reference

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.

`#include <master.h>`

Inheritance diagram for ABA_MASTER::



Public Types

- enum STATUS {
  Optimal, Error, OutOfMemory, Unprocessed,
  Processing, Guaranteed, MaxLevel, MaxCpuTime,
  MaxCowTime, ExceptionFathom }
- enum OUTLEVEL {
  Silent, Statistics, Subproblem, LinearProgram,
  Full }
- enum ENUMSTRAT { BestFirst, BreadthFirst, DepthFirst, DiveAndBest }
- enum BRANCHINGSTRAT { CloseHalf, CloseHalfExpensive }

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

- enum PRIMALBOUNDMODE { NoPrimalBound, Optimum, OptimumOne }

  This enumeration provides various methods for the initialization of the primal bound.

- enum SKIPPINGMODE { SkipByNode, SkipByLevel }
- enum CONELIMMODE { NoConElim, NonBinding, Basic }

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

- enum VARELIMMODE { NoVarElim, ReducedCost }

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

- enum VBCMODE { NoVbc, File, Pipe }

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

- enum OSISOLVER {
  Cbc, Clp, CPLEX, DyLP,
  FortMP, GLPK, MOSEK, OSL,
  SoPlex, SYMPHONY, Vol, XPRESS_MP }

  This enumeration defines which solvers can be used to solve theLP-relaxations.

Public Member Functions

- 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, bool readParamFromFile=true)

- virtual ~ABA_MASTER ()

  The destructor.

- STATUS optimize ()
- ENUMSTRAT enumerationStrategy () const
- void enumerationStrategy (ENUMSTRAT strat)

  This version of the function enumerationStrategy() *changes the enumeration strategy.*

- virtual int enumerationStrategy (const ABA_SUB ∗s1, const ABA_SUB ∗s2)

  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.*

- bool 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.

- double guarantee ()
- void printGuarantee ()
- bool check ()

  Can be used to control the correctness of the optimization if the value of the optimum solution has been loaded.

- bool knownOptimum (double &optVal)

  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.

- virtual void output ()
- bool cutting () const
- bool pricing () const
- const ABA_OPTSENSE ∗ optSense () const
- ABA_HISTORY ∗ history () const
- ABA_OPENSUB ∗ openSub () const
- ABA_STANDARDPOOL< ABA_CONSTRAINT, ABA_VARIABLE > ∗ conPool () const
- ABA_STANDARDPOOL< ABA_CONSTRAINT, ABA_VARIABLE > ∗ cutPool () const
- ABA_STANDARDPOOL< ABA_VARIABLE, ABA_CONSTRAINT > ∗ varPool () const
- ABA_SUB ∗ root () const
- ABA_SUB ∗ rRoot () const
- STATUS status () const
- const ABA_STRING ∗ problemName () const
- const ABA_COWTIMER ∗ totalCowTime () const
- bool solveApprox () const
- const ABA_CPUTIMER ∗ totalTime () const
- const ABA_CPUTIMER ∗ lpTime () const
- const ABA_CPUTIMER ∗ lpSolverTime () const
- const ABA_CPUTIMER ∗ separationTime () const
- const ABA_CPUTIMER ∗ improveTime () const
- const ABA_CPUTIMER ∗ pricingTime () const
- const ABA_CPUTIMER ∗ branchingTime () const
- int nSub () const
- int nLp () const

- int highestLevel () const
- int nNewRoot () const
- int nSubSelected () const
- void printParameters ()

  Writes all parameters of the class ABA_MASTER together with their values to the global output stream.

- BRANCHINGSTRAT branchingStrategy () const
- void branchingStrategy (BRANCHINGSTRAT strat)
- OSISOLVER defaultLpSolver () const
- void defaultLpSolver (OSISOLVER osiSolver)
- ABA_LPMASTEROSI ∗ lpMasterOsi () const
- int nBranchingVariableCandidates () const
- void nBranchingVariableCandidates (int n)

  This version of the function nbranchingVariableCandidates() *sets the number of tested branching variable candidates.*

- double requiredGuarantee () const
- void requiredGuarantee (double g)

  This version of the function requiredGuarantee() *changes the guarantee specification.*

- int maxLevel () const
- void maxLevel (int ml)

  This version of the function maxLevel() *changes the maximal enumeration depth.*

- const ABA_STRING & maxCpuTime () const
- void maxCpuTime (const ABA_STRING &t)
- const ABA_STRING & maxCowTime () const

  The function maxCowTime()*.*

- void maxCowTime (const ABA_STRING &t)

  This version of the function maxCowTime() *set the maximal wall-clock time for the optimization.*

- bool objInteger () const
- void objInteger (bool b)

  This version of function objInteger() *sets the assumption that the objective function values of all feasible solutions are integer.*

- int tailOffNLp () const

  The function tailOffNLp()*.*

- void tailOffNLp (int n)
- double tailOffPercent () const

  The function tailOffPercent()*.*

- void tailOffPercent (double p)

  This version of the function tailOffPercent() *sets the minimal change of the dual bound for the tailing off analysis.*

- OUTLEVEL outLevel () const
- void outLevel (OUTLEVEL mode)

  The version of the function outLevel() *sets the output mode.*

- OUTLEVEL logLevel () const
- void logLevel (OUTLEVEL mode)

    This version of the function logLevel() sets the output mode for the log-file.

- bool delayedBranching (int nOpt_) const
- void dbThreshold (int threshold)

    Sets the number of optimizations of a subproblem until sons are created in ABA_SUB::branching().

- int dbThreshold () const
- int minDormantRounds () const
- void minDormantRounds (int nRounds)
- PRIMALBOUNDMODE pbMode () const
- void pbMode (PRIMALBOUNDMODE mode)
- int pricingFreq () const
- void pricingFreq (int f)

    This version of the function pricingFreq() sets the number of linear programs being solved between two additional pricing steps.

- int skipFactor () const
- void skipFactor (int f)

    This version of the function skipFactor() sets the frequency for constraint and variable generation.

- void skippingMode (SKIPPINGMODE mode)

    This version of the function skippingMode() sets the skipping strategy.

- 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)
- int varElimAge () const
- void varElimAge (int eps)
- int conElimAge () const
- void conElimAge (int eps)
- bool fixSetByRedCost () const
- void fixSetByRedCost (bool on)
- bool printLP () const
- void printLP (bool on)
- int maxConAdd () const
- void maxConAdd (int max)

    Sets the maximal number of constraints that are added in an iteration of the cutting plane algorithm.

- int maxConBuffered () const
- void maxConBuffered (int max)

    Changes the maximal number of constraints that are buffered in an iteration of the cutting plane algorithm.

- int maxVarAdd () const
- void maxVarAdd (int max)

  Changes the maximal number of variables that are added in an iteration of the subproblem optimization.

- int maxVarBuffered () const
- void maxVarBuffered (int max)

  Changes the maximal number of variables that are buffered in an iteration of the subproblem optimization.

- int maxIterations () const
- void maxIterations (int max)

  Changes the default value for the maximal number of iterations of the optimization of a subproblem.

- bool eliminateFixedSet () const
- void eliminateFixedSet (bool turnOn)

  This version of the function eliminateFixedSet() *can be used to turn the elimination of fixed and set variables on or off.*

- bool newRootReOptimize () const
- void newRootReOptimize (bool on)
- const ABA_STRING & optimumFileName () const
- void optimumFileName (const char ∗name)
- bool showAverageCutDistance () const
- void showAverageCutDistance (bool on)

  Turns the output of the average distance of the added cuts from the fractional solution on or off.

- VBCMODE vbcLog () const
- void vbcLog (VBCMODE mode)
- virtual bool setSolverParameters (OsiSolverInterface ∗interface, bool solverIsApprox)

bounds

In order to embed both minimization and maximization problems in this system we work internally with primal bounds, i.e., a value which is worse than the best known solution (often a value of a feasible solution), and dual bounds, i.e., a bound which is better than the best known solution. Primal and dual bounds are then interpreted as lower or upper bounds according to the sense of the optimization.

- double lowerBound () const
- double upperBound () const
- double primalBound () const
- void primalBound (double x)

  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.*

- double dualBound () const
- void dualBound (double x)

  This version of the function dualBound() *sets the dual bound to* x *and makes a new entry in the solution history.*

- bool betterDual (double x) const
- bool primalViolated (double x) const
- bool betterPrimal (double x) const
- bool feasibleFound () const

  We use this function ,e.g., to adapt the enumeration strategy in the DiveAndBest-Strategy.

Static Public Attributes

- static const char ∗ STATUS_ [ ]

  Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { STATUS[0]=="Optimal"}).

- static const char ∗ OUTLEVEL_ [ ]

  Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { OUTLEVEL[0]=="Silent"}).

- static const char ∗ ENUMSTRAT_ [ ]

  Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { ENUMSTRAT[0]=="BestFirst"}).

- static const char ∗ BRANCHINGSTRAT_ [ ]

  Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { BRANCHINGSTRAT[0]=="CloseHalf"}).

- static const char ∗ PRIMALBOUNDMODE_ [ ]

  Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { PRIMALBOUNDMODE[0]=="None"}).

- static const char ∗ SKIPPINGMODE_ [ ]

  Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { SKIPPINGMODE[0]=="None"}).

- static const char ∗ CONELIMMODE_ [ ]

  Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { CONELIMMODE[0]=="None"}).

- static const char ∗ VARELIMMODE_ [ ]

  Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { VARELIMMODE[0]=="None"}).

- static const char ∗ VBCMODE_ [ ]

  Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { VBCMODE[0]=="None"}).

- static const char ∗ OSISOLVER_ [ ]

  Array for the literal values for possible Osi solvers.

Protected Member Functions

- 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)

  Is overloaded such that also a first set of cutting planes can be inserted into the cutting plane pool.

- void initializeOptSense (ABA_OPTSENSE::SENSE sense)

  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.

- int bestFirstSearch (const ABA_SUB ∗s1, const ABA_SUB ∗s2) const
- virtual int equalSubCompare (const ABA_SUB ∗s1, const ABA_SUB ∗s2) const

  Is called from the function bestFirstSearch() *and from the function* depthFirstSearch() *if the subproblems* s1 *and* s2 *have the same priority.*

- int depthFirstSearch (const ABA_SUB ∗s1, const ABA_SUB ∗s2) const

  Implements the depth first search enumeration strategy, i.e., the subproblem with maximum level *is selected.*

- int breadthFirstSearch (const ABA_SUB ∗s1, const ABA_SUB ∗s2) const

  Implements the breadth first search enumeration strategy, i.e., the subproblem with minimum level *is selected.*

- int diveAndBestFirstSearch (const ABA_SUB ∗s1, const ABA_SUB ∗s2) const

  Performs depth-first search until a feasible solution is found, then the search process is continued with best-first search.

- virtual void 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 ().

- virtual ABA_SUB ∗ firstSub ()=0
- virtual void initializeOptimization ()

  The default implementation of initializeOptimization() *does nothing.*

- virtual void terminateOptimization ()

  The default implementation of terminateOptimization() *does nothing.*

Private Member Functions

- void _initializeParameters ()

  Reads the parameter-file { .abacus}, which is searched in the directory given by the environment variable ABACUS_-DIR, and calls the virtual function initializeParameters() *which can initialize parameters of derived classes and overwrite parameters of this class.*

- void _createLpMasters ()
- void _deleteLpMasters ()
- void _initializeLpParameters ()
- void _setDefaultLpParameters ()

  Initializes the LP solver specific default Parameters if they are not read from the parameter-file { .abacus}.

- void _printLpParameters ()
- void _outputLpStatistics ()
- ABA_SUB ∗ select ()
- int initLP ()
- void writeTreeInterface (const char ∗info, bool time=true) const
- void treeInterfaceNewNode (ABA_SUB ∗sub) const

Adds the subproblem sub *to the stream storing information for graphical output of the enumeration tree if this logging is turned on.*

- void treeInterfacePaintNode (int id, int color) const
- void treeInterfaceLowerBound (double lb) const
- void treeInterfaceUpperBound (double ub) const
- void treeInterfaceNodeBounds (int id, double lb, double ub)

  Updates the node information in the node with number id *by writing the lower bound* lb *and the upper bound* ub *to the node.*

- void newSub (int level)
- void countLp ()

  Increments the counter for linear programs and should be called in each optimization call of the LP-relaxation.

- void newFixed (int n)

  Increments the counter of the number of fixed variables by n.

- void addCons (int n)

  Increments the counter for the total number of added constraints by n.

- void removeCons (int n)

  Increments the counter for the total number of removed constraints by n.

- void addVars (int n)

  Increments the counter for the total number of added variables by n.

- void removeVars (int n)

  Increments the counter for the total number of removed variables by n.

- ABA_FIXCAND ∗ fixCand () const
- void rRoot (ABA_SUB ∗newRoot, bool reoptimize)
- void status (STATUS stat)
- void rootDualBound (double x)
- void theFuture ()
- ABA_MASTER (const ABA_MASTER &rhs)
- const ABA_MASTER & operator= (const ABA_MASTER &rhs)

Private Attributes

- ABA_STRING problemName_
- bool readParamFromFile_
- ABA_OPTSENSE optSense_
- ABA_SUB ∗ root_
- ABA_SUB ∗ rRoot_
- ABA_OPENSUB ∗ openSub_
- ABA_HISTORY ∗ history_
- ENUMSTRAT enumerationStrategy_
- BRANCHINGSTRAT branchingStrategy_
- int nBranchingVariableCandidates_

The number of candidates that are evaluated for branching on variables.

- OSISOLVER defaultLpSolver_
- ABA_LPMASTEROSI ∗ lpMasterOsi_
- ABA_STANDARDPOOL< ABA_CONSTRAINT, ABA_VARIABLE > ∗ conPool_
- ABA_STANDARDPOOL< ABA_CONSTRAINT, ABA_VARIABLE > ∗ cutPool_
- ABA_STANDARDPOOL< ABA_VARIABLE, ABA_CONSTRAINT > ∗ varPool_
- double primalBound_
- double dualBound_
- double rootDualBound_
- ABA_FIXCAND ∗ fixCand_
- bool cutting_
- bool pricing_
- bool solveApprox_
- int nSubSelected_

  The number of subproblems already selected from the list of open subproblems.

- VBCMODE VbcLog_

  Ouput for the Tree Interface is generated depending on the value of this variable.

- ostream ∗ treeStream_
- double requiredGuarantee_

  The guarantee in percent which should be reached when the optimization stops.

- int maxLevel_
- ABA_STRING maxCpuTime_
- ABA_STRING maxCowTime_
- bool objInteger_

  true*, if all objective function values of feasible solutions are assumed to be integer.*

- int tailOffNLp_
- double tailOffPercent_
- int dbThreshold_
- int minDormantRounds_

  The minimal 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.*

- OUTLEVEL outLevel_
- OUTLEVEL logLevel_
- PRIMALBOUNDMODE pbMode_
- int pricingFreq_
- int skipFactor_

  The frequency constraints or variables are generated depending on the skipping mode.

- SKIPPINGMODE skippingMode_

  Either constraints are generated only every skipFactor_ *subproblem (*SkipByNode*) only every* skipFactor_ *level (*SkipByLevel*).*

- bool fixSetByRedCost_
- bool printLP_

- int maxConAdd_

  The maximal number of added constraints per iteration of the cutting plane algorithm.

- int maxConBuffered_
- int maxVarAdd_

  The maximal number of added variables per iteration of the column generation algorithm.

- int maxVarBuffered_
- int maxIterations_

  The maximal number of iterations of the cutting plane/column generation algorithm in the subproblem.

- bool eliminateFixedSet_
- bool newRootReOptimize_

  If true*, then an already earlier processed node is reoptimized if it becomes the new root of the remaining* \ *tree.*

- ABA_STRING optimumFileName_

  The name of a file storing a list of optimum solutions of problem instances.

- bool showAverageCutDistance_

  If true *then the average distance of the added cutting planes is output every iteration of the cutting plane algorithm.*

- CONELIMMODE conElimMode_

  The way constraints are automatically eliminated in the cutting plane algorithm.

- VARELIMMODE varElimMode_

  The way variables are automatically eliminated in the column generation algorithm.

- double conElimEps_

  The tolerance for the elimination of constraints by the mode NonBinding/.

- double varElimEps_

  The tolerance for the elimination of variables by the mode ReducedCost.

- int conElimAge_

  The number of iterations an elimination criterion must be satisfied until a constraint can be removed.

- int varElimAge_

  The number of iterations an elimination criterion must be satisfied until a variable can be removed.

- STATUS status_
- ABA_COWTIMER totalCowTime_
- ABA_CPUTIMER totalTime_
- ABA_CPUTIMER lpTime_
- ABA_CPUTIMER lpSolverTime_
- ABA_CPUTIMER separationTime_
- ABA_CPUTIMER improveTime_

  The timer for the cpu time spent in the heuristics for the computation of feasible solutions.

- ABA_CPUTIMER pricingTime_

- ABA_CPUTIMER branchingTime_
- int nSub_
- int nLp_
- int highestLevel_
- int nFixed_
- int nAddCons_
- int nRemCons_
- int nAddVars_
- int nRemVars_
- int nNewRoot_

Friends

- class ABA_SUB
- class ABA_FIXCAND

6.4.1 Detailed Description

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.

Definition at line 76 of file master.h.

6.4.2 Member Enumeration Documentation

6.4.2.1 enum ABA_MASTER::BRANCHINGSTRAT

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

Parameters:

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.

Enumeration values:

CloseHalf

CloseHalfExpensive

Definition at line 175 of file master.h.

6.4.2.2 enum ABA_MASTER::CONELIMMODE

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

Parameters:

NoConElim No constraints are eliminated.

NonBinding Nonbinding constraints are eliminated.

Basic Constraints with basic slack variable are eliminated.

Enumeration values:

NoConElim

NonBinding

Basic

Definition at line 233 of file master.h.

6.4.2.3 enum ABA_MASTER::ENUMSTRAT

Enumeration values:

BestFirst

BreadthFirst

DepthFirst

DiveAndBest

Definition at line 158 of file master.h.

6.4.2.4 enum ABA_MASTER::OSISOLVER

This enumeration defines which solvers can be used to solve theLP-relaxations.

Enumeration values:

Cbc

Clp

CPLEX

DyLP

FortMP

GLPK

MOSEK

OSL

SoPlex

SYMPHONY

Vol

XPRESS_MP

Definition at line 280 of file master.h.

6.4.2.5 enum ABA_MASTER::OUTLEVEL

This enumeration defines the different output levels:

Parameters:

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.

Enumeration values:

Silent

Statistics

Subproblem

LinearProgram

Full

Definition at line 131 of file master.h.

6.4.2.6 enum ABA_MASTER::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.

Parameters:

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.

Enumeration values:

NoPrimalBound

Optimum

OptimumOne

Definition at line 202 of file master.h.

6.4.2.7 enum ABA_MASTER::SKIPPINGMODE

The way nodes are skipped for the generation of cuts.

Parameters:

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.

Enumeration values:

SkipByNode

SkipByLevel

Definition at line 218 of file master.h.

6.4.2.8 enum ABA_MASTER::STATUS

The various statuses of the optimization process.

Parameters:

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().

Enumeration values:

Optimal

Error

OutOfMemory

Unprocessed

Processing

Guaranteed

MaxLevel

MaxCpuTime

MaxCowTime

ExceptionFathom

Definition at line 109 of file master.h.

6.4.2.9 enum ABA_MASTER::VARELIMMODE

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

Parameters:

NoVarElim No variables are eliminated.

ReducedCost Variables with high absolute reduced costs are eliminated.

Enumeration values:

NoVarElim

ReducedCost

Definition at line 249 of file master.h.

6.4.2.10 enum ABA_MASTER::VBCMODE

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

Parameters:

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.

Enumeration values:

NoVbc

File

Pipe

Definition at line 266 of file master.h.

6.4.3 Constructor & Destructor Documentation

6.4.3.1 ABA_MASTER::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`, bool *readParamFromFile* = `true`)

The constructor.

Parameters:

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).

readParamFromFile If true, then the parameter file .abacus is read, otherwise the parameters are initialized with default values (default *true*).

The members *primalBound_* and *dualBound_* stay uninitialized since this can only be done when the sense of optimization (minimization or maximization) is known. The initialization is performed automatically in the function *optimize()*.

6.4.3.2 virtual ABA_MASTER::∼ABA_MASTER () `[virtual]`

The destructor.

6.4.3.3 ABA_MASTER::ABA_MASTER (const ABA_MASTER & *rhs*) `[private]`

6.4.4 Member Function Documentation

6.4.4.1 void ABA_MASTER::_createLpMasters () `[private]`

6.4.4.2 void ABA_MASTER::_deleteLpMasters () `[private]`

6.4.4.3 void ABA_MASTER::_initializeLpParameters () `[private]`

6.4.4.4 void ABA_MASTER::_initializeParameters () `[private]`

Reads the parameter-file { .abacus}, which is searched in the directory given by the environment variable ABACUS_DIR, and calls the virtual function *initializeParameters()* which can initialize parameters of derived classes and overwrite parameters of this class.

All parameters are first inserted together with their values in a parameter table in the function *readParameters()*. If the virtual dummy function *initializeParameters()* is redefined in a derived class and also reads a parameter file with the function *readParameters()*, then already inserted parameters can be overwritten.

After all parameters are input we extract with the function *assignParameter()* all parameters. Problem specific parameters should be extracted in a redefined version of *initializeParameters()*. extracted from this table

6.4.4.5 void ABA_MASTER::_outputLpStatistics () `[private]`

Prints the LP solver specific statistics.

This function is implemented in the file *lpif.cc*.

6.4.4.6 void ABA_MASTER::_printLpParameters () `[private]`

Prints the LP solver specific parameters.

This function is implemented in the file *lpif.cc*.

6.4.4.7 void ABA_MASTER::_setDefaultLpParameters () `[private]`

Initializes the LP solver specific default Parameters if they are not read from the parameter-file { .abacus}.

This function is implemented in the file *lpif.cc*.

6.4.4.8 void ABA_MASTER::addCons (int *n*) `[inline, private]`

Increments the counter for the total number of added constraints by *n*.

Definition at line 2016 of file master.h.

6.4.4.9 void ABA_MASTER::addVars (int *n*) `[inline, private]`

Increments the counter for the total number of added variables by *n*.

Definition at line 2026 of file master.h.

6.4.4.10 int ABA_MASTER::bestFirstSearch (const ABA_SUB ∗ *s1*, const ABA_SUB ∗ *s2*) const `[protected]`

Implements the best first search enumeration.

If the bounds of both subproblems are equal, then the subproblems are compared with the function *equalSubCompare()*.

Returns:

-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.

Parameters:

s1 A subproblem.

s2 A subproblem.

6.4.4.11 bool ABA_MASTER::betterDual (double *x*) const

Returns:

true If *x* is better than the best known dual bound.
false otherwise.

Parameters:

x The value being compared with the best know dual bound.

6.4.4.12 bool ABA_MASTER::betterPrimal (double *x*) const

Can be used to check if a value is better than the best know primal bound.

Returns:

true If *x* is better than the best known primal bound,
false otherwise.

Parameters:

x The value compared with the primal bound.

6.4.4.13 void ABA_MASTER::branchingStrategy (BRANCHINGSTRAT *strat*) `[inline]`

Changes the branching strategy.

Parameters:

strat The new branching strategy.

Definition at line 2266 of file master.h.

6.4.4.14 ABA_MASTER::BRANCHINGSTRAT ABA_MASTER::branchingStrategy () const `[inline]`

Returns:

The branching strategy.

Definition at line 2261 of file master.h.

6.4.4.15 const ABA_CPUTIMER ∗ ABA_MASTER::branchingTime () const `[inline]`

Returns:

A pointer to the timer measuring the cpu time spent in finding and selecting the branching rules.

Definition at line 2001 of file master.h.

6.4.4.16 int ABA_MASTER::breadthFirstSearch (const ABA_SUB ∗ *s1*, const ABA_SUB ∗ *s2*) const `[protected]`

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*.

Returns:

-1 If subproblem *s1* has higher priority,
0 if both subproblems have equal priority,
1 otherwise.

Parameters:

s1 The first subproblem.

s2 The second subproblem.

6.4.4.17 bool ABA_MASTER::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}.

Returns:

true If the optimum solution of the problem is known and equals the primal bound,
false otherwise.

6.4.4.18 void ABA_MASTER::conElimAge (int *eps*) `[inline]`

Changes the age for the elimination of constraints.

Parameters:

eps The new age.

Definition at line 2246 of file master.h.

6.4.4.19 int ABA_MASTER::conElimAge () const `[inline]`

Returns:

The age for the elimination of constraints.

Definition at line 2241 of file master.h.

6.4.4.20 void ABA_MASTER::conElimEps (double *eps*) `[inline]`

Changes the tolerance for the elimination of constraints by the slack criterion.

Parameters:
eps The new tolerance.

Definition at line 2216 of file master.h.

6.4.4.21 double ABA_MASTER::conElimEps () const `[inline]`

Returns:
The zero tolerance for the elimination of constraints by the slack criterion.

Definition at line 2211 of file master.h.

6.4.4.22 void ABA_MASTER::conElimMode (CONELIMMODE *mode*) `[inline]`

Changes the constraint elimination mode.

Parameters:
mode The new constraint elimination mode.

Definition at line 2196 of file master.h.

6.4.4.23 ABA_MASTER::CONELIMMODE ABA_MASTER::conElimMode () const `[inline]`

Returns:
The mode for the elimination of constraints.

Definition at line 2191 of file master.h.

6.4.4.24 ABA_STANDARDPOOL< ABA_CONSTRAINT, ABA_VARIABLE > * ABA_MASTER::conPool () const `[inline]`

Returns:
A pointer to the default pool storing the constraints of the problem formulation.

Definition at line 1937 of file master.h.

6.4.4.25 void ABA_MASTER::countLp () `[inline, private]`

Increments the counter for linear programs and should be called in each optimization call of the LP-relaxation.

Definition at line 2006 of file master.h.

6.4.4.26 ABA_STANDARDPOOL< ABA_CONSTRAINT, ABA_VARIABLE > * ABA_MASTER::cutPool () const [inline]

Returns:

A pointer to the default pool for the generated cutting planes.

Definition at line 1942 of file master.h.

6.4.4.27 bool ABA_MASTER::cutting () const [inline]

Returns:

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.

Definition at line 1952 of file master.h.

6.4.4.28 int ABA_MASTER::dbThreshold () const [inline]

Returns:

The number of optimizations of a subproblem until sons are created. For further detatails we refer to *dbThreshold(int)*.

Definition at line 2366 of file master.h.

6.4.4.29 void ABA_MASTER::dbThreshold (int *threshold*) [inline]

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.

Parameters:

threshold The new value of the delayed branching threshold.

Definition at line 2361 of file master.h.

6.4.4.30 void ABA_MASTER::defaultLpSolver (OSISOLVER *osiSolver*) [inline]

Changes the default Lp solver.

Parameters:

osiSolver The new solver.

Definition at line 2276 of file master.h.

6.4.4.31 ABA_MASTER::OSISOLVER ABA_MASTER::defaultLpSolver () const `[inline]`

Returns:

The Lp Solver.

Definition at line 2271 of file master.h.

6.4.4.32 bool ABA_MASTER::delayedBranching (int *nOpt_*) const

Returns:

true If the number of optimizations *nOpt* of a subproblem exceeds the delayed branching threshold, false otherwise.

Parameters:

nOpt The number of optimizations of a subproblem.

6.4.4.33 int ABA_MASTER::depthFirstSearch (const ABA_SUB ∗ *s1*, const ABA_SUB ∗ *s2*) const `[protected]`

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()*.

Returns:

-1 If subproblem *s1* has higher priority,
0 if both subproblems have equal priority,
1 otherwise.

Parameters:

s1 The first subproblem.

s2 The second subproblem.

6.4.4.34 int ABA_MASTER::diveAndBestFirstSearch (const ABA_SUB ∗ *s1*, const ABA_SUB ∗ *s2*) const `[protected]`

Performs depth-first search until a feasible solution is found, then the search process is continued with best-first search.

Returns:

-1 If subproblem *s1* has higher priority,
0 if both subproblems have equal priority,
1 otherwise.

Parameters:

s1 The first subproblem.

s2 The second subproblem.

6.4.4.35 void ABA_MASTER::dualBound (double *x*)

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.

Parameters:

x The new value of the dual bound.

6.4.4.36 double ABA_MASTER::dualBound () const [inline]

Returns:

The value of the dual bound, i.e., the *upperBound()* for a maximization problem and the *lowerBound()* for a minimization problem, respectively.

Definition at line 1902 of file master.h.

6.4.4.37 void ABA_MASTER::eliminateFixedSet (bool *turnOn*) [inline]

This version of the function *eliminateFixedSet()* can be used to turn the elimination of fixed and set variables on or off.

Parameters:

turnOn The elimination is turned on if *turnOn* is *true*, otherwise it is turned off.

Definition at line 2156 of file master.h.

6.4.4.38 bool ABA_MASTER::eliminateFixedSet () const [inline]

Returns:

true Then we try to eliminate fixed and set variables from the linear program.
false Fixed or set variables are not eliminated.

Definition at line 2151 of file master.h.

6.4.4.39 virtual int ABA_MASTER::enumerationStrategy (const ABA_SUB $*$ *s1*, const ABA_SUB $*$ *s2*) [virtual]

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.

Returns:

1 If *s1* has higher priority than *s2*
0 if *s2* has higher priority it returns -1 , and if both subproblems have equal priority

Parameters:

s1 A pointer to subproblem.

s2 A pointer to subproblem.

6.4.4.40 void ABA_MASTER::enumerationStrategy (ENUMSTRAT *strat*) `[inline]`

This version of the function *enumerationStrategy()* changes the enumeration strategy.

Parameters:

strat The new enumeration strategy.

Definition at line 2256 of file master.h.

6.4.4.41 ABA_MASTER::ENUMSTRAT ABA_MASTER::enumerationStrategy () const `[inline]`

Returns:

The enumeration strategy.

Definition at line 2251 of file master.h.

6.4.4.42 virtual int ABA_MASTER::equalSubCompare (const ABA_SUB ∗ *s1*, const ABA_SUB ∗ *s2*) const `[protected, virtual]`

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. As the root node is compared with itself and has no branching rule, we have to insert the first line of this function.

Parameters:

s1 A subproblem.

s2 A subproblem.

Returns:

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.

6.4.4.43 bool ABA_MASTER::feasibleFound () const

We use this function ,e.g., to adapt the enumeration strategy in the *DiveAndBest-Strategy*.

This function is only correct if any primal bound better than plus/minus infinity corresponds to a feasible solution.

Returns:

true If a feasible solution of the optimization problem has been found.
false otherwise.

6.4.4.44 virtual ABA_SUB∗ ABA_MASTER::firstSub () `[protected, pure virtual]`

Returns:

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.

6.4.4.45 ABA_FIXCAND ∗ ABA_MASTER::fixCand () const `[inline, private]`

returns a pointer to the object storing the variables which are candidates for being fixed.

Definition at line 1932 of file master.h.

6.4.4.46 void ABA_MASTER::fixSetByRedCost (bool *on*) `[inline]`

Turns fixing and setting variables by reduced cost on or off.

Parameters:

on If *true*, then variable fixing and setting by reduced cost is turned on. Otherwise it is turned of.

Definition at line 2066 of file master.h.

6.4.4.47 bool ABA_MASTER::fixSetByRedCost () const `[inline]`

Returns:

true Then variables are fixed and set by reduced cost criteria.
false Then no variables are fixed or set by reduced cost criteria.

Definition at line 2061 of file master.h.

6.4.4.48 double ABA_MASTER::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.

Returns:

The guarantee for best known feasible solution in percent.

6.4.4.49 bool ABA_MASTER::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.

Returns:

true If the guarantee requirements are fulfilled,
false otherwise.

6.4.4.50 int ABA_MASTER::highestLevel () const `[inline]`

Returns:

The highest level in the tree which has been reached during the implicit enumeration.

Definition at line 2046 of file master.h.

6.4.4.51 ABA_HISTORY ∗ ABA_MASTER::history () const `[inline]`

Returns:

A pointer to the object storing the solution history of this branch and cut problem.

Definition at line 1922 of file master.h.

6.4.4.52 const ABA_CPUTIMER ∗ ABA_MASTER::improveTime () const `[inline]`

Returns:

A pointer to the timer measuring the cpu time spent in the heuristics for the computation of feasible solutions.

Definition at line 1991 of file master.h.

6.4.4.53 virtual void ABA_MASTER::initializeOptimization () `[protected, 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.

6.4.4.54 void ABA_MASTER::initializeOptSense (ABA_OPTSENSE::SENSE *sense*) `[protected]`

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.

Parameters:

sense The sense of the optimization (ABA_OPTSENSE::Min or ABA_OPTSENSE::Max).

6.4.4.55 virtual void ABA_MASTER::initializeParameters () `[protected, virtual]`

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 ().

6.4.4.56 virtual 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* = `false`) `[protected, virtual]`

Is overloaded such that also a first set of cutting planes can be inserted into the cutting plane pool.

Parameters:

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.

6.4.4.57 virtual void ABA_MASTER::initializePools (ABA_BUFFER< ABA_CONSTRAINT ∗ > & *constraints*, ABA_BUFFER< ABA_VARIABLE ∗ > & *Variables*, int *varPoolSize*, int *cutPoolSize*, bool *dynamicCutPool* = `false`) `[protected, virtual]`

Sets up the default pools for variables, constraints, and cutting planes.

Parameters:

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.

6.4.4.58 int ABA_MASTER::initLP () [private]

6.4.4.59 bool ABA_MASTER::knownOptimum (double & *optVal*)

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.

Returns:

true If a line with *problemName_* has been found,
false otherwise.

Parameters:

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.

6.4.4.60 void ABA_MASTER::logLevel (OUTLEVEL *mode*) [inline]

This version of the function *logLevel()* sets the output mode for the log-file.

Parameters:

mode The new value of the output mode.

Definition at line 2356 of file master.h.

6.4.4.61 ABA_MASTER::OUTLEVEL ABA_MASTER::logLevel () const [inline]

Returns:

The output mode for the log-file.

Definition at line 2351 of file master.h.

6.4.4.62 double ABA_MASTER::lowerBound () const [inline]

Returns:

The value of the global lower bound.

Definition at line 1885 of file master.h.

6.4.4.63 ABA_LPMASTEROSI∗ ABA_MASTER::lpMasterOsi () const [inline]

Definition at line 739 of file master.h.

6.4.4.64 const ABA_CPUTIMER ∗ ABA_MASTER::lpSolverTime () const `[inline]`

Returns:
A pointer to the timer measuring the cpu time required by the LP solver.

Definition at line 1981 of file master.h.

6.4.4.65 const ABA_CPUTIMER ∗ ABA_MASTER::lpTime () const `[inline]`

Returns:
A pointer to the timer measuring the cpu time spent in members of the LP-interface.

Definition at line 1976 of file master.h.

6.4.4.66 void ABA_MASTER::maxConAdd (int *max*) `[inline]`

Sets the maximal number of constraints that are added in an iteration of the cutting plane algorithm.

Parameters:
max The maximal number of constraints.

Definition at line 2086 of file master.h.

6.4.4.67 int ABA_MASTER::maxConAdd () const `[inline]`

Returns:
The maximal number of constraints which should be added in every iteration of the cutting plane algorithm.

Definition at line 2081 of file master.h.

6.4.4.68 void ABA_MASTER::maxConBuffered (int *max*) `[inline]`

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.

Parameters:
max The new maximal number of buffered constraints.

Definition at line 2096 of file master.h.

6.4.4.69 int ABA_MASTER::maxConBuffered () const `[inline]`

Returns:
The size of the buffer for generated constraints in the cutting plane algorithm.

Definition at line 2091 of file master.h.

6.4.4.70 void ABA_MASTER::maxCowTime (const ABA_STRING & *t*) [inline]

This version of the function *maxCowTime()* set the maximal wall-clock time for the optimization.

Parameters:

t The new value of the maximal wall-clock time in the form { hh:mm:ss}.

Definition at line 2311 of file master.h.

6.4.4.71 const ABA_STRING & ABA_MASTER::maxCowTime () const [inline]

The function *maxCowTime()*.

Returns:

The maximal wall-clock time for the optimization.

Definition at line 2306 of file master.h.

6.4.4.72 void ABA_MASTER::maxCpuTime (const ABA_STRING & *t*) [inline]

Sets the maximal usable cpu time for the optimization.

Parameters:

t The new value of the maximal cpu time in the form { "hh:mm:ss"}.

Definition at line 2301 of file master.h.

6.4.4.73 const ABA_STRING & ABA_MASTER::maxCpuTime () const [inline]

Returns:

The maximal cpu time which can be used by the optimization.

Definition at line 2296 of file master.h.

6.4.4.74 void ABA_MASTER::maxIterations (int *max*) [inline]

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().

Parameters:

max The new maximal number of iterations of the subproblem optimization (-1 means no limit).

Definition at line 2126 of file master.h.

6.4.4.75 int ABA_MASTER::maxIterations () const [inline]

Returns:

The maximal number of iterations per subproblem optimization (-1 means no iteration limit).

Definition at line 2121 of file master.h.

6.4.4.76 void ABA_MASTER::maxLevel (int *ml*)

This version of the function *maxLevel()* changes the maximal enumeration depth.

If it is set to 1 the \ algorithm becomes a pure cutting plane algorithm.

Parameters:

max The new value of the maximal enumeration level.

6.4.4.77 int ABA_MASTER::maxLevel () const [inline]

Returns:

The maximal depth up to which the enumeration should be performed. By default the maximal enumeration depth is *INT* .

Definition at line 2291 of file master.h.

6.4.4.78 void ABA_MASTER::maxVarAdd (int *max*) [inline]

Changes the maximal number of variables that are added in an iteration of the subproblem optimization.

Parameters:

max The new maximal number of added variables.

Definition at line 2106 of file master.h.

6.4.4.79 int ABA_MASTER::maxVarAdd () const [inline]

Returns:

The maximal number of variables which should be added in the column generation algorithm.

Definition at line 2101 of file master.h.

6.4.4.80 void ABA_MASTER::maxVarBuffered (int *max*) [inline]

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.

Parameters:

max The new maximal number of buffered variables.

Definition at line 2116 of file master.h.

6.4.4.81 int ABA_MASTER::maxVarBuffered () const `[inline]`

Returns:

The size of the buffer for the variables generated in the column generation algorithm.

Definition at line 2111 of file master.h.

6.4.4.82 void ABA_MASTER::minDormantRounds (int *nRounds*) `[inline]`

Sets the number of rounds a subproblem should stay dormant.

Parameters:

nRounds The new minimal number of dormant rounds.

Definition at line 2376 of file master.h.

6.4.4.83 int ABA_MASTER::minDormantRounds () const `[inline]`

Returns:

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.

Definition at line 2371 of file master.h.

6.4.4.84 void ABA_MASTER::nBranchingVariableCandidates (int *n*)

This version of the function *nbranchingVariableCandidates()* sets the number of tested branching variable candidates.

Parameters:

n The new value of the number of tested variables for becoming branching variable.

6.4.4.85 int ABA_MASTER::nBranchingVariableCandidates () const `[inline]`

Returns:

The number of variables that should be tested for the selection of the branching variable.

Definition at line 2281 of file master.h.

6.4.4.86 void ABA_MASTER::newFixed (int *n*) `[inline, private]`

Increments the counter of the number of fixed variables by *n*.

Definition at line 2011 of file master.h.

6.4.4.87 void ABA_MASTER::newRootReOptimize (bool *on*) `[inline]`

Turns the reoptimization of new root nodes of the remaining branch and bound tree on or off.

Parameters:
on If *true*, new root nodes are reoptimized.

Definition at line 2166 of file master.h.

6.4.4.88 bool ABA_MASTER::newRootReOptimize () const `[inline]`

Returns:
true Then a new root of the remaining \ tree is reoptimized such that the associated reduced costs can be used for the fixing of variables.
false A new root is not reoptimized.

Definition at line 2161 of file master.h.

6.4.4.89 void ABA_MASTER::newSub (int *level*) `[private]`

Registers a new subproblem which is on level *level* in enumeration tree.

It is called each time a new subproblem is generated.

6.4.4.90 int ABA_MASTER::nLp () const `[inline]`

Returns:
The number of optimized linear programs (only LP-relaxations).

Definition at line 2041 of file master.h.

6.4.4.91 int ABA_MASTER::nNewRoot () const `[inline]`

Returns:
The number of root changes of the remaining \ tree.

Definition at line 2051 of file master.h.

6.4.4.92 int ABA_MASTER::nSub () const `[inline]`

Returns:
The number of generated subproblems.

Definition at line 2036 of file master.h.

6.4.4.93 int ABA_MASTER::nSubSelected () const `[inline]`

Returns:
The number of subproblems which have already been selected from the set of open subproblems.

Definition at line 2056 of file master.h.

6.4.4.94 void ABA_MASTER::objInteger (bool *b*) `[inline]`

This version of function *objInteger()* sets the assumption that the objective function values of all feasible solutions are integer.

Parameters:

b The new value of the assumption.

Definition at line 2321 of file master.h.

6.4.4.95 bool ABA_MASTER::objInteger () const `[inline]`

Returns:

true Then we assume that all feasible solutions have integral objective function values, false otherwise.

Definition at line 2316 of file master.h.

6.4.4.96 ABA_OPENSUB ∗ ABA_MASTER::openSub () const `[inline]`

Returns:

A pointer to the set of open subproblems.

Definition at line 1927 of file master.h.

6.4.4.97 const ABA_MASTER& ABA_MASTER::operator= (const ABA_MASTER & *rhs*) `[private]`

6.4.4.98 STATUS ABA_MASTER::optimize ()

Performs the optimization by .

The status of the optimization.

6.4.4.99 void ABA_MASTER::optimumFileName (const char ∗ *name*) `[inline]`

Changes the name of the file in which the value of the optimum solution is searched.

Parameters:

name The new name of the file.

Definition at line 2136 of file master.h.

6.4.4.100 const ABA_STRING & ABA_MASTER::optimumFileName () const `[inline]`

Returns:

The name of the file that stores the optimum solutions.

Definition at line 2131 of file master.h.

6.4.4.101 const ABA_OPTSENSE ∗ ABA_MASTER::optSense () const `[inline]`

Returns:

A pointer to the object holding the optimization sense of the problem.

Definition at line 1917 of file master.h.

6.4.4.102 void ABA_MASTER::outLevel (OUTLEVEL *mode*) `[inline]`

The version of the function *outLevel()* sets the output mode.

Parameters:

mode The new value of the output mode.

Definition at line 2346 of file master.h.

6.4.4.103 ABA_MASTER::OUTLEVEL ABA_MASTER::outLevel () const `[inline]`

Returns:

The output mode.

Definition at line 2341 of file master.h.

6.4.4.104 virtual void ABA_MASTER::output () `[virtual]`

Does nothing but can be redefined in derived classes for output before the timing statistics.

6.4.4.105 void ABA_MASTER::pbMode (PRIMALBOUNDMODE *mode*) `[inline]`

Sets the mode of the primal bound initialization.

Parameters:

mode The new mode of the primal bound initialization.

Definition at line 2386 of file master.h.

6.4.4.106 ABA_MASTER::PRIMALBOUNDMODE ABA_MASTER::pbMode () const `[inline]`

Returns:

The mode of the primal bound initialization.

Definition at line 2381 of file master.h.

6.4.4.107 bool ABA_MASTER::pricing () const `[inline]`

Returns:

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.

Definition at line 1957 of file master.h.

6.4.4.108 void ABA_MASTER::pricingFreq (int *f*)

This version of the function *pricingFreq()* sets the number of linear programs being solved between two additional pricing steps.

Parameters:

f The pricing frequency.

6.4.4.109 int ABA_MASTER::pricingFreq () const `[inline]`

Returns:

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.

Definition at line 2391 of file master.h.

6.4.4.110 const ABA_CPUTIMER $*$ ABA_MASTER::pricingTime () const `[inline]`

Returns:

A pointer to the timer measuring the cpu time spent in pricing.

Definition at line 1996 of file master.h.

6.4.4.111 void ABA_MASTER::primalBound (double *x*)

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.

Parameters:

x The new value of the primal bound.

6.4.4.112 double ABA_MASTER::primalBound () const `[inline]`

Returns:

The value of the primal bound, i.e., the *lowerBound()* for a maximization problem and the *upperBound()* for a minimization problem, respectively.

Definition at line 1897 of file master.h.

6.4.4.113 bool ABA_MASTER::primalViolated (double *x*) const

Can be used to compare a value with the one of the best known primal bound.

If the objective function values of all feasible solutions are integer, then we do not have to be so carefully.

Returns:

true If *x* is not better than the best known primal bound,
false otherwise.

Parameters:

x The value being compared with the primal bound.

6.4.4.114 void ABA_MASTER::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.

6.4.4.115 void ABA_MASTER::printLP (bool *on*) `[inline]`

Turns the output of the linear program in every iteration on or off.

Parameters:

on If *true*, then the linear program is output, otherwise it is not output.

Definition at line 2076 of file master.h.

6.4.4.116 bool ABA_MASTER::printLP () const `[inline]`

Returns:

true Then the linear program is output every iteration of the subproblem optimization.
false The linear program is not output.

Definition at line 2071 of file master.h.

6.4.4.117 void ABA_MASTER::printParameters ()

Writes all parameters of the class ABA_MASTER together with their values to the global output stream.

6.4.4.118 const ABA_STRING$*$ ABA_MASTER::problemName () const

Returns:

A pointer to the name of the instance being optimized (as specified in the constructor of this class).

6.4.4.119 void ABA_MASTER::removeCons (int *n*) `[inline, private]`

Increments the counter for the total number of removed constraints by *n*.

Definition at line 2021 of file master.h.

6.4.4.120 void ABA_MASTER::removeVars (int *n*) `[inline, private]`

Increments the counter for the total number of removed variables by *n*.

Definition at line 2031 of file master.h.

6.4.4.121 void ABA_MASTER::requiredGuarantee (double *g*)

This version of the function *requiredGuarantee()* changes the guarantee specification.

Parameters:

g The new guarantee specification (in percent). 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.

6.4.4.122 double ABA_MASTER::requiredGuarantee () const `[inline]`

The guarantee specification for the optimization.

Definition at line 2286 of file master.h.

6.4.4.123 ABA_SUB $*$ ABA_MASTER::root () const `[inline]`

Can be used to access the root node of the \ tree.

Returns:

A pointer to the root node of the enumeration tree.

Definition at line 1907 of file master.h.

6.4.4.124 void ABA_MASTER::rootDualBound (double *x*) `[private]`

Updates the final dual bound of the root node.

This function should be only called at the end of the root node optimization.

6.4.4.125 void ABA_MASTER::rRoot (ABA_SUB $*$ *newRoot*, bool *reoptimize*) `[private]`

Sets the root of the remaining \ tree to *newRoot*.

If *reoptimize* is *true* a reoptimization of the subproblem $*$*newRoot* is performed. This is controlled via a function argument since it might not be desirable when we find a new *rRoot_* during the fathoming of a complete subtree ABA_SUB::FathomTheSubtree().

6.4.4.126 ABA_SUB $*$ ABA_MASTER::rRoot () const `[inline]`

Returns:

A pointer to the root of the remaining \ tree, i.e., the subproblem which is an ancestor of all open subproblems and has highest level in the tree.

Definition at line 1912 of file master.h.

6.4.4.127 ABA_SUB∗ ABA_MASTER::select () `[private]`

Returns a pointer to an open subproblem for further processing.

If the set of open subproblems is empty or one of the criteria for early termination of the optimization (maximal cpu time, maximal elapsed time, guarantee) is fulfilled 0 is returned.

6.4.4.128 const ABA_CPUTIMER ∗ ABA_MASTER::separationTime () const `[inline]`

Returns:

A pointer to the timer measuring the cpu time spent in the separation of cutting planes.

Definition at line 1986 of file master.h.

6.4.4.129 virtual bool ABA_MASTER::setSolverParameters (OsiSolverInterface ∗ *interface*, bool *solverIsApprox*) `[virtual]`

Set solver specific parameters. The default does nothing.

Returns:

true if an error has occured otherwise

6.4.4.130 void ABA_MASTER::showAverageCutDistance (bool *on*) `[inline]`

Turns the output of the average distance of the added cuts from the fractional solution on or off.

Parameters:

on If *true* the output is turned on, otherwise it is turned off.

Definition at line 2176 of file master.h.

6.4.4.131 bool ABA_MASTER::showAverageCutDistance () const `[inline]`

Returns:

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.

Definition at line 2171 of file master.h.

6.4.4.132 void ABA_MASTER::skipFactor (int *f*)

This version of the function *skipFactor()* sets the frequency for constraint and variable generation.

Parameters:

f The new value of the frequency.

6.4.4.133 int ABA_MASTER::skipFactor () const `[inline]`

Returns:

The frequency of subproblems in which constraints or variables should be generated.

Definition at line 2396 of file master.h.

6.4.4.134 ABA_MASTER::SKIPPINGMODE ABA_MASTER::skippingMode () const `[inline]`

Returns:

The skipping strategy.

Definition at line 2401 of file master.h.

6.4.4.135 void ABA_MASTER::skippingMode (SKIPPINGMODE *mode*) `[inline]`

This version of the function *skippingMode()* sets the skipping strategy.

Parameters:

mode The new skipping strategy.

Definition at line 2406 of file master.h.

6.4.4.136 bool ABA_MASTER::solveApprox () const `[inline]`

True, if an approximative solver should be used

Definition at line 1961 of file master.h.

6.4.4.137 void ABA_MASTER::status (STATUS *stat*) `[inline, private]`

This version of the function *status()* sets the status of the ABA_MASTER.

Definition at line 2146 of file master.h.

6.4.4.138 ABA_MASTER::STATUS ABA_MASTER::status () const `[inline]`

Returns:

The status of the ABA_MASTER.

Definition at line 2141 of file master.h.

6.4.4.139 void ABA_MASTER::tailOffNLp (int *n*) `[inline]`

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.

Parameters:

n The new number of LPs for the tailing off analysis.

Definition at line 2331 of file master.h.

6.4.4.140 int ABA_MASTER::tailOffNLp () const `[inline]`

The function *tailOffNLp()*.

Returns:

The number of linear programs considered in the tailing off analysis.

Definition at line 2326 of file master.h.

6.4.4.141 void ABA_MASTER::tailOffPercent (double *p*)

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.

Parameters:

p The new value for the tailing off analysis.

6.4.4.142 double ABA_MASTER::tailOffPercent () const `[inline]`

The function *tailOffPercent()*.

Returns:

The minimal change of the dual bound for the tailing off analysis in percent.

Definition at line 2336 of file master.h.

6.4.4.143 virtual void ABA_MASTER::terminateOptimization () `[protected, virtual]`

The default implementation of *terminateOptimization()* does nothing.

This virtual function can be used as an entrance point after the optimization process is finished.

6.4.4.144 void ABA_MASTER::theFuture () `[private]`

6.4.4.145 const ABA_COWTIMER ∗ ABA_MASTER::totalCowTime () const `[inline]`

Returns:

A pointer to the timer measuring the total wall clock time.

Definition at line 1966 of file master.h.

6.4.4.146 const ABA_CPUTIMER $*$ ABA_MASTER::totalTime () const `[inline]`

Returns:

A pointer to the timer measuring the total cpu time for the optimization.

Definition at line 1971 of file master.h.

6.4.4.147 void ABA_MASTER::treeInterfaceLowerBound (double *lb*) const `[private]`

Passes the new lower bound *lb* to the Tree Interface.

6.4.4.148 void ABA_MASTER::treeInterfaceNewNode (ABA_SUB $*$ *sub*) const `[private]`

Adds the subproblem *sub* to the stream storing information for graphical output of the enumeration tree if this logging is turned on.

6.4.4.149 void ABA_MASTER::treeInterfaceNodeBounds (int *id*, double *lb*, double *ub*) `[private]`

Updates the node information in the node with number *id* by writing the lower bound *lb* and the upper bound *ub* to the node.

6.4.4.150 void ABA_MASTER::treeInterfacePaintNode (int *id*, int *color*) const `[private]`

Assigns the *color* to the subproblem *sub* in the Tree Interface.

6.4.4.151 void ABA_MASTER::treeInterfaceUpperBound (double *ub*) const `[private]`

Passes the new upper bound *ub* to the Tree Interface.

6.4.4.152 double ABA_MASTER::upperBound () const `[inline]`

Returns:

The value of the global upper bound.

Definition at line 1891 of file master.h.

6.4.4.153 void ABA_MASTER::varElimAge (int *eps*) `[inline]`

Changes the age for the elimination of variables by the reduced cost criterion.

Parameters:

eps The new age.

Definition at line 2236 of file master.h.

6.4.4.154 int ABA_MASTER::varElimAge () const `[inline]`

Returns:
The age for the elimination of variables by the reduced cost criterion.

Definition at line 2231 of file master.h.

6.4.4.155 void ABA_MASTER::varElimEps (double *eps*) `[inline]`

Changes the tolerance for the elimination of variables by the reduced cost criterion.

Parameters:
eps The new tolerance.

Definition at line 2226 of file master.h.

6.4.4.156 double ABA_MASTER::varElimEps () const `[inline]`

Returns:
The zero tolerance for the elimination of variables by the reduced cost criterion.

Definition at line 2221 of file master.h.

6.4.4.157 void ABA_MASTER::varElimMode (VARELIMMODE *mode*) `[inline]`

Changes the variable elimination mode.

Parameters:
mode The new variable elimination mode.

Definition at line 2206 of file master.h.

6.4.4.158 ABA_MASTER::VARELIMMODE ABA_MASTER::varElimMode () const `[inline]`

Returns:
The mode for the elimination of variables.

Definition at line 2201 of file master.h.

6.4.4.159 ABA_STANDARDPOOL< ABA_VARIABLE, ABA_CONSTRAINT > * ABA_MASTER::varPool () const `[inline]`

Returns:
A pointer to the default pool storing the variables.

Definition at line 1947 of file master.h.

6.4.4.160 void ABA_MASTER::vbcLog (VBCMODE *mode*) `[inline]`

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().

Parameters:

mode The new mode.

Definition at line 2186 of file master.h.

6.4.4.161 ABA_MASTER::VBCMODE ABA_MASTER::vbcLog () const `[inline]`

Returns:

The mode of output for the Vbc-Tool.

Definition at line 2181 of file master.h.

6.4.4.162 void ABA_MASTER::writeTreeInterface (const char ∗ *info*, bool *time* = `true`) const `[private]`

Writes the string *info* to the stream associated with the Tree Interface.

A $ is preceded if the output is written to standard out for further pipelining. If *time* is true a time string is written in front of the information. The default value of *time* is *true*.

6.4.5 Friends And Related Function Documentation

6.4.5.1 friend class ABA_FIXCAND `[friend]`

Definition at line 78 of file master.h.

6.4.5.2 friend class ABA_SUB `[friend]`

Definition at line 77 of file master.h.

6.4.6 Member Data Documentation

6.4.6.1 const char∗ ABA_MASTER::BRANCHINGSTRAT_[ ] `[static]`

Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { BRANCHINGSTRAT[0]=="CloseHalf"}).

Definition at line 181 of file master.h.

6.4.6.2 BRANCHINGSTRAT ABA_MASTER::branchingStrategy_ [private]

The branching strategy.

Definition at line 1592 of file master.h.

6.4.6.3 ABA_CPUTIMER ABA_MASTER::branchingTime_ [private]

The timer for the cpu time spent in determining the branching rules.

Definition at line 1843 of file master.h.

6.4.6.4 int ABA_MASTER::conElimAge_ [private]

The number of iterations an elimination criterion must be satisfied until a constraint can be removed.

Definition at line 1804 of file master.h.

6.4.6.5 double ABA_MASTER::conElimEps_ [private]

The tolerance for the elimination of constraints by the mode *NonBinding/*.

Definition at line 1794 of file master.h.

6.4.6.6 CONELIMMODE ABA_MASTER::conElimMode_ [private]

The way constraints are automatically eliminated in the cutting plane algorithm.

Definition at line 1784 of file master.h.

6.4.6.7 const char∗ ABA_MASTER::CONELIMMODE_[ ] [static]

Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { CONELIMMODE[0]=="None"}).

Definition at line 240 of file master.h.

6.4.6.8 ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE>∗ ABA_MASTER::conPool_ [private]

The default pool with the constraints of the problem formulation.

Definition at line 1607 of file master.h.

6.4.6.9 ABA_STANDARDPOOL<ABA_CONSTRAINT, ABA_VARIABLE>∗ ABA_MASTER::cutPool_ [private]

The default pool of dynamically generated constraints.

Definition at line 1612 of file master.h.

6.4.6.10 bool ABA_MASTER::cutting_ [private]

If *true*, then constraints are generated in the optimization.

Definition at line 1636 of file master.h.

6.4.6.11 int ABA_MASTER::dbThreshold_ [private]

The number of optimizations of an ABA_SUB until branching is performed.

Definition at line 1697 of file master.h.

6.4.6.12 OSISOLVER ABA_MASTER::defaultLpSolver_ [private]

The default LP-Solver.

Definition at line 1601 of file master.h.

6.4.6.13 double ABA_MASTER::dualBound_ [private]

The best known dual bound.

Definition at line 1624 of file master.h.

6.4.6.14 bool ABA_MASTER::eliminateFixedSet_ [private]

If *true*, then nonbasic fixed and set variables are eliminated.

Definition at line 1764 of file master.h.

6.4.6.15 ENUMSTRAT ABA_MASTER::enumerationStrategy_ [private]

The enumeration strategy.

Definition at line 1588 of file master.h.

6.4.6.16 const char$*$ ABA_MASTER::ENUMSTRAT_[ ] [static]

Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { ENUMSTRAT[0]=="BestFirst"}).

Definition at line 163 of file master.h.

6.4.6.17 ABA_FIXCAND$*$ ABA_MASTER::fixCand_ [private]

The variables which are candidates for being fixed.

Definition at line 1632 of file master.h.

6.4.6.18 bool ABA_MASTER::fixSetByRedCost_ `[private]`

If *true*, then variables are fixed and set by reduced cost criteria.

Definition at line 1733 of file master.h.

6.4.6.19 int ABA_MASTER::highestLevel_ `[private]`

The highest level which has been reached in the enumeration tree.

Definition at line 1855 of file master.h.

6.4.6.20 ABA_HISTORY∗ ABA_MASTER::history_ `[private]`

The solution history.

Definition at line 1584 of file master.h.

6.4.6.21 ABA_CPUTIMER ABA_MASTER::improveTime_ `[private]`

The timer for the cpu time spent in the heuristics for the computation of feasible solutions.

Definition at line 1835 of file master.h.

6.4.6.22 OUTLEVEL ABA_MASTER::logLevel_ `[private]`

The amount of output written to the log file.

Definition at line 1711 of file master.h.

6.4.6.23 ABA_LPMASTEROSI∗ ABA_MASTER::lpMasterOsi_ `[private]`

Definition at line 1603 of file master.h.

6.4.6.24 ABA_CPUTIMER ABA_MASTER::lpSolverTime_ `[private]`

Definition at line 1826 of file master.h.

6.4.6.25 ABA_CPUTIMER ABA_MASTER::lpTime_ `[private]`

The timer for the cpu time spent in the LP-interface.

Definition at line 1825 of file master.h.

6.4.6.26 int ABA_MASTER::maxConAdd_ `[private]`

The maximal number of added constraints per iteration of the cutting plane algorithm.

Definition at line 1742 of file master.h.

6.4.6.27 int ABA_MASTER::maxConBuffered_ `[private]`

The size of the buffer for generated cutting planes.

Definition at line 1746 of file master.h.

6.4.6.28 ABA_STRING ABA_MASTER::maxCowTime_ `[private]`

The maximal available wall-clock time.

Definition at line 1680 of file master.h.

6.4.6.29 ABA_STRING ABA_MASTER::maxCpuTime_ `[private]`

The maximal available cpu time.

Definition at line 1676 of file master.h.

6.4.6.30 int ABA_MASTER::maxIterations_ `[private]`

The maximal number of iterations of the cutting plane/column generation algorithm in the subproblem.

Definition at line 1760 of file master.h.

6.4.6.31 int ABA_MASTER::maxLevel_ `[private]`

The maximal level in enumeration tree.

Up to this level subproblems are considered in the enumeration.

Definition at line 1672 of file master.h.

6.4.6.32 int ABA_MASTER::maxVarAdd_ `[private]`

The maximal number of added variables per iteration of the column generation algorithm.

Definition at line 1751 of file master.h.

6.4.6.33 int ABA_MASTER::maxVarBuffered_ `[private]`

The size of the buffer for generated variables.

Definition at line 1755 of file master.h.

6.4.6.34 int ABA_MASTER::minDormantRounds_ `[private]`

The minimal 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.

Definition at line 1703 of file master.h.

6.4.6.35 int ABA_MASTER::nAddCons_ `[private]`

The total number of added constraints.

Definition at line 1863 of file master.h.

6.4.6.36 int ABA_MASTER::nAddVars_ `[private]`

The total number of added variables.

Definition at line 1871 of file master.h.

6.4.6.37 int ABA_MASTER::nBranchingVariableCandidates_ `[private]`

The number of candidates that are evaluated for branching on variables.

Definition at line 1597 of file master.h.

6.4.6.38 bool ABA_MASTER::newRootReOptimize_ `[private]`

If *true*, then an already earlier processed node is reoptimized if it becomes the new root of the remaining \ tree.

Definition at line 1769 of file master.h.

6.4.6.39 int ABA_MASTER::nFixed_ `[private]`

The total number of fixed variables.

Definition at line 1859 of file master.h.

6.4.6.40 int ABA_MASTER::nLp_ `[private]`

The number of solved LPs.

Definition at line 1851 of file master.h.

6.4.6.41 int ABA_MASTER::nNewRoot_ `[private]`

The number of changes of the root of the remaining \ tree.

Definition at line 1879 of file master.h.

6.4.6.42 int ABA_MASTER::nRemCons_ `[private]`

The total number of removed constraints.

Definition at line 1867 of file master.h.

6.4.6.43 int ABA_MASTER::nRemVars_ `[private]`

The total number of removed variables.

Definition at line 1875 of file master.h.

6.4.6.44 int ABA_MASTER::nSub_ [private]

The number of generated subproblems.

Definition at line 1847 of file master.h.

6.4.6.45 int ABA_MASTER::nSubSelected_ [private]

The number of subproblems already selected from the list of open subproblems.

Definition at line 1650 of file master.h.

6.4.6.46 bool ABA_MASTER::objInteger_ [private]

true, if all objective function values of feasible solutions are assumed to be integer.

Definition at line 1685 of file master.h.

6.4.6.47 ABA_OPENSUB* ABA_MASTER::openSub_ [private]

The set of open subproblems.

Definition at line 1580 of file master.h.

6.4.6.48 ABA_STRING ABA_MASTER::optimumFileName_ [private]

The name of a file storing a list of optimum solutions of problem instances.

Definition at line 1774 of file master.h.

6.4.6.49 ABA_OPTSENSE ABA_MASTER::optSense_ [private]

The sense of the objective function.

Definition at line 1568 of file master.h.

6.4.6.50 const char* ABA_MASTER::OSISOLVER_[ ] [static]

Array for the literal values for possible Osi solvers.

Definition at line 284 of file master.h.

6.4.6.51 OUTLEVEL ABA_MASTER::outLevel_ [private]

The output mode.

Definition at line 1707 of file master.h.

6.4.6.52 const char∗ ABA_MASTER::OUTLEVEL_[ ] [static]

Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { OUTLEVEL[0]=="Silent"}).

Definition at line 138 of file master.h.

6.4.6.53 PRIMALBOUNDMODE ABA_MASTER::pbMode_ [private]

The mode of the primal bound initialization.

Definition at line 1715 of file master.h.

6.4.6.54 bool ABA_MASTER::pricing_ [private]

If *true*, then variables are generated in the optimization.

Definition at line 1640 of file master.h.

6.4.6.55 int ABA_MASTER::pricingFreq_ [private]

The number of solved LPs between two additional pricing steps.

Definition at line 1719 of file master.h.

6.4.6.56 ABA_CPUTIMER ABA_MASTER::pricingTime_ [private]

The timer for the cpu time spent in pricing.

Definition at line 1839 of file master.h.

6.4.6.57 double ABA_MASTER::primalBound_ [private]

The best known primal bound.

Definition at line 1620 of file master.h.

6.4.6.58 const char∗ ABA_MASTER::PRIMALBOUNDMODE_[ ] [static]

Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { PRIMALBOUNDMODE[0]=="None"}).

Definition at line 208 of file master.h.

6.4.6.59 bool ABA_MASTER::printLP_ [private]

If *true*, then the linear program is output every iteration.

Definition at line 1737 of file master.h.

6.4.6.60 ABA_STRING ABA_MASTER::problemName_ `[private]`

The name of the optimized problem.

Definition at line 1563 of file master.h.

6.4.6.61 bool ABA_MASTER::readParamFromFile_ `[private]`

Definition at line 1564 of file master.h.

6.4.6.62 double ABA_MASTER::requiredGuarantee_ `[private]`

The guarantee in percent which should be reached when the optimization stops.

If this value is 0.0 , then the optimum solution is determined.

Definition at line 1666 of file master.h.

6.4.6.63 ABA_SUB$*$ ABA_MASTER::root_ `[private]`

The root node of the enumeration tree.

Definition at line 1572 of file master.h.

6.4.6.64 double ABA_MASTER::rootDualBound_ `[private]`

The best known dual bound at the end of the optimization of the root node.

Definition at line 1628 of file master.h.

6.4.6.65 ABA_SUB$*$ ABA_MASTER::rRoot_ `[private]`

The root node of the remaining enumeration tree.

Definition at line 1576 of file master.h.

6.4.6.66 ABA_CPUTIMER ABA_MASTER::separationTime_ `[private]`

The timer for the cpu time spent in the separation

Definition at line 1830 of file master.h.

6.4.6.67 bool ABA_MASTER::showAverageCutDistance_ `[private]`

If *true* then the average distance of the added cutting planes is output every iteration of the cutting plane algorithm.

Definition at line 1779 of file master.h.

6.4.6.68 int ABA_MASTER::skipFactor_ `[private]`

The frequency constraints or variables are generated depending on the skipping mode.

Definition at line 1724 of file master.h.

6.4.6.69 SKIPPINGMODE ABA_MASTER::skippingMode_ [private]

Either constraints are generated only every *skipFactor_* subproblem (*SkipByNode*) only every *skipFactor_* level (*SkipByLevel*).

Definition at line 1729 of file master.h.

6.4.6.70 const char∗ ABA_MASTER::SKIPPINGMODE_[ ] [static]

Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { SKIPPINGMODE[0]=="None"}).

Definition at line 224 of file master.h.

6.4.6.71 bool ABA_MASTER::solveApprox_ [private]

If *true*, then an approximative solver is used to solve linear programs

Definition at line 1645 of file master.h.

6.4.6.72 STATUS ABA_MASTER::status_ [private]

The current status of the optimization.

Definition at line 1813 of file master.h.

6.4.6.73 const char∗ ABA_MASTER::STATUS_[ ] [static]

Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { STATUS[0]=="Optimal"}).

Definition at line 117 of file master.h.

6.4.6.74 int ABA_MASTER::tailOffNLp_ [private]

The number of LP-iterations for the tailing off analysis.

Definition at line 1689 of file master.h.

6.4.6.75 double ABA_MASTER::tailOffPercent_ [private]

The minimal change of the LP-value on the tailing off analysis.

Definition at line 1693 of file master.h.

6.4.6.76 ABA_COWTIMER ABA_MASTER::totalCowTime_ [private]

The timer for the total elapsed time.

Definition at line 1817 of file master.h.

6.4.6.77 ABA_CPUTIMER ABA_MASTER::totalTime_ [private]

The timer for the total cpu time for the optimization.

Definition at line 1821 of file master.h.

6.4.6.78 ostream∗ ABA_MASTER::treeStream_ [private]

A pointer to the log stream for the VBC-Tool.

Definition at line 1659 of file master.h.

6.4.6.79 int ABA_MASTER::varElimAge_ [private]

The number of iterations an elimination criterion must be satisfied until a variable can be removed.

Definition at line 1809 of file master.h.

6.4.6.80 double ABA_MASTER::varElimEps_ [private]

The tolerance for the elimination of variables by the mode *ReducedCost*.

Definition at line 1799 of file master.h.

6.4.6.81 VARELIMMODE ABA_MASTER::varElimMode_ [private]

The way variables are automatically eliminated in the column generation algorithm.

Definition at line 1789 of file master.h.

6.4.6.82 const char∗ ABA_MASTER::VARELIMMODE_[ ] [static]

Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { VARELIMMODE[0]=="None"}).

Definition at line 255 of file master.h.

6.4.6.83 ABA_STANDARDPOOL<ABA_VARIABLE, ABA_CONSTRAINT>∗ ABA_MASTER::varPool_ [private]

The default pool with the variables of the problem formulation.

Definition at line 1616 of file master.h.

6.4.6.84 VBCMODE ABA_MASTER::VbcLog_ [private]

Ouput for the Tree Interface is generated depending on the value of this variable.

Definition at line 1655 of file master.h.

6.4.6.85 const char∗ ABA_MASTER::VBCMODE_[ ] `[static]`

Literal values for the enumerators of the corresponding enumeration type. The order of the enumerators is preserved. (e.g., { VBCMODE[0]=="None"}).

Definition at line 272 of file master.h.

The documentation for this class was generated from the following file:

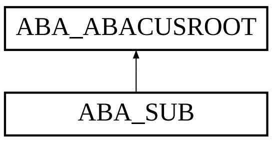
- Include/abacus/master.h

6.5 ABA_SUB Class Reference

class implements an abstract base class for a subproblem of the enumeration, i.e., a node of the \ tree.

`#include <sub.h>`

Inheritance diagram for ABA_SUB::



Public Types

- enum STATUS {
  Unprocessed, Active, Dormant, Processed,
  Fathomed }
- enum PHASE { Done, Cutting, Branching, Fathoming }

  The optimization of the subproblem can be in one of the following phases:.

Public Member Functions

- 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)

  The constructor for the root node of the enumeration tree.

- ABA_SUB (ABA_MASTER ∗master, ABA_SUB ∗father, ABA_BRANCHRULE ∗branchRule)

  The constructor for non-root nodes of the enumeration tree.

- virtual ∼ABA_SUB ()
- bool forceExactSolver () const
- 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
- void dualBound (double x)

  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.

- const ABA_SUB ∗ father () const
- ABA_LPSUB ∗ lp () const
- void maxIterations (int max)
- ABA_ACTIVE< ABA_CONSTRAINT, ABA_VARIABLE > ∗ actCon () const
- ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > ∗ actVar () const
- ABA_CONSTRAINT ∗ constraint (int i) const
- ABA_SLACKSTAT ∗ slackStat (int i) const
- ABA_VARIABLE ∗ variable (int i) const
- double lBound (int i) const
- void lBound (int i, double l)
- double uBound (int i) const
- void uBound (int i, double u)

  This version of the function uBound() *sets thef local upper bound of a variable.*

- ABA_FSVARSTAT ∗ fsVarStat (int i) const
- ABA_LPVARSTAT ∗ lpVarStat (int i) const
- double xVal (int i) const
- double yVal (int i) const
- bool ancestor (const ABA_SUB ∗sub) const
- ABA_MASTER ∗ master () const
- void removeVars (ABA_BUFFER< int > &remove)

  With function removeVars() *variables can be removed from the set of active variables.*

- void removeVar (int i)
- double nnzReserve () const
- bool relativeReserve () const
- ABA_BRANCHRULE ∗ branchRule () const
- bool 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.*

- virtual void removeCons (ABA_BUFFER< int > &remove)

  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.

- virtual void removeCon (int i)

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.

- int addConBufferSpace () const

  Can be used to determine the maximal number of the constraints which still can be added to the constraint buffer.

- int addVarBufferSpace () const

  Can be used to determine the maximal number of the variables which still can be added to the variable buffer.

- int nDormantRounds () const
- void ignoreInTailingOff ()
- virtual int addBranchingConstraint (ABA_POOLSLOT< ABA_CONSTRAINT, ABA_VARIABLE > ∗slot)

  Adds a branching constraint to the constraint buffer such that it is automatically added at the beginning of the cutting plane algorithm.

Protected Member Functions

- 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 ()

  Does nothing but can be used as an entrance point for problem specific activations by a reimplementation in derived classes.

- virtual void deactivate ()

  Can be used as entrance point for problem specific deactivations after the subproblem optimization.

- virtual int generateBranchRules (ABA_BUFFER< ABA_BRANCHRULE ∗ > &rules)
- virtual int branchingOnVariable (ABA_BUFFER< ABA_BRANCHRULE ∗ > &rules)

  Generates branching rules for two new subproblems by selecting a branching variable with the function selectBranchingVariable().

- virtual int selectBranchingVariable (int &variable)
- virtual int selectBranchingVariableCandidates (ABA_BUFFER< int > &candidates)

  Selects depending on the branching variable strategy given by the parameter { BranchingStrategy} in the file { .abacus} candidates that for branching variables.

- virtual int selectBestBranchingSample (int nSamples, ABA_BUFFER< ABA_BRANCHRULE ∗ > ∗∗samples)

  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().

- virtual void rankBranchingSample (ABA_BUFFER< ABA_BRANCHRULE ∗ > &sample, ABA_ARRAY< double > &rank)

  Computes for each branching rule of a branching sample a rank with the function rankBranchingRule().

- virtual double rankBranchingRule (ABA_BRANCHRULE ∗branchRule)
- double lpRankBranchingRule (ABA_BRANCHRULE ∗branchRule, int iterLimit=-1)

  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.

- virtual int compareBranchingSampleRanks (ABA_ARRAY< double > &rank1, ABA_ARRAY< double > &rank2)
- int closeHalfExpensive (int &branchVar, ABA_VARTYPE::TYPE branchVarType)

  Selects a single branching variable of type branchVarType, *with fractional part close to* 0.5 *and high absolute objective function coefficient.*

- int closeHalfExpensive (ABA_BUFFER< int > &variables, ABA_VARTYPE::TYPE branchVarType)

  This version of the function closeHalfExpensive() *selects several candidates for branching variables of type* branchVarType.

- int closeHalf (int &branchVar, ABA_VARTYPE::TYPE branchVarType)

  Searches a branching variable of type branchVarType, *with fraction as close to* 0.5 *as possible.*

- int closeHalf (ABA_BUFFER< int > &branchVar, ABA_VARTYPE::TYPE branchVarType)

  Searches searches several possible branching variable of type branchVarType, *with fraction as close to* 0.5 *as possible.*

- 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)

  The default implementation of the virtual initMakeFeas() *does nothing.*

- virtual int makeFeasible ()

  The default implementation of makeFeasible() *does nothing.*

- 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)

  The default implementation of setByLogImp() *does nothing.*

- virtual bool feasible ()=0

  The pure virtual function feasible() *checks for the feasibility of a solution of the LP-relaxation.*

- bool 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.

- virtual bool 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.

- virtual int separate ()
- virtual void conEliminate (ABA_BUFFER< int > &remove)

  Can be used as an entry point for application specific elimination of constraints by redefinig it in derived classes.

- virtual void nonBindingConEliminate (ABA_BUFFER< int > &remove)

  Retrieves the dynamic constraints with slack exceeding the value given by the parameter { ConElimEps}.

- virtual void basicConEliminate (ABA_BUFFER< int > &remove)
- virtual void varEliminate (ABA_BUFFER< int > &remove)

  Provides an entry point for application specific variable elimination that can be implemented by redefining this function in a derived class.

- void redCostVarEliminate (ABA_BUFFER< int > &remove)
- virtual int pricing ()
- virtual int improve (double &primalValue)

  Can be redefined in derived classes in order to implement primal heuristics for finding feasible solutions.

- virtual ABA_SUB ∗ generateSon (ABA_BRANCHRULE ∗rule)=0

  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.

- bool boundCrash () const
- virtual bool pausing ()
- bool infeasible ()
- virtual void varRealloc (int newSize)

  Reallocates memory that at most newSize *variables can be handled in the subproblem.*

- virtual void conRealloc (int newSize)

  Reallocates memory that at most newSize *constraints can be handled in the subproblem.*

- virtual ABA_LP::METHOD chooseLpMethod (int nVarRemoved, int nConRemoved, int nVarAdded, int nConAdded)
- virtual bool tailingOff ()
- bool betterDual (double x) const
- 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)

  Should collect the numbers of the variables to be fixed in variable *and the respective statuses in* status.

- virtual int fixAndSet (bool &newValues)

  Tries to fix and set variables both by logical implications and reduced cost criteria.

- virtual int fixing (bool &newValues, bool saveCand=false)
- virtual int setting (bool &newValues)

  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.

- 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 ()

  Is called if the global lower bound of a \ 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).

- virtual PHASE fathoming ()

  Fathoms the node, and if certain conditions are satisfied, also its ancestor.

- virtual PHASE cutting ()

  Iteratively solves the LP-relaxation, generates constraints and/or variables.

- virtual ABA_LPSUB ∗ generateLp ()
- virtual int initializeLp ()
- virtual int solveLp ()
- virtual bool exceptionFathom ()

  Can be used to specify a problem specific fathoming criterium that is checked before the separation or pricing.

- virtual bool exceptionBranch ()
- virtual bool solveApproxNow ()

Protected Attributes

- 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_

  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.

- ABA_ARRAY< ABA_LPVARSTAT ∗ > ∗ lpVarStat_

  A pointer to an array storing the status of each active variable in the linear program.

- ABA_ARRAY< double > ∗ lBound_
- ABA_ARRAY< double > ∗ uBound_
- ABA_ARRAY< ABA_SLACKSTAT ∗ > ∗ slackStat_

  A pointer to an array storing the statuses of the slack variables of the last solved linear program.

- ABA_TAILOFF ∗ tailOff_
- double dualBound_
- int nIter_
- int lastIterConAdd_
- int lastIterVarAdd_
- ABA_BRANCHRULE ∗ branchRule_
- 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.*

- ABA_LP::METHOD lpMethod_
- ABA_CUTBUFFER< ABA_VARIABLE, ABA_CONSTRAINT > ∗ addVarBuffer_
- ABA_CUTBUFFER< ABA_CONSTRAINT, ABA_VARIABLE > ∗ addConBuffer_
- ABA_BUFFER< int > ∗ removeVarBuffer_

  The buffer of the variables which are removed at the beginning of the next iteration.

- ABA_BUFFER< int > ∗ removeConBuffer_

  The buffer of the constraints which are removed at the beginning of the next iteration.

- double ∗ xVal_
- double ∗ yVal_
- double ∗ bInvRow_

  A row of the basis inverse associated with the infeasible variable infeasVar_ *or slack variable* infeasCon_.

- int infeasCon_
- int infeasVar_
- bool genNonLiftCons_

Private Member Functions

- virtual int _separate ()
- virtual int _conEliminate ()
- virtual int _varEliminate ()
- virtual int _pricing (bool &newValues, bool doFixSet=true)
- virtual int _improve (double &primalValue)
- virtual int _fixByLogImp (bool &newValues)

Returns 1, if a contradiction has been found, 0 otherwise.

- virtual void updateBoundInLp (int i)
- virtual double fixSetNewBound (int i)

  Returns the value which the upper and lower bounds of a variable should take after it is fixed or set.

- virtual void newDormantRound ()
- virtual PHASE _activate ()

  Allocates and initializes memory of the subproblem at the beginning of the optimization.

- virtual void _deactivate ()

  Deallocates the memory which is not required after the optimization of the subproblem.

- virtual int _initMakeFeas ()

  Tries to add variables to restore infeasibilities detected at initialization time.

- virtual int _makeFeasible ()

  Is called if the LP *is infeasible and adds inactive variables, which can make the* LP *feasible again, to the set of active variables.*

- virtual int _setByLogImp (bool &newValues)

  Tries to set variables according to logical implications of already set and fixed variables.

- virtual void infeasibleSub ()
- virtual void getBase ()
- virtual void activateVars (ABA_BUFFER< ABA_POOLSLOT< ABA_VARIABLE, ABA_CONSTRAINT > ∗ > &newVars)

  Adds the variables stored in the pool slots of newVars *to the set of active variables, but not to the linear program.*

- virtual void addVarsToLp (ABA_BUFFER< ABA_POOLSLOT< ABA_VARIABLE, ABA_CONSTRAINT > ∗ > &newVars, ABA_BUFFER< ABA_FSVARSTAT ∗ > ∗localStatus=0)

  Adds the variables stored in the pool slots of newVars *to the linear program.* localStatus *can specify a local status of fixing and setting.*

- virtual void _selectVars (ABA_BUFFER< ABA_POOLSLOT< ABA_VARIABLE, ABA_CONSTRAINT > ∗ > &newVars)
- virtual void _selectCons (ABA_BUFFER< ABA_POOLSLOT< ABA_CONSTRAINT, ABA_VARIABLE > ∗ > &newCons)

  Selects the master_->maxConAdd() *best constraints from the buffered constraints and stores them in* newCons*.*

- virtual int _removeVars (ABA_BUFFER< int > &remove)
- virtual int _removeCons (ABA_BUFFER< int > &remove)
- ABA_SUB (const ABA_SUB &rhs)
- const ABA_SUB & operator= (const ABA_SUB &rhs)

Private Attributes

- int level_
- int id_
- STATUS status_

- ABA_BUFFER< ABA_SUB ∗ > ∗ sons_
- int maxIterations_
- int nOpt_
- bool relativeReserve_

  If this member is true *then the space reserve of the following three members* varReserve_, conReserve_, *and* nnz-Reserve_ *is relative to the initial numbers of constraints, variables, and nonzeros, respectively. Otherwise, the values are casted to integers and regarded as absolute values.*

- double varReserve_
- double conReserve_
- double nnzReserve_
- int nDormantRounds_

  The number of subproblem optimizations the subproblem has already the status Dormant*.*

- bool activated_

  The variable is true *if the function* activate() *has been called from the function* _activate()*. This memorization is required such that a* deactivate() *is only called when* activate() *has been called.*

- bool ignoreInTailingOff_

  If this flag is set to true *then the next LP-solution is ignored in the tailing-off control. The default value of the variable is* false*. It can be set to* true *by the function* ignoreInTailingOff()*.*

- ABA_LP::METHOD lastLP_

  The method that was used to solve the last LP.

- ABA_CPUTIMER localTimer_
- bool forceExactSolver_

  Indicates whether to force the use of an exact solver to prepare branching etc.

Friends

- class ABA_MASTER
- class ABA_BOUNDBRANCHRULE
- class ABA_OPENSUB
- class ABA_LPSOLUTION< ABA_CONSTRAINT, ABA_VARIABLE >
- class ABA_LPSOLUTION< ABA_VARIABLE, ABA_CONSTRAINT >

6.5.1 Detailed Description

class implements an abstract base class for a subproblem of the enumeration, i.e., a node of the \ tree.

Definition at line 75 of file sub.h.

6.5.2 Member Enumeration Documentation

6.5.2.1 enum ABA_SUB::PHASE

The optimization of the subproblem can be in one of the following phases:.

Parameters:

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.

Enumeration values:

Done

Cutting

Branching

Fathoming

Definition at line 111 of file sub.h.

6.5.2.2 enum ABA_SUB::STATUS

A subproblem can have different statuses:

Parameters:

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.

Enumeration values:

Unprocessed

Active

Dormant

Processed

Fathomed

Definition at line 98 of file sub.h.

6.5.3 Constructor & Destructor Documentation

6.5.3.1 ABA_SUB::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`)

The constructor for the root node of the enumeration tree.

Parameters:

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.

6.5.3.2 ABA_SUB::ABA_SUB (ABA_MASTER ∗ *master*, ABA_SUB ∗ *father*, ABA_BRANCHRULE ∗ *branchRule*)

The constructor for non-root nodes of the enumeration tree.

Parameters:

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.

6.5.3.3 virtual ABA_SUB::∼ABA_SUB () `[virtual]`

The destructor only deletes the sons of the node.

The deletion of allocated memory is already performed when the node is fathomed. We recursively call the destructors of all subproblems contained in the enumeration tree below this subproblem itself.

If a subproblem has no sons and its status is either *Unprocessed* or *Dormant*, then it is still contained in the set of open subproblems, where it is removed from.

6.5.3.4 ABA_SUB::ABA_SUB (const ABA_SUB & *rhs*) `[private]`

6.5.4 Member Function Documentation

6.5.4.1 virtual PHASE ABA_SUB::_activate () `[private, virtual]`

Allocates and initializes memory of the subproblem at the beginning of the optimization.

The function returns the next phase of the optimization. This is either *Cutting* or *Fathoming* if the subproblem immediately turns out to be infeasible.

Since many objects of the class ABA_SUB can exist at the same time, yet in a sequential algorithm only one problem is active, a lot of memory can be saved if some memory is dynamically allocated when the subproblem becomes active and other information is stored in a compressed format for dormant problems.

These allocations and decompressions are performed by the function *_activate()*, the respective deallocations and compression of data is executed by the function *_deactivate()*.

Currently for all subproblems which have not been processed already (except for the root) we initialize the active constraints and variables with the respective data from the father node adapted by the branching information since so we can make sure that all fixed and set variables are active. A more flexible strategy might be desirable but also dangerous.

The virtual function *activate()* can perform problem specific activations. It is called before variables are fixed by logical implications, because, e.g., for problems on graphs, the graph associated with the subproblem might have to be activated.

Moreover, the function *_activate()* is redundant in the sense that it is called only once and could be substituted by a function. However, having a future generalization to non \ in mind, we keep this function.

6.5.4.2 virtual int ABA_SUB::_conEliminate () `[private, virtual]`

Returns the number of eliminated constraints.

Only dynamic constraints are eliminated from the *LP*.

It might be worth to implement problem specific versions of this function.

6.5.4.3 virtual void ABA_SUB::_deactivate () `[private, virtual]`

Deallocates the memory which is not required after the optimization of the subproblem.

The virtual dummy function *deactivate()* can perform problem specific deactivations.

As the function *_activate()*, the function *_deactivate()* is redundant in the sense that it is called only once and could be substituted by a function. However, having a future generalization to non \ in mind, we keep this function.

6.5.4.4 virtual int ABA_SUB::_fixByLogImp (bool & *newValues*) `[private, virtual]`

Returns 1, if a contradiction has been found, 0 otherwise.

The parameter *newValues* is set to *true* if a variable is fixed to value different from its value in the last solved linear program.

6.5.4.5 virtual int ABA_SUB::_improve (double & *primalValue*) `[private, virtual]`

Tries to find a better feasible solution.

If a better solution is found its value is stored in *primalValue* and we return 1, otherwise we return 0.

> If the upper bound has been initialized with the optimum solution or with the optimum solution plus/minus one these primal heuristics are skipped.

> The primal bound, if improved, is either updated in the function *cutting()*, from which *_improved()* is called, are can be updated in the function *improve()* of an application in a derived class.

6.5.4.6 virtual int ABA_SUB::_initMakeFeas () `[private, virtual]`

Tries to add variables to restore infeasibilities detected at initialization time.

It returns 0 if variables could be activated which might restore feasibility, otherwise it returns 1.

> The function should analyse the constraints stored in ABA_LPSUB::infeasCons_ and try to add inactive variables which could restore the infeasibility.

> The new variables are only added to the set of active variables but not to the linear program since no linear program exists when this function is called.

6.5.4.7 virtual int ABA_SUB::_makeFeasible () `[private, virtual]`

Is called if the *LP* is infeasible and adds inactive variables, which can make the *LP* feasible again, to the set of active variables.

The function returns *0* if feasibility might have been restored and *1* if it is guaranteed that the linear program is infeasible on the complete variable set.

6.5.4.8 virtual int ABA_SUB::_pricing (bool & *newValues*, bool *doFixSet* = `true`) `[private, virtual]`

If *doFixSet* is *true*, then we try to fix and set variables, if all inactive variables price out correctly. In this case *newValues* becomes *true* of a variable is set or fixed to a value different from its value in the last linear program.

In a pricing step the reduced costs of inactive variables are computed and variables with positive (negative) reduced costs in a maximization (minimization) problem are activated.

The function *_pricing()* returns the 1 if no global optimality can be guaranteed, since variables have negative reduced costs, it returns 2 if before a pricing step can be performed, non-liftable constraints have to be removed, and 0 if the LP-solution is global dual feasible.

Also if there are no inactive variables, this function is called since it will also try to fix and set variables.

true is the default value of *doFixSet*. No variables should be fixed or set if *_pricing()* is called from *_makeFeasible()*.

6.5.4.9 virtual int ABA_SUB::_removeCons (ABA_BUFFER< int > & *remove*) `[private, virtual]`

Removes the constraints with numbers *remove* from the set of active constraints.

6.5.4.10 virtual int ABA_SUB::_removeVars (ABA_BUFFER< int > & *remove*) `[private, virtual]`

6.5.4.11 virtual void ABA_SUB::_selectCons (ABA_BUFFER< ABA_POOLSLOT< ABA_CONSTRAINT, ABA_VARIABLE > ∗ > & *newCons*) `[private, virtual]`

Selects the *master_->maxConAdd()* best constraints from the buffered constraints and stores them in *newCons*.

6.5.4.12 virtual void ABA_SUB::_selectVars (ABA_BUFFER< ABA_POOLSLOT< ABA_VARIABLE, ABA_CONSTRAINT > ∗ > & *newVars*) `[private, virtual]`

Selects the *master_->maxVarAdd()* best variables from the buffered variables.

Parameters:

newVars Holds the selected variables after the call.

6.5.4.13 virtual int ABA_SUB::_separate () `[private, virtual]`

Returns the number of generated cutting planes.

6.5.4.14 virtual int ABA_SUB::_setByLogImp (bool & *newValues*) `[private, virtual]`

Tries to set variables according to logical implications of already set and fixed variables.

Since logical implications are problem specific the virtual function *setByLogImp()* is called to find variables which can be set. If a variable is set to a new value, i.e., a value different from the one in the last solved LP, *newValues* is set to *true*. If such a setting implies a contradiction, *_setByLogImp()* returns 1, otherwise it returns 0.

6.5.4.15 virtual int ABA_SUB::_varEliminate () `[private, virtual]`

Returns the number of eliminated variables.

Only dynamic variables can be eliminated.

6.5.4.16 ABA_ACTIVE< ABA_CONSTRAINT, ABA_VARIABLE > ∗ ABA_SUB::actCon () const `[inline]`

Returns:

A pointer to the currently active constraints.

Definition at line 2249 of file sub.h.

6.5.4.17 virtual void ABA_SUB::activate () `[protected, virtual]`

Does nothing but can be used as an entrance point for problem specific activations by a reimplementation in derived classes.

6.5.4.18 virtual void ABA_SUB::activateVars (ABA_BUFFER< ABA_POOLSLOT< ABA_VARIABLE, ABA_CONSTRAINT > ∗ > & *newVars*) `[private, virtual]`

Adds the variables stored in the pool slots of *newVars* to the set of active variables, but not to the linear program.

If the new number of variables exceeds the maximal number of variables an automatic reallocation is performed.

6.5.4.19 ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > ∗ ABA_SUB::actVar () const `[inline]`

Returns:

A pointer to the currently active variables.

Definition at line 2254 of file sub.h.

6.5.4.20 int ABA_SUB::addBranchingConstraint (ABA_POOLSLOT< ABA_CONSTRAINT, ABA_VARIABLE > ∗ *slot*) `[inline, virtual]`

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()*.

Returns:

0 If the constraint could be added,
1 otherwise.

Parameters:

slot A pointer to the pools slot containing the branching constraint.

Definition at line 2143 of file sub.h.

6.5.4.21 int ABA_SUB::addConBufferSpace () const `[inline]`

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.

Returns:

The number of constraints which still can be inserted into the constraint buffer.

Definition at line 2148 of file sub.h.

6.5.4.22 virtual int ABA_SUB::addCons (ABA_BUFFER< ABA_POOLSLOT< ABA_CONSTRAINT, ABA_VARIABLE > ∗ > & *newCons*) `[protected, virtual]`

Adds constraints to the active constraints and the linear program.

Returns:

The number of added constraints.

Parameters:

newCons A buffer storing the pool slots of the new constraints.

6.5.4.23 virtual int ABA_SUB::addCons (ABA_BUFFER< ABA_CONSTRAINT ∗ > & *constraints*, ABA_POOL< ABA_CONSTRAINT, ABA_VARIABLE > ∗ *pool* = `0`, ABA_BUFFER< bool > ∗ *keepInPool* = `0`, ABA_BUFFER< double > ∗ *rank* = `0`) `[protected, virtual]`

Tries to add new constraints to the constraint buffer and a pool.

The memory management of added constraints is passed to \ by calling this function.

Returns:

The number of added constraints.

Parameters:

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.

6.5.4.24 int ABA_SUB::addVarBufferSpace () const `[inline]`

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.

Returns:

The number of variables which still can be inserted into the variable buffer.

Definition at line 2153 of file sub.h.

6.5.4.25 virtual int ABA_SUB::addVars (ABA_BUFFER< ABA_POOLSLOT< ABA_VARIABLE, ABA_CONSTRAINT > ∗ > & *newVars*) `[protected, virtual]`

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.

Returns:

The number of added variables. We require this feature in derived classes if variables of *newVars* can be discarded if they are already active.

Parameters:

newVars A buffer storing the pool slots of the new variables.

6.5.4.26 virtual int ABA_SUB::addVars (ABA_BUFFER< ABA_VARIABLE ∗ > & *variables*, ABA_POOL< ABA_VARIABLE, ABA_CONSTRAINT > ∗ *pool* = `0`, ABA_BUFFER< bool > ∗ *keepInPool* = `0`, ABA_BUFFER< double > ∗ *rank* = `0`) `[protected, virtual]`

Tries to add new variables to the variable buffer and a pool.

The memory management of added variables is passed to \ by calling this function.

Returns:

The number of added variables.

Parameters:

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.

6.5.4.27 virtual void ABA_SUB::addVarsToLp (ABA_BUFFER< ABA_POOLSLOT< ABA_VARIABLE, ABA_CONSTRAINT > * > & *newVars*, ABA_BUFFER< ABA_FSVARSTAT * > * *localStatus* = `0`) `[private, virtual]`

Adds the variables stored in the pool slots of *newVars* to the linear program. *localStatus* can specify a local status of fixing and setting.

If the local ABA_FSVARSTAT of the added variables differs from their global status, then this local status can be stated in *localStatus*. Per default the value of *localStatus* is 0.

6.5.4.28 bool ABA_SUB::ancestor (const ABA_SUB * *sub*) const

Returns:

true If *this* subproblem is an ancestor of the subproblem *sub*. We define that a subproblem is its own ancestor, false otherwise.

Parameters:

sub A pointer to a subproblem.

6.5.4.29 virtual void ABA_SUB::basicConEliminate (ABA_BUFFER< int > & *remove*) `[protected, virtual]`

Retrieves all dynamic constraints having basic slack variable.

Parameters:

remove Stores the nonbinding constraints.

6.5.4.30 bool ABA_SUB::betterDual (double *x*) const `[protected]`

Returns:

true If *x* is better than the best known dual bound of the subproblem,
false otherwise.

6.5.4.31 bool ABA_SUB::boundCrash () const `[protected]`

Returns:

true If the dual bound is worse than the best known primal bound,
false otherwise.

6.5.4.32 virtual PHASE ABA_SUB::branching () `[protected, virtual]`

Is called if the global lower bound of a \ 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 \ nodes without generating sons of this node.

Finally if none of the previous conditions is satisfied we generate new subproblems.

Returns:

Done If sons of the subproblem could be generated,
Fathoming otherwise.

6.5.4.33 virtual int ABA_SUB::branchingOnVariable (ABA_BUFFER< ABA_BRANCHRULE ∗ > & *rules*) `[protected, virtual]`

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.

Returns:

0 If branching rules could be found,
1 otherwise}

Parameters:

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.

6.5.4.34 ABA_BRANCHRULE ∗ ABA_SUB::branchRule () const `[inline]`

Returns:

A pointer to the branching rule of the subproblem.

Definition at line 2138 of file sub.h.

6.5.4.35 virtual ABA_LP::METHOD ABA_SUB::chooseLpMethod (int *nVarRemoved*, int *nConRemoved*, int *nVarAdded*, int *nConAdded*) `[protected, virtual]`

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.

Returns:

The method the next linear programming relaxation is solved with.

Parameters:

nVarRemoved The number of removed variables.

nConRemoved The number of removed constraints.

nVarAdded The number of added variables.

nConAdded The number of added constraint.

6.5.4.36 int ABA_SUB::closeHalf (ABA_BUFFER< int > & *branchVar*, ABA_VARTYPE::TYPE *branchVarType*) `[protected]`

Searches searches several possible branching variable of type *branchVarType*, with fraction as close to 0.5 as possible.

Returns:

0 If at least one branching variable is found,
1 otherwise.

Parameters:

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.

6.5.4.37 int ABA_SUB::closeHalf (int & *branchVar*, ABA_VARTYPE::TYPE *branchVarType*) `[protected]`

Searches a branching variable of type *branchVarType*, with fraction as close to 0.5 as possible.

Returns:

0 If a branching variable is found,
1 otherwise.

Parameters:

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.

6.5.4.38 int ABA_SUB::closeHalfExpensive (ABA_BUFFER< int > & *variables*, ABA_VARTYPE::TYPE *branchVarType*) `[protected]`

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..

Returns:

0 If at least one branching variable is found,
1 otherwise.

Parameters:

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.

6.5.4.39 int ABA_SUB::closeHalfExpensive (int & *branchVar*, ABA_VARTYPE::TYPE *branchVarType*) `[protected]`

Selects a single branching variable of type *branchVarType*, with fractional part close to 0.5 and high absolute objective function coefficient.

This is the default strategy from the TSP project JRT94}.

Returns:

0 If a branching variable is found,
1 otherwise.

Parameters:

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.

6.5.4.40 virtual int ABA_SUB::compareBranchingSampleRanks (ABA_ARRAY< double > & *rank1*, ABA_ARRAY< double > & *rank2*) `[protected, virtual]`

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.

Returns:

1 If *rank1* is better.
0 If both ranks are equal.
-1 If *rank2* is better.

6.5.4.41 virtual void ABA_SUB::conEliminate (ABA_BUFFER< int > & *remove*) `[protected, virtual]`

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.

Parameters:

remove The constraints that should be eliminated must be inserted in this buffer.

6.5.4.42 virtual void ABA_SUB::conRealloc (int *newSize*) `[protected, virtual]`

Reallocates memory that at most *newSize* constraints can be handled in the subproblem.

Parameters:

newSize The new maximal number of constraints of the subproblem.

6.5.4.43 ABA_CONSTRAINT* ABA_SUB::constraint (int *i*) const

Returns:

A pointer to the *i-th* active constraint.

Parameters:

i The constraint being accessed.

6.5.4.44 virtual int ABA_SUB::constraintPoolSeparation (int *ranking* = `0`, ABA_POOL< ABA_CONSTRAINT, ABA_VARIABLE > * *pool* = `0`, double *minViolation* = `0.001`) `[protected, virtual]`

Tries to generate inactive constraints from a pool.

Returns:

The number of generated constraints.

Parameters:

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, 3: rank determined by ABA_CONVAR::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.

6.5.4.45 virtual PHASE ABA_SUB::cutting () `[protected, virtual]`

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. A reader even studying these lines has been very brave! Therefore, the first reader of these lines is invited to a beer from the author.

Returns:

Fathoming If one of the conditions for fathoming the subproblem is satisfied.
Branching If the subproblem should be splitted in further subproblems.

6.5.4.46 virtual void ABA_SUB::deactivate () `[protected, virtual]`

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. This function is called from *_deactivate()*.

6.5.4.47 void ABA_SUB::dualBound (double *x*)

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 \ during the subproblem optimization.

Parameters:

x The new value of the dual bound.

6.5.4.48 double ABA_SUB::dualBound () const `[inline]`

Returns:

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.

Definition at line 2229 of file sub.h.

6.5.4.49 virtual double ABA_SUB::dualRound (double *x*) `[protected, virtual]`

Returns:

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*.

Parameters:

x The value that should be rounded if possible.

6.5.4.50 virtual bool ABA_SUB::exceptionBranch () [protected, virtual]

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*.

Returns:

true If the subproblem should be fathomed,
false otherwise.

6.5.4.51 virtual bool ABA_SUB::exceptionFathom () [protected, virtual]

Can be used to specify a problem specific fathoming criterium that is checked before the separation or pricing.

The default implementation always returns *false*.

Returns:

true If the subproblem should be fathomed,
false otherwise.

6.5.4.52 const ABA_SUB ∗ ABA_SUB::father () const [inline]

Returns:

A pointer to the father of the subproblem in the \ tree.

Definition at line 2234 of file sub.h.

6.5.4.53 virtual void ABA_SUB::fathom (bool *reoptimize*) [protected, virtual]

Fathoms a node and recursively tries to fathom its father.

If the root of the remaining \ tree is fathomed we are done since the optimization problem has been solved.

Otherwise, we count the number of unfathomed sons of the father of the subproblem being fathomed. If all sons of the father are fathomed it is recursively fathomed, too. If the father is the root of the remaining \ tree and only one of its sons is unfathomed, then this unfathomed son becomes the new root of the remaining \ tree.

We could stop the recursive fathoming already at the root of the remaining \ tree. But, we proceed until the root of the complete tree was visited to be really correct.

Note:

Use the function *ExceptionFathom()* for specifying problem specific fathoming criteria.

Parameters:

reoptimize If *reoptimize* is *true*, then we perform a reoptimization in the new root. This is controlled via a parameter since it might not be desirable when we find a new root during the fathoming of a complete subtree with the function *fathomTheSubTree()*.

6.5.4.54 virtual PHASE ABA_SUB::fathoming () `[protected, virtual]`

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.

The called function *fathom()* fathoms the subproblem itself and recursively also tries to fathom its father in the enumeration tree. The argument of *fathom()* is *true* as a possibly detected new root should be reoptimized in order to receive better criteria for fixing variables by reduced costs.

In the parallel version, only the subproblem itself is fathomed. No processed unfathomed nodes are kept in memory (father_=0).

Returns:

The function always returns *Done*.

6.5.4.55 virtual void ABA_SUB::fathomTheSubTree () `[protected, virtual]`

Fathoms all nodes in the subtree rooted at this subproblem.

Dormant and *Unprocessed* nodes are also removed from the set of open subproblems.

If the subproblem is already *Fathomed* we do not have to proceed in this branch. Otherwise, we fathom the node and continue with all its sons. The actual fathoming starts at the unfathomed leaves of the subtree and recursively goes up through the tree.

6.5.4.56 virtual bool ABA_SUB::feasible () `[protected, pure 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.

Returns:

true If the LP-solution is feasible,
false otherwise.

6.5.4.57 int ABA_SUB::findNonFixedSet (int & *branchVar*, ABA_VARTYPE::TYPE *branchVarType*) `[protected]`

Selects the first variable that is neither fixed nor set.

Returns:
0 If a variable neither fixed nor set is found,
1 otherwise.

Parameters:
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).

6.5.4.58 int ABA_SUB::findNonFixedSet (ABA_BUFFER< int > & *branchVar*, ABA_VARTYPE::TYPE *branchVarType*) `[protected]`

Selects the first variables that are neither fixed nor set.

Returns:
0 If at least one variable neither fixed nor set is found,
1 otherwise.

Parameters:
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.

6.5.4.59 virtual int ABA_SUB::fix (int *i*, ABA_FSVARSTAT $*$ *newStat*, bool & *newValue*) `[protected, virtual]`

Fixes a variable.

If the variable which is currently fixed is already set then we must not change its bounds in the LP since it might be eliminated.

Returns:
1 If a contradiction is found,
0 otherwise.

Parameters:
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*.

6.5.4.60 virtual int ABA_SUB::fixAndSet (bool & *newValues*) `[protected, virtual]`

Tries to fix and set variables both by logical implications and reduced cost criteria.

Actually, variables fixed or set to 0 could be eliminated. However, this could lead to a loss of important structural information for fixing and setting further variables later, for the computation of feasible solutions, for the separation and for detecting contradictions. Therefore, we do not eliminate these variables per default.

Returns:
1 If a contradiction is found,
0 otherwise.

Parameters:
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*.

6.5.4.61 virtual bool ABA_SUB::fixAndSetTime () `[protected, virtual]`

Controls if variables should be fixed or set when all variables price out corretly.

The default implementation always returns *true*.

Returns:
true If variables should be fixed and set,
false otherwise.

6.5.4.62 virtual void ABA_SUB::fixByLogImp (ABA_BUFFER< int > & *variable*, ABA_BUFFER< ABA_FSVARSTAT ∗ > & *status*) `[protected, virtual]`

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.

Parameters:
variables The variables which should be fixed.
status The statuses these variables should be fixed to.

6.5.4.63 virtual int ABA_SUB::fixByRedCost (bool & *newValues*, bool *saveCand*) `[protected, virtual]`

Tries to fix variables according to the reduced cost criterion.

Returns:
1 If a contradiction is found,
0 otherwise.

Parameters:
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 \ is processed.

6.5.4.64 virtual int ABA_SUB::fixing (bool & *newValues*, bool *saveCand* = `false`) `[protected, virtual]`

Tries to fix variables by reduced cost criteria and logical implications.

Returns:

1 If a contradiction is found,
0 otherwise.

Parameters:

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.

6.5.4.65 virtual double ABA_SUB::fixSetNewBound (int *i*) `[private, virtual]`

Returns the value which the upper and lower bounds of a variable should take after it is fixed or set.

6.5.4.66 bool ABA_SUB::forceExactSolver () const `[inline]`

Returns:

Whether using the exact solver is forced.

Definition at line 2207 of file sub.h.

6.5.4.67 ABA_FSVARSTAT $*$ ABA_SUB::fsVarStat (int *i*) const `[inline]`

In a $\backslash$ algorithm we also would have to refer to the global variable status. While this subproblem is processed another subproblem could change the global status.

Returns:

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()*).

Parameters:

i The number of the variable.

Definition at line 2192 of file sub.h.

6.5.4.68 virtual int ABA_SUB::generateBranchRules (ABA_BUFFER< ABA_BRANCHRULE * > & *rules*) [protected, virtual]

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.

Returns:
0 If branching rules could be found,
1 otherwise.

Parameters:
rules If branching rules are found, then they are stored in this buffer.

6.5.4.69 virtual ABA_LPSUB* ABA_SUB::generateLp () [protected, virtual]

Instantiates an *LP* for the solution of the *LP-relaxation* in this subproblem.

This function is redefined in a derived class for a specific *LP-solver* interface

This function is defined in the file *lpif.cc*.

Returns:
A pointer to an object of type ABA_LPSUB.

6.5.4.70 virtual ABA_SUB* ABA_SUB::generateSon (ABA_BRANCHRULE * *rule*) [protected, pure virtual]

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*.

Parameters:
rule The branching rule with which the subproblem is generated.

6.5.4.71 virtual void ABA_SUB::getBase () [private, virtual]

Updates the status of the variables and the slack variables.

6.5.4.72 virtual bool ABA_SUB::goodCol (ABA_COLUMN & *col*, ABA_ARRAY< double > & *row*, double *x*, double *lb*, double *ub*) [protected, virtual]

Returns:
true If the column *col* might restore feasibiblity if the variable with value *x* turns out to be infeasible,
false otherwise.

Parameters:

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.

6.5.4.73 virtual double ABA_SUB::guarantee () `[protected, virtual]`

May not be called if the lower bound is 0 and upper bound not equal to 0.

The guarantee that can be given for the subproblem.

6.5.4.74 virtual bool ABA_SUB::guaranteed () `[protected, virtual]`

Returns:

true If the lower and the upper bound of the subproblem satisfies the guarantee requirements,
false otherwise.

6.5.4.75 int ABA_SUB::id () const `[inline]`

Returns:

The identity number of the subproblem.

Definition at line 2217 of file sub.h.

6.5.4.76 void ABA_SUB::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).

6.5.4.77 virtual int ABA_SUB::improve (double & *primalValue*) `[protected, virtual]`

Can be redefined in derived classes in order to implement primal heuristics for finding feasible solutions.

The default implementation does nothing.

Returns:

0 If no better solution could be found,
1 otherwise.

Parameters:

primalValue Should hold the value of the feasible solution, if a better one is found.

6.5.4.78 bool ABA_SUB::infeasible () `[protected]`

Returns:

true If the subproblem does not contain a feasible solution,
false otherwise.

6.5.4.79 virtual void ABA_SUB::infeasibleSub () `[private, virtual]`

Should be called if a subproblem turns out to be infeasible.

It sets the dual bound of the subproblem correctly.

6.5.4.80 virtual void ABA_SUB::initializeCons (int *maxCon*) `[protected, virtual]`

Initializes the active constraint set.

Parameters:

maxCon The maximal number of constraints of the subproblem.

6.5.4.81 virtual int ABA_SUB::initializeLp () `[protected, virtual]`

Initializes the linear program.

Since not all variables might be active we first have to try making the *LP* feasible again by the addition of variables. If this fails, i.e., *_initMakeFeas()* has a nonzero return value, we return 1 in order to indicate that the corresponding subproblem can be fathomed. Otherwise, we continue with the initialization of the *LP*.

Returns:

0 If the linear program could be initialized successfully.
1 If the linear program turns out to be infeasible.

6.5.4.82 virtual void ABA_SUB::initializeVars (int *maxVar*) `[protected, virtual]`

Initializes the active variable set.

Parameters:

maxVar The maximal number of variables of the subproblem.

6.5.4.83 virtual int ABA_SUB::initMakeFeas (ABA_BUFFER< ABA_INFEASCON * > & *infeasCon*, ABA_BUFFER< ABA_VARIABLE * > & *newVars*, ABA_POOL< ABA_VARIABLE, ABA_CONSTRAINT > ** *pool*) `[protected, 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.

Returns:

0 If the feasibility might have been restored,
1 otherwise.

Parameters:

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.

6.5.4.84 bool ABA_SUB::integerFeasible () `[protected]`

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.

Returns:

true If the LP-value of all binary and integer variables is integral,
false otherwise.

6.5.4.85 void ABA_SUB::lBound (int *i*, double *l*) `[inline]`

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.

Parameters:

i The number of the variable.

x The new value of the lower bound.

Definition at line 2173 of file sub.h.

6.5.4.86 double ABA_SUB::lBound (int *i*) const `[inline]`

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.

Returns:
The lower bound of the *i-th* variable.

Parameters:
i The number of the variable.

Definition at line 2168 of file sub.h.

6.5.4.87 int ABA_SUB::level () const `[inline]`

Returns:
The level of the subproblem in the \ tree.

Definition at line 2212 of file sub.h.

6.5.4.88 double ABA_SUB::lowerBound () const

Returns:
A lower bound on the optimal solution of the subproblem.

6.5.4.89 ABA_LPSUB ∗ ABA_SUB::lp () const `[inline]`

Returns:
A pointer to the linear program of the subproblem.

Definition at line 2239 of file sub.h.

6.5.4.90 double ABA_SUB::lpRankBranchingRule (ABA_BRANCHRULE ∗ *branchRule*, int *iterLimit* = `-1`) `[protected]`

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.

Returns:
The value of he linear programming relaxation of the subproblem modified by the branching rule.

Parameters:
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*.

6.5.4.91 ABA_LPVARSTAT * ABA_SUB::lpVarStat (int *i*) const [inline]

Returns:

A pointer to the status of the variable *i* in the last solvedlinear program.

Parameters:

i The number of the variable.

Definition at line 2197 of file sub.h.

6.5.4.92 virtual int ABA_SUB::makeFeasible () [protected, 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.

Returns:

0 If feasibility can be restored,
1 otherwise.

6.5.4.93 ABA_MASTER * ABA_SUB::master () const [inline]

Definition at line 2118 of file sub.h.

6.5.4.94 int ABA_SUB::maxCon () const [inline]

Returns:

The maximum number of constraints which can be handled without reallocation.

Definition at line 2274 of file sub.h.

6.5.4.95 void ABA_SUB::maxIterations (int *max*)

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.

Parameters:

max The maximal number of iterations.

6.5.4.96 int ABA_SUB::maxVar () const `[inline]`

Returns:

The maximum number of variables which can be handled without reallocation.

Definition at line 2269 of file sub.h.

6.5.4.97 int ABA_SUB::nCon () const `[inline]`

Returns:

The number of active constraints.

Definition at line 2264 of file sub.h.

6.5.4.98 int ABA_SUB::nDormantRounds () const `[inline]`

Returns:

The number of subproblem optimization the subproblem is already dormant.

Definition at line 2158 of file sub.h.

6.5.4.99 void ABA_SUB::newDormantRound () `[inline, private, virtual]`

Increments the counter for the number of rounds the subproblem is dormant.

This function is called, when the set of open subproblems is scanned for the selection of the next subproblem.

Definition at line 2163 of file sub.h.

6.5.4.100 double ABA_SUB::nnzReserve () const `[inline]`

Returns:

The additional space for nonzero elements of the constraint matrix when it is passed to the LP-solver.

Definition at line 2128 of file sub.h.

6.5.4.101 virtual void ABA_SUB::nonBindingConEliminate (ABA_BUFFER< int > & *remove*) `[protected, virtual]`

Retrieves the dynamic constraints with slack exceeding the value given by the parameter { ConElimEps}.

Parameters:

remove Stores the nonbinding constraints.

6.5.4.102 int ABA_SUB::nVar () const `[inline]`

Returns:

The number of active variables.

Definition at line 2259 of file sub.h.

6.5.4.103 bool ABA_SUB::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.

Returns:
true If this condition is satisfied by the currently active variable set,
false otherwise.

6.5.4.104 const ABA_SUB& ABA_SUB::operator= (const ABA_SUB & *rhs*) `[private]`

6.5.4.105 virtual int ABA_SUB::optimize () `[protected, virtual]`

Performs the optimization of the subproblem.

After activating the subproblem, i.e., allocating and initializing memory, and initializing the *LP*, the optimization process constitutes of the three phases *Cutting*, *Branching*, and *Fathoming*, which are alternately processed. The function *fathoming()* always returns *Done*. However, we think that the code is better readable instead of taking it out of the *while* loop. The optimization stops if the *PHASE Done* is reached. Note, *Done* does not necessarily mean that the subproblem is solved to optimality!

After the node is processed we deallocate memory, which is not required for further computations or of which the corresponding data can be easily reconstructed. This is performed in *_deactivate()*.

Returns:
0 If the optimization has been performed without error,
1 otherwise.

6.5.4.106 virtual bool ABA_SUB::pausing () `[protected, 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.

Returns:
true The function *pausing()* should return *true* if this condition is satisfied,
false otherwise.

6.5.4.107 virtual int ABA_SUB::prepareBranching (bool & *lastIteration*) `[protected, virtual]`

Is called before a branching step to remove constraints.

Returns:
1 If constraints have been removed,
0 otherwise.

Parameters:
lastIteration This argument is always set to *true* in the function call.

6.5.4.108 virtual int ABA_SUB::pricing () `[protected, virtual]`

Should generate inactive variables which do not price out correctly.

The default implementation does nothing and returns 0.

Returns:
The number of new variables.

6.5.4.109 virtual bool ABA_SUB::primalSeparation () `[protected, virtual]`

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()*.

Returns:
true Then cutting planes are generated in this iteration.
false Then columns are generated in this iteration.

6.5.4.110 virtual double ABA_SUB::rankBranchingRule (ABA_BRANCHRULE $*$ *branchRule*) `[protected, virtual]`

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.

Returns:
The rank of the branching rule.

Parameters:
branchRule A pointer to a branching rule.

6.5.4.111 virtual void ABA_SUB::rankBranchingSample (ABA_BUFFER< ABA_BRANCHRULE $*$ > & *sample*, ABA_ARRAY< double > & *rank*) `[protected, virtual]`

Computes for each branching rule of a branching sample a rank with the function *rankBranchingRule()*.

Parameters:

sample A branching sample.

rank An array storing the rank for each branching rule in the sample after the function call.

6.5.4.112 void ABA_SUB::redCostVarEliminate (ABA_BUFFER< int > & *remove*) `[protected]`

Retrieves all variables with “wrong” reduced costs.

Parameters:

remove The variables with “wrong” reduced cost are stored in this buffer.

6.5.4.113 bool ABA_SUB::relativeReserve () const `[inline]`

Returns:

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.

Definition at line 2133 of file sub.h.

6.5.4.114 virtual void ABA_SUB::removeCon (int *i*) `[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.

Parameters:

i The number of the constraint being removed.

6.5.4.115 virtual void ABA_SUB::removeCons (ABA_BUFFER< int > & *remove*) `[virtual]`

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.

Parameters:

remove The constraints which should be removed.

6.5.4.116 virtual bool ABA_SUB::removeNonLiftableCons () `[protected, virtual]`

Returns:

true If all active constraints can be lifted.
false otherwise. In this case the non-liftable constraints are removed and *genNonLiftCons_* is set to *false* to avoid the generation of non-liftable constraints in the next cutting plane iterations.

6.5.4.117 void ABA_SUB::removeVar (int *i*) `[inline]`

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.

Parameters:

i The variable which should be removed.

Definition at line 2123 of file sub.h.

6.5.4.118 void ABA_SUB::removeVars (ABA_BUFFER< int > & *remove*)

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.

Parameters:

remove The variables which should be removed.

6.5.4.119 virtual void ABA_SUB::reoptimize () `[protected, virtual]`

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 \ tree.

As the subproblem has been processed already earlier it is sufficient to perform the cutting plane algorithm. If the subproblem is fathomed the complete subtree rooted at this subproblem can be fathomed, too. Since this function is usually only called for the root of the remaining \ tree, we are done in this case.

It is not guaranteed that all constraints and variables of this subproblem can be regenerated in *_activate()*. Therefore, the result of a call to *reoptimize()* can differ from the result obtained by the cutting plane algorithm in *optimize()*.

6.5.4.120 virtual int ABA_SUB::selectBestBranchingSample (int *nSamples*, ABA_BUFFER< ABA_BRANCHRULE ∗ > ∗∗ *samples*) `[protected, virtual]`

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()*.

Returns:

The number of the best branching sample, or *-1* in case of an internal error.

Parameters:

nSamples The number of branching samples.

samples An array of pointer to buffers storing the branching rules of each sample.

6.5.4.121 virtual int ABA_SUB::selectBranchingVariable (int & *variable*) `[protected, virtual]`

Chooses a branching variable.

The function *selectBranchingVariableCandidates()* is asked to generate depending in the parameter { NBranching-VariableCandidates} 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()*.

Returns:

0 If a branching variable is found,
1 otherwise.

Parameters:

variable Holds the branching variable if one is found.

6.5.4.122 virtual int ABA_SUB::selectBranchingVariableCandidates (ABA_BUFFER< int > & *candidates*) `[protected, virtual]`

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.

Returns:

0 If a candidate is found,
1 otherwise.

Parameters:

candidates The candidates for branching variables are stored in this buffer. We try to find as many variables as fit into the buffer.

6.5.4.123 virtual void ABA_SUB::selectCons () `[protected, virtual]`

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.

6.5.4.124 virtual void ABA_SUB::selectVars () `[protected, virtual]`

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.

6.5.4.125 virtual int ABA_SUB::separate () `[protected, virtual]`

Must be redefined in derived classes for the generation of cutting planes.

The default implementation does nothing.

Returns:

The number of generated cutting planes.

6.5.4.126 virtual int ABA_SUB::set (int *i*, ABA_FSVARSTAT::STATUS *newStat*, double *value*, bool & *newValue*) `[protected, virtual]`

Sets a variable.

Returns:

1 If a contradiction is found,
0 otherwise.

Parameters:

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*.

6.5.4.127 virtual int ABA_SUB::set (int *i*, ABA_FSVARSTAT::STATUS *newStat*, bool & *newValue*) `[protected, virtual]`

Sets a variable.

Returns:

1 If a contradiction is found,
0 otherwise.

Parameters:

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*.

6.5.4.128 virtual int ABA_SUB::set (int *i*, ABA_FSVARSTAT $*$ *newStat*, bool & *newValue*) `[protected, virtual]`

Sets a variable.

Returns:

1 If a contradiction is found,
0 otherwise.

Parameters:

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*.

6.5.4.129 virtual void ABA_SUB::setByLogImp (ABA_BUFFER< int > & *variable*, ABA_BUFFER< ABA_FSVARSTAT $*$ > & *status*) `[protected, virtual]`

The default implementation of *setByLogImp()* does nothing.

In derived classes this function can be reimplemented.

Parameters:

variable The variables which should be set have to be inserted in this buffer.

status The status of the set variables.

6.5.4.130 virtual int ABA_SUB::setByRedCost () `[protected, virtual]`

Tries to set variables according to the reduced cost criterion.

Returns:

1 If a contradiction is found,
0 otherwise.

6.5.4.131 virtual int ABA_SUB::setting (bool & *newValues*) `[protected, virtual]`

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.

Returns:

1 If a contradiction has been found,
0 otherwise.

Parameters:

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).

6.5.4.132 ABA_SLACKSTAT $*$ ABA_SUB::slackStat (int *i*) const `[inline]`

Returns:

A pointer to the status of the slack variable *i* in the last solved linear program.

Parameters:

i The number of the slack variable.

Definition at line 2202 of file sub.h.

6.5.4.133 virtual bool ABA_SUB::solveApproxNow () `[protected, virtual]`

Returns:

True, if the approximative solver should be used to solve the next linear program, false otherwise.

The default implementation always returns false.

6.5.4.134 virtual int ABA_SUB::solveLp () `[protected, virtual]`

Solves the LP-relaxation of the subproblem.

As soon as the *LP-relaxation* becomes infeasible in a static branch and cut algorithm the respective subproblem can be fathomed. Otherwise, we memorize the value of the LP-solution to control the tailing off effect.

{ We assume that the *LP* is never primal unbounded.

Returns:

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.

6.5.4.135 ABA_SUB::STATUS ABA_SUB::status () const `[inline]`

Returns:

The status of the subproblem optimization.

Definition at line 2244 of file sub.h.

6.5.4.136 virtual bool ABA_SUB::tailingOff () `[protected, virtual]`

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).

Returns:

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.

6.5.4.137 void ABA_SUB::uBound (int *i*, double *u*) `[inline]`

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.

Parameters:

i The number of the variable.

x The new value of the upper bound.

Definition at line 2185 of file sub.h.

6.5.4.138 double ABA_SUB::uBound (int *i*) const `[inline]`

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.

Returns:

The upper bound of the *i-th* variable.

Parameters:

i The number of the variable.

Definition at line 2180 of file sub.h.

6.5.4.139 virtual void ABA_SUB::updateBoundInLp (int *i*) `[private, virtual]`

Adapts the bound of a fixed or set variable *i* also in the linear program.

This can be only done if a linear program is available and the variable is not eliminated.

6.5.4.140 double ABA_SUB::upperBound () const

Returns:

An upper bound on the optimal solution of the subproblem.

6.5.4.141 virtual void ABA_SUB::varEliminate (ABA_BUFFER< int > & *remove*) `[protected, virtual]`

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()*.

Parameters:

remove The variables that should be removed have to be stored in this buffer.

6.5.4.142 ABA_VARIABLE* ABA_SUB::variable (int *i*) const

Returns:

A pointer to the *i-th* active variable.

Parameters:

i The number of the variable being accessed.

6.5.4.143 virtual int ABA_SUB::variablePoolSeparation (int *ranking* = `0`, ABA_POOL< ABA_VARIABLE, ABA_CONSTRAINT > * *pool* = `0`, double *minViolation* = `0.001`) `[protected, virtual]`

Tries to generate inactive variables from a pool.

Returns:

The number of generated variables.

Parameters:

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 3: rank determined by ABA_CONVAR::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.

6.5.4.144 virtual void ABA_SUB::varRealloc (int *newSize*) `[protected, virtual]`

Reallocates memory that at most *newSize* variables can be handled in the subproblem.

Parameters:

newSize The new maximal number of variables in the subproblem.

6.5.4.145 double ABA_SUB::xVal (int *i*) const `[inline]`

Parameters:

i The number of the variable under consideration.

Returns:

The value of the *i-th* variable in the last solved linear program.

Definition at line 2108 of file sub.h.

6.5.4.146 double ABA_SUB::yVal (int *i*) const `[inline]`

Parameters:

i The number of the variable under consideration.

Returns:

The value of the *i-th* dual variable in the last solved linear program.

Definition at line 2113 of file sub.h.

6.5.5 Friends And Related Function Documentation

6.5.5.1 friend class ABA_BOUNDBRANCHRULE `[friend]`

Definition at line 77 of file sub.h.

6.5.5.2 friend class ABA_LPSOLUTION< ABA_CONSTRAINT, ABA_VARIABLE > [friend]

Definition at line 79 of file sub.h.

6.5.5.3 friend class ABA_LPSOLUTION< ABA_VARIABLE, ABA_CONSTRAINT > [friend]

Definition at line 80 of file sub.h.

6.5.5.4 friend class ABA_MASTER [friend]

Definition at line 76 of file sub.h.

6.5.5.5 friend class ABA_OPENSUB [friend]

Definition at line 78 of file sub.h.

6.5.6 Member Data Documentation

6.5.6.1 ABA_ACTIVE<ABA_CONSTRAINT, ABA_VARIABLE>* ABA_SUB::actCon_ [protected]

The active constraints of the subproblem.

Definition at line 1660 of file sub.h.

6.5.6.2 bool ABA_SUB::activated_ [private]

The variable is *true* if the function *activate()* has been called from the function *_activate()*. This memorization is required such that a *deactivate()* is only called when *activate()* has been called.

Definition at line 2080 of file sub.h.

6.5.6.3 ABA_ACTIVE<ABA_VARIABLE, ABA_CONSTRAINT>* ABA_SUB::actVar_ [protected]

The active variables of the subproblem.

Definition at line 1664 of file sub.h.

6.5.6.4 ABA_CUTBUFFER<ABA_CONSTRAINT, ABA_VARIABLE>* ABA_SUB::addConBuffer_ [protected]

The buffer of the newly generated constraints.

Definition at line 1741 of file sub.h.

6.5.6.5 ABA_CUTBUFFER<ABA_VARIABLE, ABA_CONSTRAINT>* ABA_SUB::addVarBuffer_ [protected]

The buffer of the newly generated variables.

Definition at line 1737 of file sub.h.

6.5.6.6 bool ABA_SUB::allBranchOnSetVars_ [protected]

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.

Definition at line 1729 of file sub.h.

6.5.6.7 double* ABA_SUB::bInvRow_ [protected]

A row of the basis inverse associated with the infeasible variable *infeasVar_* or slack variable *infeasCon_*.

Definition at line 1764 of file sub.h.

6.5.6.8 ABA_BRANCHRULE* ABA_SUB::branchRule_ [protected]

The branching rule for the subproblem.

Definition at line 1723 of file sub.h.

6.5.6.9 double ABA_SUB::conReserve_ [private]

The additional space for constraints.

Definition at line 2064 of file sub.h.

6.5.6.10 double ABA_SUB::dualBound_ [protected]

The dual bound of the subproblem.

Definition at line 1707 of file sub.h.

6.5.6.11 ABA_SUB* ABA_SUB::father_ [protected]

A pointer to the father in the \ tree.

Definition at line 1668 of file sub.h.

6.5.6.12 bool ABA_SUB::forceExactSolver_ [private]

Indicates whether to force the use of an exact solver to prepare branching etc.

Definition at line 2096 of file sub.h.

6.5.6.13 ABA_ARRAY<ABA_FSVARSTAT*>* ABA_SUB::fsVarStat_ [protected]

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.

Definition at line 1681 of file sub.h.

6.5.6.14 bool ABA_SUB::genNonLiftCons_ [protected]

If *true*, then the management of non-liftable constraints is performed.

Definition at line 1776 of file sub.h.

6.5.6.15 int ABA_SUB::id_ [private]

The number of the subproblem.

Definition at line 2028 of file sub.h.

6.5.6.16 bool ABA_SUB::ignoreInTailingOff_ [private]

If this flag is set to *true* then the next LP-solution is ignored in the tailing-off control. The default value of the variable is *false*. It can be set to *true* by the function *ignoreInTailingOff()*.

Definition at line 2086 of file sub.h.

6.5.6.17 int ABA_SUB::infeasCon_ [protected]

The number of an infeasible constraint.

Definition at line 1768 of file sub.h.

6.5.6.18 int ABA_SUB::infeasVar_ [protected]

The number of an infeasible variable.

Definition at line 1772 of file sub.h.

6.5.6.19 int ABA_SUB::lastIterConAdd_ [protected]

The last iteration in which constraints have been added.

Definition at line 1715 of file sub.h.

6.5.6.20 int ABA_SUB::lastIterVarAdd_ [protected]

The last iteration in which variables have been added.

Definition at line 1719 of file sub.h.

6.5.6.21 ABA_LP::METHOD ABA_SUB::lastLP_ `[private]`

The method that was used to solve the last LP.

Definition at line 2089 of file sub.h.

6.5.6.22 ABA_ARRAY<double>∗ ABA_SUB::lBound_ `[protected]`

A pointer to an array with the local lower bound of the active variables.

Definition at line 1690 of file sub.h.

6.5.6.23 int ABA_SUB::level_ `[private]`

The level of the subproblem in the enumeration tree.

Definition at line 2024 of file sub.h.

6.5.6.24 ABA_CPUTIMER ABA_SUB::localTimer_ `[private]`

Definition at line 2091 of file sub.h.

6.5.6.25 ABA_LPSUB∗ ABA_SUB::lp_ `[protected]`

A pointer to the corresponding linear program.

Definition at line 1672 of file sub.h.

6.5.6.26 ABA_LP::METHOD ABA_SUB::lpMethod_ `[protected]`

The solution method for the next linear program.

Definition at line 1733 of file sub.h.

6.5.6.27 ABA_ARRAY<ABA_LPVARSTAT∗>∗ ABA_SUB::lpVarStat_ `[protected]`

A pointer to an array storing the status of each active variable in the linear program.

Definition at line 1686 of file sub.h.

6.5.6.28 ABA_MASTER∗ ABA_SUB::master_ `[protected]`

A pointer to the corresponding master of the optimization.

Definition at line 1656 of file sub.h.

6.5.6.29 int ABA_SUB::maxIterations_ `[private]`

The maximum number of iterations in the cutting plane phase.

Definition at line 2043 of file sub.h.

6.5.6.30 int ABA_SUB::nDormantRounds_ [private]

The number of subproblem optimizations the subproblem has already the status *Dormant*.

Definition at line 2073 of file sub.h.

6.5.6.31 int ABA_SUB::nIter_ [protected]

The number of iterations in the cutting plane phase.

Definition at line 1711 of file sub.h.

6.5.6.32 double ABA_SUB::nnzReserve_ [private]

The additional space for nonzeros.

Definition at line 2068 of file sub.h.

6.5.6.33 int ABA_SUB::nOpt_ [private]

The number of optimizations of the subproblem.

Definition at line 2047 of file sub.h.

6.5.6.34 bool ABA_SUB::relativeReserve_ [private]

If this member is *true* then the space reserve of the following three members *varReserve_*, *conReserve_*, and *nnz-Reserve_* is relative to the initial numbers of constraints, variables, and nonzeros, respectively. Otherwise, the values are casted to integers and regarded as absolute values.

Definition at line 2056 of file sub.h.

6.5.6.35 ABA_BUFFER<int>∗ ABA_SUB::removeConBuffer_ [protected]

The buffer of the constraints which are removed at the beginning of the next iteration.

Definition at line 1751 of file sub.h.

6.5.6.36 ABA_BUFFER<int>∗ ABA_SUB::removeVarBuffer_ [protected]

The buffer of the variables which are removed at the beginning of the next iteration.

Definition at line 1746 of file sub.h.

6.5.6.37 ABA_ARRAY<ABA_SLACKSTAT∗>∗ ABA_SUB::slackStat_ [protected]

A pointer to an array storing the statuses of the slack variables of the last solved linear program.

Definition at line 1699 of file sub.h.

6.5.6.38 ABA_BUFFER<ABA_SUB*>* ABA_SUB::sons_ [private]

The sons of the node in the \ tree.

Definition at line 2039 of file sub.h.

6.5.6.39 STATUS ABA_SUB::status_ [private]

The status of the subproblem.

Definition at line 2035 of file sub.h.

6.5.6.40 ABA_TAILOFF* ABA_SUB::tailOff_ [protected]

A pointer to the tailing off manager.

Definition at line 1703 of file sub.h.

6.5.6.41 ABA_ARRAY<double>* ABA_SUB::uBound_ [protected]

A pointer to an array with the local upper bounds of the active variables.

Definition at line 1694 of file sub.h.

6.5.6.42 double ABA_SUB::varReserve_ [private]

The additional space for variables.

Definition at line 2060 of file sub.h.

6.5.6.43 double* ABA_SUB::xVal_ [protected]

The last LP-solution.

Definition at line 1755 of file sub.h.

6.5.6.44 double* ABA_SUB::yVal_ [protected]

The dual variables of the last linear program.

Definition at line 1759 of file sub.h.

The documentation for this class was generated from the following file:

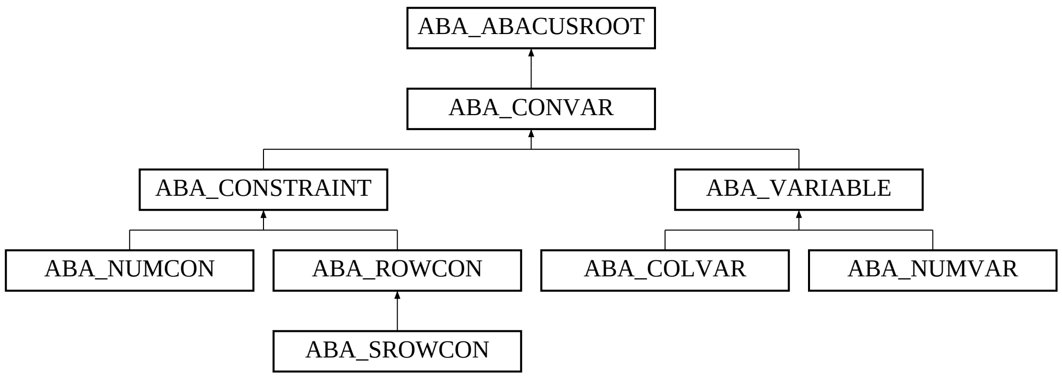
- Include/abacus/sub.h

6.6 ABA_CONVAR Class Reference

ABA_CONVAR is the common base class for constraints and variables, which are implemented in the derived classes ABA_CONSTRAINT and ABA_VARIABLE, respectively.

`#include <convar.h>`

Inheritance diagram for ABA_CONVAR::



Public Member Functions

- ABA_CONVAR (ABA_MASTER ∗master, const ABA_SUB ∗sub, bool dynamic, bool local)
- virtual ∼ABA_CONVAR ()
- bool active () const
- bool local () const
- bool global () const
- virtual bool dynamic () const
- const ABA_SUB ∗ sub () const
- void sub (ABA_SUB ∗sub)

  This version of the function sub() associates a new subproblem with the constraint/variable.

- virtual unsigned hashKey ()

  Should provide a key for the constraint/variable that can be used to insert it into a hash table.

- virtual const char ∗ name ()
- virtual bool equal (ABA_CONVAR ∗cv)

  Should compare if the constraint/variable is identical (in a mathematical sense) with the constraint/variable cv.

- virtual double rank ()

  The function should return a rank associated with the constraint/variable. The default implementation returns 0.

Constraints/Variables often have to be stored in a format different from the format used in the linear program. One reason is to save memory and the other reason is that if constraints and/or variable sets are dynamic, then we require a format to compute the coefficients of later activated variables/constraints.

The disadvantage of such a constraint format is that the computation of a single constraint coefficient can be very time consuming. Often it cannot be done in constant time. Hence we provide a mechanism which converts a constraint/variable to a format enabling efficient computation of coefficients. The following functions provide this feature.

- bool expanded () const
- void _expand ()
- void _compress ()
- virtual void print (ostream &out)

Protected Attributes

- ABA_MASTER ∗ master_
- const ABA_SUB ∗ sub_

  A pointer to the subproblem associated with the constraint/variable. This may be also the 0-pointer.

- bool expanded_
- int nReferences_

  The number of references to the pool slot the constraint is stored ABA_POOLSLOTREF.

- 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.

- int nActive_

  The number of active subproblems of which the constraint/variable belongs to the set of active constraints/variables.

- int nLocks_
- bool local_

  This flag is true *if the constraint/variable is only locally valid, otherwise it is false.*

Private Member Functions

- void activate ()

  Must be called if the constraint/variable is added to the active formulation of an active subproblem.

- void deactivate ()

  Is the counterpart to activate() *and is also called within members of the class* ABA_SUB *to indicate that the constraint/variable does not belong any more to the active formulation of an active subproblem.*

- int nReferences () const

  Returns the number of references to the pool slot ABA_POOLSLOTREF storing this constraint/variable.

- void addReference ()

  Indicates that there is a new reference to the pool slot storing this constraint/variable.

- void removeReference ()

  Is the counterpart of the function addReference() *and indicates the removal of a reference to this constraint.*

- virtual bool deletable () const
- virtual void expand ()
- virtual void compress ()

If a constraint/variable has just been separated and added to the buffer of currently separated constraints/variables, then this item should not be removed before the buffer is emptied at the beginning of the next iteration. Hence, we provide a locking mechanism for constraints/variables by the following three functions.

- bool locked () const

- void lock ()
  Adds an additional lock to the constraint/variable.
- void unlock ()

Friends

- class ABA_POOLSLOT< ABA_CONSTRAINT, ABA_VARIABLE >
- class ABA_POOLSLOT< ABA_VARIABLE, ABA_CONSTRAINT >
- class ABA_POOLSLOTREF< ABA_CONSTRAINT, ABA_VARIABLE >
- class ABA_POOLSLOTREF< ABA_VARIABLE, ABA_CONSTRAINT >
- class ABA_STANDARDPOOL< ABA_CONSTRAINT, ABA_VARIABLE >
- class ABA_STANDARDPOOL< ABA_VARIABLE, ABA_CONSTRAINT >
- class ABA_CUTBUFFER< ABA_CONSTRAINT, ABA_VARIABLE >
- class ABA_CUTBUFFER< ABA_VARIABLE, ABA_CONSTRAINT >
- class ABA_SUB

6.6.1 Detailed Description

ABA_CONVAR is the common base class for constraints and variables, which are implemented in the derived classes ABA_CONSTRAINT and ABA_VARIABLE, respectively.

Definition at line 79 of file convar.h.

6.6.2 Constructor & Destructor Documentation

6.6.2.1 ABA_CONVAR::ABA_CONVAR (ABA_MASTER ∗ *master*, const ABA_SUB ∗ *sub*, bool *dynamic*, bool *local*)

The constructor.

Parameters:

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 paramument 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.

6.6.2.2 virtual ABA_CONVAR::∼ABA_CONVAR () `[virtual]`

The destructor.

6.6.3 Member Function Documentation

6.6.3.1 void ABA_CONVAR::_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.

6.6.3.2 void ABA_CONVAR::_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.

6.6.3.3 void ABA_CONVAR::activate () `[inline, private]`

Must be called if the constraint/variable is added to the active formulation of an active subproblem.

This function is only called within member functions of the class ABA_SUB.

Definition at line 473 of file convar.h.

6.6.3.4 bool ABA_CONVAR::active () const `[inline]`

Checks if the constraint/variable is active in at least one active subproblem.

In the parallel implementation this can be more than one subproblem when multithreading occurs. Only those subproblems are taken into account which are related to the pool in which the constraint/variable is stored.

Returns:

true If the constraint/variable is active,
false otherwise.

Definition at line 467 of file convar.h.

6.6.3.5 void ABA_CONVAR::addReference () `[inline, private]`

Indicates that there is a new reference to the pool slot storing this constraint/variable.

The function is only called from members of the class ABA_POOLSLOTREF.

Definition at line 483 of file convar.h.

6.6.3.6 virtual void ABA_CONVAR::compress () `[private, virtual]`

Also the default implementation of the function *compress()* is void. It should be redefined in derived classes.

6.6.3.7 void ABA_CONVAR::deactivate () `[private]`

Is the counterpart to *activate()* and is also called within members of the class ABA_SUB to indicate that the constraint/variable does not belong any more to the active formulation of an active subproblem.

6.6.3.8 virtual bool ABA_CONVAR::deletable () const `[private, virtual]`

Returns *true* if the constraint/variable can be destructed.

This is per default only possible if the reference counter is 0 and no lock is set. The function is declared virtual such that problem specific implementations are possible.

6.6.3.9 virtual bool ABA_CONVAR::dynamic () const `[virtual]`

Returns:

true If the constraint/variable can be also removed from the set of active constraints/var\-i\-a\-bles after it has been activated,
false otherwise.

6.6.3.10 virtual bool ABA_CONVAR::equal (ABA_CONVAR ∗ *cv*) `[virtual]`

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 \ it is not required.

The redundant return statement is required to suppress compiler warnings.

Returns:

true If the constraint/variable represented by this object represents the same item as the constraint/variable *cv*,
false otherwise.

Parameters:

cv The constraint/variable that should be compared with this object.

6.6.3.11 virtual void ABA_CONVAR::expand () `[private, virtual]`

The default implementation of the function *expand()* is void. It should be redefined in derived classes.

6.6.3.12 bool ABA_CONVAR::expanded () const `[inline]`

Returns:

true If the expanded format of a constraint/variable is available,
false otherwise.

Definition at line 524 of file convar.h.

6.6.3.13 bool ABA_CONVAR::global () const `[inline]`

Returns:

true If the constraint/variable is globally valid,
false otherwise.

Definition at line 504 of file convar.h.

6.6.3.14 virtual unsigned ABA_CONVAR::hashKey () `[virtual]`

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 \ it is not required.

We do not use *double* as result type because typical problems in floating point arithmetic might give slightly different hash keys for two constraints that are equal from a mathematical point of view.

The redundant return statement is required to suppress compiler warnings.

Returns:

An integer providing a hash key for the constraint/variable.

6.6.3.15 bool ABA_CONVAR::local () const `[inline]`

Returns:

true If the constraint/variable is only locally valid,
false otherwise.

Definition at line 499 of file convar.h.

6.6.3.16 void ABA_CONVAR::lock () `[inline, private]`

Adds an additional lock to the constraint/variable.

Definition at line 494 of file convar.h.

6.6.3.17 bool ABA_CONVAR::locked () const `[inline, private]`

Returns:

true if at least one lock is set on the constraint/variable
false otherwise.

Definition at line 488 of file convar.h.

6.6.3.18 virtual const char$*$ ABA_CONVAR::name () `[virtual]`

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 /. 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 $\backslash$ it is not required.

The redundant return statement is required to suppress compiler warnings.

Returns:

The name of the constraint/variable.

6.6.3.19 int ABA_CONVAR::nReferences () const `[inline, private]`

Returns the number of references to the pool slot ABA_POOLSLOTREF storing this constraint/variable.

We require the bookkeeping of the references in order to determine if a constraint/variable can be deleted without causing any harm.

Definition at line 478 of file convar.h.

6.6.3.20 virtual void ABA_CONVAR::print (ostream & *out*) `[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.

Parameters:
out The output stream.

Reimplemented in ABA_COLVAR, ABA_NUMCON, and ABA_ROWCON.

6.6.3.21 virtual double ABA_CONVAR::rank () `[virtual]`

The function should return a rank associated with the constraint/variable. The default implementation returns 0.

Returns:
The rank of the constraint/variable.

6.6.3.22 void ABA_CONVAR::removeReference () `[private]`

Is the counterpart of the function *addReference()* and indicates the removal of a reference to this constraint.

It is only called from members of the class ABA_POOLSLOTREF.

6.6.3.23 void ABA_CONVAR::sub (ABA_SUB ∗ *sub*) `[inline]`

This version of the function *sub()* associates a new subproblem with the constraint/variable.

Parameters:
sub The new subproblem associated with the constraint/variable.

Definition at line 519 of file convar.h.

6.6.3.24 const ABA_SUB ∗ ABA_CONVAR::sub () const `[inline]`

Returns:
A pointer to the subproblem associated with the constraint/variable. Note, this can also be the 0-pointer.

Definition at line 514 of file convar.h.

6.6.3.25 void ABA_CONVAR::unlock () `[private]`

Removes one lock from the constraint/variable.

6.6.4 Friends And Related Function Documentation

6.6.4.1 friend class ABA_CUTBUFFER< ABA_CONSTRAINT, ABA_VARIABLE > `[friend]`

Definition at line 86 of file convar.h.

6.6.4.2 friend class ABA_CUTBUFFER< ABA_VARIABLE, ABA_CONSTRAINT > `[friend]`

Definition at line 87 of file convar.h.

6.6.4.3 friend class ABA_POOLSLOT< ABA_CONSTRAINT, ABA_VARIABLE > `[friend]`

Definition at line 80 of file convar.h.

6.6.4.4 friend class ABA_POOLSLOT< ABA_VARIABLE, ABA_CONSTRAINT > `[friend]`

Definition at line 81 of file convar.h.

6.6.4.5 friend class ABA_POOLSLOTREF< ABA_CONSTRAINT, ABA_VARIABLE > `[friend]`

Definition at line 82 of file convar.h.

6.6.4.6 friend class ABA_POOLSLOTREF< ABA_VARIABLE, ABA_CONSTRAINT > `[friend]`

Definition at line 83 of file convar.h.

6.6.4.7 friend class ABA_STANDARDPOOL< ABA_CONSTRAINT, ABA_VARIABLE > `[friend]`

Definition at line 84 of file convar.h.

6.6.4.8 friend class ABA_STANDARDPOOL< ABA_VARIABLE, ABA_CONSTRAINT > `[friend]`

Definition at line 85 of file convar.h.

6.6.4.9 friend class ABA_SUB `[friend]`

Definition at line 88 of file convar.h.

6.6.5 Member Data Documentation

6.6.5.1 bool ABA_CONVAR::dynamic_ `[protected]`

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*.

Definition at line 356 of file convar.h.

6.6.5.2 bool ABA_CONVAR::expanded_ [protected]

true, if expanded version of constraint/variables available.

Definition at line 342 of file convar.h.

6.6.5.3 bool ABA_CONVAR::local_ [protected]

This flag is *true* if the constraint/variable is only locally valid, otherwise it is false.

Definition at line 377 of file convar.h.

6.6.5.4 ABA_MASTER∗ ABA_CONVAR::master_ [protected]

A pointer to the corresponding master of the optimization.

Definition at line 333 of file convar.h.

6.6.5.5 int ABA_CONVAR::nActive_ [protected]

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.

Definition at line 368 of file convar.h.

6.6.5.6 int ABA_CONVAR::nLocks_ [protected]

The number of locks which have been set on the constraint/variable.

Definition at line 372 of file convar.h.

6.6.5.7 int ABA_CONVAR::nReferences_ [protected]

The number of references to the pool slot the constraint is stored ABA_POOLSLOTREF.

Definition at line 347 of file convar.h.

6.6.5.8 const ABA_SUB∗ ABA_CONVAR::sub_ [protected]

A pointer to the subproblem associated with the constraint/variable. This may be also the 0-pointer.

Definition at line 338 of file convar.h.

The documentation for this class was generated from the following file:

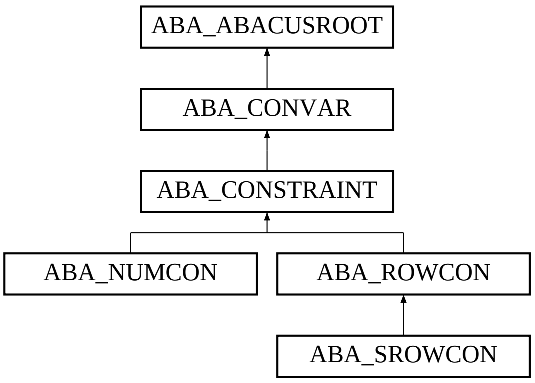
- Include/abacus/convar.h

6.7 ABA_CONSTRAINT Class Reference

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.

`#include <constraint.h>`

Inheritance diagram for ABA_CONSTRAINT::



Public Member Functions

- ABA_CONSTRAINT (ABA_MASTER ∗master, const 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)

  Computes the slack of the vector x *associated with the variable set* variables.

- virtual bool violated (ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > ∗variables, double ∗x, double ∗sl=0)
- virtual bool violated (double slack) const

  This version of function violated() checks for the violation given the slack of a vector.

- void printRow (ostream &out, ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > ∗var)

  Writes the row format of the constraint associated with the variable set var *on an output stream.*

- virtual double distance (double ∗x, ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > ∗actVar)
- ABA_CONSTRAINT ∗ duplicate ()
- ABA_CONCLASS ∗ classification (ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > ∗var=0)

Protected Member Functions

- virtual ABA_INFEASCON::INFEAS voidLhsViolated (double newRhs) const

  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.

- virtual ABA_CONCLASS $*$ classify (ABA_ACTIVE$<$ ABA_VARIABLE, ABA_CONSTRAINT $>$ $*$var)

  The default implementation of the function classify() *returns a 0 pointer.*

Protected Attributes

- ABA_CSENSE sense_
- double rhs_
- ABA_CONCLASS $*$ conClass_
- bool liftable_

  This member is true *if also coefficients of variables which have been inactive at generation time can be computed,* false *otherwise.*

Private Member Functions

- const ABA_CONSTRAINT & operator= (const ABA_CONSTRAINT &rhs)

Friends

- class ABA_LPSUB

6.7.1 Detailed Description

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.

Definition at line 55 of file constraint.h.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 ABA_CONSTRAINT::ABA_CONSTRAINT (ABA_MASTER $*$ *master*, const ABA_SUB $*$ *sub*, ABA_CSENSE::SENSE *sense*, double *rhs*, bool *dynamic*, bool *local*, bool *liftable*)

The constructor.

Parameters:

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 paramument 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 paramument is *true*, then the constraint is considered to be only locally valid. In this case the paramument sub must not be 0 as each locally valid constraint is associated with a subproblem. }

liftable If this paramument 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.

6.7.2.2 ABA_CONSTRAINT::ABA_CONSTRAINT (ABA_MASTER $*$ *master*)

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.

Parameters:

master A pointer to the corresponding master of the optimization.

6.7.2.3 ABA_CONSTRAINT::ABA_CONSTRAINT (const ABA_CONSTRAINT & *rhs*)

The copy constructor.

Parameters:

rhs The constraint being copied.

6.7.2.4 virtual ABA_CONSTRAINT::~ABA_CONSTRAINT () `[virtual]`

The destructor.

6.7.3 Member Function Documentation

6.7.3.1 ABA_CONCLASS$*$ ABA_CONSTRAINT::classification (ABA_ACTIVE$<$ ABA_VARIABLE, ABA_CONSTRAINT $>$ $*$ *var* = `0`)

Returns a pointer to the classification of the constraint.

If no classification is available then we try to classify the constraint. In this case *var* must not be a 0-pointer. Per default *var* is 0.

A constraint classification can only be generated if the function *classify()* is redefined in a derived class.

6.7.3.2 virtual ABA_CONCLASS$*$ ABA_CONSTRAINT::classify (ABA_ACTIVE$<$ ABA_VARIABLE, ABA_CONSTRAINT $>$ $*$ *var*) `[protected, virtual]`

The default implementation of the function *classify()* returns a 0 pointer.

6.7.3.3 virtual double ABA_CONSTRAINT::coeff (ABA_VARIABLE $*$ *v*) `[pure virtual]`

Parameters:

v A pointer to a variable.

Returns:

The coefficient of the variable *v* in the constraint.

Implemented in ABA_NUMCON, and ABA_ROWCON.

6.7.3.4 virtual double ABA_CONSTRAINT::distance (double $*$ *x*, ABA_ACTIVE$<$ ABA_VARIABLE, ABA_CONSTRAINT $>$ $*$ *actVar*) `[virtual]`

The distance of a point $\overline{x}$ and a hyperplane $a^T x = \beta$ can be computed in the following way: Let y be the intersection of the hyperplane $a^T x = \beta$ and the line defined by $\overline{x}$ and the vector a . Then the distance d is the length of the vector $||\overline{x} - y||$.

Returns:

The Euclidean distance of the vector *x* associated with the variable set *actVar* to the hyperplane induced by the constraint.

Parameters:

x The point for which the distance should be computed.

actVar The variables associated with *x*.

6.7.3.5 ABA_CONSTRAINT$*$ ABA_CONSTRAINT::duplicate () `[inline]`

Definition at line 245 of file constraint.h.

6.7.3.6 virtual int ABA_CONSTRAINT::genRow (ABA_ACTIVE$<$ ABA_VARIABLE, ABA_CONSTRAINT $>$ $*$ *var*, ABA_ROW & *row*) `[virtual]`

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.

All nonzero coefficients are added to the row format. Before we generate the coefficients we try to expand the constraint, afterwards it is compressed again.

Returns:

The number of nonzero elements in the row format *row*.

Parameters:

var The variable set for which the row format should be computed.

row Stores the row format after calling this function.

Reimplemented in ABA_SROWCON.

6.7.3.7 bool ABA_CONSTRAINT::liftable () const `[inline]`

Checks if the constraint is liftable,

i.e., if the coefficients of variables inactive at generation time of the constraint can be computed later.

Returns:

true If the constraint can be lifted,
false otherwise.

Definition at line 308 of file constraint.h.

6.7.3.8 const ABA_CONSTRAINT& ABA_CONSTRAINT::operator= (const ABA_CONSTRAINT & *rhs*) `[private]`

6.7.3.9 void ABA_CONSTRAINT::printRow (ostream & *out*, ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > ∗ *var*)

Writes the row format of the constraint associated with the variable set *var* on an output stream.

Parameters:

out The output stream.

var The variables for which the row format should be written.

6.7.3.10 virtual double ABA_CONSTRAINT::rhs () `[virtual]`

Returns:

The right hand side of the constraint.

6.7.3.11 ABA_CSENSE ∗ ABA_CONSTRAINT::sense () `[inline]`

Returns:

A pointer to the sense of the constraint.

Definition at line 303 of file constraint.h.

6.7.3.12 virtual double ABA_CONSTRAINT::slack (ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > ∗ *variables*, double ∗ *x*) `[virtual]`

Computes the slack of the vector *x* associated with the variable set *variables*.

Returns:
The slack induced by the vector *x*.

Parameters:
variables The variable set associated with the vector *x*.
x The values of the variables.

Reimplemented in ABA_SROWCON.

6.7.3.13 virtual bool ABA_CONSTRAINT::valid (ABA_SUB ∗ *sub*) `[virtual]`

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.

Returns:
true If the constraint is valid for the subproblem sub,
false otherwise.

Parameters:
sub The subproblem for which the validity is checked.

6.7.3.14 virtual bool ABA_CONSTRAINT::violated (double *slack*) const `[virtual]`

This version of function *violated()* checks for the violation given the slack of a vector.

Returns:
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.

Parameters:
slack The slack of a vector.

6.7.3.15 virtual bool ABA_CONSTRAINT::violated (ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > ∗ *variables*, double ∗ *x*, double ∗ *sl* = `0`) `[virtual]`

Checks if a constraint is violated by a vector *x* associated with a variable set.

Returns:
true If the constraint is violated,
false otherwise.

Parameters:

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.

6.7.3.16 virtual ABA_INFEASCON::INFEAS ABA_CONSTRAINT::voidLhsViolated (double *newRhs*) const `[protected, virtual]`

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.

Returns:

{TooLparame or TooSmall} If the value *newRhs* violates the sense of the constraint, i.e, it is $<$ / $>$ / != 0 and the sense of the constraint is $>=$ / $<=$ / =,
Feasible otherwise.

Parameters:

newRhs The right hand side of the constraint after the elimination of the variables.

6.7.4 Friends And Related Function Documentation

6.7.4.1 friend class ABA_LPSUB `[friend]`

Definition at line 56 of file constraint.h.

6.7.5 Member Data Documentation

6.7.5.1 ABA_CONCLASS* ABA_CONSTRAINT::conClass_ `[protected]`

Definition at line 290 of file constraint.h.

6.7.5.2 bool ABA_CONSTRAINT::liftable_ `[protected]`

This member is *true* if also coefficients of variables which have been inactive at generation time can be computed, *false* otherwise.

Definition at line 296 of file constraint.h.

6.7.5.3 double ABA_CONSTRAINT::rhs_ `[protected]`

The right hand side of the constraint.

Definition at line 289 of file constraint.h.

6.7.5.4 ABA_CSENSE ABA_CONSTRAINT::sense_ `[protected]`

The sense of the constraint.

Definition at line 285 of file constraint.h.

The documentation for this class was generated from the following file:

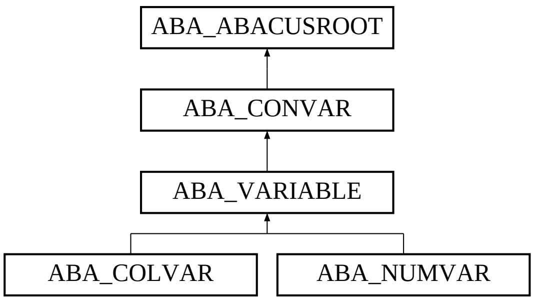
- Include/abacus/constraint.h

6.8 ABA_VARIABLE Class Reference

class forms the virtual base class for all possible variables given in pool format

`#include <variable.h>`

Inheritance diagram for ABA_VARIABLE::



Public Member Functions

- ABA_VARIABLE (ABA_MASTER ∗master, const ABA_SUB ∗sub, bool dynamic, bool local, double obj, double lBound, double uBound, ABA_VARTYPE::TYPE type)
- virtual ~ABA_VARIABLE ()

  The destructor.

- ABA_VARTYPE::TYPE varType () const
- bool discrete ()
- bool binary ()
- bool integer ()
- virtual double obj ()
- double uBound () const
- void uBound (double newValue)

  This version of the function uBound() sets the upper bound of the variable.

- double lBound () const
- void lBound (double newValue)

  This version of the function lBound() sets the lower bound of the variable.

- ABA_FSVARSTAT $*$ fsVarStat ()
- virtual bool valid (ABA_SUB $*$sub)
- virtual int genColumn (ABA_ACTIVE< ABA_CONSTRAINT, ABA_VARIABLE > $*$actCon, ABA_COLUMN &col)

  Computes the column col *of the variable associated with the active constraints* $*$actCon*.*

- virtual double coeff (ABA_CONSTRAINT $*$con)
- virtual bool violated (double rc) const

  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.*

- virtual bool violated (ABA_ACTIVE< ABA_CONSTRAINT, ABA_VARIABLE > $*$constraints, double $*$y, double $*$slack=0)

  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.*

- virtual double redCost (ABA_ACTIVE< ABA_CONSTRAINT, ABA_VARIABLE > $*$actCon, double $*$y)

  Computes the reduced cost of the variable corresponding the constraint set actCon *and the dual variables* y*.*

- virtual bool useful (ABA_ACTIVE< ABA_CONSTRAINT, ABA_VARIABLE > $*$actCon, double $*$y, double lpVal)

  An (inactive) discrete variable is considered as useful() *if its activation might not produce only solutions worse than the best known feasible solution.*

- void printCol (ostream &out, ABA_ACTIVE< ABA_CONSTRAINT, ABA_VARIABLE > $*$constraints)

  Writes the column of the variable corresponding to the constraints *on the stream* out*.*

Protected Attributes

- ABA_FSVARSTAT fsVarStat_
- double obj_
- double lBound_
- double uBound_
- ABA_VARTYPE type_

6.8.1 Detailed Description

class forms the virtual base class for all possible variables given in pool format

Definition at line 55 of file variable.h.

6.8.2 Constructor & Destructor Documentation

6.8.2.1 ABA_VARIABLE::ABA_VARIABLE (ABA_MASTER ∗ *master*, const ABA_SUB ∗ *sub*, bool *dynamic*, bool *local*, double *obj*, double *lBound*, double *uBound*, ABA_VARTYPE::TYPE *type*)

The constructor.

Parameters:

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.

6.8.2.2 virtual ABA_VARIABLE::∼ABA_VARIABLE () `[virtual]`

The destructor.

6.8.3 Member Function Documentation

6.8.3.1 bool ABA_VARIABLE::binary () `[inline]`

Returns:

true If the type of the variable is *Binary*,
false otherwise.

Definition at line 312 of file variable.h.

6.8.3.2 virtual double ABA_VARIABLE::coeff (ABA_CONSTRAINT ∗ *con*) `[virtual]`

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.

Returns:

The coefficient of the variable in the constraint *con*.

Parameters:

con The constraint of which the coefficient should be computed.

Reimplemented in ABA_COLVAR.

6.8.3.3 bool ABA_VARIABLE::discrete () `[inline]`

Returns:

true If the type of the variable is *Integer* or *Binary*,
false otherwise.

Definition at line 307 of file variable.h.

6.8.3.4 ABA_FSVARSTAT $*$ ABA_VARIABLE::fsVarStat () `[inline]`

Returns:

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().

Definition at line 342 of file variable.h.

6.8.3.5 virtual int ABA_VARIABLE::genColumn (ABA_ACTIVE$<$ ABA_CONSTRAINT, ABA_VARIABLE $>$ $*$ *actCon*, ABA_COLUMN & *col*) `[virtual]`

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.

Returns:

The number of nonzero entries in *col*.

Parameters:

actCon The constraints for which the column of the variable should be computed.

col Stores the column when the function terminates.

6.8.3.6 bool ABA_VARIABLE::integer () `[inline]`

Returns:

true If the type of the variable is *Integer*,
false otherwise.

Definition at line 317 of file variable.h.

6.8.3.7 void ABA_VARIABLE::lBound (double *newValue*) `[inline]`

This version of the function *lBound()* sets the lower bound of the variable.

Parameters:

newBound The new value of the lower bound.

Definition at line 327 of file variable.h.

6.8.3.8 double ABA_VARIABLE::lBound () const `[inline]`

Returns:
The lower bound of the variable.

Definition at line 322 of file variable.h.

6.8.3.9 virtual double ABA_VARIABLE::obj () `[virtual]`

Returns:
The objective function coefficient.

6.8.3.10 void ABA_VARIABLE::printCol (ostream & *out*, ABA_ACTIVE< ABA_CONSTRAINT, ABA_VARIABLE > ∗ *constraints*)

Writes the column of the variable corresponding to the *constraints* on the stream *out*.

Parameters:
out The output stream.
constraints The constraints for which the column should be written.

6.8.3.11 virtual double ABA_VARIABLE::redCost (ABA_ACTIVE< ABA_CONSTRAINT, ABA_VARIABLE > ∗ *actCon*, double ∗ *y*) `[virtual]`

Computes the reduced cost of the variable corresponding the constraint set *actCon* and the dual variables *y*.

Given the dual variables y , then the reduced cost of a variable with objective function coefficient c_e , column $a_{.e}$ are defined as $c_e - y^{\mathrm{T}} a_{.e}$.

Returns:
The reduced cost of the variable.

Parameters:
actCon The constraints associated with the dual variables *y*.
y The dual variables of the constraint.

6.8.3.12 void ABA_VARIABLE::uBound (double *newValue*) `[inline]`

This version of the function *uBound()* sets the upper bound of the variable.

Parameters:
newBound The new value of the upper bound.

Definition at line 337 of file variable.h.

6.8.3.13 double ABA_VARIABLE::uBound () const `[inline]`

Returns:
The upper bound of the variable.

Definition at line 332 of file variable.h.

6.8.3.14 virtual bool ABA_VARIABLE::useful (ABA_ACTIVE< ABA_CONSTRAINT, ABA_VARIABLE > * *actCon*, double * *y*, double *lpVal*) `[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.

Returns:
true If the variable is considered as useful,
false otherwise.

Parameters:
actCon The active constraints.
y The dual variables of these constraints.
lpVal The value of the linear program.

6.8.3.15 virtual bool ABA_VARIABLE::valid (ABA_SUB * *sub*) `[virtual]`

Returns:
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.

Parameters:
sub The subproblem for which validity of the variable is checked.

6.8.3.16 ABA_VARTYPE::TYPE ABA_VARIABLE::varType () const `[inline]`

Returns:
The type of the variable.

Definition at line 302 of file variable.h.

6.8.3.17 virtual bool ABA_VARIABLE::violated (ABA_ACTIVE< ABA_CONSTRAINT, ABA_VARIABLE > * *constraints*, double * *y*, double * *slack* = `0`) `[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.

Returns:

true If the variable does not price out correctly.
false otherwise.

Parameters:

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.

6.8.3.18 virtual bool ABA_VARIABLE::violated (double *rc*) const `[virtual]`

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.

Returns:

true If the variable does not price out correctly.
false otherwise.

Parameters:

rc The reduced cost of the variable.

6.8.4 Member Data Documentation

6.8.4.1 ABA_FSVARSTAT ABA_VARIABLE::fsVarStat_ `[protected]`

The global status of fixing and setting of the variable.

Definition at line 282 of file variable.h.

6.8.4.2 double ABA_VARIABLE::lBound_ `[protected]`

The lower bound of the variable.

Definition at line 290 of file variable.h.

6.8.4.3 double ABA_VARIABLE::obj_ `[protected]`

The objective function coefficient of the variable.

Definition at line 286 of file variable.h.

6.8.4.4 ABA_VARTYPE ABA_VARIABLE::type_ `[protected]`

The type of the variable.

Definition at line 298 of file variable.h.

6.8.4.5 double ABA_VARIABLE::uBound_ `[protected]`

The upper bound of the variable.

Definition at line 294 of file variable.h.

The documentation for this class was generated from the following file:

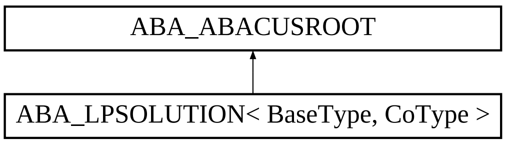
- Include/abacus/variable.h

6.9 ABA_LPSOLUTION< BaseType, CoType > Class Template Reference

template class implements an LP solution. This class is necessary when using the class ABA_SEPARATOR for separation.

`#include <lpsolution.h>`

Inheritance diagram for ABA_LPSOLUTION< BaseType, CoType >::



Public Member Functions

- ABA_LPSOLUTION (ABA_SUB ∗sub, bool primalVariables, ABA_ACTIVE< BaseType, CoType > ∗active)
- ABA_LPSOLUTION (ABA_MASTER ∗master)
- ABA_LPSOLUTION (const ABA_LPSOLUTION< BaseType, CoType > &rhs)
- ∼ABA_LPSOLUTION ()

    The destructor.

- int nVarCon () const
- double ∗ zVal ()
- ABA_ACTIVE< BaseType, CoType > ∗ active ()
- int idSub () const
- int idLp () const

Protected Attributes

- ABA_MASTER ∗ master_
- int nVarCon_
- int idSub_
- int idLp_
- ABA_ARRAY< double > zVal_
- ABA_ACTIVE< BaseType, CoType > ∗ active_

Private Member Functions

- const ABA_LPSOLUTION< BaseType, CoType > & operator= (const ABA_LPSOLUTION< BaseType, CoType > &rhs)

Friends

- class ABA_SEPARATOR< CoType, BaseType >
- ostream & operator<< (ostream &out, const ABA_LPSOLUTION< BaseType, CoType > &rhs)

6.9.1 Detailed Description

template<class BaseType, class CoType> class ABA_LPSOLUTION< BaseType, CoType >

template class implements an LP solution. This class is necessary when using the class ABA_SEPARATOR for separation.

Parameters:

ABA_MASTER ∗master_ A pointer to the corresponding master of the optimization.

int nVarCon_ The number of variables/constraints.

int idSub_ The Id of the subproblem in which the LP solution was generated.

int idLp_ The Id of the LP in which the LP solution was generated.

ABA_ARRAY<double> ∗ zVal_ The primal/dual variables of the LP solution.

ABA_ACTIVE<BaseType,CoType> ∗active_ The active variables/constraints.

Definition at line 61 of file lpsolution.h.

6.9.2 Constructor & Destructor Documentation

6.9.2.1 template<class BaseType, class CoType> ABA_LPSOLUTION< BaseType, CoType >::ABA_LPSOLUTION (ABA_SUB ∗ *sub*, bool *primalVariables*, ABA_ACTIVE< BaseType, CoType > ∗ *active*)

The constructor.

Parameters:

sub A pointer to the subproblem in which the LP solution is generated.

primalVariables True if ABA_LPSOLUTION contains the primal variables. In this case *BaseType* must be ABA_VARIABLE. If *primaVariables* is false, then *BaseType* must be ABA_CONSTRAINT.

active The active variables/constraints that are associated with the LP solution. The default argument is 0. Then the set of active variables/constraints is not stored, but is assumed to be fixed and known.

6.9.2.2 template<class BaseType, class CoType> ABA_LPSOLUTION< BaseType, CoType >::ABA_LPSOLUTION (ABA_MASTER * *master*)

The constructor.

Parameters:

master A pointer to ABA_MASTER.

6.9.2.3 template<class BaseType, class CoType> ABA_LPSOLUTION< BaseType, CoType >::ABA_LPSOLUTION (const ABA_LPSOLUTION< BaseType, CoType > & *rhs*)

The copy constructor.

Parameters:

rhs The LP solution that is copied.

6.9.2.4 template<class BaseType, class CoType> ABA_LPSOLUTION< BaseType, CoType >::~ABA_LPSOLUTION ()

The destructor.

6.9.3 Member Function Documentation

6.9.3.1 template<class BaseType, class CoType> ABA_ACTIVE<BaseType, CoType>* ABA_LPSOLUTION< BaseType, CoType >::active ()

Returns:

The active variables/constraints.

6.9.3.2 template<class BaseType, class CoType> int ABA_LPSOLUTION< BaseType, CoType >::idLp () const

Returns:

The Id of the LP in which the LP solution was generated.

6.9.3.3 template<class BaseType, class CoType> int ABA_LPSOLUTION< BaseType, CoType >::idSub () const

Returns:

The Id of the subproblem in which the LP solution was generated.

6.9.3.4 template<class BaseType, class CoType> int ABA_LPSOLUTION< BaseType, CoType >::nVarCon () const

Returns:

The number of variables (if *BaseType* is ABA_VARIABLE) or the number of constraints (if *BaseType* is ABA_CONSTRAINT), resp.

6.9.3.5 template<class BaseType, class CoType> const ABA_LPSOLUTION<BaseType, CoType>& ABA_LPSOLUTION< BaseType, CoType >::operator= (const ABA_LPSOLUTION< BaseType, CoType > & *rhs*) `[private]`

6.9.3.6 template<class BaseType, class CoType> double* ABA_LPSOLUTION< BaseType, CoType >::zVal ()

Returns:

The primal/dual variables of the LP solution.

6.9.4 Friends And Related Function Documentation

6.9.4.1 template<class BaseType, class CoType> friend class ABA_SEPARATOR< CoType, BaseType > `[friend]`

Definition at line 62 of file lpsolution.h.

6.9.4.2 template<class BaseType, class CoType> ostream& operator<< (ostream & *out*, const ABA_LPSOLUTION< BaseType, CoType > & *rhs*) `[friend]`

The output operator writes the lpsolution to an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The lpsolution being output.

6.9.5 Member Data Documentation

6.9.5.1 template<class BaseType, class CoType> ABA_ACTIVE<BaseType, CoType>* ABA_LPSOLUTION< BaseType, CoType >::active_ [protected]

Definition at line 159 of file lpsolution.h.

6.9.5.2 template<class BaseType, class CoType> int ABA_LPSOLUTION< BaseType, CoType >::idLp_ [protected]

Definition at line 154 of file lpsolution.h.

6.9.5.3 template<class BaseType, class CoType> int ABA_LPSOLUTION< BaseType, CoType >::idSub_ [protected]

Definition at line 153 of file lpsolution.h.

6.9.5.4 template<class BaseType, class CoType> ABA_MASTER* ABA_LPSOLUTION< BaseType, CoType >::master_ [protected]

Definition at line 151 of file lpsolution.h.

6.9.5.5 template<class BaseType, class CoType> int ABA_LPSOLUTION< BaseType, CoType >::nVarCon_ [protected]

Definition at line 152 of file lpsolution.h.

6.9.5.6 template<class BaseType, class CoType> ABA_ARRAY<double> ABA_LPSOLUTION< BaseType, CoType >::zVal_ [protected]

Definition at line 158 of file lpsolution.h.

The documentation for this class was generated from the following file:

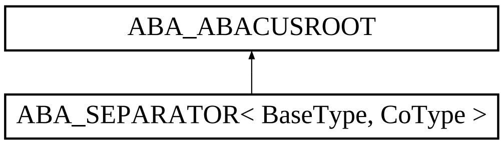
- Include/abacus/lpsolution.h

6.10 ABA_SEPARATOR< BaseType, CoType > Class Template Reference

abstract template class can be used to implement a separation routine

`#include <separator.h>`

Inheritance diagram for ABA_SEPARATOR< BaseType, CoType >::



Public Member Functions

- ABA_SEPARATOR (ABA_LPSOLUTION< CoType, BaseType > ∗lpSolution, bool nonDuplications, int maxGen=300)
- virtual ∼ABA_SEPARATOR ()

  The destructor.

- virtual void separate ()=0

  This function has to be redefined and should implement the separation routine.

- ABA_SEPARATOR_CUTFOUND cutFound (BaseType ∗)

  The function cutFound(BaseType ∗cv) *passes a cut (constraint or variable) to the buffer.*

- virtual bool terminateSeparation ()
- ABA_BUFFER< BaseType ∗ > & cutBuffer ()
- int nGen () const
- int nDuplications () const
- int nCollisions () const
- int maxGen () const
- double minAbsViolation () const
- void minAbsViolation (double minAbsVio)

  Set a new value for minAbsViolation*.*

- ABA_LPSOLUTION< CoType, BaseType > ∗ lpSolution ()

  The lpSolution to be separated.

- void watchNonDuplPool (ABA_NONDUPLPOOL< BaseType, CoType > ∗pool)

  If the separator checks for duplication of cuts, the test is also done for constraints/variables that are in the pool passed as argument.

Protected Member Functions

- bool find (BaseType ∗)

Protected Attributes

- ABA_MASTER ∗ master_
- ABA_LPSOLUTION< CoType, BaseType > ∗ lpSol_

Private Member Functions

- ABA_SEPARATOR (const ABA_SEPARATOR< BaseType, CoType > &rhs)
- const ABA_SEPARATOR< BaseType, CoType > & operator= (const ABA_SEPARATOR< BaseType, CoType > &rhs)

Private Attributes

- double minAbsViolation_
- ABA_BUFFER< BaseType ∗ > newCons_
- ABA_HASH< unsigned, BaseType ∗ > ∗ hash_
- int nDuplications_
- bool sendConstraints_
- ABA_NONDUPLPOOL< BaseType, CoType > ∗ pool_

6.10.1 Detailed Description

template<class BaseType, class CoType> class ABA_SEPARATOR< BaseType, CoType >

abstract template class can be used to implement a separation routine

Parameters:

ABA_MASTER ∗master A pointer to the corresponding master of the optimization.

ABA_LPSOLUTION<CoType,BaseType> ∗lpSol The LP solution to be separated.

Definition at line 67 of file separator.h.

6.10.2 Constructor & Destructor Documentation

6.10.2.1 template<class BaseType, class CoType> ABA_SEPARATOR< BaseType, CoType >::ABA_SEPARATOR (ABA_LPSOLUTION< CoType, BaseType > ∗ *lpSolution*, bool *nonDuplications*, int *maxGen* = `300`)

The constructor.

Parameters:

master A pointer to the corresponding master of the optimization.

lpSolution The LP solution to be separated.

maxGen The maximal number of cutting planes which are stored.

nonDuplications If this flag is set, then the same constraint/variable is stored at most once in the buffer. In this case for constraints/variables the virtual member functions *name()*, *hashKey()*, and *equal()* of the base class ABA_CONVAR have to be defined. Using these three functions, we check in the function *cutFound* if a constraint or variable is already stored in the buffer.

sendConstraint In the parallel version this parameter determines if the constraints should be sent to their corresponding stores.

6.10.2.2 template<class BaseType, class CoType> virtual ABA_SEPARATOR< BaseType, CoType >::~ABA_SEPARATOR () [virtual]

The destructor.

6.10.2.3 template<class BaseType, class CoType> ABA_SEPARATOR< BaseType, CoType >::ABA_SEPARATOR (const ABA_SEPARATOR< BaseType, CoType > & *rhs*) [private]

6.10.3 Member Function Documentation

6.10.3.1 template<class BaseType, class CoType> ABA_BUFFER<BaseType ∗>& ABA_SEPARATOR< BaseType, CoType >::cutBuffer ()

Returns:

The buffer with the generated constraints/variable.

6.10.3.2 template<class BaseType, class CoType> ABA_SEPARATOR_CUTFOUND ABA_SEPARATOR< BaseType, CoType >::cutFound (BaseType ∗)

The function *cutFound(BaseType ∗cv)* passes a cut (constraint or variable) to the buffer.

If the buffer is full or the cut already exists, the cut is deleted.

Returns:

ABAAdded, if the cut is added to the buffer;
ABADuplication, if the cut is already in the buffer;
ABAFull, if the buffer is full.

Parameters:

cv A pointer to a new constraint/variable found by the separation algorithm.

6.10.3.3 template<class BaseType, class CoType> bool ABA_SEPARATOR< BaseType, CoType >::find (BaseType ∗) [protected]

Returns:

The function checks if a constraint/variable that is equivalent to *cv* according to the function ABA_CONVAR::equal() is already stored in the buffer by using the hashtable.

Parameters:

cv A pointer to a constraint/variable for which it should be checked if an equivalent item is already contained in the buffer.

6.10.3.4 template<class BaseType, class CoType> ABA_LPSOLUTION<CoType, BaseType>∗ ABA_SEPARATOR< BaseType, CoType >::lpSolution () `[inline]`

The lpSolution to be separated.

Definition at line 149 of file separator.h.

6.10.3.5 template<class BaseType, class CoType> int ABA_SEPARATOR< BaseType, CoType >::maxGen () const

Returns:

The maximal number of generated cutting planes.

6.10.3.6 template<class BaseType, class CoType> void ABA_SEPARATOR< BaseType, CoType >::minAbsViolation (double *minAbsVio*) `[inline]`

Set a new value for *minAbsViolation*.

Definition at line 145 of file separator.h.

6.10.3.7 template<class BaseType, class CoType> double ABA_SEPARATOR< BaseType, CoType >::minAbsViolation () const

Returns:

The absolute value for considering a constraint/variable as violated.

6.10.3.8 template<class BaseType, class CoType> int ABA_SEPARATOR< BaseType, CoType >::nCollisions () const

Returns:

The number of collisions in the hash table.

6.10.3.9 template<class BaseType, class CoType> int ABA_SEPARATOR< BaseType, CoType >::nDuplications () const

Returns:

The number of duplicated constraints/variables which are discarded.

6.10.3.10 template<class BaseType, class CoType> int ABA_SEPARATOR< BaseType, CoType >::nGen () const

Returns:

The number of generated cutting planes.

6.10.3.11 template<class BaseType, class CoType> const ABA_SEPARATOR<BaseType, CoType>& ABA_SEPARATOR< BaseType, CoType >::operator= (const ABA_SEPARATOR< BaseType, CoType > & *rhs*) `[private]`

6.10.3.12 template<class BaseType, class CoType> virtual void ABA_SEPARATOR< BaseType, CoType >::separate () `[pure virtual]`

This function has to be redefined and should implement the separation routine.

6.10.3.13 template<class BaseType, class CoType> virtual bool ABA_SEPARATOR< BaseType, CoType >::terminateSeparation () `[inline, virtual]`

Returns:

The function returns true if the separation should be terminated. In the default implementation, this is the case if *maxGen* constraints/variables are in the cutBuffer.

Definition at line 117 of file separator.h.

6.10.3.14 template<class BaseType, class CoType> void ABA_SEPARATOR< BaseType, CoType >::watchNonDuplPool (ABA_NONDUPLPOOL< BaseType, CoType > ∗ *pool*) `[inline]`

If the separator checks for duplication of cuts, the test is also done for constraints/variables that are in the pool passed as argument.

This can be useful if already cuts are generated by performing constraint pool separation of this pool.

Definition at line 160 of file separator.h.

6.10.4 Member Data Documentation

6.10.4.1 template<class BaseType, class CoType> ABA_HASH<unsigned, BaseType∗>∗ ABA_SEPARATOR< BaseType, CoType >::hash_ `[private]`

Definition at line 178 of file separator.h.

6.10.4.2 template<class BaseType, class CoType> ABA_LPSOLUTION<CoType, BaseType>∗ ABA_SEPARATOR< BaseType, CoType >::lpSol_ `[protected]`

Definition at line 174 of file separator.h.

6.10.4.3 template<class BaseType, class CoType> ABA_MASTER∗ ABA_SEPARATOR< BaseType, CoType >::master_ [protected]

Definition at line 173 of file separator.h.

6.10.4.4 template<class BaseType, class CoType> double ABA_SEPARATOR< BaseType, CoType >::minAbsViolation_ [private]

Definition at line 176 of file separator.h.

6.10.4.5 template<class BaseType, class CoType> int ABA_SEPARATOR< BaseType, CoType >::nDuplications_ [private]

Definition at line 179 of file separator.h.

6.10.4.6 template<class BaseType, class CoType> ABA_BUFFER<BaseType∗> ABA_SEPARATOR< BaseType, CoType >::newCons_ [private]

Definition at line 177 of file separator.h.

6.10.4.7 template<class BaseType, class CoType> ABA_NONDUPLPOOL<BaseType, CoType>∗ ABA_SEPARATOR< BaseType, CoType >::pool_ [private]

Definition at line 181 of file separator.h.

6.10.4.8 template<class BaseType, class CoType> bool ABA_SEPARATOR< BaseType, CoType >::sendConstraints_ [private]

Definition at line 180 of file separator.h.

The documentation for this class was generated from the following file:

- Include/abacus/separator.h

%

6.11 System Classes

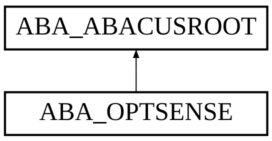
This section documents (almost) all internal system classes of ABACUS. This classes are usually not involved in the derivation process for the implementation. However for retrieving special information (e.g., about the linear program) or for advanced usage we provide here a detailed documentation.

6.12 ABA_OPTSENSE Class Reference

We can either minimize or maximize the objective function. We encapsulate this information in a class since it is required in various classes.

`#include <optsense.h>`

Inheritance diagram for ABA_OPTSENSE::



Public Types

- enum SENSE { Min, Max, Unknown }

Public Member Functions

- ABA_OPTSENSE (SENSE s=Unknown)
- void sense (SENSE s)

  This version of the function sense() *sets the optimization sense.*

- SENSE sense () const
- bool min () const
- bool max () const
- bool unknown () const

Private Attributes

- SENSE sense_

Friends

- ostream & operator<< (ostream &out, const ABA_OPTSENSE &rhs)

  The output operator writes the optimization sense on an output stream in the form { maximize}, { minimize}, or { unknown}.

6.12.1 Detailed Description

We can either minimize or maximize the objective function. We encapsulate this information in a class since it is required in various classes.

Definition at line 43 of file optsense.h.

6.12.2 Member Enumeration Documentation

6.12.2.1 enum ABA_OPTSENSE::SENSE

The enumeration defining the sense of optimization.

Parameters:

Min Minimization problem.

Max Maximization problem.

Unknown Unknown optimization sense, required to recognize uninitialized object.

Enumeration values:

Min

Max

Unknown

Definition at line 53 of file optsense.h.

6.12.3 Constructor & Destructor Documentation

6.12.3.1 ABA_OPTSENSE::ABA_OPTSENSE (SENSE *s* = `Unknown`) `[inline]`

The constructor initializes the optimization sense.

Parameters:

s The sense of the optimization. The default value is *Unknown*.

Definition at line 106 of file optsense.h.

6.12.4 Member Function Documentation

6.12.4.1 bool ABA_OPTSENSE::max () const `[inline]`

Returns:

true If it is maximization problem,
false otherwise.

Definition at line 126 of file optsense.h.

6.12.4.2 bool ABA_OPTSENSE::min () const `[inline]`

Returns:

true If it is minimization problem,
false otherwise.

Definition at line 121 of file optsense.h.

6.12.4.3 ABA_OPTSENSE::SENSE ABA_OPTSENSE::sense () const [inline]

Returns:
The sense of the optimization.

Definition at line 111 of file optsense.h.

6.12.4.4 void ABA_OPTSENSE::sense (SENSE *s*) [inline]

This version of the function *sense()* sets the optimization sense.

Parameters:
s The new sense of the optimization.

Definition at line 116 of file optsense.h.

6.12.4.5 bool ABA_OPTSENSE::unknown () const [inline]

Returns:
true If the optimization sense is unknown,
false otherwise.

Definition at line 131 of file optsense.h.

6.12.5 Friends And Related Function Documentation

6.12.5.1 ostream& operator<< (ostream & *out*, const ABA_OPTSENSE & *rhs*) [friend]

The output operator writes the optimization sense on an output stream in the form { maximize}, { minimize}, or { unknown}.

Returns:
The output stream.

Parameters:
out The output stream.
rhs The sense being output.

6.12.6 Member Data Documentation

6.12.6.1 SENSE ABA_OPTSENSE::sense_ `[private]`

The optimization sense.

Definition at line 102 of file optsense.h.

The documentation for this class was generated from the following file:

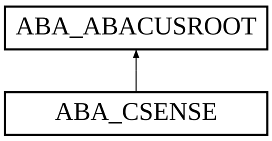
- Include/abacus/optsense.h

6.13 ABA_CSENSE Class Reference

we implement the sense of optimization as a class since we require it both in the classes ABA_CONSTRAINT and ABA_ROW.

`#include <csense.h>`

Inheritance diagram for ABA_CSENSE::



Public Types

- enum SENSE { Less, Equal, Greater }

Public Member Functions

- ABA_CSENSE (ABA_GLOBAL ∗glob)
- ABA_CSENSE (ABA_GLOBAL ∗glob, SENSE s)
- ABA_CSENSE (ABA_GLOBAL ∗glob, char s)

  With this constructor the sense of the constraint can also be initialized with a single letter.

- const ABA_CSENSE & operator= (SENSE rhs)

  The default assignment operator is overloaded such that also the enumeration SENSE *can be used on the right hand side.*

- SENSE sense () const
- void sense (SENSE s)

  This overloaded version of sense() *changes the sense of the constraint.*

- void sense (char s)

  The sense can also be changed by a character as in the constructor ABA_CSENSE(ABA_GLOBAL ∗glob, char s).

Private Attributes

- ABA_GLOBAL $*$ glob_
- SENSE sense_

Friends

- ostream & operator<< (ostream &out, const ABA_CSENSE &rhs)

  The output operator writes the sense on an output stream in the form <=, =, or >=.

6.13.1 Detailed Description

we implement the sense of optimization as a class since we require it both in the classes ABA_CONSTRAINT and ABA_ROW.

Definition at line 50 of file csense.h.

6.13.2 Member Enumeration Documentation

6.13.2.1 enum ABA_CSENSE::SENSE

Parameters:

Less $\leq$

Equal $=$

Greater $\geq$

Enumeration values:

Less

Equal

Greater

Definition at line 57 of file csense.h.

6.13.3 Constructor & Destructor Documentation

6.13.3.1 ABA_CSENSE::ABA_CSENSE (ABA_GLOBAL $*$ *glob*)

If the default constructor is used, the sense is undefined.

Parameters:

glob A pointer to the corresponding global object.

6.13.3.2 ABA_CSENSE::ABA_CSENSE (ABA_GLOBAL $*$ *glob*, SENSE *s*)

This constructor initializes the sense.

Parameters:

glob A pointer to the corresponding global object.

s The sense.

6.13.3.3 ABA_CSENSE::ABA_CSENSE (ABA_GLOBAL $*$ *glob*, char *s*)

With this constructor the sense of the constraint can also be initialized with a single letter.

Parameters:

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*.

6.13.4 Member Function Documentation

6.13.4.1 const ABA_CSENSE & ABA_CSENSE::operator= (SENSE *rhs*) `[inline]`

The default assignment operator is overloaded such that also the enumeration *SENSE* can be used on the right hand side.

Returns:

A reference to the sense.

Parameters:

rhs The new sense.

Definition at line 146 of file csense.h.

6.13.4.2 void ABA_CSENSE::sense (char *s*)

The sense can also be changed by a character as in the constructor ABA_CSENSE(ABA_GLOBAL $*$glob, char s).

Parameters:

s The new sense.

6.13.4.3 void ABA_CSENSE::sense (SENSE *s*) `[inline]`

This overloaded version of *sense()* changes the sense of the constraint.

Parameters:
s The new sense.

Definition at line 157 of file csense.h.

6.13.4.4 ABA_CSENSE::SENSE ABA_CSENSE::sense () const `[inline]`

Returns:
The sense of the constraint.

Definition at line 152 of file csense.h.

6.13.5 Friends And Related Function Documentation

6.13.5.1 ostream& operator<< (ostream & *out*, const ABA_CSENSE & *rhs*) `[friend]`

The output operator writes the sense on an output stream in the form <=, =, or >=.

Returns:
The output stream.

Parameters:
out The output stream.
rhs The sense being output.

6.13.6 Member Data Documentation

6.13.6.1 ABA_GLOBAL$*$ ABA_CSENSE::glob_ `[private]`

Definition at line 138 of file csense.h.

6.13.6.2 SENSE ABA_CSENSE::sense_ `[private]`

Stores the sense of a constraint.

Definition at line 142 of file csense.h.

The documentation for this class was generated from the following file:

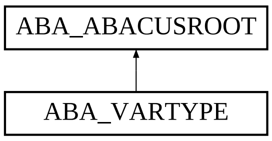
- Include/abacus/csense.h

6.14 ABA_VARTYPE Class Reference

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).

`#include <vartype.h>`

Inheritance diagram for ABA_VARTYPE::



Public Types

- enum TYPE { Continuous, Integer, Binary }

Public Member Functions

- ABA_VARTYPE ()

  The default constructor lets the type of the variable uninitialized.

- ABA_VARTYPE (TYPE t)
- TYPE type () const
- void type (TYPE t)

  This version of the function type() *sets the variable type.*

- bool discrete () const
- bool binary () const
- bool integer () const

Private Attributes

- TYPE type_

Friends

- ostream & operator<< (ostream &out, const ABA_VARTYPE &rhs)

  The output operator writes the variable type to an output stream in the format { Continuous}, { Integer}, or { Binary}.

6.14.1 Detailed Description

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).

Definition at line 45 of file vartype.h.

6.14.2 Member Enumeration Documentation

6.14.2.1 enum ABA_VARTYPE::TYPE

The enumeration with the different variable types.

Parameters:

Continuous A continuous variable.

Integer A general integer variable.

Binary A variable having value 0 or 1.

Enumeration values:

Continuous

Integer

Binary

Definition at line 54 of file vartype.h.

6.14.3 Constructor & Destructor Documentation

6.14.3.1 ABA_VARTYPE::ABA_VARTYPE () `[inline]`

The default constructor lets the type of the variable uninitialized.

Definition at line 126 of file vartype.h.

6.14.3.2 ABA_VARTYPE::ABA_VARTYPE (TYPE *t*) `[inline]`

This constructor initializes the variable type.

Parameters:

t The variable type.

Definition at line 130 of file vartype.h.

6.14.4 Member Function Documentation

6.14.4.1 bool ABA_VARTYPE::binary () const `[inline]`

Returns:

true If the type of the variable *Binary*,
false otherwise.

Definition at line 151 of file vartype.h.

6.14.4.2 bool ABA_VARTYPE::discrete () const `[inline]`

Returns:

true If the type of the variable is *Integer* or *Binary*,
false otherwise.

Definition at line 145 of file vartype.h.

6.14.4.3 bool ABA_VARTYPE::integer () const

Returns:

true If the type of the variable is *Integer*,
false otherwise.

6.14.4.4 void ABA_VARTYPE::type (TYPE *t*) `[inline]`

This version of the function *type()* sets the variable type.

Parameters:

t The new type of the variable.

Definition at line 140 of file vartype.h.

6.14.4.5 ABA_VARTYPE::TYPE ABA_VARTYPE::type () const `[inline]`

Returns:

The type of the variable.

Definition at line 135 of file vartype.h.

6.14.5 Friends And Related Function Documentation

6.14.5.1 ostream& operator<< (ostream & *out*, const ABA_VARTYPE & *rhs*) `[friend]`

The output operator writes the variable type to an output stream in the format { Continuous}, { Integer}, or { Binary}.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The variable type being output.

6.14.6 Member Data Documentation

6.14.6.1 TYPE ABA_VARTYPE::type_ `[private]`

The type of the variable.

Definition at line 122 of file vartype.h.

The documentation for this class was generated from the following file:

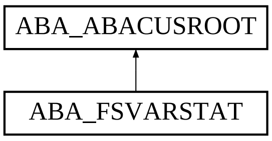
- Include/abacus/vartype.h

6.15 ABA_FSVARSTAT Class Reference

status of fixed and set variables.

`#include <fsvarstat.h>`

Inheritance diagram for ABA_FSVARSTAT::



Public Types

- enum STATUS {
  Free, SetToLowerBound, Set, SetToUpperBound,
  FixedToLowerBound, Fixed, FixedToUpperBound }

  The enumeration defining the different statuses of variables from the point of view of fixing and setting:.

Public Member Functions

- ABA_FSVARSTAT (ABA_GLOBAL ∗glob)

  This constructor initializes the status as Free.

- ABA_FSVARSTAT (ABA_GLOBAL ∗glob, STATUS status)

  This constructor initializes the status explicitely.

- ABA_FSVARSTAT (ABA_GLOBAL ∗glob, STATUS status, double value)

  This constructor initializes the status explicitely to Fixed *or* Set.

- ABA_FSVARSTAT (ABA_FSVARSTAT ∗fsVarStat)
- STATUS status () const
- void status (STATUS stat)

  This version of the function status() *assigns a new status.*

- void status (STATUS stat, double val)

  This version of the function status() *can assign a new status also for the statuses* Fixed *and* Set.

- void status (const ABA_FSVARSTAT ∗stat)

  A version of status() *for assigning the status of an other object of the class ABA_FSVARSTAT.*

- double value () const
- void value (double val)

  This version of value() *assigns a new value of fixing or setting.*

- bool fixed () const
- bool set () const
- bool fixedOrSet () const
- bool contradiction (ABA_FSVARSTAT ∗fsVarStat) const

  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 contradiction (STATUS status, double value=0) const

Private Attributes

- ABA_GLOBAL ∗ glob_
- STATUS status_
- double value_

Friends

- ostream & operator<< (ostream &out, const ABA_FSVARSTAT &rhs)

  The output operator writes the status and, if the status is Fixed *or* Set, *also its value on an output stream.*

6.15.1 Detailed Description

status of fixed and set variables.

Definition at line 49 of file fsvarstat.h.

6.15.2 Member Enumeration Documentation

6.15.2.1 enum ABA_FSVARSTAT::STATUS

The enumeration defining the different statuses of variables from the point of view of fixing and setting:.

Parameters:

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.

Enumeration values:

Free

SetToLowerBound

Set

SetToUpperBound

FixedToLowerBound

Fixed

FixedToUpperBound

Definition at line 65 of file fsvarstat.h.

6.15.3 Constructor & Destructor Documentation

6.15.3.1 ABA_FSVARSTAT::ABA_FSVARSTAT (ABA_GLOBAL $*$ *glob*) `[inline]`

This constructor initializes the status as *Free*.

Parameters:

glob A pointer to a global object.

Definition at line 227 of file fsvarstat.h.

6.15.3.2 ABA_FSVARSTAT::ABA_FSVARSTAT (ABA_GLOBAL $*$ *glob*, STATUS *status*)

This constructor initializes the status explicitely.

Parameters:

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.

6.15.3.3 ABA_FSVARSTAT::ABA_FSVARSTAT (ABA_GLOBAL ∗ *glob*, STATUS *status*, double *value*)

This constructor initializes the status explicitely to *Fixed* or *Set*.

Parameters:

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*.

6.15.3.4 ABA_FSVARSTAT::ABA_FSVARSTAT (ABA_FSVARSTAT ∗ *fsVarStat*)

This constructor makes a copy.

Parameters:

fsVarStat The status is initialized with a copy of ∗*fsVarStat*.

6.15.4 Member Function Documentation

6.15.4.1 bool ABA_FSVARSTAT::contradiction (STATUS *status*, double *value* = `0`) const

Another version of the function *contradiction()*.

Returns:

true If there is a contradiction between the status of this object and (*status*, *value*),
false otherwise.

Parameters:

status The status with which contradiction is checked.

value The value with which contradiction is checked. The default value of *value* is 0.

6.15.4.2 bool ABA_FSVARSTAT::contradiction (ABA_FSVARSTAT ∗ *fsVarStat*) const

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.

Returns:

true If there is a contradiction between the status of this object and *fsVarStat*,
false otherwise.

Parameters:

fsVarStat A pointer to the status with which contradiction is is tested.

6.15.4.3 bool ABA_FSVARSTAT::fixed () const

Returns:

true If the status is *FixedToLowerBound*, *Fixed*, or *FixedToUpperBound*,
false otherwise.

6.15.4.4 bool ABA_FSVARSTAT::fixedOrSet () const `[inline]`

Returns:

false If the status is *Free*,
true otherwise.

Definition at line 265 of file fsvarstat.h.

6.15.4.5 bool ABA_FSVARSTAT::set () const

Returns:

true If the status is *SetToLowerBound*, *Set*, or *SetToUpperBound*,
false otherwise.

6.15.4.6 void ABA_FSVARSTAT::status (const ABA_FSVARSTAT $*$ *stat*) `[inline]`

A version of *status()* for assigning the status of an other object of the class ABA_FSVARSTAT.

Parameters:

stat A pointer to the object that status and value is copied.

Definition at line 249 of file fsvarstat.h.

6.15.4.7 void ABA_FSVARSTAT::status (STATUS *stat*, double *val*) `[inline]`

This version of the function *status()* can assign a new status also for the statuses *Fixed* and *Set*.

Parameters:

stat The new status.

val A value associated with the new status.

Definition at line 243 of file fsvarstat.h.

6.15.4.8 void ABA_FSVARSTAT::status (STATUS *stat*) `[inline]`

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.

Parameters:

stat The new status.

Definition at line 238 of file fsvarstat.h.

6.15.4.9 ABA_FSVARSTAT::STATUS ABA_FSVARSTAT::status () const `[inline]`

Returns:

The status of fixing or setting.

Definition at line 233 of file fsvarstat.h.

6.15.4.10 void ABA_FSVARSTAT::value (double *val*) `[inline]`

This version of *value()* assigns a new value of fixing or setting.

Parameters:

val The new value.

Definition at line 260 of file fsvarstat.h.

6.15.4.11 double ABA_FSVARSTAT::value () const `[inline]`

Returns:

The value of fixing or setting if the variable has status *Fixed* or *Set*.

Definition at line 255 of file fsvarstat.h.

6.15.5 Friends And Related Function Documentation

6.15.5.1 ostream& operator<< (ostream & *out*, const ABA_FSVARSTAT & *rhs*) `[friend]`

The output operator writes the status and, if the status is *Fixed* or *Set*, also its value on an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The variable status being output.

6.15.6 Member Data Documentation

6.15.6.1 ABA_GLOBAL∗ ABA_FSVARSTAT::glob_ `[private]`

A pointer to the corresponding global object.

Definition at line 213 of file fsvarstat.h.

6.15.6.2 STATUS ABA_FSVARSTAT::status_ `[private]`

The status of the variable.

Definition at line 217 of file fsvarstat.h.

6.15.6.3 double ABA_FSVARSTAT::value_ `[private]`

The value the variable is fixed/set to.

This member is only used for the statuses *Fixed* and *Set*.

Definition at line 223 of file fsvarstat.h.

The documentation for this class was generated from the following file:

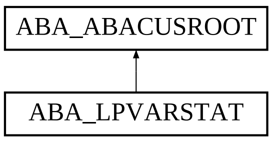
- Include/abacus/fsvarstat.h

6.16 ABA_LPVARSTAT Class Reference

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.

`#include <lpvarstat.h>`

Inheritance diagram for ABA_LPVARSTAT::



Public Types

- enum STATUS {
  AtLowerBound, Basic, AtUpperBound, NonBasicFree,
  Eliminated, Unknown }

  The enumeration of the statuses a variable gets from the linear program solver:.

Public Member Functions

- ABA_LPVARSTAT (ABA_GLOBAL ∗glob)
- ABA_LPVARSTAT (ABA_GLOBAL ∗glob, STATUS status)
- ABA_LPVARSTAT (ABA_LPVARSTAT ∗lpVarStat)
- STATUS status () const
- void status (STATUS stat)

  This version of status() *sets the status.*

- void status (const ABA_LPVARSTAT ∗stat)

  Another version of the function status() *for setting the status.*

- bool atBound () const
- bool basic () const

Private Attributes

- ABA_GLOBAL ∗ glob_
- STATUS status_

Friends

- ostream & operator<< (ostream &out, const ABA_LPVARSTAT &rhs)

  The output operator writes the STATUS *to an output stream in the form { AtLowerBound}, { Basic}, { AtUpper\-Bound}, { NonBasicFree}, { Eliminated}, { Unknown}.*

6.16.1 Detailed Description

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.

Definition at line 51 of file lpvarstat.h.

6.16.2 Member Enumeration Documentation

6.16.2.1 enum ABA_LPVARSTAT::STATUS

The enumeration of the statuses a variable gets from the linear program solver:.

Parameters:

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.

Enumeration values:

AtLowerBound

Basic

AtUpperBound

NonBasicFree

Eliminated

Unknown

Definition at line 73 of file lpvarstat.h.

6.16.3 Constructor & Destructor Documentation

6.16.3.1 ABA_LPVARSTAT::ABA_LPVARSTAT (ABA_GLOBAL $*$ *glob*) `[inline]`

This constructor initializes the status as *Unknown*.

Parameters:

glob A pointer to the corresponding global object.

Definition at line 164 of file lpvarstat.h.

6.16.3.2 ABA_LPVARSTAT::ABA_LPVARSTAT (ABA_GLOBAL $*$ *glob*, STATUS *status*) `[inline]`

This constructor initializes the ABA_LPVARSTAT.

Parameters:

glob A pointer to the corresponding global object.

status The initial status.

Definition at line 170 of file lpvarstat.h.

6.16.3.3 ABA_LPVARSTAT::ABA_LPVARSTAT (ABA_LPVARSTAT $*$ *lpVarStat*) `[inline]`

This constructor make a copy of $*$*lpVarStat*.

Parameters:

lpVarStat A copy of this object is made.

Definition at line 176 of file lpvarstat.h.

6.16.4 Member Function Documentation

6.16.4.1 bool ABA_LPVARSTAT::atBound () const `[inline]`

Returns:

true If the variable status is *AtUpperBound* or *AtLowerBound*,
false otherwise.

Definition at line 197 of file lpvarstat.h.

6.16.4.2 bool ABA_LPVARSTAT::basic () const `[inline]`

Returns:

true If the status is *Basic*,
false otherwise.

Definition at line 203 of file lpvarstat.h.

6.16.4.3 void ABA_LPVARSTAT::status (const ABA_LPVARSTAT ∗ *stat*) `[inline]`

Another version of the function *status()* for setting the status.

Parameters:

stat The new LP-status.

Definition at line 192 of file lpvarstat.h.

6.16.4.4 void ABA_LPVARSTAT::status (STATUS *stat*) `[inline]`

This version of *status()* sets the status.

Parameters:

stat The new LP-status.

Definition at line 187 of file lpvarstat.h.

6.16.4.5 ABA_LPVARSTAT::STATUS ABA_LPVARSTAT::status () const `[inline]`

Returns:

The LP-status.

Definition at line 182 of file lpvarstat.h.

6.16.5 Friends And Related Function Documentation

6.16.5.1 ostream& operator<< (ostream & *out*, const ABA_LPVARSTAT & *rhs*) [friend]

The output operator writes the *STATUS* to an output stream in the form { AtLowerBound}, { Basic}, { AtUpper\-Bound}, { NonBasicFree}, { Eliminated}, { Unknown}.

Returns:
A reference to the output stream.

Parameters:
out The output stream.
rhs The status being output.

6.16.6 Member Data Documentation

6.16.6.1 ABA_GLOBAL∗ ABA_LPVARSTAT::glob_ [private]

A pointer to the corresponding global object.

Definition at line 156 of file lpvarstat.h.

6.16.6.2 STATUS ABA_LPVARSTAT::status_ [private]

The LP-status.

Definition at line 160 of file lpvarstat.h.

The documentation for this class was generated from the following file:

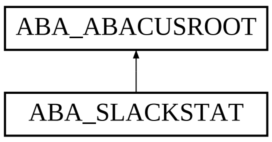
- Include/abacus/lpvarstat.h

6.17 ABA_SLACKSTAT Class Reference

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.

`#include <slackstat.h>`

Inheritance diagram for ABA_SLACKSTAT::



Public Types

- enum STATUS { Basic, NonBasicZero, NonBasicNonZero, Unknown }

Public Member Functions

- ABA_SLACKSTAT (const ABA_GLOBAL ∗glob)

  This constructor initializes the status as Unknown.

- ABA_SLACKSTAT (const ABA_GLOBAL ∗glob, STATUS status)
- STATUS status () const
- void status (STATUS stat)

  This version of the function status() *sets the status of the slack variable.*

- void status (const ABA_SLACKSTAT ∗stat)

  This version of the function status() *sets the status to the one of* ∗stat.

Private Attributes

- const ABA_GLOBAL ∗ glob_
- STATUS status_

Friends

- ostream & operator<< (ostream &out, const ABA_SLACKSTAT &rhs)

  The output operator writes the status to an output stream in the format { Basic}, { NonBasicZero}, { Non\-Basic\-Non\-Zero}, or { Unknown}.

6.17.1 Detailed Description

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.

Definition at line 50 of file slackstat.h.

6.17.2 Member Enumeration Documentation

6.17.2.1 enum ABA_SLACKSTAT::STATUS

The different statuses of a slack variable:

Parameters:

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.

Enumeration values:

Basic

NonBasicZero

NonBasicNonZero

Unknown

Definition at line 64 of file slackstat.h.

6.17.3 Constructor & Destructor Documentation

6.17.3.1 ABA_SLACKSTAT::ABA_SLACKSTAT (const ABA_GLOBAL ∗ *glob*) `[inline]`

This constructor initializes the status as *Unknown*.

Parameters:

glob A pointer to the corresponding global object.

Definition at line 137 of file slackstat.h.

6.17.3.2 ABA_SLACKSTAT::ABA_SLACKSTAT (const ABA_GLOBAL ∗ *glob*, STATUS *status*) `[inline]`

A constructor with initialization.

Parameters:

glob A pointer to the corresponding global object.

status The slack variable receives the status *status*.

Definition at line 143 of file slackstat.h.

6.17.4 Member Function Documentation

6.17.4.1 void ABA_SLACKSTAT::status (const ABA_SLACKSTAT ∗ *stat*) `[inline]`

This version of the function *status()* sets the status to the one of ∗*stat*.

Parameters:

stat The status of the slack variable is set to ∗*stat*.

Definition at line 159 of file slackstat.h.

6.17.4.2 void ABA_SLACKSTAT::status (STATUS *stat*) [inline]

This version of the function *status()* sets the status of the slack variable.

Parameters:
 stat The new status of the slack variable.

Definition at line 154 of file slackstat.h.

6.17.4.3 ABA_SLACKSTAT::STATUS ABA_SLACKSTAT::status () const [inline]

Returns:
 The status of the slack variable.

Definition at line 149 of file slackstat.h.

6.17.5 Friends And Related Function Documentation

6.17.5.1 ostream& operator<< (ostream & *out*, const ABA_SLACKSTAT & *rhs*) [friend]

The output operator writes the status to an output stream in the format { Basic}, { NonBasicZero}, { Non\-Basic\-Non\-Zero}, or { Unknown}.

Returns:
 A reference to the output stream.

Parameters:
 out The output stream.
 rhs The status being output.

6.17.6 Member Data Documentation

6.17.6.1 const ABA_GLOBAL∗ ABA_SLACKSTAT::glob_ [private]

A pointer to the corresponding global object.

Definition at line 129 of file slackstat.h.

6.17.6.2 STATUS ABA_SLACKSTAT::status_ [private]

The status of the slack variable.

Definition at line 133 of file slackstat.h.

The documentation for this class was generated from the following file:

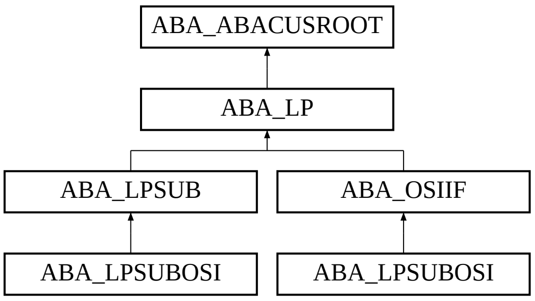
- Include/abacus/slackstat.h

6.18 ABA_LP Class Reference

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 \ algorithm and for interfaces to LP-solvers (ABA_OSIIF).

`#include <lp.h>`

Inheritance diagram for ABA_LP::



Public Types

- enum OPTSTAT {
  Optimal, Unoptimized, Error, Feasible,
  Infeasible, Unbounded }
- enum SOLSTAT { Available, Missing }

  This enumeration describes if parts of the solution like x -values, reduced costs, etc. are available.

- enum METHOD {
  Primal, Dual, BarrierAndCrossover, BarrierNoCrossover,
  Approximate }

Public Member Functions

- ABA_LP (ABA_MASTER ∗master)
- virtual ∼ABA_LP ()

  The destructor.

- 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)

This version of the function initialize() performs like its previous version, but also initializes the basis with the arguments:.

- 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)

  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.

- 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)

  Writes the complete basis of an optimal linear program to a file.

- int setSimplexIterationLimit (int limit)
- int getSimplexIterationLimit (int &limit)
- ABA_CPUTIMER ∗ lpSolverTime ()

Protected Member Functions

- 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
- virtual ABA_OPTSENSE _sense () const =0

  The pure virtual function _sense() must be defined by the used LP-solver and return the sense of the optimization.

- virtual void _sense (const ABA_OPTSENSE &newSense)=0
- virtual int _nRow () const =0

  The pure virtual function _nRow() must be defined by the used LP-solver and return the number of rows of the problem.

- virtual int _maxRow () const =0

  The pure virtual function _maxRow() must be defined by the used LP-solver and return the maximal number of rows.

- virtual int _nCol () const =0

  The pure virtual function _nCol() must be defined by the used LP-solver and return the number of columns.

- virtual int _maxCol () const =0

  The pure virtual function _maxCol() must be defined by the the used LP-solver and return the maximal number of columns.

- virtual int _nnz () const =0

  The pure virtual function _nnz() must be defined by the used LP-solver and return the number of nonzero elements of the constraint matrix not including the right hand side and the bounds of the variables.

- virtual double _obj (int i) const =0

  The pure virtual function _obj() must be defined by the used LP-solver and return the objective function coefficient of variable i.

- virtual double _lBound (int i) const =0

  The pure virtual function _lBound() must be defined by the used LP-solver and return the lower bound of variable i.

- virtual double _uBound (int i) const =0

  The pure virtual function _uBound() must be defined by the used LP-solver and return the upper bound of variable i.

- virtual double _rhs (int i) const =0

  The pure virtual function _rhs() must be defined by the used LP-solver and return the right hand side of constraint i.

- virtual void _row (int i, ABA_ROW &r) const =0

- virtual 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)=0

  The pure virtual function _initialize() *must be defined by the used LP-solver and should initialize the LP-solver with.*

- virtual void _loadBasis (ABA_ARRAY< ABA_LPVARSTAT::STATUS > &lpVarStat, ABA_ARRAY< ABA_SLACKSTAT::STATUS > &slackStat)=0
- virtual OPTSTAT _primalSimplex ()=0

  The pure virtual function _primalSimplex() *must be defined by the used LP-solver and should call the primal simplex method of the used LP-solver.*

- virtual OPTSTAT _dualSimplex ()=0

  The pure virtual function _dualSimplex() *must be defined by the used LP-solver and should call the dual simplex method of the used LP-solver.*

- virtual OPTSTAT _barrier (bool doCrossover)=0

  The pure virtual function _barrier() *must be defined by the used LP-solver and should call the barrier method of the used LP-solver.*

- virtual OPTSTAT _approx ()=0

  The pure virtual function _approx() *must be defined by the used LP-solver and should call the approximative method of the used LP-solver.*

- virtual double _value () const =0

  The pure virtual function _value() *must be defined by the used LP-solver and should return the optimum value of the linear program after it has been solved.*

- virtual double _xVal (int i)=0

  The pure virtual function _xVal() *must be defined by the used LP-solver and should return the value of variable* i *in the LP-solution.*

- virtual double _barXVal (int i)=0
- virtual double _reco (int i)=0

  The pure virtual function _reco() *must be defined by the used LP-solver and should return the reduced cost of variable* i.

- virtual double _slack (int i)=0

  The pure virtual function _slack() *must be defined by the used LP-solver and should return the value of the slack variable* i.

- virtual double _yVal (int i)=0

  The pure virtual function _yVal() *must be defined by the used LP-solver and should return the value of the dual variable of the constraint* i.

- virtual ABA_LPVARSTAT::STATUS _lpVarStat (int i)=0

  The pure virtual function _lpVarStat() *must be defined by the used LP-solver and should return the status of the variable* i *in the LP-solution.*

- virtual ABA_SLACKSTAT::STATUS _slackStat (int i)=0

  The pure virtual function _slackStat() *must be defined by the used LP-solver and should return the status of the slack variable* i *in the LP-solution.*

- virtual int _getInfeas (int &infeasRow, int &infeasCol, double ∗bInvRow)=0

  The pure virtual function _getInfeas() must be defined by the used LP-solver and can be called if the last linear program has been solved with the dual simplex method and is infeasible.

- virtual void _remRows (ABA_BUFFER< int > &ind)=0

  The pure virtual function _remRows() must be defined by the used LP-solver and should remove the rows with numbers given in the buffer ind *from the LP-solver.*

- virtual void _addRows (ABA_BUFFER< ABA_ROW ∗ > &newRows)=0

  The pure virtual function _addRows() must be defined by the used LP-solver and should add the rows given in the buffer newRows *to the LP.*

- virtual void _remCols (ABA_BUFFER< int > &vars)=0

  The pure virtual function _remCols() must be defined by the used LP-solver and should remove the columns with numbers given in vars *from the LP.*

- virtual void _addCols (ABA_BUFFER< ABA_COLUMN ∗ > &newCols)=0

  The pure virtual function _addCols() must be defined by the used LP-solver and should add the columns newCols *to the LP.*

- virtual void _changeRhs (ABA_ARRAY< double > &newRhs)=0

  The pure virtual function _changeRhs() must be defined by the used LP-solver and should set the right hand side of the constraint matrix of the LP to newRhs.

- virtual void _changeLBound (int i, double newLb)=0

  The pure virtual function _changeLBound() must be defined by the used LP-solver and should set the lower bound of variable i *to* newLb.

- virtual void _changeUBound (int i, double newUb)=0

  The pure virtual function _changeLBound() must be defined by the used LP-solver and should set the upper bound of variable i *to* newUb.

- virtual int _pivotSlackVariableIn (ABA_BUFFER< int > &rows)=0

  The function pivotSlackVariableIn() pivots the slack variables stored in the buffer rows *into the basis.*

- virtual void _rowRealloc (int newSize)=0

  The pure virtual function _rowRealloc() must be defined in the used LP-solver and should reallocate its memory such that up to newSize *rows can be handled.*

- virtual void _colRealloc (int newSize)=0

  The pure virtual function _colRealloc() must be defined by the used LP-solver and should reallocate its memory such that up to newSize *columns can be handled.*

- virtual int _setSimplexIterationLimit (int limit)=0

  The function setSimplexIterationLimit() changes the iteration limit of the Simplex algorithm.

- virtual int _getSimplexIterationLimit (int &limit)=0

  The function getSimplexIterationLimit() retrieves the value of the iteration limit of the simplex algorithm.

Protected Attributes

- ABA_MASTER $*$ master_
- OPTSTAT optStat_
- SOLSTAT xValStatus_

  This member becomes Available *if the x -values of the optimal solution can be accessed with the function* xVal()*, otherwise it has the value* Missing.

- SOLSTAT barXValStatus_
- SOLSTAT yValStatus_

  This member becomes Available *if the values of the dual variables of the optimal solution can be accessed with the function* yVal()*, otherwise it has the value* Missing/.

- SOLSTAT recoStatus_

  This member becomes Available *if the reduced costs of the optimal solution can be accessed with the function* reco()*, otherwise it has the value* Missing.

- SOLSTAT slackStatus_

  This member becomes Available *if the values of the slack variables of the optimal solution can be accessed with the function* slack()*, otherwise it has the value* Missing.

- SOLSTAT basisStatus_

  This member becomes Available *if the status of the variables and the slack variables of the optimal solution can be accessed with the functions* lpVarStat() *and* slackStat()*, otherwise it has the value* Missing.

- int nOpt_
- ABA_CPUTIMER lpSolverTime_

Private Member Functions

- void initPostOpt ()

  Resets the optimization status and the availability statuses of the solution.

- ABA_LP (const ABA_LP &rhs)
- const ABA_LP & operator= (const ABA_LP &rhs)

Friends

- ostream & operator<< (ostream &out, const ABA_LP &rhs)

  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.

6.18.1 Detailed Description

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 \ algorithm and for interfaces to LP-solvers (ABA_OSIIF).

Definition at line 70 of file lp.h.

6.18.2 Member Enumeration Documentation

6.18.2.1 enum ABA_LP::METHOD

The solution method for the linear program.

Parameters:

Primal The primal simplex method.

Dual The dual simplex method.

BarrierAndCrossover The barrier method followed by a crossover to a basis.

BarrierNoCrossover The barrier method without crossover.

Approximate An approximative solver

Enumeration values:

Primal

Dual

BarrierAndCrossover

BarrierNoCrossover

Approximate

Definition at line 107 of file lp.h.

6.18.2.2 enum ABA_LP::OPTSTAT

The optimization status of the linear program.

Parameters:

Unoptimized Optimization is still required, this is also the case for reoptimization.

Optimized The optimization has been performed, yet only a call to |()| 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.

Enumeration values:

Optimal

Unoptimized

Error

Feasible

Infeasible

Unbounded

Definition at line 87 of file lp.h.

6.18.2.3 enum ABA_LP::SOLSTAT

This enumeration describes if parts of the solution like x -values, reduced costs, etc. are available.

Parameters:

Available The part of the solution is available.

Missing The part of the solution is missing.

Enumeration values:

Available

Missing

Definition at line 96 of file lp.h.

6.18.3 Constructor & Destructor Documentation

6.18.3.1 ABA_LP::ABA_LP (ABA_MASTER $*$ *master*)

The constructor.

Parameters:

master A pointer to the corresponding master of the optimization.

6.18.3.2 virtual ABA_LP::~ABA_LP () `[virtual]`

The destructor.

6.18.3.3 ABA_LP::ABA_LP (const ABA_LP & *rhs*) `[private]`

6.18.4 Member Function Documentation

6.18.4.1 virtual void ABA_LP::_addCols (ABA_BUFFER< ABA_COLUMN $*$ > & *newCols*) `[protected, pure virtual]`

The pure virtual function *_addCols()* must be defined by the used LP-solver and should add the columns *newCols* to the LP.

Implemented in ABA_OSIIF.

6.18.4.2 virtual void ABA_LP::_addRows (ABA_BUFFER< ABA_ROW * > & *newRows*) `[protected, pure virtual]`

The pure virtual function *_addRows()* must be defined by the used LP-solver and should add the rows given in the buffer *newRows* to the LP.

Implemented in ABA_OSIIF.

6.18.4.3 virtual OPTSTAT ABA_LP::_approx () `[protected, pure virtual]`

The pure virtual function *_approx()* must be defined by the used LP-solver and should call the approximative method of the used LP-solver.

Implemented in ABA_OSIIF.

6.18.4.4 virtual OPTSTAT ABA_LP::_barrier (bool *doCrossover*) `[protected, pure virtual]`

The pure virtual function *_barrier()* must be defined by the used LP-solver and should call the barrier method of the used LP-solver.

Implemented in ABA_OSIIF.

6.18.4.5 virtual double ABA_LP::_barXVal (int *i*) `[protected, pure virtual]`

Implemented in ABA_OSIIF.

6.18.4.6 virtual void ABA_LP::_changeLBound (int *i*, double *newLb*) `[protected, pure virtual]`

The pure virtual function *_changeLBound()* must be defined by the used LP-solver and should set the lower bound of variable *i* to *newLb*.

Implemented in ABA_OSIIF.

6.18.4.7 virtual void ABA_LP::_changeRhs (ABA_ARRAY< double > & *newRhs*) `[protected, pure virtual]`

The pure virtual function *_changeRhs()* must be defined by the used LP-solver and should set the right hand side of the constraint matrix of the LP to *newRhs*.

Implemented in ABA_OSIIF.

6.18.4.8 virtual void ABA_LP::_changeUBound (int *i*, double *newUb*) `[protected, pure virtual]`

The pure virtual function *_changeLBound()* must be defined by the used LP-solver and should set the upper bound of variable *i* to *newUb*.

Implemented in ABA_OSIIF.

6.18.4.9 virtual void ABA_LP::_colRealloc (int *newSize*) `[protected, pure virtual]`

The pure virtual function *_colRealloc()* must be defined by the used LP-solver and should reallocate its memory such that up to *newSize* columns can be handled.

Implemented in ABA_OSIIF.

6.18.4.10 virtual OPTSTAT ABA_LP::_dualSimplex () `[protected, pure virtual]`

The pure virtual function *_dualSimplex()* must be defined by the used LP-solver and should call the dual simplex method of the used LP-solver.

Implemented in ABA_OSIIF.

6.18.4.11 virtual int ABA_LP::_getInfeas (int & *infeasRow*, int & *infeasCol*, double $*$ *bInvRow*) `[protected, pure virtual]`

The pure virtual function *_getInfeas()* must be defined by the used LP-solver and can be called if the last linear program has been solved with the dual simplex method and is infeasible.

In this case it should compute the infeasible basic variable or constraint and the corresponding row *bInvRow* of the basis inverse. Either *infeasRow* or *infeasCol* is nonnegative. The nonnegative argument is an infeasible row or column, respectively.

Returns:

0 if it is successful
1 otherwise.

Implemented in ABA_OSIIF.

6.18.4.12 virtual int ABA_LP::_getSimplexIterationLimit (int & *limit*) `[protected, pure virtual]`

The function *getSimplexIterationLimit()* retrieves the value of the iteration limit of the simplex algorithm.

Returns:

0 If the iteration limit could be get,
1 otherwise.

Parameters:

limit Stores the value of the iteration limit if the function returns 0.

Implemented in ABA_OSIIF.

6.18.4.13 virtual 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*) `[protected, pure virtual]`

The pure virtual function *_initialize()* must be defined by the used LP-solver and should initialize the LP-solver with.

Parameters:

sense The sense of the optimization.

nRow The number of rows.

maxRow The maximal number of rows.

nCol The number of columns.

maxCol The maximal number of columns.

obj An array with the objective functions coefficients.

lBound An array with the lower bounds of the variables.

uBound An array with the upper bounds of the variables.

rows An array storing the constraint matrix in row format.

Implemented in ABA_OSIIF.

6.18.4.14 virtual double ABA_LP::_lBound (int *i*) const `[protected, pure virtual]`

The pure virtual function *_lBound()* must be defined by the used LP-solver and return the lower bound of variable *i*.

Implemented in ABA_OSIIF.

6.18.4.15 virtual void ABA_LP::_loadBasis (ABA_ARRAY< ABA_LPVARSTAT::STATUS > & *lpVarStat*, ABA_ARRAY< ABA_SLACKSTAT::STATUS > & *slackStat*) `[protected, pure virtual]`

This pure virtual function should load a basis into the LP-solver.

Parameters:

lpVarStat An array storing the status of the variables.

slackStat An array storing the status of the slack variables.

Implemented in ABA_OSIIF.

6.18.4.16 virtual ABA_LPVARSTAT::STATUS ABA_LP::_lpVarStat (int *i*) `[protected, pure virtual]`

The pure virtual function *_lpVarStat()* must be defined by the used LP-solver and should return the status of the variable *i* in the LP-solution.

Implemented in ABA_OSIIF.

6.18.4.17 virtual int ABA_LP::_maxCol () const `[protected, pure virtual]`

The pure virtual function *_maxCol()* must be defined by the the used LP-solver and return the maximal number of columns.

Implemented in ABA_OSIIF.

6.18.4.18 virtual int ABA_LP::_maxRow () const `[protected, pure virtual]`

The pure virtual function *_maxRow()* must be defined by the used LP-solver and return the maximal number of rows.

Implemented in ABA_OSIIF.

6.18.4.19 virtual int ABA_LP::_nCol () const `[protected, pure virtual]`

The pure virtual function *_nCol()* must be defined by the used LP-solver and return the number of columns.

Implemented in ABA_OSIIF.

6.18.4.20 virtual int ABA_LP::_nnz () const `[protected, pure virtual]`

The pure virtual function *_nnz()* must be defined by the used LP-solver and return the number of nonzero elements of the constraint matrix not including the right hand side and the bounds of the variables.

Implemented in ABA_OSIIF.

6.18.4.21 virtual int ABA_LP::_nRow () const `[protected, pure virtual]`

The pure virtual function *_nRow()* must be defined by the used LP-solver and return the number of rows of the problem.

Implemented in ABA_OSIIF.

6.18.4.22 virtual double ABA_LP::_obj (int *i*) const `[protected, pure virtual]`

The pure virtual function *_obj()* must be defined by the used LP-solver and return the objective function coefficient of variable *i*.

Implemented in ABA_OSIIF.

6.18.4.23 virtual int ABA_LP::_pivotSlackVariableIn (ABA_BUFFER< int > & *rows*) `[protected, pure virtual]`

The function *pivotSlackVariableIn()* pivots the slack variables stored in the buffer *rows* into the basis.

Returns:

0 All variables could be pivoted in,
1 otherwise.

Parameters:

rows The numbers of the slack variables that should be pivoted in.

Implemented in ABA_OSIIF.

6.18.4.24 virtual OPTSTAT ABA_LP::_primalSimplex () `[protected, pure virtual]`

The pure virtual function *_primalSimplex()* must be defined by the used LP-solver and should call the primal simplex method of the used LP-solver.

Implemented in ABA_OSIIF.

6.18.4.25 virtual double ABA_LP::_reco (int *i*) `[protected, pure virtual]`

The pure virtual function *_reco()* must be defined by the used LP-solver and should return the reduced cost of variable *i*.

Implemented in ABA_OSIIF.

6.18.4.26 virtual void ABA_LP::_remCols (ABA_BUFFER< int > & *vars*) `[protected, pure virtual]`

The pure virtual function *_remCols()* must be defined by the used LP-solver and should remove the columns with numbers given in *vars* from the LP.

Implemented in ABA_OSIIF.

6.18.4.27 virtual void ABA_LP::_remRows (ABA_BUFFER< int > & *ind*) `[protected, pure virtual]`

The pure virtual function *_remRows()* must be defined by the used LP-solver and should remove the rows with numbers given in the buffer *ind* from the LP-solver.

Implemented in ABA_OSIIF.

6.18.4.28 virtual double ABA_LP::_rhs (int *i*) const `[protected, pure virtual]`

The pure virtual function *_rhs()* must be defined by the used LP-solver and return the right hand side of constraint *i*.

Implemented in ABA_OSIIF.

6.18.4.29 virtual void ABA_LP::_row (int *i*, ABA_ROW & *r*) const `[protected, pure virtual]`

The pure virtual function *_row()* must be defined by the used LP-solver and store the *i-th* row of the problem in the row *r*.

Implemented in ABA_OSIIF.

6.18.4.30 virtual void ABA_LP::_rowRealloc (int *newSize*) `[protected, pure virtual]`

The pure virtual function *_rowRealloc()* must be defined in the used LP-solver and should reallocate its memory such that up to *newSize* rows can be handled.

Implemented in ABA_OSIIF.

6.18.4.31 virtual void ABA_LP::_sense (const ABA_OPTSENSE & *newSense*) `[protected, pure virtual]`

Implemented in ABA_OSIIF.

6.18.4.32 virtual ABA_OPTSENSE ABA_LP::_sense () const `[protected, pure virtual]`

The pure virtual function *_sense()* must be defined by the used LP-solver and return the sense of the optimization.

Implemented in ABA_OSIIF.

6.18.4.33 virtual int ABA_LP::_setSimplexIterationLimit (int *limit*) `[protected, pure virtual]`

The function *setSimplexIterationLimit()* changes the iteration limit of the Simplex algorithm.

Returns:
0 If the iteration limit could be set,
1 otherwise.

Parameters:
limit The new value of the iteration limit.

Implemented in ABA_OSIIF.

6.18.4.34 virtual double ABA_LP::_slack (int *i*) `[protected, pure virtual]`

The pure virtual function *_slack()* must be defined by the used LP-solver and should return the value of the slack variable *i*.

Implemented in ABA_OSIIF.

6.18.4.35 virtual ABA_SLACKSTAT::STATUS ABA_LP::_slackStat (int *i*) `[protected, pure virtual]`

The pure virtual function *_slackStat()* must be defined by the used LP-solver and should return the status of the slack variable *i* in the LP-solution.

Implemented in ABA_OSIIF.

6.18.4.36 virtual double ABA_LP::_uBound (int *i*) const `[protected, pure virtual]`

The pure virtual function *_uBound()* must be defined by the used LP-solver and return the upper bound of variable *i*.

Implemented in ABA_OSIIF.

6.18.4.37 virtual double ABA_LP::_value () const `[protected, pure virtual]`

The pure virtual function *_value()* must be defined by the used LP-solver and should return the optimum value of the linear program after it has been solved.

Implemented in ABA_OSIIF.

6.18.4.38 virtual double ABA_LP::_xVal (int *i*) `[protected, pure virtual]`

The pure virtual function *_xVal()* must be defined by the used LP-solver and should return the value of variable *i* in the LP-solution.

Implemented in ABA_OSIIF.

6.18.4.39 virtual double ABA_LP::_yVal (int *i*) `[protected, pure virtual]`

The pure virtual function *_yVal()* must be defined by the used LP-solver and should return the value of the dual variable of the constraint *i*.

Implemented in ABA_OSIIF.

6.18.4.40 void ABA_LP::addCols (ABA_BUFFER< ABA_COLUMN * > & *newCols*)

Adds columns to the linear program.

If the new number of columns exceeds the maximal number of columns a reallocation is performed.

Parameters:

newCols The new columns that are added.

6.18.4.41 void ABA_LP::addRows (ABA_BUFFER< ABA_ROW * > & *newRows*)

Adds rows to the linear program.

If the new number of rows exceeds the maximal number of rows a reallocation is performed.

Parameters:

newRows The rows that should be added to the linear program.

6.18.4.42 double ABA_LP::barXVal (int *i*) `[inline, virtual]`

Reimplemented in ABA_LPSUB.

Definition at line 793 of file lp.h.

6.18.4.43 ABA_LP::SOLSTAT ABA_LP::barXValStatus () const `[inline]`

Definition at line 830 of file lp.h.

6.18.4.44 ABA_LP::SOLSTAT ABA_LP::basisStatus () const `[inline]`

Definition at line 850 of file lp.h.

6.18.4.45 virtual void ABA_LP::changeLBound (int *i*, double *newLb*) `[virtual]`

Changes the lower bound of a single column.

Parameters:

i The column.

newLb The new lower bound of the column.

Reimplemented in ABA_LPSUB.

6.18.4.46 void ABA_LP::changeRhs (ABA_ARRAY< double > & *newRhs*)

Changes the complete right hand side of the linear program.

Parameters:

newRhs The new right hand side of the rows.

6.18.4.47 virtual void ABA_LP::changeUBound (int *i*, double *newUb*) `[virtual]`

Changes the upper bound of a single column.

Parameters:

i The column.

newUb The new upper bound of the column.

Reimplemented in ABA_LPSUB.

6.18.4.48 void ABA_LP::colRangeCheck (int *i*) const `[protected]`

Terminates the program if there is no column with index *i*.

Parameters:

i The number of a column.

6.18.4.49 void ABA_LP::colRealloc (int *newSize*)

Performs a reallocation of the column space of the linear program.

Parameters:

newSize The new maximal number of columns of the linear program.

Reimplemented in ABA_LPSUB.

6.18.4.50 void ABA_LP::colsNnz (int *nRow*, ABA_ARRAY< ABA_ROW ∗ > & *rows*, ABA_ARRAY< int > & *nnz*) `[protected]`

Computes the number of nonzero elements in each column of a given set of rows.

Parameters:

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.

6.18.4.51 virtual int ABA_LP::getInfeas (int & *infeasRow*, int & *infeasCol*, double $*$ *bInvRow*) `[virtual]`

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.

Returns:

0 On success,
1 otherwise.

Parameters:

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.

Reimplemented in ABA_LPSUB.

6.18.4.52 int ABA_LP::getSimplexIterationLimit (int & *limit*)

Returns:

0 If the iteration limit could be get,
1 otherwise.

Parameters:

limit Stores the iteration limit if the return value is 0.

6.18.4.53 bool ABA_LP::infeasible () const `[inline, virtual]`

Reimplemented in ABA_LPSUB.

Definition at line 860 of file lp.h.

6.18.4.54 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*)

This version of the function *initialize()* performs like its previous version, but also initializes the basis with the arguments:.

Parameters:

lpVarStat An array storing the status of the columns.

slackStat An array storing the status of the slack variables.

Reimplemented in ABA_LPSUB.

6.18.4.55 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*)

Loads the linear program defined by its arguments.

We do not perform the initialization via arguments of a constructor, since for the most frequent application of linear programs within , the solution of the linear programming relaxations in the subproblems, the problem data is preprocessed before it is loaded. Only after the preprocessing in the constructor of the derived class, we can call *initialize()*.

Of course, it would be possible to provide an extra constructor with automatic initialization if required.

Parameters:

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.

Reimplemented in ABA_LPSUB.

6.18.4.56 void ABA_LP::initPostOpt () `[private]`

Resets the optimization status and the availability statuses of the solution.

The function *initPostOpt()* must be called after each modification of the linear program. It resets the optimization status and the availability status of the solution.

6.18.4.57 double ABA_LP::lBound (int *i*) const `[inline]`

Reimplemented in ABA_LPSUB.

Definition at line 748 of file lp.h.

6.18.4.58 virtual void ABA_LP::loadBasis (ABA_ARRAY< ABA_LPVARSTAT::STATUS > & *lpVarStat*, ABA_ARRAY< ABA_SLACKSTAT::STATUS > & *slackStat*) `[virtual]`

Loads a new basis for the linear program.

Parameters:

lpVarStat An array storing the status of the columns.

slackStat An array storing the status of the slack variables.

Reimplemented in ABA_LPSUB.

6.18.4.59 ABA_CPUTIMER∗ ABA_LP::lpSolverTime () `[inline]`

Definition at line 347 of file lp.h.

6.18.4.60 ABA_LPVARSTAT::STATUS ABA_LP::lpVarStat (int *i*) `[inline, virtual]`

Reimplemented in ABA_LPSUB.

Definition at line 866 of file lp.h.

6.18.4.61 int ABA_LP::maxCol () const `[inline]`

Reimplemented in ABA_LPSUB.

Definition at line 730 of file lp.h.

6.18.4.62 int ABA_LP::maxRow () const `[inline]`

Definition at line 720 of file lp.h.

6.18.4.63 int ABA_LP::nCol () const `[inline]`

Reimplemented in ABA_LPSUB.

Definition at line 725 of file lp.h.

6.18.4.64 int ABA_LP::nnz () const `[inline]`

Reimplemented in ABA_LPSUB.

Definition at line 735 of file lp.h.

6.18.4.65 int ABA_LP::nOpt () const `[inline]`

Definition at line 855 of file lp.h.

6.18.4.66 int ABA_LP::nRow () const `[inline]`

Definition at line 715 of file lp.h.

6.18.4.67 double ABA_LP::obj (int *i*) const `[inline]`

Reimplemented in ABA_LPSUB.

Definition at line 740 of file lp.h.

6.18.4.68 const ABA_LP& ABA_LP::operator= (const ABA_LP & *rhs*) `[private]`

6.18.4.69 virtual OPTSTAT ABA_LP::optimize (METHOD *method*) `[virtual]`

Performs the optimization of the linear program.

Returns:
The status of the optimization.

Parameters:
method The method with which the optimization is performed.

Reimplemented in ABA_LPSUB.

6.18.4.70 virtual int ABA_LP::pivotSlackVariableIn (ABA_BUFFER< int > & *rows*) `[virtual]`

Pivots the slack variables stored in the buffer *rows* into the basis.

Returns:
0 All variables could be pivoted in,
1 otherwise.

Parameters:
rows The numbers of the slack variables that should be pivoted in.

6.18.4.71 double ABA_LP::reco (int *i*) `[inline, virtual]`

Reimplemented in ABA_LPSUB.

Definition at line 801 of file lp.h.

6.18.4.72 ABA_LP::SOLSTAT ABA_LP::recoStatus () const [inline]

Definition at line 835 of file lp.h.

6.18.4.73 void ABA_LP::remCols (ABA_BUFFER< int > & *cols*)

Removes columns from the linear program.

Parameters:

cols The numbers of the columns that should be removed.

6.18.4.74 void ABA_LP::remRows (ABA_BUFFER< int > & *ind*)

Removes rows of the linear program.

Parameters:

ind The numbers of the rows that should be removed.

6.18.4.75 double ABA_LP::rhs (int *i*) const [inline]

Definition at line 772 of file lp.h.

6.18.4.76 void ABA_LP::row (int *i*, ABA_ROW & *r*) const [inline]

Definition at line 764 of file lp.h.

6.18.4.77 void ABA_LP::rowRangeCheck (int *r*) const [protected]

Terminates the program if there is no row with index *r*.

Parameters:

r The number of a row of the linear program.

6.18.4.78 void ABA_LP::rowRealloc (int *newSize*)

Performs a reallocation of the row space of the linear program.

Parameters:

newSize The new maximal number of rows of the linear program.

Reimplemented in ABA_LPSUB.

6.18.4.79 void ABA_LP::rows2cols (int *nRow*, ABA_ARRAY< ABA_ROW ∗ > & *rows*, ABA_ARRAY< ABA_SPARVEC ∗ > & *cols*) `[protected]`

Computes the columnwise representation of the row matrix.

Parameters:

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()*.

6.18.4.80 void ABA_LP::sense (const ABA_OPTSENSE & *newSense*) `[inline]`

Definition at line 710 of file lp.h.

6.18.4.81 ABA_OPTSENSE ABA_LP::sense () const `[inline]`

Definition at line 705 of file lp.h.

6.18.4.82 int ABA_LP::setSimplexIterationLimit (int *limit*)

Changes the iteration limit of the Simplex algorithm.

Returns:

0 If the iteration limit could be set,
1 otherwise.

Parameters:

limit The new value of the iteration limit.

6.18.4.83 double ABA_LP::slack (int *c*) `[inline, virtual]`

Definition at line 817 of file lp.h.

6.18.4.84 ABA_SLACKSTAT::STATUS ABA_LP::slackStat (int *i*) `[inline, virtual]`

Definition at line 874 of file lp.h.

6.18.4.85 ABA_LP::SOLSTAT ABA_LP::slackStatus () const `[inline]`

Definition at line 845 of file lp.h.

6.18.4.86 double ABA_LP::uBound (int *i*) const `[inline]`

Reimplemented in ABA_LPSUB.

Definition at line 756 of file lp.h.

6.18.4.87 double ABA_LP::value () const `[inline, virtual]`

Reimplemented in ABA_LPSUB.

Definition at line 780 of file lp.h.

6.18.4.88 int ABA_LP::writeBasisMatrix (const char ∗ *fileName*)

Writes the complete basis of an optimal linear program to a file.

Returns:
0 If a basis is available and could be written,
1 otherwise.

Parameters:
fileName The name of the file the basis is written to.

6.18.4.89 double ABA_LP::xVal (int *i*) `[inline, virtual]`

Reimplemented in ABA_LPSUB.

Definition at line 785 of file lp.h.

6.18.4.90 ABA_LP::SOLSTAT ABA_LP::xValStatus () const `[inline]`

Definition at line 825 of file lp.h.

6.18.4.91 double ABA_LP::yVal (int *c*) `[inline, virtual]`

Definition at line 809 of file lp.h.

6.18.4.92 ABA_LP::SOLSTAT ABA_LP::yValStatus () const `[inline]`

Definition at line 840 of file lp.h.

6.18.5 Friends And Related Function Documentation

6.18.5.1 ostream& operator<< (ostream & *out*, const ABA_LP & *rhs*) `[friend]`

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.

Every ten output columns we perform a line break for better readability. This has also the advantage that LP-solvers with an input function requiring a limited length of a line (e.g., Cplex 255 characters) have a higher chance to read a file generated by this output operator.

Returns:
A reference to the output stream.

Parameters:
out The output stream.

rhs The linear program being output.

6.18.6 Member Data Documentation

6.18.6.1 SOLSTAT ABA_LP::barXValStatus_ `[protected]`

Definition at line 655 of file lp.h.

6.18.6.2 SOLSTAT ABA_LP::basisStatus_ `[protected]`

This member becomes *Available* if the status of the variables and the slack variables of the optimal solution can be accessed with the functions *lpVarStat()* and *slackStat()*, otherwise it has the value *Missing*.

Definition at line 683 of file lp.h.

6.18.6.3 ABA_CPUTIMER ABA_LP::lpSolverTime_ `[protected]`

Definition at line 688 of file lp.h.

6.18.6.4 ABA_MASTER∗ ABA_LP::master_ `[protected]`

A pointer to the corresponding master of the optimization.

Definition at line 644 of file lp.h.

6.18.6.5 int ABA_LP::nOpt_ `[protected]`

The number of optimizations of the linear program.

Definition at line 687 of file lp.h.

6.18.6.6 OPTSTAT ABA_LP::optStat_ `[protected]`

The status of the linear program.

Definition at line 648 of file lp.h.

6.18.6.7 SOLSTAT ABA_LP::recoStatus_ `[protected]`

This member becomes *Available* if the reduced costs of the optimal solution can be accessed with the function *reco()*, otherwise it has the value *Missing*.

Definition at line 668 of file lp.h.

6.18.6.8 SOLSTAT ABA_LP::slackStatus_ `[protected]`

This member becomes *Available* if the values of the slack variables of the optimal solution can be accessed with the function *slack()*, otherwise it has the value *Missing*.

Definition at line 675 of file lp.h.

6.18.6.9 SOLSTAT ABA_LP::xValStatus_ `[protected]`

This member becomes *Available* if the x -values of the optimal solution can be accessed with the function *xVal()*, otherwise it has the value *Missing*.

Definition at line 654 of file lp.h.

6.18.6.10 SOLSTAT ABA_LP::yValStatus_ `[protected]`

This member becomes *Available* if the values of the dual variables of the optimal solution can be accessed with the function *yVal()*, otherwise it has the value *Missing/*.

Definition at line 662 of file lp.h.

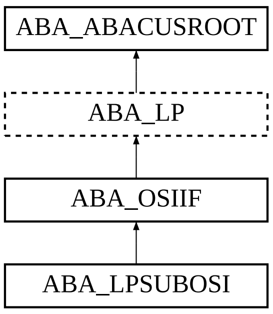
The documentation for this class was generated from the following file:

- Include/abacus/lp.h

6.19 ABA_OSIIF Class Reference

`#include <osiif.h>`

Inheritance diagram for ABA_OSIIF::



Public Types

- enum SOLVERTYPE { Exact, Approx }

Public Member Functions

- ABA_OSIIF (ABA_MASTER ∗master)

  This constructor does not initialize the problem data of the linear program. It must be loaded later with the function initialize().

- ABA_OSIIF (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_OSIIF ()

  The destructor.

- SOLVERTYPE currentSolverType ()
- OsiSolverInterface ∗ osiLP ()

Private Member Functions

- void freeDouble (const double ∗&)
- void freeDouble (double ∗&)
- void freeInt (int ∗&)
- void freeChar (char ∗&)
- void freeChar (const char ∗&)
- void freeStatus (CoinWarmStartBasis::Status ∗&)
- virtual 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)

  Implements the corresponding pure virtual function of the base class LP *and loads the linear program defined by the following arguments to the solver.*

- virtual void _loadBasis (ABA_ARRAY< ABA_LPVARSTAT::STATUS > &lpVarStat, ABA_ARRAY< ABA_SLACKSTAT::STATUS > &slackStat)
- virtual ABA_OPTSENSE _sense () const
- virtual void _sense (const ABA_OPTSENSE &newSense)

  This version of the function _sense() changes the sense of the optimization.

- virtual int _nRow () const
- virtual int _maxRow () const
- virtual int _nCol () const
- virtual int _maxCol () const
- virtual double _obj (int i) const
- virtual double _lBound (int i) const
- virtual double _uBound (int i) const
- virtual double _rhs (int i) const
- virtual void _row (int i, ABA_ROW &r) const
- virtual int _nnz () const

    Returns the number of nonzero elements in the constraint matrix (not including the right hand side).

- virtual OPTSTAT _primalSimplex ()
- virtual OPTSTAT _dualSimplex ()
- virtual OPTSTAT _barrier (bool doCrossover)
- virtual OPTSTAT _approx ()
- virtual double _value () const
- virtual double _xVal (int i)
- virtual double _barXVal (int i)
- virtual double _reco (int i)
- virtual double _slack (int i)
- virtual double _yVal (int i)
- virtual ABA_LPVARSTAT::STATUS _lpVarStat (int i)
- virtual ABA_SLACKSTAT::STATUS _slackStat (int i)
- virtual int _getInfeas (int &infeasRow, int &infeasCol, double ∗bInvRow)

    Can be called if the last linear program has been solved with the dual simplex method and is infeasible. This function is currently not supported by the interface.

- virtual void _remRows (ABA_BUFFER< int > &ind)
- virtual void _addRows (ABA_BUFFER< ABA_ROW ∗ > &newRows)
- virtual void _remCols (ABA_BUFFER< int > &vars)
- virtual void _addCols (ABA_BUFFER< ABA_COLUMN ∗ > &newVars)
- virtual 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)

    Pivots the slack variables stored in the buffer rows *into the basis. This function defines the pure virtual function of the base class* LP*. This function is currently not supported by the interface.*

- void getSol ()

    Extracts the solution, i.e., the value, the status, the values of the variables, slack variables, and dual variables, the reduced costs, and the statuses of the variables and slack variables form the internal solver data structure.

- char csense2osi (ABA_CSENSE ∗sense) const

    Converts the ABACUS representation of the row sense to the Osi representation.

- ABA_CSENSE::SENSE osi2csense (char sense) const

    Converts the OSI representation of the row sense to the ABACUS representation.

- CoinWarmStartBasis::Status lpVarStat2osi (ABA_LPVARSTAT::STATUS stat) const

  Converts the ABACUS variable status to OSI format.

- ABA_LPVARSTAT::STATUS osi2lpVarStat (CoinWarmStartBasis::Status stat) const

  Converts the OSI variable status to ABACUS format.

- CoinWarmStartBasis::Status slackStat2osi (ABA_SLACKSTAT::STATUS stat) const

  Converts the ABACUS slack status to OSI format.

- ABA_SLACKSTAT::STATUS osi2slackStat (CoinWarmStartBasis::Status stat) const

  Converts the OSI slack status to ABACUS format.

- OsiSolverInterface ∗ getDefaultInterface ()

  Allocates an Open Solver Interface of type defaultOsiSolver.

- OsiSolverInterface ∗ switchInterfaces (SOLVERTYPE newMethod)

  Switches between exact and approximate solvers.

- void loadDummyRow (OsiSolverInterface ∗s2, const double ∗lbounds, const double ∗ubounds, const double ∗objectives)

  Initializes the problem with a dummy row To be used with CPLEX if there are no rows.

- void _rowRealloc (int newSize)
- void _colRealloc (int newSize)
- virtual int _setSimplexIterationLimit (int limit)
- virtual int _getSimplexIterationLimit (int &limit)
- ABA_OSIIF (const ABA_OSIIF &rhs)
- const ABA_OSIIF & operator= (const ABA_OSIIF &rhs)
- void convertSenseToBound (double inf, const char sense, const double right, const double range, double &lower, double &upper) const

Private Attributes

- OsiSolverInterface ∗ osiLP_
- ABA_LPMASTEROSI ∗ lpMasterOsi_
- double value_
- const double ∗ xVal_

  An array storing the values of the variables after the linear program has been optimized.

- const double ∗ barXVal_
- const double ∗ reco_

  An array storing the values of the reduced costs after the linear program has been optimized.

- const double ∗ yVal_

  An array storing the values of the dual variables after the linear program has been optimized.

- const char ∗ cStat_

  An array storing the statuses of the variables after the linear program has been optimized.

- int numCols_

  The number of columns currently used in the LP.

- int numRows_

  The number of rows currently used in the LP.

- const char ∗ rStat_

  An array storing the statuses of the slack variables after the linear program has been optimized.

- const double ∗ rhs_

  An array storing the right hand sides of the linear program.

- const double ∗ rowactivity_

  An array storing the row activity of the linear program.

- const char ∗ rowsense_

  An array storing the row senses of the linear program.

- const double ∗ colupper_

  An array storing the column upper bounds of the linear program.

- const double ∗ collower_

  An array storing the column lower bounds of the linear program.

- const double ∗ objcoeff_

  An array storing the objective function coefficients of the linear program.

- CoinWarmStartBasis ∗ ws_

  A warm start object storing information about a basis of the linear program.

- SOLVERTYPE currentSolverType_

  The type of the current solver interface.

6.19.1 Member Enumeration Documentation

6.19.1.1 enum ABA_OSIIF::SOLVERTYPE

The enumeration of possible solver types

Enumeration values:

Exact

Approx

Definition at line 85 of file osiif.h.

6.19.2 Constructor & Destructor Documentation

6.19.2.1 ABA_OSIIF::ABA_OSIIF (ABA_MASTER $*$ *master*)

This constructor does not initialize the problem data of the linear program. It must be loaded later with the function *initialize()*.

Parameters:

master A pointer to the corresponding master of the optimization.

6.19.2.2 ABA_OSIIF::ABA_OSIIF (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*)

A constructor with initialization.

Parameters:

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.

6.19.2.3 virtual ABA_OSIIF::$\sim$ABA_OSIIF () `[virtual]`

The destructor.

6.19.2.4 ABA_OSIIF::ABA_OSIIF (const ABA_OSIIF & *rhs*) `[private]`

6.19.3 Member Function Documentation

6.19.3.1 virtual void ABA_OSIIF::_addCols (ABA_BUFFER< ABA_COLUMN $*$ > & *newVars*) `[private, virtual]`

Adds the columns *newCols* to the linear program.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.2 virtual void ABA_OSIIF::_addRows (ABA_BUFFER< ABA_ROW $*$ > & *newRows*) `[private, virtual]`

Adds the *rows* to the linear program.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.3 virtual OPTSTAT ABA_OSIIF::_approx () `[private, virtual]`

Calls an approximate method.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.4 virtual OPTSTAT ABA_OSIIF::_barrier (bool *doCrossover*) `[private, virtual]`

Calls the barrier method.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.5 virtual double ABA_OSIIF::_barXVal (int *i*) `[private, virtual]`

Returns the value of the column *i*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.6 virtual void ABA_OSIIF::_changeLBound (int *i*, double *newLb*) `[private, virtual]`

Sets the lower bound of column *i* to *newLb*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.7 virtual void ABA_OSIIF::_changeRhs (ABA_ARRAY< double > & *newRhs*) `[private, virtual]`

Sets the right hand side of the linear program to *newRhs*.

This array must have at least length of the number of rows. This function implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.8 virtual void ABA_OSIIF::_changeUBound (int *i*, double *newUb*) `[private, virtual]`

Sets the upper bound of column *i* to *newLb*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.9 void ABA_OSIIF::_colRealloc (int *newSize*) `[private, virtual]`

Reallocates the internal memory such that *newSize* columns can be stored. This function is obsolete, as memory management is completely handled by Osi.

It implements the corresponding pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.10 virtual OPTSTAT ABA_OSIIF::_dualSimplex () `[private, virtual]`

Calls the dual simplex method.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.11 virtual int ABA_OSIIF::_getInfeas (int & *infeasRow*, int & *infeasCol*, double $*$ *bInvRow*) `[private, virtual]`

Can be called if the last linear program has been solved with the dual simplex method and is infeasible. This function is currently not supported by the interface.

In this case it computes the infeasible basic variable or constraint and the corresponding row *nInvRow* of the basis inverse. Either *infeasRow* or *infeasCol* is nonnegative. Then this number refers to an infeasible variable or slack variable, respectively. The function returns 0 if it is successful, 1 otherwise.

Currently this featureis not supported by the Open Solver Interface, therefore a call to this function always returns an error status.

> It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.12 virtual int ABA_OSIIF::_getSimplexIterationLimit (int & *limit*) `[private, virtual]`

Defines a pure virtual function of the base class *LP*.

Returns:

0 If the iteration limit could be retrieved,
1 otherwise.

Parameters:

limit Stores the iteration limit if the return value is 0.

Implements ABA_LP.

6.19.3.13 virtual void ABA_OSIIF::_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*) `[private, virtual]`

Implements the corresponding pure virtual function of the base class *LP* and loads the linear program defined by the following arguments to the solver.

Parameters:

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.

Implements ABA_LP.

6.19.3.14 virtual double ABA_OSIIF::_lBound (int *i*) const `[private, virtual]`

Returns the lower bound of column *i*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.15 virtual void ABA_OSIIF::_loadBasis (ABA_ARRAY< ABA_LPVARSTAT::STATUS > & *lpVarStat*, ABA_ARRAY< ABA_SLACKSTAT::STATUS > & *slackStat*) `[private, virtual]`

Loads a basis to the solver

Parameters:

lpVarStat An array storing the status of the columns.

lpVarStat An array storing the status of the slack variables.

Implements ABA_LP.

6.19.3.16 virtual ABA_LPVARSTAT::STATUS ABA_OSIIF::_lpVarStat (int *i*) `[private, virtual]`

Returns the status of the column *i*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.17 virtual int ABA_OSIIF::_maxCol () const `[private, virtual]`

Returns the maximal number of columns of the linear program.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.18 virtual int ABA_OSIIF::_maxRow () const `[private, virtual]`

Returns the maximal number of rows of the linear program.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.19 virtual int ABA_OSIIF::_nCol () const `[private, virtual]`

Returns the number of columns of the linear program.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.20 virtual int ABA_OSIIF::_nnz () const `[private, virtual]`

Returns the number of nonzero elements in the constraint matrix (not including the right hand side).

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.21 virtual int ABA_OSIIF::_nRow () const `[private, virtual]`

Returns the number of rows of the linear program.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.22 virtual double ABA_OSIIF::_obj (int *i*) const `[private, virtual]`

Returns the objective function coefficient of column *i*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.23 virtual int ABA_OSIIF::_pivotSlackVariableIn (ABA_BUFFER< int > & *rows*) `[private, virtual]`

Pivots the slack variables stored in the buffer *rows* into the basis. This function defines the pure virtual function of the base class *LP*. This function is currently not supported by the interface.

Returns:

0 All variables could be pivoted in,
1 otherwise.

Parameters:

rows The numbers of the slack variables that should be pivoted in.

Implements ABA_LP.

6.19.3.24 virtual OPTSTAT ABA_OSIIF::_primalSimplex () `[private, virtual]`

Calls the primal simplex method.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.25 virtual double ABA_OSIIF::_reco (int *i*) `[private, virtual]`

Returns the reduced cost of the column *i*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.26 virtual void ABA_OSIIF::_remCols (ABA_BUFFER< int > & *vars*) `[private, virtual]`

Removes the columns listed in *vars*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.27 virtual void ABA_OSIIF::_remRows (ABA_BUFFER< int > & *ind*) `[private, virtual]`

Removes the rows listed in *ind*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.28 virtual double ABA_OSIIF::_rhs (int *i*) const `[private, virtual]`

Returns the right hand side of row *i*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.29 virtual void ABA_OSIIF::_row (int *i*, ABA_ROW & *r*) const `[private, virtual]`

Stores a copy of row *i* in *r*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.30 void ABA_OSIIF::_rowRealloc (int *newSize*) `[private, virtual]`

Reallocates the internal memory such that newSize rows can be stored. This function is obsolete, as memory management is completely handled by Osi.

It implements the corresponding pure virtual function of the base class *LP*. If a reallocation is performed in the base class *LP*, we reinitialize the internal data structure. Actually this reinitialization is redundant since it would be performed automatically if *addRows()* or *addCols()* fail. However, to be consistent, and if a reallocation is performed to decrease the size of the arrays we call *reinitialize()*.

Implements ABA_LP.

6.19.3.31 virtual void ABA_OSIIF::_sense (const ABA_OPTSENSE & *newSense*) `[private, virtual]`

This version of the function *_sense()* changes the sense of the optimization.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.32 virtual ABA_OPTSENSE ABA_OSIIF::_sense () const `[private, virtual]`

Returns the sense of the optimization.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.33 virtual int ABA_OSIIF::_setSimplexIterationLimit (int *limit*) `[private, virtual]`

Changes the iteration limit of the Simplex algorithm.

This function defines a pure virtual function of the base class *LP*.

Returns:

0 If the iteration limit could be set,
1 otherwise.

Parameters:

limit The new value of the iteration limit.

Implements ABA_LP.

6.19.3.34 virtual double ABA_OSIIF::_slack (int *i*) `[private, virtual]`

Returns the value of the slack column of the row *i*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.35 virtual ABA_SLACKSTAT::STATUS ABA_OSIIF::_slackStat (int *i*) `[private, virtual]`

Returns the status of the slack column *i*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.36 virtual double ABA_OSIIF::_uBound (int *i*) const `[private, virtual]`

Returns the upper bound of column *i*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.37 virtual double ABA_OSIIF::_value () const `[private, virtual]`

Returns the optimum value of the linear program.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.38 virtual double ABA_OSIIF::_xVal (int *i*) `[private, virtual]`

Returns the value of the column *i*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.39 virtual double ABA_OSIIF::_yVal (int *i*) `[private, virtual]`

Returns the value of the dual column of the row *i*.

It implements the pure virtual function of the base class *LP*.

Implements ABA_LP.

6.19.3.40 void ABA_OSIIF::convertSenseToBound (double *inf*, const char *sense*, const double *right*, const double *range*, double & *lower*, double & *upper*) const `[inline, private]`

Definition at line 530 of file osiif.h.

6.19.3.41 char ABA_OSIIF::csense2osi (ABA_CSENSE $*$ *sense*) const `[private]`

Converts the ABACUS representation of the row sense to the Osi representation.

6.19.3.42 SOLVERTYPE ABA_OSIIF::currentSolverType () `[inline]`

Definition at line 87 of file osiif.h.

6.19.3.43 void ABA_OSIIF::freeChar (const char ∗&) `[private]`

6.19.3.44 void ABA_OSIIF::freeChar (char ∗&) `[private]`

6.19.3.45 void ABA_OSIIF::freeDouble (double ∗&) `[private]`

6.19.3.46 void ABA_OSIIF::freeDouble (const double ∗&) `[private]`

6.19.3.47 void ABA_OSIIF::freeInt (int ∗&) `[private]`

6.19.3.48 void ABA_OSIIF::freeStatus (CoinWarmStartBasis::Status ∗&) `[private]`

6.19.3.49 OsiSolverInterface∗ ABA_OSIIF::getDefaultInterface () `[private]`

Allocates an Open Solver Interface of type defaultOsiSolver.

6.19.3.50 void ABA_OSIIF::getSol () `[private]`

Extracts the solution, i.e., the value, the status, the values of the variables, slack variables, and dual variables, the reduced costs, and the statuses of the variables and slack variables form the internal solver data structure.

6.19.3.51 void ABA_OSIIF::loadDummyRow (OsiSolverInterface ∗ *s2*, const double ∗ *lbounds*, const double ∗ *ubounds*, const double ∗ *objectives*) `[private]`

Initializes the problem with a dummy row To be used with CPLEX if there are no rows.

6.19.3.52 CoinWarmStartBasis::Status ABA_OSIIF::lpVarStat2osi (ABA_LPVARSTAT::STATUS *stat*) const [private]

Converts the ABACUS variable status to OSI format.

6.19.3.53 const ABA_OSIIF& ABA_OSIIF::operator= (const ABA_OSIIF & *rhs*) [private]

6.19.3.54 ABA_CSENSE::SENSE ABA_OSIIF::osi2csense (char *sense*) const [private]

Converts the OSI representation of the row sense to the ABACUS representation.

6.19.3.55 ABA_LPVARSTAT::STATUS ABA_OSIIF::osi2lpVarStat (CoinWarmStartBasis::Status *stat*) const [private]

Converts the OSI variable status to ABACUS format.

6.19.3.56 ABA_SLACKSTAT::STATUS ABA_OSIIF::osi2slackStat (CoinWarmStartBasis::Status *stat*) const [private]

Converts the OSI slack status to ABACUS format.

6.19.3.57 OsiSolverInterface ∗ ABA_OSIIF::osiLP () [inline]

Definition at line 559 of file osiif.h.

6.19.3.58 CoinWarmStartBasis::Status ABA_OSIIF::slackStat2osi (ABA_SLACKSTAT::STATUS *stat*) const [private]

Converts the ABACUS slack status to OSI format.

6.19.3.59 OsiSolverInterface∗ ABA_OSIIF::switchInterfaces (SOLVERTYPE *newMethod*) [private]

Switches between exact and approximate solvers.

6.19.4 Member Data Documentation

6.19.4.1 const double∗ ABA_OSIIF::barXVal_ [private]

Definition at line 464 of file osiif.h.

6.19.4.2 const double∗ ABA_OSIIF::collower_ [private]

An array storing the column lower bounds of the linear program.

Definition at line 512 of file osiif.h.

6.19.4.3 const double∗ ABA_OSIIF::colupper_ [private]

An array storing the column upper bounds of the linear program.

Definition at line 508 of file osiif.h.

6.19.4.4 const char∗ ABA_OSIIF::cStat_ [private]

An array storing the statuses of the variables after the linear program has been optimized.

Definition at line 479 of file osiif.h.

6.19.4.5 SOLVERTYPE ABA_OSIIF::currentSolverType_ [private]

The type of the current solver interface.

Definition at line 524 of file osiif.h.

6.19.4.6 ABA_LPMASTEROSI∗ ABA_OSIIF::lpMasterOsi_ [private]

Definition at line 454 of file osiif.h.

6.19.4.7 int ABA_OSIIF::numCols_ [private]

The number of columns currently used in the LP.

Definition at line 483 of file osiif.h.

6.19.4.8 int ABA_OSIIF::numRows_ [private]

The number of rows currently used in the LP.

Definition at line 487 of file osiif.h.

6.19.4.9 const double∗ ABA_OSIIF::objcoeff_ [private]

An array storing the objective function coefficients of the linear program.

Definition at line 516 of file osiif.h.

6.19.4.10 OsiSolverInterface∗ ABA_OSIIF::osiLP_ [private]

Definition at line 100 of file osiif.h.

6.19.4.11 const double∗ ABA_OSIIF::reco_ [private]

An array storing the values of the reduced costs after the linear program has been optimized.

Definition at line 469 of file osiif.h.

6.19.4.12 const double∗ ABA_OSIIF::rhs_ [private]

An array storing the right hand sides of the linear program.

Definition at line 496 of file osiif.h.

6.19.4.13 const double∗ ABA_OSIIF::rowactivity_ [private]

An array storing the row activity of the linear program.

Definition at line 500 of file osiif.h.

6.19.4.14 const char∗ ABA_OSIIF::rowsense_ [private]

An array storing the row senses of the linear program.

Definition at line 504 of file osiif.h.

6.19.4.15 const char∗ ABA_OSIIF::rStat_ [private]

An array storing the statuses of the slack variables after the linear program has been optimized.

Definition at line 492 of file osiif.h.

6.19.4.16 double ABA_OSIIF::value_ [private]

The value of the optimal solution.

Definition at line 458 of file osiif.h.

6.19.4.17 CoinWarmStartBasis∗ ABA_OSIIF::ws_ [private]

A warm start object storing information about a basis of the linear program.

Definition at line 520 of file osiif.h.

6.19.4.18 const double∗ ABA_OSIIF::xVal_ [private]

An array storing the values of the variables after the linear program has been optimized.

Definition at line 463 of file osiif.h.

6.19.4.19 const double∗ ABA_OSIIF::yVal_ [private]

An array storing the values of the dual variables after the linear program has been optimized.

Definition at line 474 of file osiif.h.

The documentation for this class was generated from the following file:

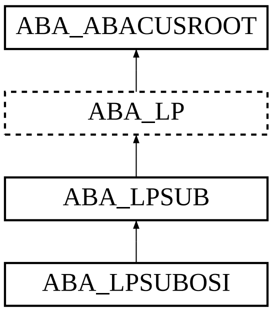
- Include/abacus/osiif.h

6.20 ABA_LPSUB Class Reference

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*.

`#include <lpsub.h>`

Inheritance diagram for ABA_LPSUB::



Public Member Functions

- ABA_LPSUB (ABA_MASTER ∗master, const ABA_SUB ∗sub)
- virtual ∼ABA_LPSUB ()

  The destructor deletes the components of infeasCons_ *since they might have been allocated in the constructor and ABA_SUB::initializeLp() deletes after having tried to add variables restoring feasibility immediately ABA_LPSUB. Afterwards the constructor of ABA_LPSUB is called again.*

- const ABA_SUB ∗ sub () const
- int trueNCol () const
- int trueNnz () const
- double lBound (int i) const

  We have to redefine the function lBound(i) *since variables may have been eliminated.*

- double uBound (int i) const

  We have to redefine the function uBound(i) *since variables may have been eliminated.*

- virtual double value () const

Since variables might be eliminated we have to add to the solution value of the LP-solver the objective function part of the eliminated variables, to get the right value of value()*.*

- virtual double xVal (int i)

  We have to redefine the function xVal(i) *since variables may have been eliminated.*

- virtual double barXVal (int i)

  We have to redefine the function barXVal(i) *since variables may have been eliminated.*

- virtual double reco (int i)
- virtual ABA_LPVARSTAT::STATUS lpVarStat (int i)
- virtual int getInfeas (int &infeasCon, int &infeasVar, double ∗bInvRow)

  Is called if the last linear program has been solved with the dual simplex method and is infeasible.

- 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 Member Functions

- virtual void 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.*

Private Member Functions

- virtual OPTSTAT optimize (METHOD method)
- virtual void removeCons (ABA_BUFFER< int > &ind)
- virtual void removeVars (ABA_BUFFER< int > &vars)
- virtual void addCons (ABA_BUFFER< ABA_CONSTRAINT ∗ > &newCons)
- virtual void addVars (ABA_BUFFER< ABA_VARIABLE ∗ > &vars, ABA_BUFFER< ABA_FSVARSTAT ∗ > &fsVarStat, ABA_BUFFER< double > &lb, ABA_BUFFER< double > &ub)
- virtual void changeLBound (int i, double newLb)
- virtual void changeUBound (int i, double newUb)
- virtual void varRealloc (int newSize)
- virtual void conRealloc (int newSize)
- void constraint2row (ABA_BUFFER< ABA_CONSTRAINT ∗ > &newCons, ABA_BUFFER< ABA_ROW ∗ > &newRows)
- bool eliminable (int i) const
- bool eliminated (int i) const

  Returns true *if the variable* i *is actually eliminated from the* LP*.*

- virtual double elimVal (int i) const
- virtual double elimVal (ABA_FSVARSTAT ∗stat, double lb, double ub) const
- 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)

  This version of the function initialize() *performs like its previous version, but also initializes the basis with the arguments:.*

- int nCol () const
- int maxCol () const
- int nnz () const
- double obj (int i) const
- void rowRealloc (int newSize)
- void colRealloc (int newSize)
- ABA_LPSUB (const ABA_LPSUB &rhs)
- const ABA_LPSUB & operator= (const ABA_LPSUB &rhs)

Private Attributes

- const ABA_SUB ∗ sub_
- ABA_ARRAY< int > orig2lp_

  After the elimination of variables the internal variables are again numbered consecutively starting with 0. orig2lp_-*[i] is the internal number of the variable* i*. This is* -1 *if the variable is eliminated.*

- ABA_ARRAY< int > lp2orig_
- ABA_BUFFER< ABA_INFEASCON ∗ > infeasCons_
- double valueAdd_

  The constant which has been added to the objective function value due to the elimination of variables.

- int nOrigVar_

Friends

- class ABA_SUB
- class ABA_SETBRANCHRULE
- class ABA_BOUNDBRANCHRULE
- class ABA_VALBRANCHRULE
- class ABA_CONBRANCHRULE
- class COPBRANCHRULE

6.20.1 Detailed Description

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*.

Definition at line 57 of file lpsub.h.

6.20.2 Constructor & Destructor Documentation

6.20.2.1 ABA_LPSUB::ABA_LPSUB (ABA_MASTER $*$ *master*, const ABA_SUB $*$ *sub*)

The constructor.

Parameters:

master A pointer to the corresponding master of the optimization.

sub The subproblem of which the LP-relaxation is solved.

6.20.2.2 virtual ABA_LPSUB::∼ABA_LPSUB () `[virtual]`

The destructor deletes the components of *infeasCons_* since they might have been allocated in the constructor and ABA_SUB::initializeLp() deletes after having tried to add variables restoring feasibility immediately ABA_-LPSUB. Afterwards the constructor of ABA_LPSUB is called again.

6.20.2.3 ABA_LPSUB::ABA_LPSUB (const ABA_LPSUB & *rhs*) `[private]`

6.20.3 Member Function Documentation

6.20.3.1 virtual void ABA_LPSUB::addCons (ABA_BUFFER$<$ ABA_CONSTRAINT $*$ $>$ & *newCons*) `[private, virtual]`

Adds the constraints *newCons* to the linear program.

6.20.3.2 virtual void ABA_LPSUB::addVars (ABA_BUFFER$<$ ABA_VARIABLE $*$ $>$ & *vars*, ABA_BUFFER$<$ ABA_FSVARSTAT $*$ $>$ & *fsVarStat*, ABA_BUFFER$<$ double $>$ & *lb*, ABA_BUFFER$<$ double $>$ & *ub*) `[private, virtual]`

Parameters:

vars The new variables which are added to the linear program.

fsVarstat The status of fixing/setting of the new variables.

lb The lower bounds of the new variables.

ub The upper bounds of the new variables.

6.20.3.3 virtual double ABA_LPSUB::barXVal (int *i*) `[virtual]`

We have to redefine the function *barXVal(i)* since variables may have been eliminated.

Returns:

The x -value of variable *i* after the solution of the linear program before crossing over to a basic solution.

Reimplemented from ABA_LP.

6.20.3.4 virtual void ABA_LPSUB::changeLBound (int *i*, double *newLb*) `[private, virtual]`

Sets the lower bound of variable *i* to *newLb*.

It is not allowed to change the lower bound of an eliminated variable. This will cause a run-time error.

Reimplemented from ABA_LP.

6.20.3.5 virtual void ABA_LPSUB::changeUBound (int *i*, double *newUb*) `[private, virtual]`

Sets the upper bound of variable *i* to *newUb*.

It is not allowed to change the upper bound of an eliminated variable. This will cause a run-time error.

Reimplemented from ABA_LP.

6.20.3.6 void ABA_LPSUB::colRealloc (int *newSize*) `[private]`

Performs a reallocation of the column space of the linear program.

Parameters:

newSize The new maximal number of columns of the linear program.

Reimplemented from ABA_LP.

6.20.3.7 virtual void ABA_LPSUB::conRealloc (int *newSize*) `[private, virtual]`

Sets the maximal number of constraints to *newSize*.

6.20.3.8 void ABA_LPSUB::constraint2row (ABA_BUFFER< ABA_CONSTRAINT ∗ > & *newCons*, ABA_BUFFER< ABA_ROW ∗ > & *newRows*) `[private]`

Generates the row format of the constraint *cons* and stores it in *rows*.

6.20.3.9 bool ABA_LPSUB::eliminable (int *i*) const `[private]`

Returns *true* if the function can be eliminated.

This function may be only applied to variables which are fixed or set! It is sufficient for turning off any variable elimination to return always *false* by this function.

6.20.3.10 bool ABA_LPSUB::eliminated (int *i*) const `[inline, private]`

Returns *true* if the variable *i* is actually eliminated from the *LP*.

This function can give different results than the function *eliminate(i)* since the condition to eliminate a variable might have become *true* after the *LP* has been set up.

Definition at line 362 of file lpsub.h.

6.20.3.11 virtual double ABA_LPSUB::elimVal (ABA_FSVARSTAT $*$ *stat*, double *lb*, double *ub*) const `[private, virtual]`

Returns the value a variable is fixed or set to.

Parameters:

- ***fsVarStat*** A pointer to the status of the variable.
- ***lb*** The lower bound of the variable.
- ***ub*** The upper bound of the variable.

6.20.3.12 virtual double ABA_LPSUB::elimVal (int *i*) const `[private, virtual]`

Returns the value the variable *i* to which it is fixed or set to.

The value of an eliminated variable is defined by the bound to which it is fixed or set. There is no reason to distinguish between *sub_* and *master_* in the *switch* statement, since both values should be equal.

6.20.3.13 virtual int ABA_LPSUB::getInfeas (int & *infeasCon*, int & *infeasVar*, double $*$ *bInvRow*) `[virtual]`

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.

Returns:

0 If no error occurs,
1 otherwise.

Parameters:

- ***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.

Reimplemented from ABA_LP.

6.20.3.14 ABA_BUFFER$<$ ABA_INFEASCON $* > *$ ABA_LPSUB::infeasCon () `[inline]`

return A pointer to the buffer holding the infeasible constraints.

Definition at line 383 of file lpsub.h.

6.20.3.15 virtual bool ABA_LPSUB::infeasible () const `[virtual]`

Returns:

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.

Reimplemented from ABA_LP.

6.20.3.16 void ABA_LPSUB::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*) `[private]`

This version of the function *initialize()* performs like its previous version, but also initializes the basis with the arguments:.

Parameters:

lpVarStat An array storing the status of the columns.

slackStat An array storing the status of the slack variables.

Reimplemented from ABA_LP.

6.20.3.17 void ABA_LPSUB::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*) `[private]`

Loads the linear program defined by its arguments.

We do not perform the initialization via arguments of a constructor, since for the most frequent application of linear programs within , the solution of the linear programming relaxations in the subproblems, the problem data is preprocessed before it is loaded. Only after the preprocessing in the constructor of the derived class, we can call *initialize()*.

Of course, it would be possible to provide an extra constructor with automatic initialization if required.

Parameters:

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.

Reimplemented from ABA_LP.

6.20.3.18 virtual void ABA_LPSUB::initialize () `[protected, virtual]`

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.

6.20.3.19 double ABA_LPSUB::lBound (int *i*) const

We have to redefine the function *lBound(i)* since variables may have been eliminated.

Returns:

The lower bound of variable i. If a variable is eliminated, we return the value the eliminated variable is fixed or set to.

Parameters:

i The number of a variable.

Reimplemented from ABA_LP.

6.20.3.20 virtual void ABA_LPSUB::loadBasis (ABA_ARRAY< ABA_LPVARSTAT::STATUS > & *lpVarStat*, ABA_ARRAY< ABA_SLACKSTAT::STATUS > & *slackStat*) `[virtual]`

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.

Parameters:

lpVarStat An array storing the status of the columns.

slackStat An array storing the status of the slack variables.

Reimplemented from ABA_LP.

6.20.3.21 virtual ABA_LPVARSTAT::STATUS ABA_LPSUB::lpVarStat (int *i*) `[virtual]`

Returns:

The status of the variable in the linear program. If the variable i is eliminated, then ABA_LPVARSTAT::Eliminated is returned.

Reimplemented from ABA_LP.

6.20.3.22 int ABA_LPSUB::maxCol () const `[private]`

Reimplemented from ABA_LP.

6.20.3.23 int ABA_LPSUB::nCol () const `[private]`

Reimplemented from ABA_LP.

6.20.3.24 int ABA_LPSUB::nnz () const `[private]`

Reimplemented from ABA_LP.

6.20.3.25 double ABA_LPSUB::obj (int *i*) const `[private]`

Reimplemented from ABA_LP.

6.20.3.26 const ABA_LPSUB& ABA_LPSUB::operator= (const ABA_LPSUB & *rhs*) `[private]`

6.20.3.27 virtual OPTSTAT ABA_LPSUB::optimize (METHOD *method*) `[private, virtual]`

Performs the optimization of the linear program with method *method*.

This function redefines a virtual function of the base class *LP*.

> We have to reimplement *optimize()* since there might be infeasible constraints. If a linear program turns out to be infeasible but has not been solved with the dual simplex method we solve it again to find a dual feasible basis which can be used to determine inactive variables restoring feasibility. Before the optimization can be performed the infeasible constraints must be removed with the function *_initMakeFeas()*, then the *LP* should be deleted and reconstructed. This is done by the function *solveLp()* in the cutting plane algorithm of the class ABA_SUB.

Reimplemented from ABA_LP.

6.20.3.28 virtual double ABA_LPSUB::reco (int *i*) `[virtual]`

We define the reduced costs of eliminated variables as 0.

Returns:

The reduced cost of variable *i*.

Reimplemented from ABA_LP.

6.20.3.29 virtual void ABA_LPSUB::removeCons (ABA_BUFFER< int > & *ind*) `[private, virtual]`

Removes all constraints listed in the buffer *ind* from the linear program.

6.20.3.30 virtual void ABA_LPSUB::removeVars (ABA_BUFFER< int > & *vars*) `[private, virtual]`

Removes the variables with names given in *vars* from the linear program.

6.20.3.31 void ABA_LPSUB::rowRealloc (int *newSize*) `[private]`

Performs a reallocation of the row space of the linear program.

Parameters:

newSize The new maximal number of rows of the linear program.

Reimplemented from ABA_LP.

6.20.3.32 const ABA_SUB ∗ ABA_LPSUB::sub () const `[inline]`

Definition at line 357 of file lpsub.h.

6.20.3.33 int ABA_LPSUB::trueNCol () const `[inline]`

Returns:

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.

Definition at line 368 of file lpsub.h.

6.20.3.34 int ABA_LPSUB::trueNnz () const `[inline]`

Returns:

The number of nonzeros which are currently present in the constraint matrix of the LP-solver.

Definition at line 373 of file lpsub.h.

6.20.3.35 double ABA_LPSUB::uBound (int *i*) const

We have to redefine the function *uBound(i)* since variables may have been eliminated.

Returns:

The upper bound of variable *i*. If a variable is eliminated, we return the value the eliminated variable is fixed or set to.

Parameters:

i The number of a variable.$*$

Reimplemented from ABA_LP.

6.20.3.36 double ABA_LPSUB::value () const `[inline, virtual]`

Since variables might be eliminated we have to add to the solution value of the LP-solver the objective function part of the eliminated variables, to get the right value of *value()*.

Returns:

The objective function value of the linear program.

Reimplemented from ABA_LP.

Definition at line 378 of file lpsub.h.

6.20.3.37 virtual void ABA_LPSUB::varRealloc (int *newSize*) `[private, virtual]`

Sets the maximal number of variables to *newSize*.

6.20.3.38 virtual double ABA_LPSUB::xVal (int *i*) `[virtual]`

We have to redefine the function *xVal(i)* since variables may have been eliminated.

Returns:

The x -value of variable *i* after the solution of the linear program.

Reimplemented from ABA_LP.

6.20.4 Friends And Related Function Documentation

6.20.4.1 friend class ABA_BOUNDBRANCHRULE `[friend]`

Definition at line 60 of file lpsub.h.

6.20.4.2 friend class ABA_CONBRANCHRULE `[friend]`

Definition at line 62 of file lpsub.h.

6.20.4.3 friend class ABA_SETBRANCHRULE `[friend]`

Definition at line 59 of file lpsub.h.

6.20.4.4 friend class ABA_SUB `[friend]`

Definition at line 58 of file lpsub.h.

6.20.4.5 friend class ABA_VALBRANCHRULE `[friend]`

Definition at line 61 of file lpsub.h.

6.20.4.6 friend class COPBRANCHRULE `[friend]`

Definition at line 63 of file lpsub.h.

6.20.5 Member Data Documentation

6.20.5.1 ABA_BUFFER<ABA_INFEASCON*> ABA_LPSUB::infeasCons_ `[private]`

Buffer storing the infeasible constraints found be the constructor.

Definition at line 342 of file lpsub.h.

6.20.5.2 ABA_ARRAY<int> ABA_LPSUB::lp2orig_ `[private]`

Orignial number of a (non-eliminated) variable.

Definition at line 338 of file lpsub.h.

6.20.5.3 int ABA_LPSUB::nOrigVar_ [private]

The number of original variables of the linear program.

Definition at line 351 of file lpsub.h.

6.20.5.4 ABA_ARRAY<int> ABA_LPSUB::orig2lp_ [private]

After the elimination of variables the internal variables are again numbered consecutively starting with 0. *orig2lp_-*[i] is the internal number of the variable *i*. This is *-1* if the variable is eliminated.

Definition at line 334 of file lpsub.h.

6.20.5.5 const ABA_SUB* ABA_LPSUB::sub_ [private]

A pointer to the corresponding subproblem.

Definition at line 327 of file lpsub.h.

6.20.5.6 double ABA_LPSUB::valueAdd_ [private]

The constant which has been added to the objective function value due to the elimination of variables.

Definition at line 347 of file lpsub.h.

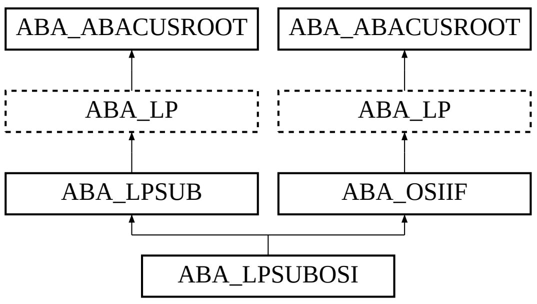
The documentation for this class was generated from the following file:

- Include/abacus/lpsub.h

6.21 ABA_LPSUBOSI Class Reference

`#include <lpsubosi.h>`

Inheritance diagram for ABA_LPSUBOSI::



Public Member Functions

- ABA_LPSUBOSI (ABA_MASTER ∗master, ABA_SUB ∗sub)

The constructor calls the function initialize() *of the base classABA_LPSUB, which sets up the linear program and passes the data to the LP-solver.*

- virtual ~ABA_LPSUBOSI ()

  The destructor.

Private Member Functions

- ABA_LPSUBOSI (const ABA_LPSUBOSI &rhs)
- const ABA_LPSUBOSI & operator= (const ABA_LPSUBOSI &rhs)

6.21.1 Constructor & Destructor Documentation

6.21.1.1 ABA_LPSUBOSI::ABA_LPSUBOSI (ABA_MASTER $*$ *master*, ABA_SUB $*$ *sub*)

The constructor calls the function *initialize()* of the base classABA_LPSUB, which sets up the linear program and passes the data to the LP-solver.

Parameters:

master A pointer to the corresponding master of the optimization.

sub The subproblem of which the LP-relaxation is solved.

6.21.1.2 virtual ABA_LPSUBOSI::~ABA_LPSUBOSI () `[virtual]`

The destructor.

6.21.1.3 ABA_LPSUBOSI::ABA_LPSUBOSI (const ABA_LPSUBOSI & *rhs*) `[private]`

6.21.2 Member Function Documentation

6.21.2.1 const ABA_LPSUBOSI& ABA_LPSUBOSI::operator= (const ABA_LPSUBOSI & *rhs*) `[private]`

The documentation for this class was generated from the following file:

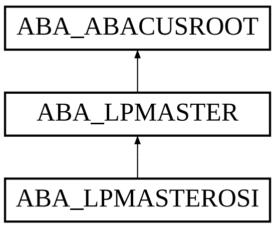
- Include/abacus/lpsubosi.h

6.22 ABA_LPMASTER Class Reference

The class ABA_LPMASTER is an abstract base class. A LP solver specific master class has to be derived from this class.

`#include <lpmaster.h>`

Inheritance diagram for ABA_LPMASTER::



Public Member Functions

- ABA_LPMASTER (ABA_MASTER ∗master)
- virtual ∼ABA_LPMASTER ()
- virtual void initializeLpParameters ()=0
- virtual void setDefaultLpParameters ()=0
- virtual void printLpParameters ()=0
- virtual void outputLpStatistics ()=0

Protected Attributes

- ABA_MASTER ∗ master_

6.22.1 Detailed Description

The class ABA_LPMASTER is an abstract base class. A LP solver specific master class has to be derived from this class.

Definition at line 40 of file lpmaster.h.

6.22.2 Constructor & Destructor Documentation

6.22.2.1 ABA_LPMASTER::ABA_LPMASTER (ABA_MASTER ∗ *master*) `[inline]`

Definition at line 42 of file lpmaster.h.

6.22.2.2 virtual ABA_LPMASTER::∼ABA_LPMASTER () `[inline, virtual]`

Definition at line 43 of file lpmaster.h.

6.22.3 Member Function Documentation

6.22.3.1 virtual void ABA_LPMASTER::initializeLpParameters () `[pure virtual]`

Implemented in ABA_LPMASTEROSI.

6.22.3.2 virtual void ABA_LPMASTER::outputLpStatistics () `[pure virtual]`

Implemented in ABA_LPMASTEROSI.

6.22.3.3 virtual void ABA_LPMASTER::printLpParameters () `[pure virtual]`

Implemented in ABA_LPMASTEROSI.

6.22.3.4 virtual void ABA_LPMASTER::setDefaultLpParameters () `[pure virtual]`

Implemented in ABA_LPMASTEROSI.

6.22.4 Member Data Documentation

6.22.4.1 ABA_MASTER∗ ABA_LPMASTER::master_ `[protected]`

Definition at line 50 of file lpmaster.h.

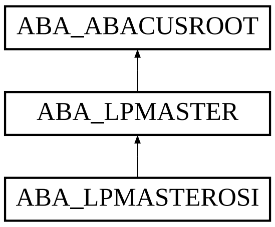
The documentation for this class was generated from the following file:

- Include/abacus/lpmaster.h

6.23 ABA_LPMASTEROSI Class Reference

`#include <lpmasterosi.h>`

Inheritance diagram for ABA_LPMASTEROSI::



Public Member Functions

- ABA_LPMASTEROSI (ABA_MASTER ∗master)
- virtual ∼ABA_LPMASTEROSI ()

  The destructor.

- virtual void initializeLpParameters ()
- virtual void setDefaultLpParameters ()
- virtual void printLpParameters ()
- virtual void outputLpStatistics ()

Friends

- class ABA_OSIIF

6.23.1 Constructor & Destructor Documentation

6.23.1.1 ABA_LPMASTEROSI::ABA_LPMASTEROSI (ABA_MASTER ∗ *master*)

The constructor.

Parameters:
  master The master of the optimization.

6.23.1.2 virtual ABA_LPMASTEROSI::∼ABA_LPMASTEROSI () `[virtual]`

The destructor.

6.23.2 Member Function Documentation

6.23.2.1 virtual void ABA_LPMASTEROSI::initializeLpParameters () `[virtual]`

Initializes the LP solver specific Parameters.

Implements ABA_LPMASTER.

6.23.2.2 virtual void ABA_LPMASTEROSI::outputLpStatistics () `[virtual]`

Prints LP solver specific Statistics.

Implements ABA_LPMASTER.

6.23.2.3 virtual void ABA_LPMASTEROSI::printLpParameters () `[virtual]`

Prints the settings of the LP solver specific Parameters.

Implements ABA_LPMASTER.

6.23.2.4 virtual void ABA_LPMASTEROSI::setDefaultLpParameters () `[virtual]`

Sets default values of the LP solver specific Parameters.

Implements ABA_LPMASTER.

6.23.3 Friends And Related Function Documentation

6.23.3.1 friend class ABA_OSIIF `[friend]`

Definition at line 40 of file lpmasterosi.h.

The documentation for this class was generated from the following file:

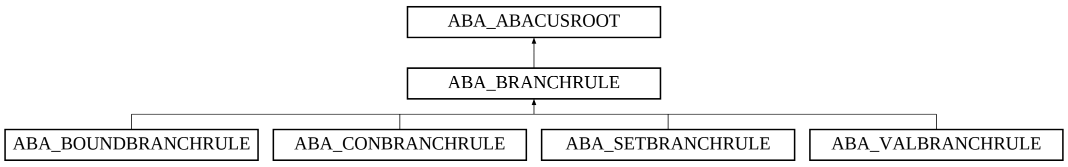
- Include/abacus/lpmasterosi.h

6.24 ABA_BRANCHRULE Class Reference

class is an abstract base class for all branching rules within this framework.

`#include <branchrule.h>`

Inheritance diagram for ABA_BRANCHRULE::



Public Member Functions

- ABA_BRANCHRULE (ABA_MASTER ∗master)
- virtual ∼ABA_BRANCHRULE ()
- virtual int extract (ABA_SUB ∗sub)=0
- virtual void extract (ABA_LPSUB ∗lp)

  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.

- virtual void unExtract (ABA_LPSUB ∗lp)
- virtual bool branchOnSetVar ()

  Should indicate if the branching is performed by setting a binary variable.

- virtual void initialize (ABA_SUB ∗sub)

  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.

Protected Attributes

- ABA_MASTER ∗ master_

6.24.1 Detailed Description

class is an abstract base class for all branching rules within this framework.

Parameters:

ABA_MASTER ∗master_ A pointer to the corresponding master of the optimization.

Definition at line 63 of file branchrule.h.

6.24.2 Constructor & Destructor Documentation

6.24.2.1 ABA_BRANCHRULE::ABA_BRANCHRULE (ABA_MASTER ∗ *master*)

The constructor.

Parameters:

master A pointer to the corresponding master of the optimization.

6.24.2.2 virtual ABA_BRANCHRULE::∼ABA_BRANCHRULE () `[virtual]`

The destructor.

6.24.3 Member Function Documentation

6.24.3.1 virtual bool ABA_BRANCHRULE::branchOnSetVar () [virtual]

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.

This function is currently required to determine global validity of Gomory cuts for general *s*.

Returns:
The default implementation returns always false. This function must be redefined in the class ABA_SETBRANCHRULE, where it has to return *true*.

Reimplemented in ABA_SETBRANCHRULE.

6.24.3.2 virtual void ABA_BRANCHRULE::extract (ABA_LPSUB ∗ *lp*) [virtual]

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.

Parameters:
lp A pointer to a the linear programming relaxtion of a a subproblem.

Reimplemented in ABA_BOUNDBRANCHRULE, ABA_CONBRANCHRULE, ABA_SETBRANCHRULE, and ABA_VALBRANCHRULE.

6.24.3.3 virtual int ABA_BRANCHRULE::extract (ABA_SUB ∗ *sub*) [pure virtual]

Modifies a subproblem by setting the branching variable.

Returns:
0 If the subproblem can be modified according to the branching rule.
1 If a contradiction occurs.

Parameters:
sub The subproblem being modified.

Implemented in ABA_BOUNDBRANCHRULE, ABA_CONBRANCHRULE, ABA_SETBRANCHRULE, and ABA_VALBRANCHRULE.

6.24.3.4 virtual void ABA_BRANCHRULE::initialize (ABA_SUB ∗ *sub*) [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.

Parameters:
sub A pointer to the subproblem that should be used for the initialization.}

Reimplemented in ABA_CONBRANCHRULE.

6.24.3.5 virtual void ABA_BRANCHRULE::unExtract (ABA_LPSUB ∗ *lp*) [virtual]

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.

Parameters:

lp A pointer to a the linear programming relaxtion of a a subproblem.

Reimplemented in ABA_BOUNDBRANCHRULE, ABA_CONBRANCHRULE, ABA_SETBRANCHRULE, and ABA_VALBRANCHRULE.

6.24.4 Member Data Documentation

6.24.4.1 ABA_MASTER∗ ABA_BRANCHRULE::master_ [protected]

Definition at line 157 of file branchrule.h.

The documentation for this class was generated from the following file:

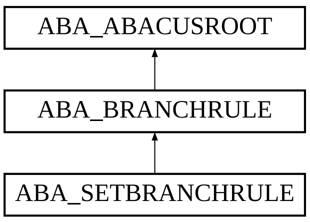
- Include/abacus/branchrule.h

6.25 ABA_SETBRANCHRULE Class Reference

The data members of the class ABA_SETBRANCHRULE.

`#include <setbranchrule.h>`

Inheritance diagram for ABA_SETBRANCHRULE::



Public Member Functions

- ABA_SETBRANCHRULE (ABA_MASTER ∗master, int variable, ABA_FSVARSTAT::STATUS status)
- virtual ∼ABA_SETBRANCHRULE ()

  The destructor.

- virtual int extract (ABA_SUB ∗sub)

- virtual void extract (ABA_LPSUB ∗lp)

  The function extract() is overloaded to modify directly the linear programming relaxation.

- virtual void unExtract (ABA_LPSUB ∗lp)

  The function unExtract().

- virtual bool branchOnSetVar ()

  Redefines the virtual function of the base class ABA_BRANCHRULE as this branching rule is setting a binary variable.

- bool setToUpperBound () const
- int variable () const

Private Attributes

- int variable_
- ABA_FSVARSTAT::STATUS status_
- double oldLpBound_

Friends

- ostream & operator<< (ostream &out, const ABA_SETBRANCHRULE &rhs)

  The output operator writes the number of the branching variable and its status on an output stream.

6.25.1 Detailed Description

The data members of the class ABA_SETBRANCHRULE.

Parameters:

int variable_ The branching variable.

ABA_FSVARSTAT::STATUS status_ The status of the branching variable (*SetToLowerBound* or *SetToUpperBound*).

double oldLpbound_ The bound of the branching variable in the linear program, before it is temporarily modified for testing the quality of this branching rule.

Definition at line 43 of file setbranchrule.h.

6.25.2 Constructor & Destructor Documentation

6.25.2.1 ABA_SETBRANCHRULE::ABA_SETBRANCHRULE (ABA_MASTER ∗ *master*, int *variable*, ABA_FSVARSTAT::STATUS *status*)

The constructor.

Parameters:

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*).

6.25.2.2 virtual ABA_SETBRANCHRULE::∼ABA_SETBRANCHRULE () `[virtual]`

The destructor.

6.25.3 Member Function Documentation

6.25.3.1 virtual bool ABA_SETBRANCHRULE::branchOnSetVar () `[virtual]`

Redefines the virtual function of the base class ABA_BRANCHRULE as this branching rule is setting a binary variable.

Returns:

Always *true*.

Reimplemented from ABA_BRANCHRULE.

6.25.3.2 virtual void ABA_SETBRANCHRULE::extract (ABA_LPSUB ∗ *lp*) `[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.

Parameters:

lp A pointer to the linear programming relaxation of a subproblem.

Reimplemented from ABA_BRANCHRULE.

6.25.3.3 virtual int ABA_SETBRANCHRULE::extract (ABA_SUB ∗ *sub*) `[virtual]`

Modifies a subproblem by setting the branching variable.

Returns:

0 If the subproblem can be modified according to the branching rule.
1 If a contradiction occurs.

Parameters:

sub The subproblem being modified.

Implements ABA_BRANCHRULE.

6.25.3.4 bool ABA_SETBRANCHRULE::setToUpperBound () const

Returns:

true If the branching variable is set to the upper bound,
false otherwise.

6.25.3.5 virtual void ABA_SETBRANCHRULE::unExtract (ABA_LPSUB ∗ *lp*) [virtual]

The function *unExtract()*.

Reimplemented from ABA_BRANCHRULE.

6.25.3.6 int ABA_SETBRANCHRULE::variable () const [inline]

Returns:

The number of the branching variable.

Definition at line 151 of file setbranchrule.h.

6.25.4 Friends And Related Function Documentation

6.25.4.1 ostream& operator<< (ostream & *out*, const ABA_SETBRANCHRULE & *rhs*) [friend]

The output operator writes the number of the branching variable and its status on an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The branching rule being output.

6.25.5 Member Data Documentation

6.25.5.1 double ABA_SETBRANCHRULE::oldLpBound_ [private]

Definition at line 146 of file setbranchrule.h.

6.25.5.2 ABA_FSVARSTAT::STATUS ABA_SETBRANCHRULE::status_ [private]

Definition at line 145 of file setbranchrule.h.

6.25.5.3 int ABA_SETBRANCHRULE::variable_ `[private]`

Definition at line 144 of file setbranchrule.h.

The documentation for this class was generated from the following file:

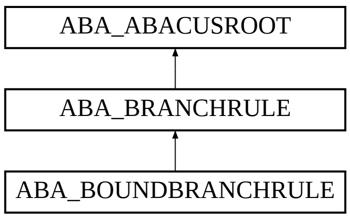
- Include/abacus/setbranchrule.h

6.26 ABA_BOUNDBRANCHRULE Class Reference

class implements a branching rule for modifying the lower and the upper bound of a variable.

`#include <boundbranchrule.h>`

Inheritance diagram for ABA_BOUNDBRANCHRULE::



Public Member Functions

- ABA_BOUNDBRANCHRULE (ABA_MASTER ∗master, int variable, double lBound, double uBound)
- virtual ∼ABA_BOUNDBRANCHRULE ()
- virtual int extract (ABA_SUB ∗sub)

  Modifies a subproblem by changing the lower and the upper bound of the branching variable.

- virtual void extract (ABA_LPSUB ∗lp)

  Is overloaded to modify directly the linear programming relaxation.

- virtual void unExtract (ABA_LPSUB ∗lp)
- int variable () const
- double lBound () const
- double uBound () const

Private Attributes

- int variable_
- double lBound_
- double uBound_
- double oldLpLBound_
- double oldLpUBound_

Friends

- ostream & operator<< (ostream &out, const ABA_BOUNDBRANCHRULE &rhs)

  The output operator writes the branching variable together with its lower and upper bound to an output stream.

6.26.1 Detailed Description

class implements a branching rule for modifying the lower and the upper bound of a variable.

Parameters:

int variable_ The branching variable.

double lBound_ The lower bound of the branching variable.

double uBound_ The upper bound of the branching variable.

Definition at line 40 of file boundbranchrule.h.

6.26.2 Constructor & Destructor Documentation

6.26.2.1 ABA_BOUNDBRANCHRULE::ABA_BOUNDBRANCHRULE (ABA_MASTER $*$ *master*, int *variable*, double *lBound*, double *uBound*)

The constructor.

Parameters:

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.

6.26.2.2 virtual ABA_BOUNDBRANCHRULE::∼ABA_BOUNDBRANCHRULE () `[virtual]`

The destructor.

6.26.3 Member Function Documentation

6.26.3.1 virtual void ABA_BOUNDBRANCHRULE::extract (ABA_LPSUB $*$ *lp*) `[virtual]`

Is overloaded to modify directly the linear programming relaxation.

This required to evaluate the quality of a branching rule.

Reimplemented from ABA_BRANCHRULE.

6.26.3.2 virtual int ABA_BOUNDBRANCHRULE::extract (ABA_SUB $*$ *sub*) `[virtual]`

Modifies a subproblem by changing the lower and the upper bound of the branching variable.

Returns:
0 If the subproblem is successfully modified.
1 If the modification causes a contradiction.

Parameters:
sub The subproblem being modified.

Implements ABA_BRANCHRULE.

6.26.3.3 double ABA_BOUNDBRANCHRULE::lBound () const `[inline]`

Returns:
The lower bound of the branching variable.

Definition at line 139 of file boundbranchrule.h.

6.26.3.4 double ABA_BOUNDBRANCHRULE::uBound () const `[inline]`

Returns:
The upper bound of the branching variable.

Definition at line 144 of file boundbranchrule.h.

6.26.3.5 virtual void ABA_BOUNDBRANCHRULE::unExtract (ABA_LPSUB $*$ *lp*) `[virtual]`

Should undo the modifications of the linear programming relaxtion |lp|.

This function has to be redefined in a derived class, if also extract(ABA_LPSUB$*$) is redefined there.

Parameters:
lp A pointer to a the linear programming relaxtion of a a subproblem.

Reimplemented from ABA_BRANCHRULE.

6.26.3.6 int ABA_BOUNDBRANCHRULE::variable () const `[inline]`

Returns:
The number of the branching variable.

Definition at line 134 of file boundbranchrule.h.

6.26.4 Friends And Related Function Documentation

6.26.4.1 ostream& operator<< (ostream & *out*, const ABA_BOUNDBRANCHRULE & *rhs*) `[friend]`

The output operator writes the branching variable together with its lower and upper bound to an output stream.

Returns:
A reference to the output stream.

Parameters:
out The output stream.
rhs The branch rule being output.

6.26.5 Member Data Documentation

6.26.5.1 double ABA_BOUNDBRANCHRULE::lBound_ `[private]`

Definition at line 127 of file boundbranchrule.h.

6.26.5.2 double ABA_BOUNDBRANCHRULE::oldLpLBound_ `[private]`

Definition at line 129 of file boundbranchrule.h.

6.26.5.3 double ABA_BOUNDBRANCHRULE::oldLpUBound_ `[private]`

Definition at line 130 of file boundbranchrule.h.

6.26.5.4 double ABA_BOUNDBRANCHRULE::uBound_ `[private]`

Definition at line 128 of file boundbranchrule.h.

6.26.5.5 int ABA_BOUNDBRANCHRULE::variable_ `[private]`

Definition at line 126 of file boundbranchrule.h.

The documentation for this class was generated from the following file:

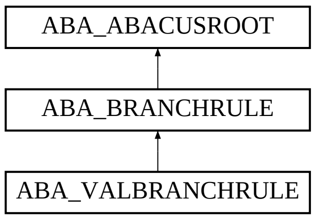
- Include/abacus/boundbranchrule.h

6.27 ABA_VALBRANCHRULE Class Reference

class implements a branching rule for setting a variable to a certain value.

`#include <valbranchrule.h>`

Inheritance diagram for ABA_VALBRANCHRULE::



Public Member Functions

- ABA_VALBRANCHRULE (ABA_MASTER ∗master, int variable, double value)
- virtual ∼ABA_VALBRANCHRULE ()

  The destructor.

- virtual int extract (ABA_SUB ∗sub)
- virtual void extract (ABA_LPSUB ∗lp)

  The function extract() *is overloaded to modify directly the linear programming relaxation. This required to evaluate the quality of a branching rule.*

- virtual void unExtract (ABA_LPSUB ∗lp)

  The function unExtract().

- int variable () const
- double value () const

Private Attributes

- int variable_
- double value_
- double oldLpLBound_
- double oldLpUBound_

Friends

- ostream & operator<< (ostream &out, const ABA_VALBRANCHRULE &rhs)

  The output operator writes the branching variable together with its value to an output stream.

6.27.1 Detailed Description

class implements a branching rule for setting a variable to a certain value.

Parameters:

int variable_ The branching variable.

double value_ The value the branching variable is set to.

Definition at line 42 of file valbranchrule.h.

6.27.2 Constructor & Destructor Documentation

6.27.2.1 ABA_VALBRANCHRULE::ABA_VALBRANCHRULE (ABA_MASTER ∗ *master*, int *variable*, double *value*)

The constructor.

Parameters:

master The corresponding master of the optimization.

variable The branching variable.

value The value the branching variable is set to.

6.27.2.2 virtual ABA_VALBRANCHRULE::∼ABA_VALBRANCHRULE () `[virtual]`

The destructor.

6.27.3 Member Function Documentation

6.27.3.1 virtual void ABA_VALBRANCHRULE::extract (ABA_LPSUB ∗ *lp*) `[virtual]`

The function *extract()* is overloaded to modify directly the linear programming relaxation. This required to evaluate the quality of a branching rule.

Reimplemented from ABA_BRANCHRULE.

6.27.3.2 virtual int ABA_VALBRANCHRULE::extract (ABA_SUB ∗ *sub*) `[virtual]`

Modifies a subproblem by setting the branching variable.

Returns:

0 If the subproblem can be modified according to the branching rule.
1 If a contradiction occurs.

Parameters:

sub The subproblem being modified.

Implements ABA_BRANCHRULE.

6.27.3.3 virtual void ABA_VALBRANCHRULE::unExtract (ABA_LPSUB ∗ *lp*) `[virtual]`

The function *unExtract()*.

Reimplemented from ABA_BRANCHRULE.

6.27.3.4 double ABA_VALBRANCHRULE::value () const `[inline]`

Returns:
The value of the branching variable.

Definition at line 136 of file valbranchrule.h.

6.27.3.5 int ABA_VALBRANCHRULE::variable () const `[inline]`

Returns:
The number of the branching variable.

Definition at line 131 of file valbranchrule.h.

6.27.4 Friends And Related Function Documentation

6.27.4.1 ostream& operator<< (ostream & *out*, const ABA_VALBRANCHRULE & *rhs*) `[friend]`

The output operator writes the branching variable together with its value to an output stream.

Returns:
A reference to the output stream.

Parameters:
out The output stream.
rhs The branching rule being output.

6.27.5 Member Data Documentation

6.27.5.1 double ABA_VALBRANCHRULE::oldLpLBound_ `[private]`

Definition at line 126 of file valbranchrule.h.

6.27.5.2 double ABA_VALBRANCHRULE::oldLpUBound_ `[private]`

Definition at line 127 of file valbranchrule.h.

6.27.5.3 double ABA_VALBRANCHRULE::value_ `[private]`

Definition at line 125 of file valbranchrule.h.

6.27.5.4 int ABA_VALBRANCHRULE::variable_ `[private]`

Definition at line 124 of file valbranchrule.h.

The documentation for this class was generated from the following file:

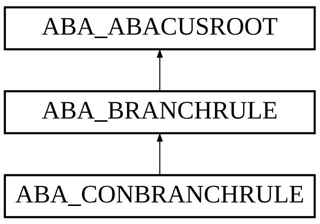
- Include/abacus/valbranchrule.h

6.28 ABA_CONBRANCHRULE Class Reference

class implements the branching by adding a constraint to the set of active constraints.

`#include <conbranchrule.h>`

Inheritance diagram for ABA_CONBRANCHRULE::



Public Member Functions

- ABA_CONBRANCHRULE (ABA_MASTER ∗master, ABA_POOLSLOT< ABA_CONSTRAINT, ABA_VARIABLE > ∗poolSlot)
- virtual ∼ABA_CONBRANCHRULE ()
- virtual int extract (ABA_SUB ∗sub)
- virtual void extract (ABA_LPSUB ∗lp)

  The function extract() is overloaded to modify directly the linear programming relaxation.

- virtual void unExtract (ABA_LPSUB ∗lp)
- virtual void initialize (ABA_SUB ∗sub)

  Redefines the virtual function of the base class ABA_BRANCHRULE in order to initialize the subproblem associated with the branching constraint.

- ABA_CONSTRAINT ∗ constraint ()

Private Member Functions

- const ABA_CONBRANCHRULE & operator= (const ABA_CONBRANCHRULE &rhs)

Private Attributes

- ABA_POOLSLOTREF< ABA_CONSTRAINT, ABA_VARIABLE > poolSlotRef_

Friends

- ostream & operator<< (ostream &out, const ABA_CONBRANCHRULE &rhs)

6.28.1 Detailed Description

class implements the branching by adding a constraint to the set of active constraints.

Parameters:

ABA_POOLSLOTREF poolSlotRef_ A reference to the pool slot of the branching constraints.

Definition at line 46 of file conbranchrule.h.

6.28.2 Constructor & Destructor Documentation

6.28.2.1 ABA_CONBRANCHRULE::ABA_CONBRANCHRULE (ABA_MASTER $*$ *master*, ABA_POOLSLOT< ABA_CONSTRAINT, ABA_VARIABLE > $*$ *poolSlot*)

The 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.

Parameters:

master A pointer to the corresponding master of the optimization.

poolSlot A pointer to the pool slot of the branching constraint.

6.28.2.2 virtual ABA_CONBRANCHRULE::~ABA_CONBRANCHRULE () `[virtual]`

The destructor.

6.28.3 Member Function Documentation

6.28.3.1 ABA_CONSTRAINT$*$ ABA_CONBRANCHRULE::constraint ()

Returns:

A pointer to the branching constraint or a 0-pointer, if this constraint is not available.

6.28.3.2 virtual void ABA_CONBRANCHRULE::extract (ABA_LPSUB ∗ *lp*) `[virtual]`

The function *extract()* is overloaded to modify directly the linear programming relaxation.

This required to evaluate the quality of a branching rule.

Reimplemented from ABA_BRANCHRULE.

6.28.3.3 virtual int ABA_CONBRANCHRULE::extract (ABA_SUB ∗ *sub*) `[virtual]`

Adds the branching constraint to the subproblem.

Instead of adding it directly to the set of active constraints it is added to the cut buffer.

Returns:
Always 0, since there cannot be a contractiction.

Parameters:
sub The subproblem being modified.

Implements ABA_BRANCHRULE.

6.28.3.4 virtual void ABA_CONBRANCHRULE::initialize (ABA_SUB ∗ *sub*) `[virtual]`

Redefines the virtual function of the base class ABA_BRANCHRULE in order to initialize the subproblem associated with the branching constraint.

Parameters:
sub A pointer to the subproblem that is associated with the branching constraint.

Reimplemented from ABA_BRANCHRULE.

6.28.3.5 const ABA_CONBRANCHRULE& ABA_CONBRANCHRULE::operator= (const ABA_CONBRANCHRULE & *rhs*) `[private]`

6.28.3.6 virtual void ABA_CONBRANCHRULE::unExtract (ABA_LPSUB ∗ *lp*) `[virtual]`

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.

Parameters:
lp A pointer to a the linear programming relaxtion of a a subproblem.

Reimplemented from ABA_BRANCHRULE.

6.28.4 Friends And Related Function Documentation

6.28.4.1 ostream& operator<< (ostream & *out*, const ABA_CONBRANCHRULE & *rhs*) `[friend]`

The output operator writes the branching constraint on an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The branch rule being output.

6.28.5 Member Data Documentation

6.28.5.1 ABA_POOLSLOTREF<ABA_CONSTRAINT, ABA_VARIABLE> ABA_CONBRANCHRULE::poolSlotRef_ `[private]`

Definition at line 135 of file conbranchrule.h.

The documentation for this class was generated from the following file:

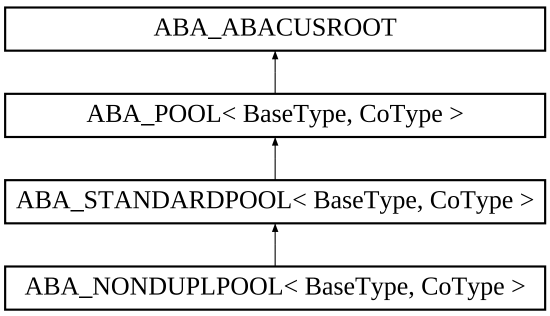
- Include/abacus/conbranchrule.h

6.29 ABA_POOL< BaseType, CoType > Class Template Reference

The public enumerations of ABA_POOL.

`#include <pool.h>`

Inheritance diagram for ABA_POOL< BaseType, CoType >::


ABA_ABACUSROOT
ABA_POOL< BaseType, CoType >
ABA_STANDARDPOOL< BaseType, CoType >
ABA_NONDUPLPOOL< BaseType, CoType >

Public Types

- enum RANKING { NO_RANK, RANK, ABS_RANK }

Public Member Functions

- ABA_POOL (ABA_MASTER ∗master)
- virtual ∼ABA_POOL ()

  The destructor.

- virtual ABA_POOLSLOT< BaseType, CoType > ∗ insert (BaseType ∗cv)=0
- void removeConVar (ABA_POOLSLOT< BaseType, CoType > ∗slot)

  Removes the constraint/variable stored in a pool slot and adds the slot to the list of free slots.

- 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 Member Functions

- virtual int softDeleteConVar (ABA_POOLSLOT< BaseType, CoType > ∗slot)

  Removes the constraint/variable stored in the pool slot slot *from the pool if the constraint/variable can be deleted.*

- virtual void hardDeleteConVar (ABA_POOLSLOT< BaseType, CoType > ∗slot)

  Removes a constraint/variable from the pool and adds the slot to the set of free slots.

- virtual ABA_POOLSLOT< BaseType, CoType > ∗ getSlot ()=0
- virtual void putSlot (ABA_POOLSLOT< BaseType, CoType > ∗slot)=0

Protected Attributes

- ABA_MASTER ∗ master_
- int number_

6.29.1 Detailed Description

template<class BaseType, class CoType> class ABA_POOL< BaseType, CoType >

The public enumerations of ABA_POOL.

Definition at line 65 of file pool.h.

6.29.2 Member Enumeration Documentation

6.29.2.1 template<class BaseType, class CoType> enum ABA_POOL::RANKING

Enumeration values:

NO_RANK

RANK

ABS_RANK

Definition at line 67 of file pool.h.

6.29.3 Constructor & Destructor Documentation

6.29.3.1 template<class BaseType, class CoType> ABA_POOL< BaseType, CoType >::ABA_POOL (ABA_MASTER ∗ *master*)

The constructor initializes an empty pool.

Parameters:

master A pointer to the corresponding master of the optimization.

6.29.3.2 template<class BaseType, class CoType> virtual ABA_POOL< BaseType, CoType >::~ABA_POOL () `[virtual]`

The destructor.

6.29.4 Member Function Documentation

6.29.4.1 template<class BaseType, class CoType> virtual ABA_POOLSLOT<BaseType, CoType>∗ ABA_POOL< BaseType, CoType >::getSlot () `[protected, pure virtual]`

Implemented in ABA_STANDARDPOOL< BaseType, CoType >, ABA_STANDARDPOOL< ABA_VARIABLE, ABA_CONSTR
and ABA_STANDARDPOOL< ABA_CONSTRAINT, ABA_VARIABLE >.

6.29.4.2 template<class BaseType, class CoType> virtual void ABA_POOL< BaseType, CoType >::hardDeleteConVar (ABA_POOLSLOT< BaseType, CoType > ∗ *slot*) `[protected, virtual]`

Removes a constraint/variable from the pool and adds the slot to the set of free slots.

Parameters:

slot A pointer to the pool slot from wich the constraint/variable should be deleted.

Reimplemented in ABA_NONDUPLPOOL< BaseType, CoType >.

6.29.4.3 template<class BaseType, class CoType> virtual ABA_POOLSLOT<BaseType, CoType>∗ ABA_POOL< BaseType, CoType >::insert (BaseType ∗ *cv*) `[pure virtual]`

Implemented in ABA_NONDUPLPOOL< BaseType, CoType >, and ABA_STANDARDPOOL< BaseType, CoType >.

6.29.4.4 template<class BaseType, class CoType> int ABA_POOL< BaseType, CoType >::number () const

Returns:

The current number of items in the pool.

6.29.4.5 template<class BaseType, class CoType> virtual void ABA_POOL< BaseType, CoType >::putSlot (ABA_POOLSLOT< BaseType, CoType > ∗ *slot*) `[protected, pure virtual]`

Implemented in ABA_STANDARDPOOL< BaseType, CoType >.

6.29.4.6 template<class BaseType, class CoType> void ABA_POOL< BaseType, CoType >::removeConVar (ABA_POOLSLOT< BaseType, CoType > ∗ *slot*)

Removes the constraint/variable stored in a pool slot and adds the slot to the list of free slots.

Parameters:

slot The pool slot from which the constraint/variable is removed.

6.29.4.7 template<class BaseType, class CoType> virtual int ABA_POOL< BaseType, CoType >::separate (double ∗ *z*, ABA_ACTIVE< CoType, BaseType > ∗ *active*, ABA_SUB ∗ *sub*, ABA_CUTBUFFER< BaseType, CoType > ∗ *cutBuffer*, double *minAbsViolation* = `0.001`, int *ranking* = `0`) `[pure virtual]`

Implemented in ABA_STANDARDPOOL< BaseType, CoType >.

6.29.4.8 template<class BaseType, class CoType> virtual int ABA_POOL< BaseType, CoType >::softDeleteConVar (ABA_POOLSLOT< BaseType, CoType > ∗ *slot*) `[protected, virtual]`

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.

Returns:

0 If the constraint/variable could be deleted.
1 otherwise.

Parameters:

slot A pointer to the pool slot from wich the constraint/variable should be deleted.

Reimplemented in ABA_NONDUPLPOOL< BaseType, CoType >.

6.29.5 Member Data Documentation

6.29.5.1 template<class BaseType, class CoType> ABA_MASTER∗ ABA_POOL< BaseType, CoType >::master_ `[protected]`

Definition at line 136 of file pool.h.

6.29.5.2 template<class BaseType, class CoType> int ABA_POOL< BaseType, CoType >::number_ `[protected]`

Definition at line 137 of file pool.h.

The documentation for this class was generated from the following file:

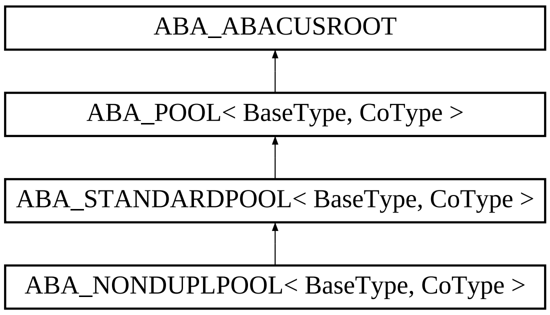
- Include/abacus/pool.h

6.30 ABA_STANDARDPOOL< BaseType, CoType > Class Template Reference

class provides a very simple implementation of a pool which is sufficient for a large class of applications. pool slots stored in array, set of free slots is managed by a linear list

`#include <standardpool.h>`

Inheritance diagram for ABA_STANDARDPOOL< BaseType, CoType >::


ABA_ABACUSROOT
ABA_POOL< BaseType, CoType >
ABA_STANDARDPOOL< BaseType, CoType >
ABA_NONDUPLPOOL< BaseType, CoType >

Public Member Functions

- ABA_STANDARDPOOL (ABA_MASTER ∗master, int size, bool autoRealloc=false)
- virtual ∼ABA_STANDARDPOOL ()

  The destructor deletes all slots. The destructor of a pool slot deletes then also the respective constraint or variable.

- virtual ABA_POOLSLOT< BaseType, CoType > ∗ insert (BaseType ∗cv)
- virtual void increase (int size)
- int 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 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)

  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.

Protected Member Functions

- int removeNonActive (int maxRemove)
- virtual ABA_POOLSLOT< BaseType, CoType > ∗ getSlot ()

  Returns a free slot, or 0 if no free slot is available. A returned slot is removed from the list of free slots.

- virtual void putSlot (ABA_POOLSLOT< BaseType, CoType > ∗slot)

Protected Attributes

- ABA_ARRAY< ABA_POOLSLOT< BaseType, CoType > ∗ > pool_
- ABA_LIST< ABA_POOLSLOT< BaseType, CoType > ∗ > freeSlots_
- bool autoRealloc_

Private Member Functions

- ABA_STANDARDPOOL (const ABA_STANDARDPOOL &rhs)
- const ABA_STANDARDPOOL & operator= (const ABA_STANDARDPOOL &rhs)

Friends

- ostream & operator<< (ostream &out, const ABA_STANDARDPOOL &rhs)

  The output operator calls the output operator of each item of a non-void pool slot.

6.30.1 Detailed Description

template<class BaseType, class CoType> class ABA_STANDARDPOOL< BaseType, CoType >

class provides a very simple implementation of a pool which is sufficient for a large class of applications. pool slots stored in array, set of free slots is managed by a linear list

Definition at line 58 of file standardpool.h.

6.30.2 Constructor & Destructor Documentation

6.30.2.1 template<class BaseType, class CoType> ABA_STANDARDPOOL< BaseType, CoType >::ABA_STANDARDPOOL (ABA_MASTER * *master*, int *size*, bool *autoRealloc* = `false`)

The constructor for an empty pool.

All slots are inserted in the linked list of free slots.

Parameters:

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.

6.30.2.2 template<class BaseType, class CoType> virtual ABA_STANDARDPOOL< BaseType, CoType >::~ABA_STANDARDPOOL ()

The destructor deletes all slots. The destructor of a pool slot deletes then also the respective constraint or variable.

6.30.2.3 template<class BaseType, class CoType> ABA_STANDARDPOOL< BaseType, CoType >::ABA_STANDARDPOOL (const ABA_STANDARDPOOL< BaseType, CoType > & *rhs*) `[private]`

6.30.3 Member Function Documentation

6.30.3.1 template<class BaseType, class CoType> int ABA_STANDARDPOOL< BaseType, CoType >::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.

Returns:

The number of "cleaned" slots.

6.30.3.2 template<class BaseType, class CoType> virtual ABA_POOLSLOT<BaseType,CoType>* ABA_STANDARDPOOL< BaseType, CoType >::getSlot () `[protected, virtual]`

Returns a free slot, or 0 if no free slot is available. A returned slot is removed from the list of free slots.

This function defines the pure virtual function of the base class ABA_POOL.

Implements ABA_POOL< BaseType, CoType >.

6.30.3.3 template<class BaseType, class CoType> virtual void ABA_STANDARDPOOL< BaseType, CoType >::increase (int *size*) `[virtual]`

Enlarges the pool to store.

To avoid fatal errors we do not allow decreasing the size of the pool.

Parameters:

size The new size of the pool.

Reimplemented in ABA_NONDUPLPOOL< BaseType, CoType >.

6.30.3.4 template<class BaseType, class CoType> virtual ABA_POOLSLOT<BaseType,CoType>∗ ABA_STANDARDPOOL< BaseType, CoType >::insert (BaseType ∗ *cv*) `[virtual]`

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.

Returns:

A pointer to the pool slot where the item has been inserted, or 0 if the insertion failed.

Parameters:

cv The constraint/variable being inserted.

Implements ABA_POOL< BaseType, CoType >.

Reimplemented in ABA_NONDUPLPOOL< BaseType, CoType >.

6.30.3.5 template<class BaseType, class CoType> const ABA_STANDARDPOOL& ABA_STANDARDPOOL< BaseType, CoType >::operator= (const ABA_STANDARDPOOL< BaseType, CoType > & *rhs*) `[private]`

6.30.3.6 template<class BaseType, class CoType> virtual void ABA_STANDARDPOOL< BaseType, CoType >::putSlot (ABA_POOLSLOT< BaseType, CoType > ∗ *slot*) `[protected, virtual]`

Inserts the *slot* in the list of free slots.

It is an error to insert a slot which is not empty.

This function defines the pure virtual function of the base class ABA_POOL.

Implements ABA_POOL< BaseType, CoType >.

6.30.3.7 template<class BaseType, class CoType> int ABA_STANDARDPOOL< BaseType, CoType >::removeNonActive (int *maxRemove*) `[protected]`

Tries to remove at most *maxRemove* inactive items from the pool.

A minimum heap of the items with the reference counter as key is built up and items are removed in this order.

6.30.3.8 template<class BaseType, class CoType> virtual int ABA_STANDARDPOOL< BaseType, CoType >::separate (double ∗ *x*, ABA_ACTIVE< CoType, BaseType > ∗ *active*, ABA_SUB ∗ *sub*, ABA_CUTBUFFER< BaseType, CoType > ∗ *cutBuffer*, double *minAbsViolation* = `0.001`, int *ranking* = `0`) `[virtual]`

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.

Returns:

The number of violated items.

Parameters:

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 3 the function ABA_CONVAR::rank() is used, if 0 no rank is associated with the item.

Implements ABA_POOL< BaseType, CoType >.

6.30.3.9 template<class BaseType, class CoType> int ABA_STANDARDPOOL< BaseType, CoType >::size () const

Returns:

The maximal number of constraints/variables that can be inserted in the pool.

6.30.3.10 template<class BaseType, class CoType> ABA_POOLSLOT<BaseType,CoType>∗ ABA_STANDARDPOOL< BaseType, CoType >::slot (int *i*)

Returns:

A pointer to the *i-th* slot in the pool.

Parameters:

i The number of the slot being accessed.

6.30.4 Friends And Related Function Documentation

6.30.4.1 template<class BaseType, class CoType> ostream& operator<< (ostream & *out*, const ABA_STANDARDPOOL< BaseType, CoType > & *rhs*) `[friend]`

The output operator calls the output operator of each item of a non-void pool slot.

Returns:
A reference to the output stream.

Parameters:
out The output stream.

rhs The pool being output.

6.30.5 Member Data Documentation

6.30.5.1 template<class BaseType, class CoType> bool ABA_STANDARDPOOL< BaseType, CoType >::autoRealloc_ `[protected]`

If the pool becomes full and this member is *true*, then an automatic reallocation is performed.

Definition at line 245 of file standardpool.h.

6.30.5.2 template<class BaseType, class CoType> ABA_LIST<ABA_POOLSLOT<BaseType,CoType> ∗> ABA_STANDARDPOOL< BaseType, CoType >::freeSlots_ `[protected]`

The linked lists of unused slots.

Definition at line 239 of file standardpool.h.

6.30.5.3 template<class BaseType, class CoType> ABA_ARRAY<ABA_POOLSLOT<BaseType,CoType> ∗> ABA_STANDARDPOOL< BaseType, CoType >::pool_ `[protected]`

The array with the pool slots.

Definition at line 235 of file standardpool.h.

The documentation for this class was generated from the following file:

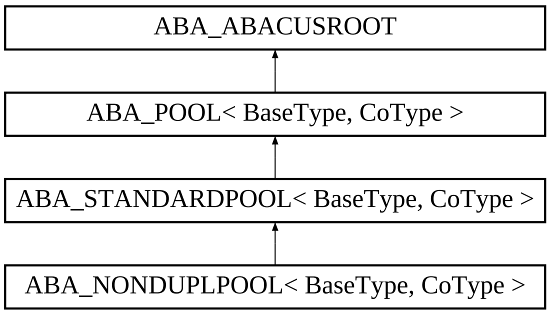
- Include/abacus/standardpool.h

6.31 ABA_NONDUPLPOOL< BaseType, CoType > Class Template Reference

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

`#include <nonduplpool.h>`

Inheritance diagram for ABA_NONDUPLPOOL< BaseType, CoType >::


ABA_ABACUSROOT
ABA_POOL< BaseType, CoType >
ABA_STANDARDPOOL< BaseType, CoType >
ABA_NONDUPLPOOL< BaseType, CoType >

Public Member Functions

- ABA_NONDUPLPOOL (ABA_MASTER ∗master, int size, bool autoRealloc=false)
- virtual ∼ABA_NONDUPLPOOL ()

  The destructor.

- virtual ABA_POOLSLOT< BaseType, CoType > ∗ insert (BaseType ∗cv)

  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.*

- virtual ABA_POOLSLOT< BaseType, CoType > ∗ present (BaseType ∗cv)
- virtual void increase (int size)
- void statistics (int &nDuplications, int &nCollisions) const

  Determines the number of constraints that have not been inserted into the pool, because an equivalent was already present.

Private Member Functions

- virtual int softDeleteConVar (ABA_POOLSLOT< BaseType, CoType > ∗slot)

  Has to be redefined because the slot has to be removed from the hash table if the constraint/variable can be deleted.

- virtual void hardDeleteConVar (ABA_POOLSLOT< BaseType, CoType > ∗slot)
- ABA_NONDUPLPOOL (const ABA_NONDUPLPOOL &rhs)
- const ABA_NONDUPLPOOL & operator= (const ABA_NONDUPLPOOL &rhs)

Private Attributes

- ABA_HASH< unsigned, ABA_POOLSLOT< BaseType, CoType > ∗ > hash_
- int nDuplications_

6.31.1 Detailed Description

template<class BaseType, class CoType> class ABA_NONDUPLPOOL< BaseType, CoType >

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

Parameters:

hash_ A hash table for a fast access to the pool slot storing a constraint/variable.

nDuplications_ The number of insertions of constraints/variables that were rejected since the constraint/variable is stored already in the pool.

Definition at line 52 of file nonduplpool.h.

6.31.2 Constructor & Destructor Documentation

6.31.2.1 template<class BaseType, class CoType> ABA_NONDUPLPOOL< BaseType, CoType >::ABA_NONDUPLPOOL (ABA_MASTER ∗ *master*, int *size*, bool *autoRealloc* = `false`)

The constructor for an empty pool.

Parameters:

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.

6.31.2.2 template<class BaseType, class CoType> virtual ABA_NONDUPLPOOL< BaseType, CoType >::~ABA_NONDUPLPOOL () `[virtual]`

The destructor.

6.31.2.3 template<class BaseType, class CoType> ABA_NONDUPLPOOL< BaseType, CoType >::ABA_NONDUPLPOOL (const ABA_NONDUPLPOOL< BaseType, CoType > & *rhs*) `[private]`

6.31.3 Member Function Documentation

6.31.3.1 template<class BaseType, class CoType> virtual void ABA_NONDUPLPOOL< BaseType, CoType >::hardDeleteConVar (ABA_POOLSLOT< BaseType, CoType > ∗ *slot*) `[private, virtual]`

Has to be redefined because the pool slot has to be removed from the hash table.

Parameters:

slot A pointer to the pool slot from wich the constraint/variable should be deleted.

Reimplemented from ABA_POOL< BaseType, CoType >.

6.31.3.2 template<class BaseType, class CoType> virtual void ABA_NONDUPLPOOL< BaseType, CoType >::increase (int *size*) `[virtual]`

Enlarges the pool to store.

To avoid fatal errors we do not allow decreasing the size of the pool. This function redefines the virtual function of the base class ABA_STANDARDPOOL because we have to reallocate the hash table.

Parameters:

size The new size of the pool.

Reimplemented from ABA_STANDARDPOOL< BaseType, CoType >.

6.31.3.3 template<class BaseType, class CoType> virtual ABA_POOLSLOT<BaseType, CoType>∗ ABA_NONDUPLPOOL< BaseType, CoType >::insert (BaseType ∗ *cv*) `[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.

Returns:

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.

Parameters:

cv The constraint/variable being inserted.

Reimplemented from ABA_STANDARDPOOL< BaseType, CoType >.

6.31.3.4 template<class BaseType, class CoType> const ABA_NONDUPLPOOL& ABA_NONDUPLPOOL< BaseType, CoType >::operator= (const ABA_NONDUPLPOOL< BaseType, CoType > & *rhs*) `[private]`

6.31.3.5 template<class BaseType, class CoType> virtual ABA_POOLSLOT<BaseType, CoType>∗ ABA_NONDUPLPOOL< BaseType, CoType >::present (BaseType ∗ *cv*) `[virtual]`

Checks if a constraint/variables is already contained in the pool.

Returns:

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.

Parameters:

cv A pointer to a constraint/variable for which it should be checked if an equivalent item is already contained in the pool.

6.31.3.6 template<class BaseType, class CoType> virtual int ABA_NONDUPLPOOL< BaseType, CoType >::softDeleteConVar (ABA_POOLSLOT< BaseType, CoType > ∗ *slot*) `[private, virtual]`

Has to be redefined because the slot has to be removed from the hash table if the constraint/variable can be deleted.

Returns:

0 If the constraint/variable could be deleted.
1 otherwise.

Parameters:

slot A pointer to the pool slot from wich the constraint/variable should be deleted.

Reimplemented from ABA_POOL< BaseType, CoType >.

6.31.3.7 template<class BaseType, class CoType> void ABA_NONDUPLPOOL< BaseType, CoType >::statistics (int & *nDuplications*, int & *nCollisions*) const

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.

Parameters:

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.

6.31.4 Member Data Documentation

6.31.4.1 template<class BaseType, class CoType> ABA_HASH<unsigned, ABA_POOLSLOT<BaseType, CoType>∗> ABA_NONDUPLPOOL< BaseType, CoType >::hash_ `[private]`

Definition at line 142 of file nonduplpool.h.

6.31.4.2 template<class BaseType, class CoType> int ABA_NONDUPLPOOL< BaseType, CoType >::nDuplications_ `[private]`

Definition at line 143 of file nonduplpool.h.

The documentation for this class was generated from the following file:

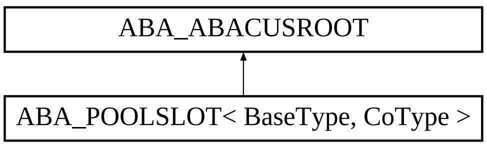
- Include/abacus/nonduplpool.h

6.32 ABA_POOLSLOT< BaseType, CoType > Class Template Reference

Constraints or variables are not directly stored in a pool. But are stored in a pool slot.

`#include <poolslot.h>`

Inheritance diagram for ABA_POOLSLOT< BaseType, CoType >::



Public Member Functions

- ABA_POOLSLOT (ABA_MASTER ∗master, ABA_POOL< BaseType, CoType > ∗pool, BaseType ∗convar=0)

  The constructor sets the version number to 1, if already a constraint is inserted in this slot, otherwise it is set to 0.

- ∼ABA_POOLSLOT ()

  The destructor for the poolslot must not be called if there are references to its constraint/variable.

- BaseType ∗ conVar () const

Private Member Functions

- void insert (BaseType ∗convar)
- int softDelete ()
- void hardDelete ()
- void removeConVarFromPool ()
- unsigned long version () const
- ABA_MASTER ∗ master ()
- ABA_POOLSLOT (const ABA_POOLSLOT< BaseType, CoType > &rhs)
- const ABA_POOLSLOT< BaseType, CoType > & operator= (const ABA_POOLSLOT< BaseType, CoType > &rhs)

Private Attributes

- ABA_MASTER ∗ master_
- BaseType ∗ conVar_
- unsigned long version_
- ABA_POOL< BaseType, CoType > ∗ pool_

Friends

- class ABA_POOLSLOTREF< BaseType, CoType >
- class ABA_POOL< BaseType, CoType >
- class ABA_STANDARDPOOL< BaseType, CoType >
- class ABA_CUTBUFFER< BaseType, CoType >
- class ABA_SUB
- class ABA_POOLSLOTREF< ABA_CONSTRAINT, ABA_VARIABLE >
- class ABA_POOLSLOTREF< ABA_VARIABLE, ABA_CONSTRAINT >
- class ABA_POOL< ABA_CONSTRAINT, ABA_VARIABLE >
- class ABA_POOL< ABA_VARIABLE, ABA_CONSTRAINT >
- class ABA_STANDARDPOOL< ABA_CONSTRAINT, ABA_VARIABLE >
- class ABA_STANDARDPOOL< ABA_VARIABLE, ABA_CONSTRAINT >
- class ABA_NONDUPLPOOL< ABA_CONSTRAINT, ABA_VARIABLE >
- class ABA_NONDUPLPOOL< ABA_VARIABLE, ABA_CONSTRAINT >
- class ABA_CUTBUFFER< ABA_CONSTRAINT, ABA_VARIABLE >
- class ABA_CUTBUFFER< ABA_VARIABLE, ABA_CONSTRAINT >

6.32.1 Detailed Description

template<class BaseType, class CoType> class ABA_POOLSLOT< BaseType, CoType >

Constraints or variables are not directly stored in a pool. But are stored in a pool slot.

Definition at line 76 of file poolslot.h.

6.32.2 Constructor & Destructor Documentation

6.32.2.1 template<class BaseType, class CoType> ABA_POOLSLOT< BaseType, CoType >::ABA_POOLSLOT (ABA_MASTER ∗ *master*, ABA_POOL< BaseType, CoType > ∗ *pool*, BaseType ∗ *convar* = `0`)

The constructor sets the version number to 1, if already a constraint is inserted in this slot, otherwise it is set to 0.

Parameters:

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.

6.32.2.2 template<class BaseType, class CoType> ABA_POOLSLOT< BaseType, CoType >::~ABA_POOLSLOT ()

The destructor for the poolslot must not be called if there are references to its constraint/variable.

6.32.2.3 template<class BaseType, class CoType> ABA_POOLSLOT< BaseType, CoType >::ABA_POOLSLOT (const ABA_POOLSLOT< BaseType, CoType > & *rhs*) `[private]`

6.32.3 Member Function Documentation

6.32.3.1 template<class BaseType, class CoType> BaseType∗ ABA_POOLSLOT< BaseType, CoType >::conVar () const

Returns:
A pointer to the constraint/variable in the pool slot.

6.32.3.2 template<class BaseType, class CoType> void ABA_POOLSLOT< BaseType, CoType >::hardDelete () `[private]`

Deletes the constraint/variable in the slot.

Warning:
This function should be used very carefully.

6.32.3.3 template<class BaseType, class CoType> void ABA_POOLSLOT< BaseType, CoType >::insert (BaseType ∗ *convar*) `[private]`

Inserts a constraint/variable in the slot, and updates the version number.

If the slot still contains a constraint, the program stops.

The constant *ULONG_MAX* is defined in the file { limits.h}.

Parameters:
conVar The constraint/variable that is inserted.

6.32.3.4 template<class BaseType, class CoType> ABA_MASTER∗ ABA_POOLSLOT< BaseType, CoType >::master () `[private]`

Returns:
A pointer to the corresponding master of the optimization.

6.32.3.5 template<class BaseType, class CoType> const ABA_POOLSLOT<BaseType, CoType>& ABA_POOLSLOT< BaseType, CoType >::operator= (const ABA_POOLSLOT< BaseType, CoType > & *rhs*) `[private]`

6.32.3.6 template<class BaseType, class CoType> void ABA_POOLSLOT< BaseType, CoType >::removeConVarFromPool () `[private]`

Removes the constraint contained in this ABA_POOLSLOT from its own ABA_POOL.

6.32.3.7 template<class BaseType, class CoType> int ABA_POOLSLOT< BaseType, CoType >::softDelete () `[private]`

Tries to remove the item from the slot.

This is possible if the function ABA_CONVAR::deletable() returns *true*.

Returns:

0 If the constraint/variable in the slot could be deleted,
1 otherwise.

6.32.3.8 template<class BaseType, class CoType> unsigned long ABA_POOLSLOT< BaseType, CoType >::version () const `[private]`

Returns:

The version number of the constraint/variable of the slot.

6.32.4 Friends And Related Function Documentation

6.32.4.1 template<class BaseType, class CoType> friend class ABA_CUTBUFFER< ABA_CONSTRAINT, ABA_VARIABLE > `[friend]`

Definition at line 91 of file poolslot.h.

6.32.4.2 template<class BaseType, class CoType> friend class ABA_CUTBUFFER< ABA_VARIABLE, ABA_CONSTRAINT > `[friend]`

Definition at line 92 of file poolslot.h.

6.32.4.3 template<class BaseType, class CoType> friend class ABA_CUTBUFFER< BaseType, CoType > `[friend]`

Definition at line 80 of file poolslot.h.

6.32.4.4 template<class BaseType, class CoType> friend class ABA_NONDUPLPOOL< ABA_CONSTRAINT, ABA_VARIABLE > `[friend]`

Definition at line 89 of file poolslot.h.

6.32.4.5 template<class BaseType, class CoType> friend class ABA_NONDUPLPOOL< ABA_VARIABLE, ABA_CONSTRAINT > `[friend]`

Definition at line 90 of file poolslot.h.

6.32.4.6 template<class BaseType, class CoType> friend class ABA_POOL< ABA_CONSTRAINT, ABA_VARIABLE > `[friend]`

Definition at line 85 of file poolslot.h.

6.32.4.7 template<class BaseType, class CoType> friend class ABA_POOL< ABA_VARIABLE, ABA_CONSTRAINT > `[friend]`

Definition at line 86 of file poolslot.h.

6.32.4.8 template<class BaseType, class CoType> friend class ABA_POOL< BaseType, CoType > `[friend]`

Definition at line 78 of file poolslot.h.

6.32.4.9 template<class BaseType, class CoType> friend class ABA_POOLSLOTREF< ABA_CONSTRAINT, ABA_VARIABLE > `[friend]`

Definition at line 83 of file poolslot.h.

6.32.4.10 template<class BaseType, class CoType> friend class ABA_POOLSLOTREF< ABA_VARIABLE, ABA_CONSTRAINT > `[friend]`

Definition at line 84 of file poolslot.h.

6.32.4.11 template<class BaseType, class CoType> friend class ABA_POOLSLOTREF< BaseType, CoType > `[friend]`

Definition at line 77 of file poolslot.h.

6.32.4.12 template<class BaseType, class CoType> friend class ABA_STANDARDPOOL< ABA_CONSTRAINT, ABA_VARIABLE > `[friend]`

Definition at line 87 of file poolslot.h.

6.32.4.13 template<class BaseType, class CoType> friend class ABA_STANDARDPOOL< ABA_VARIABLE, ABA_CONSTRAINT > [friend]

Definition at line 88 of file poolslot.h.

6.32.4.14 template<class BaseType, class CoType> friend class ABA_STANDARDPOOL< BaseType, CoType > [friend]

Definition at line 79 of file poolslot.h.

6.32.4.15 template<class BaseType, class CoType> friend class ABA_SUB [friend]

Definition at line 82 of file poolslot.h.

6.32.5 Member Data Documentation

6.32.5.1 template<class BaseType, class CoType> BaseType* ABA_POOLSLOT< BaseType, CoType >::conVar_ [private]

A pointer to the constraint/variable.

Definition at line 186 of file poolslot.h.

6.32.5.2 template<class BaseType, class CoType> ABA_MASTER* ABA_POOLSLOT< BaseType, CoType >::master_ [private]

A pointer to the corresponding master of the optimization.

Definition at line 182 of file poolslot.h.

6.32.5.3 template<class BaseType, class CoType> ABA_POOL<BaseType, CoType>* ABA_POOLSLOT< BaseType, CoType >::pool_ [private]

A pointer to the corresponding pool.

Definition at line 194 of file poolslot.h.

6.32.5.4 template<class BaseType, class CoType> unsigned long ABA_POOLSLOT< BaseType, CoType >::version_ [private]

The version of the constraint in the slot.

Definition at line 190 of file poolslot.h.

The documentation for this class was generated from the following file:

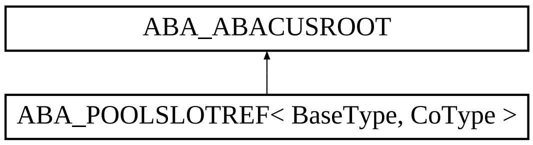
- Include/abacus/poolslot.h

6.33 ABA_POOLSLOTREF< BaseType, CoType > Class Template Reference

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.

```
#include <poolslotref.h>
```

Inheritance diagram for ABA_POOLSLOTREF< BaseType, CoType >::



Public Member Functions

- ABA_POOLSLOTREF (ABA_MASTER ∗master)
- ABA_POOLSLOTREF (ABA_POOLSLOT< BaseType, CoType > ∗slot)
- ABA_POOLSLOTREF (const ABA_POOLSLOTREF< BaseType, CoType > &rhs)

  The copy constructor may increments the reference counter of the constraint/variable only if version number of the slot and version number of the reference are equal, since otherwise this is not a correct reference to slot_->conVar().

- ∼ABA_POOLSLOTREF ()

  The destructor sends a message to the constraint that it will no longer be referred from this place in the program.

- BaseType ∗ conVar () const
- unsigned long version () const
- ABA_POOLSLOT< BaseType, CoType > ∗ slot () const
- void slot (ABA_POOLSLOT< BaseType, CoType > ∗s)

  This version of the function slot() *initializes the referenced pool slot.*

Private Member Functions

- void printDifferentVersionError () const
- const ABA_POOLSLOTREF< BaseType, CoType > & operator= (const ABA_POOLSLOTREF< BaseType, CoType > &rhs)

Private Attributes

- ABA_MASTER ∗ master_
- ABA_POOLSLOT< BaseType, CoType > ∗ slot_
- unsigned long version_

  The version number of the slot at construction/initialization time of this reference.

Friends

- ostream & operator<< (ostream &out, const ABA_POOLSLOTREF &rhs)
  The output operator writes the constraint/variable stored in the referenced slot to an output stream.

6.33.1 Detailed Description

template<class BaseType, class CoType> class ABA_POOLSLOTREF< BaseType, CoType >

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.

Definition at line 54 of file poolslotref.h.

6.33.2 Constructor & Destructor Documentation

6.33.2.1 template<class BaseType, class CoType> ABA_POOLSLOTREF< BaseType, CoType >::ABA_POOLSLOTREF (ABA_MASTER * *master*)

This constructor generates an object referencing to no pool slot.

Parameters:
master A pointer to the corresponding master of the optimization.

6.33.2.2 template<class BaseType, class CoType> ABA_POOLSLOTREF< BaseType, CoType >::ABA_POOLSLOTREF (ABA_POOLSLOT< BaseType, CoType > * *slot*)

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.

Parameters:
slot The pool slot that is referenced now.

6.33.2.3 template<class BaseType, class CoType> ABA_POOLSLOTREF< BaseType, CoType >::ABA_POOLSLOTREF (const ABA_POOLSLOTREF< BaseType, CoType > & *rhs*)

The copy constructor may increments the reference counter of the constraint/variable only if version number of the slot and version number of the reference are equal, since otherwise this is not a correct reference to *slot_->conVar()*.

Parameters:
rhs The pool slot that is copied in the initialization process.

6.33.2.4 template<class BaseType, class CoType> ABA_POOLSLOTREF< BaseType, CoType >::~ABA_POOLSLOTREF ()

The destructor sends a message to the constraint that it will no longer be referred from this place in the program.

If the version number of the reference and the version number of the slot do not equal, we must not decrement the reference counter of *slot_->conVar()* because this is not a correct reference to this constraint/variable.

6.33.3 Member Function Documentation

6.33.3.1 template<class BaseType, class CoType> BaseType∗ ABA_POOLSLOTREF< BaseType, CoType >::conVar () const

Returns:

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.

6.33.3.2 template<class BaseType, class CoType> const ABA_POOLSLOTREF<BaseType, CoType>& ABA_POOLSLOTREF< BaseType, CoType >::operator= (const ABA_POOLSLOTREF< BaseType, CoType > & *rhs*) `[private]`

6.33.3.3 template<class BaseType, class CoType> void ABA_POOLSLOTREF< BaseType, CoType >::printDifferentVersionError () const `[private]`

6.33.3.4 template<class BaseType, class CoType> void ABA_POOLSLOTREF< BaseType, CoType >::slot (ABA_POOLSLOT< BaseType, CoType > ∗ *s*)

This version of the function *slot()* initializes the referenced pool slot.

The function *slot()* may decrement the reference counter of *slot_->conVar()* only if the version number of the reference and the version number of the slot are equal since otherwise this is not a valid reference.

Parameters:

s The new slot that is referenced. This must not be a 0-pointer.

6.33.3.5 template<class BaseType, class CoType> ABA_POOLSLOT<BaseType, CoType>∗ ABA_POOLSLOTREF< BaseType, CoType >::slot () const

Returns:

A pointer to the referenced slot.

6.33.3.6 template<class BaseType, class CoType> unsigned long ABA_POOLSLOTREF< BaseType, CoType >::version () const

Returns:
The version number of the constraint/variable stored in the referenced slot at construction time of the reference to this slot.

6.33.4 Friends And Related Function Documentation

6.33.4.1 template<class BaseType, class CoType> ostream& operator<< (ostream & *out*, const ABA_POOLSLOTREF< BaseType, CoType > & *rhs*) `[friend]`

The output operator writes the constraint/variable stored in the referenced slot to an output stream.

Returns:
A reference to the output stream.

Parameters:

out The output stream.

rhs The reference to a pool slot being output.

6.33.5 Member Data Documentation

6.33.5.1 template<class BaseType, class CoType> ABA_MASTER* ABA_POOLSLOTREF< BaseType, CoType >::master_ `[private]`

A pointer to the corresponding master of the optimization.

Definition at line 151 of file poolslotref.h.

6.33.5.2 template<class BaseType, class CoType> ABA_POOLSLOT<BaseType, CoType>* ABA_POOLSLOTREF< BaseType, CoType >::slot_ `[private]`

A pointer to the referenced pool slot.

Definition at line 155 of file poolslotref.h.

6.33.5.3 template<class BaseType, class CoType> unsigned long ABA_POOLSLOTREF< BaseType, CoType >::version_ `[private]`

The version number of the slot at construction/initialization time of this reference.

Definition at line 160 of file poolslotref.h.

The documentation for this class was generated from the following file:

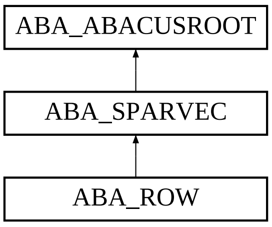
- Include/abacus/poolslotref.h

6.34 ABA_ROW Class Reference

class refines its base class ABA_SPARVEC for the representation of constraints in the row format

`#include <row.h>`

Inheritance diagram for ABA_ROW::



Public Member Functions

- 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)

  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_GLOBAL ∗glob, int nnz, int ∗s, double ∗c, ABA_CSENSE::SENSE sense, double r)

  This is also an equivalent constructor except that s *and* c *are C-style arrays.*

- ABA_ROW (ABA_GLOBAL ∗glob, int size)
- ∼ABA_ROW ()

  The destructor.

- double rhs () const
- void rhs (double r)

  This version of rhs() *sets the right hand side of the row.*

- ABA_CSENSE ∗ sense ()
- void sense (ABA_CSENSE &s)

  This version of sense() *sets the sense of the row.*

- void sense (ABA_CSENSE::SENSE s)

  And another version of sense() *to set the sense of the row.*

- void copy (const ABA_ROW &row)

    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 delInd (ABA_BUFFER< int > &buf, double rhsDelta)

    Removes the indices listed in buf *from the support of the row and subtracts* rhsDelta *from its right hand side.*

Protected Attributes

- ABA_CSENSE sense_
- double rhs_

Friends

- ostream & operator<< (ostream &out, const ABA_ROW &rhs)

    The output operator writes the row on an output stream in format like { -2.5 x1 + 3 x3 <= 7}.

6.34.1 Detailed Description

class refines its base class ABA_SPARVEC for the representation of constraints in the row format

Definition at line 48 of file row.h.

6.34.2 Constructor & Destructor Documentation

6.34.2.1 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*)

A constructor.

Parameters:

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.

6.34.2.2 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*)

This is an equivalent constructor using ABA_CSENSE::SENSE instead of an object of the class *SENSE* to initialize the sense of the constraint.

6.34.2.3 ABA_ROW::ABA_ROW (ABA_GLOBAL $*$ *glob*, int *nnz*, int $*$ *s*, double $*$ *c*, ABA_CSENSE::SENSE *sense*, double *r*)

This is also an equivalent constructor except that *s* and *c* are C-style arrays.

6.34.2.4 ABA_ROW::ABA_ROW (ABA_GLOBAL $*$ *glob*, int *size*)

A constructor without initialization of the nonzeros of the row.

Parameters:

glob A pointer to the corresponding global object.

size The maximal numbers of nonzeros.

6.34.2.5 ABA_ROW::~ABA_ROW ()

The destructor.

6.34.3 Member Function Documentation

6.34.3.1 void ABA_ROW::copy (const ABA_ROW & *row*)

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*.

Parameters:

row The row that is copied.

6.34.3.2 void ABA_ROW::delInd (ABA_BUFFER$<$ int $>$ & *buf*, double *rhsDelta*)

Removes the indices listed in *buf* from the support of the row and subtracts *rhsDelta* from its right hand side.

Parameters:

buf The components being removed from the row.

rhsDelta The correction of the right hand side of the row.

6.34.3.3 void ABA_ROW::rhs (double *r*) `[inline]`

This version of *rhs()* sets the right hand side of the row.

Parameters:

r The new value of the right hand side.

Definition at line 195 of file row.h.

6.34.3.4 double ABA_ROW::rhs () const `[inline]`

Returns:

The right hand side stored in the row format.

Definition at line 190 of file row.h.

6.34.3.5 void ABA_ROW::sense (ABA_CSENSE::SENSE *s*) `[inline]`

And another version of *sense()* to set the sense of the row.

Parameters:

s The new sense of the row.

Definition at line 210 of file row.h.

6.34.3.6 void ABA_ROW::sense (ABA_CSENSE & *s*) `[inline]`

This version of *sense()* sets the sense of the row.

Parameters:

s The new sense of the row.

Definition at line 205 of file row.h.

6.34.3.7 ABA_CSENSE ∗ ABA_ROW::sense () `[inline]`

Returns:

A pointer to the sense of the row.

Definition at line 200 of file row.h.

6.34.4 Friends And Related Function Documentation

6.34.4.1 ostream& operator<< (ostream & *out*, const ABA_ROW & *rhs*) `[friend]`

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.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The row being output.

6.34.5 Member Data Documentation

6.34.5.1 double ABA_ROW::rhs_ `[protected]`

The right hand side of the row.

Definition at line 186 of file row.h.

6.34.5.2 ABA_CSENSE ABA_ROW::sense_ `[protected]`

The sense of the row.

Definition at line 182 of file row.h.

The documentation for this class was generated from the following file:

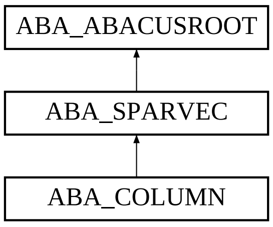
- Include/abacus/row.h

6.35 ABA_COLUMN Class Reference

class ABA_COLUMN refines ABA_SPARVEC for the representation of variables in column format.

`#include <column.h>`

Inheritance diagram for ABA_COLUMN::



Public Member Functions

- 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 ()
- double obj () const
- void obj (double c)

  This version of the function obj() sets the objective function coefficient of the column.

- double lBound () const
- void lBound (double l)

  This version of the function lBound() sets the lower bound of the column.

- double uBound () const
- void uBound (double u)

  This version of the function uBound() sets the upper bound of the column.

- void copy (const ABA_COLUMN &col)

  Is very similar to the assignment operator, yet the columns do not have to be of equal size. A reallocation is performed if required.

Private Attributes

- double obj_
- double lBound_
- double uBound_

Friends

- ostream & operator<< (ostream &out, const ABA_COLUMN &rhs)

6.35.1 Detailed Description

class ABA_COLUMN refines ABA_SPARVEC for the representation of variables in column format.

Definition at line 44 of file column.h.

6.35.2 Constructor & Destructor Documentation

6.35.2.1 ABA_COLUMN::ABA_COLUMN (ABA_GLOBAL ∗ *glob*, double *obj*, double *lb*, double *ub*, int *nnz*, ABA_ARRAY< int > & *s*, ABA_ARRAY< double > & *c*)

A constructor.

Parameters:

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|.

6.35.2.2 ABA_COLUMN::ABA_COLUMN (ABA_GLOBAL ∗ *glob*, int *maxNnz*)

Another constructor generating an uninitialized column.

Parameters:

glob A pointer to the corresponding global object.

maxNnz The maximal number of nonzero elements that can be stored in the row.

6.35.2.3 ABA_COLUMN::ABA_COLUMN (ABA_GLOBAL ∗ *glob*, double *obj*, double *lb*, double *ub*, ABA_SPARVEC & *vec*)

A constructor using a sparse vector for the initialization.

Parameters:

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.

6.35.2.4 ABA_COLUMN::∼ABA_COLUMN ()

6.35.3 Member Function Documentation

6.35.3.1 void ABA_COLUMN::copy (const ABA_COLUMN & *col*)

Is very similar to the assignment operator, yet the columns do not have to be of equal size. A reallocation is performed if required.

Parameters:

col The column that is copied.

6.35.3.2 void ABA_COLUMN::lBound (double *l*) `[inline]`

This version of the function *lBound()* sets the lower bound of the column.

Parameters:

l The new value of the lower bound.

Definition at line 187 of file column.h.

6.35.3.3 double ABA_COLUMN::lBound () const `[inline]`

Returns:

The lower bound of the column.

Definition at line 182 of file column.h.

6.35.3.4 void ABA_COLUMN::obj (double *c*) `[inline]`

This version of the function *obj()* sets the objective function coefficient of the column.

Parameters:

c The new value of the objective function coefficient.

Definition at line 177 of file column.h.

6.35.3.5 double ABA_COLUMN::obj () const `[inline]`

Returns:

The objective function coefficient of the column.

Definition at line 172 of file column.h.

6.35.3.6 void ABA_COLUMN::uBound (double *u*) `[inline]`

This version of the function *uBound()* sets the upper bound of the column.

Parameters:

u The new value of the upper bound.

Definition at line 197 of file column.h.

6.35.3.7 double ABA_COLUMN::uBound () const [inline]

Returns:

The upper bound of the column.

Definition at line 192 of file column.h.

6.35.4 Friends And Related Function Documentation

6.35.4.1 ostream& operator<< (ostream & *out*, const ABA_COLUMN & *rhs*) [friend]

The output operator.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The column being output.

6.35.5 Member Data Documentation

6.35.5.1 double ABA_COLUMN::lBound_ [private]

The lower bound of the column.

Definition at line 164 of file column.h.

6.35.5.2 double ABA_COLUMN::obj_ [private]

The objective function coefficient of the column.

Definition at line 160 of file column.h.

6.35.5.3 double ABA_COLUMN::uBound_ [private]

The upper bound of the column.

Definition at line 168 of file column.h.

The documentation for this class was generated from the following file:

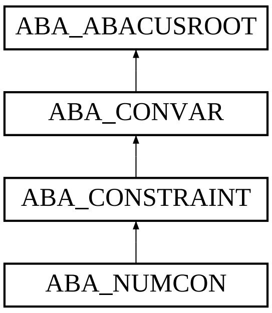
- Include/abacus/column.h

6.36 ABA_NUMCON Class Reference

Like the class ABA_NUMVAR for variables we provide the class ABA_NUMCON for constraints which are uniquely defined by an integer number.

`#include <numcon.h>`

Inheritance diagram for ABA_NUMCON::



Public Member Functions

- ABA_NUMCON (ABA_MASTER ∗master, const ABA_SUB ∗sub, ABA_CSENSE::SENSE sense, bool dynamic, bool local, bool liftable, int number, double rhs)
- virtual ∼ABA_NUMCON ()

  The destructor.

- virtual double coeff (ABA_VARIABLE ∗v)
- virtual void print (ostream &out)
- int number () const

Private Attributes

- int number_

Friends

- ostream & operator<< (ostream &out, const ABA_NUMCON &rhs)

6.36.1 Detailed Description

Like the class ABA_NUMVAR for variables we provide the class ABA_NUMCON for constraints which are uniquely defined by an integer number.

Parameters:

int number_ The identification number of the constraint.

Definition at line 38 of file numcon.h.

6.36.2 Constructor & Destructor Documentation

6.36.2.1 ABA_NUMCON::ABA_NUMCON (ABA_MASTER $*$ *master*, const ABA_SUB $*$ *sub*, ABA_CSENSE::SENSE *sense*, bool *dynamic*, bool *local*, bool *liftable*, int *number*, double *rhs*)

The constructor.

Parameters:

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.

6.36.2.2 virtual ABA_NUMCON::∼ABA_NUMCON () `[virtual]`

The destructor.

6.36.3 Member Function Documentation

6.36.3.1 virtual double ABA_NUMCON::coeff (ABA_VARIABLE $*$ *v*) `[virtual]`

Returns:

The coefficient of the variable *v*.

Parameters:

v The variable of which the coefficient is determined. It must point to an object of the class ABA_COLVAR.

Implements ABA_CONSTRAINT.

6.36.3.2 int ABA_NUMCON::number () const `[inline]`

Returns:

Returns the identification number of the constraint.

Definition at line 130 of file numcon.h.

6.36.3.3 virtual void ABA_NUMCON::print (ostream & *out*) `[virtual]`

Writes the row format of the constraint on an output stream.

It redefines the virtual function *print()* of the base class ABA_CONVAR.

Parameters:

out The output stream.

Reimplemented from ABA_CONVAR.

6.36.4 Friends And Related Function Documentation

6.36.4.1 ostream& operator<< (ostream & *out*, const ABA_NUMCON & *rhs*) `[friend]`

The output operator writes the identification number and the right hand side to an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The variable being output.

6.36.5 Member Data Documentation

6.36.5.1 int ABA_NUMCON::number_ `[private]`

Definition at line 126 of file numcon.h.

The documentation for this class was generated from the following file:

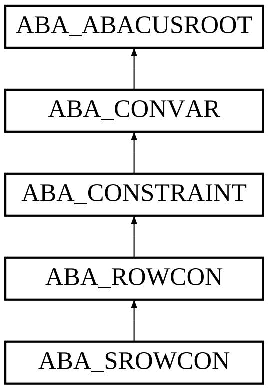
- Include/abacus/numcon.h

6.37 ABA_ROWCON Class Reference

class ABA_ROWCON implements constraints stored in the class ABA_ROW.

`#include <rowcon.h>`

Inheritance diagram for ABA_ROWCON::



Public Member Functions

- ABA_ROWCON (ABA_MASTER ∗master, const 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, const ABA_SUB ∗sub, ABA_CSENSE::SENSE sense, int nnz, int ∗support, double ∗coeff, double rhs, bool dynamic, bool local, bool liftable)

  This constructor is equivalent to the previous constructor except that it uses C-style arrays for support *and* coeff.

- virtual ∼ABA_ROWCON ()

  The destructor.

- virtual double coeff (ABA_VARIABLE ∗v)

  Computes the coefficient of a variable which must be of type ABA_NUMVAR.

- virtual void print (ostream &out)
- ABA_ROW ∗ row ()

Protected Attributes

- ABA_ROW row_

6.37.1 Detailed Description

class ABA_ROWCON implements constraints stored in the class ABA_ROW.

Definition at line 44 of file rowcon.h.

6.37.2 Constructor & Destructor Documentation

6.37.2.1 ABA_ROWCON::ABA_ROWCON (ABA_MASTER $*$ *master*, const 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*)

The constructor.

Parameters:

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.

6.37.2.2 ABA_ROWCON::ABA_ROWCON (ABA_MASTER $*$ *master*, const ABA_SUB $*$ *sub*, ABA_CSENSE::SENSE *sense*, int *nnz*, int $*$ *support*, double $*$ *coeff*, double *rhs*, bool *dynamic*, bool *local*, bool *liftable*)

This constructor is equivalent to the previous constructor except that it uses C-style arrays for *support* and *coeff*.

6.37.2.3 virtual ABA_ROWCON::$\sim$ABA_ROWCON () `[virtual]`

The destructor.

6.37.3 Member Function Documentation

6.37.3.1 virtual double ABA_ROWCON::coeff (ABA_VARIABLE $*$ *v*) `[virtual]`

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.

Returns:

The coefficient of the variable *v*.

Parameters:

v The variable of which the coefficient is determined.

Implements ABA_CONSTRAINT.

6.37.3.2 virtual void ABA_ROWCON::print (ostream & *out*) `[virtual]`

Writes the row format of the constraint on an output stream.

It redefines the virtual function *print()* of the base class ABA_CONVAR.

Parameters:

out The output stream.

Reimplemented from ABA_CONVAR.

6.37.3.3 ABA_ROW ∗ ABA_ROWCON::row () `[inline]`

Returns:

A pointer to the object of the class ABA_ROW representing the constraint.

Definition at line 158 of file rowcon.h.

6.37.4 Member Data Documentation

6.37.4.1 ABA_ROW ABA_ROWCON::row_ `[protected]`

The representation of the constraint.

Definition at line 154 of file rowcon.h.

The documentation for this class was generated from the following file:

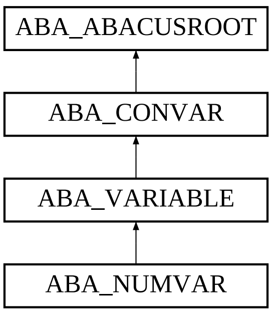
- Include/abacus/rowcon.h

6.38 ABA_NUMVAR Class Reference

class is derived from the class ABA_VARIABLE and implements a variable which is uniquely defined by a number

`#include <numvar.h>`

Inheritance diagram for ABA_NUMVAR::



Public Member Functions

- ABA_NUMVAR (ABA_MASTER ∗master, const ABA_SUB ∗sub, int number, bool dynamic, bool local, double obj, double lBound, double uBound, ABA_VARTYPE::TYPE type)
- virtual ∼ABA_NUMVAR ()

  The destructor.

- int number () const

Protected Attributes

- int number_

Friends

- ostream & operator<< (ostream &out, const ABA_NUMVAR &rhs)

6.38.1 Detailed Description

class is derived from the class ABA_VARIABLE and implements a variable which is uniquely defined by a number

Definition at line 38 of file numvar.h.

6.38.2 Constructor & Destructor Documentation

6.38.2.1 ABA_NUMVAR::ABA_NUMVAR (ABA_MASTER ∗ *master*, const ABA_SUB ∗ *sub*, int *number*, bool *dynamic*, bool *local*, double *obj*, double *lBound*, double *uBound*, ABA_VARTYPE::TYPE *type*)

The constructor.

Parameters:

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.

6.38.2.2 virtual ABA_NUMVAR::∼ABA_NUMVAR () `[virtual]`

The destructor.

6.38.3 Member Function Documentation

6.38.3.1 int ABA_NUMVAR::number () const `[inline]`

Returns:

The number of the variable.

Definition at line 133 of file numvar.h.

6.38.4 Friends And Related Function Documentation

6.38.4.1 ostream& operator<< (ostream & *out*, const ABA_NUMVAR & *rhs*) `[friend]`

Writes the number of the variable to an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The variable being output.

6.38.5 Member Data Documentation

6.38.5.1 int ABA_NUMVAR::number_ [protected]

The identification number of the variable.

Definition at line 129 of file numvar.h.

The documentation for this class was generated from the following file:

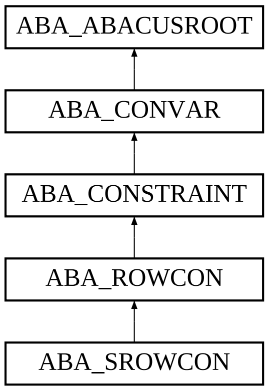
- Include/abacus/numvar.h

6.39 ABA_SROWCON Class Reference

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.

`#include <srowcon.h>`

Inheritance diagram for ABA_SROWCON::



Public Member Functions

- ABA_SROWCON (ABA_MASTER ∗master, const 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, const ABA_SUB ∗sub, ABA_CSENSE::SENSE sense, int nnz, int ∗support, double ∗coeff, double rhs, bool dynamic, bool local, bool liftable)

  This constructor is equivalent to the previous constructor except that it uses C-style arrays for support *and* coeff.

- virtual ∼ABA_SROWCON ()

  The destructor.

- virtual int genRow (ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > ∗var, ABA_ROW &row)

  Generates the row format of the constraint associated with the variable set var.

- virtual double slack (ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > ∗variables, double ∗x)

6.39.1 Detailed Description

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.

Definition at line 39 of file srowcon.h.

6.39.2 Constructor & Destructor Documentation

6.39.2.1 ABA_SROWCON::ABA_SROWCON (ABA_MASTER $*$ *master*, const 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*)

The constructor.

Parameters:

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.

6.39.2.2 ABA_SROWCON::ABA_SROWCON (ABA_MASTER $*$ *master*, const ABA_SUB $*$ *sub*, ABA_CSENSE::SENSE *sense*, int *nnz*, int $*$ *support*, double $*$ *coeff*, double *rhs*, bool *dynamic*, bool *local*, bool *liftable*)

This constructor is equivalent to the previous constructor except that it uses C-style arrays for *support* and *coeff*.

6.39.2.3 virtual ABA_SROWCON::~ABA_SROWCON () `[virtual]`

The destructor.

6.39.3 Member Function Documentation

6.39.3.1 virtual int ABA_SROWCON::genRow (ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > * *var*, ABA_ROW & *row*) `[virtual]`

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.

Returns:

It returns the number of nonzero elements in the row format.

Parameters:

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.

Reimplemented from ABA_CONSTRAINT.

6.39.3.2 virtual double ABA_SROWCON::slack (ABA_ACTIVE< ABA_VARIABLE, ABA_CONSTRAINT > * *variables*, double * *x*) `[virtual]`

Computes the slack of a vector associated with the variable set *variables*.

This function redefines a virtual function of the base class ABA_ROWCON.

Returns:

The slack of the vector *x*.

Parameters:

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.

Reimplemented from ABA_CONSTRAINT.

The documentation for this class was generated from the following file:

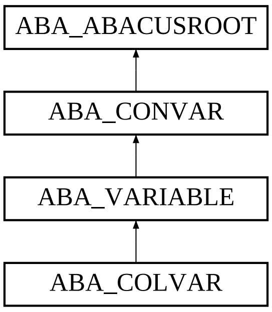
- Include/abacus/srowcon.h

6.40 ABA_COLVAR Class Reference

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.

`#include <colvar.h>`

Inheritance diagram for ABA_COLVAR::



Public Member Functions

- ABA_COLVAR (ABA_MASTER ∗master, const 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, const ABA_SUB ∗sub, bool dynamic, bool local, double lBound, double uBound, ABA_VARTYPE::TYPE varType, double obj, ABA_SPARVEC &vector)

  A constructor substituting nnz, support, *and* coeff *of the previous constructor by an object of the class ABA_SPARVEC.*

- virtual ∼ABA_COLVAR ()
- virtual void print (ostream &out)
- virtual double coeff (ABA_CONSTRAINT ∗con)
- double coeff (int i)
- ABA_COLUMN ∗ column ()

Protected Attributes

- ABA_COLUMN column_

Friends

- ostream & operator<< (ostream &out, const ABA_COLVAR &rhs)

6.40.1 Detailed Description

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.

Parameters:
ABA_COLUMN column_ The column representing the variable.

Definition at line 49 of file colvar.h.

6.40.2 Constructor & Destructor Documentation

6.40.2.1 ABA_COLVAR::ABA_COLVAR (ABA_MASTER $*$ *master*, const 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*)

The constructor.

Parameters:

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*.

6.40.2.2 ABA_COLVAR::ABA_COLVAR (ABA_MASTER $*$ *master*, const ABA_SUB $*$ *sub*, bool *dynamic*, bool *local*, double *lBound*, double *uBound*, ABA_VARTYPE::TYPE *varType*, double *obj*, ABA_SPARVEC & *vector*)

A constructor substituting *nnz*, *support*, and *coeff* of the previous constructor by an object of the class ABA_SPARVEC.

6.40.2.3 virtual ABA_COLVAR::$\sim$ABA_COLVAR () `[virtual]`

The destructor.

6.40.3 Member Function Documentation

6.40.3.1 double ABA_COLVAR::coeff (int *i*)

This version of the function *coeff()* computes the coefficient of a constraint with a given number.

Returns:

The coefficient of constraint *i*.

Parameters:

i The number of the constraint.

6.40.3.2 virtual double ABA_COLVAR::coeff (ABA_CONSTRAINT ∗ *con*) `[virtual]`

Returns:

The coefficient of the constraint *con*.

Parameters:

con The constraint of which the coefficient is computed. This must be a pointer to the class ABA_NUMCON.

Reimplemented from ABA_VARIABLE.

6.40.3.3 ABA_COLUMN∗ ABA_COLVAR::column ()

Returns:

A pointer to the column representing the variable.

6.40.3.4 virtual void ABA_COLVAR::print (ostream & *out*) `[virtual]`

Writes the column representing the variable to an output stream.

It redefines the virtual function *print()* of the base class ABA_CONVAR.

Parameters:

out The output stream.

Reimplemented from ABA_CONVAR.

6.40.4 Friends And Related Function Documentation

6.40.4.1 ostream& operator<< (ostream & *out*, const ABA_COLVAR & *rhs*) `[friend]`

The output operator writes the column representing the variable to an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The variable being output.

6.40.5 Member Data Documentation

6.40.5.1 ABA_COLUMN ABA_COLVAR::column_ `[protected]`

Definition at line 161 of file colvar.h.

The documentation for this class was generated from the following file:

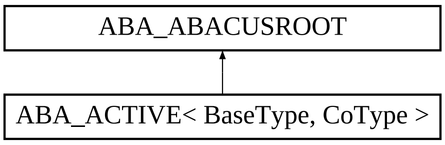
- Include/abacus/colvar.h

6.41 ABA_ACTIVE< BaseType, CoType > Class Template Reference

template class implements the sets of act ive constraints and variables which are associated w ith each subproblem

`#include <active.h>`

Inheritance diagram for ABA_ACTIVE< BaseType, CoType >::



Public Member Functions

- ABA_ACTIVE (ABA_MASTER ∗master, int max)
- ABA_ACTIVE (ABA_MASTER ∗master, ABA_ACTIVE ∗a, int max)

  In addition to the previous constructor, this constructor initializes the active set.

- ABA_ACTIVE (const ABA_ACTIVE< BaseType, CoType > &rhs)
- ∼ABA_ACTIVE ()

  The destructor.

- 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)

  Is overloaded that also several items can be added at the same time.

- void remove (ABA_BUFFER< int > &del)

  Removes items from the list of active items.

- void realloc (int newSize)

  Changes the maximum number of active items which can be stored in an object of this class.

- int redundantAge (int i) const
- void incrementRedundantAge (int i)

  Increments the number of iterations the item i *is already redundant by 1.*

- void resetRedundantAge (int i)

Private Member Functions

- const ABA_ACTIVE< BaseType, CoType > & operator= (const ABA_ACTIVE< BaseType, CoType > &rhs)

Private Attributes

- ABA_MASTER ∗ master_
- int n_
- ABA_ARRAY< ABA_POOLSLOTREF< BaseType, CoType > ∗ > active_
- ABA_ARRAY< int > redundantAge_

Friends

- ostream & operator<< (ostream &out, const ABA_ACTIVE< BaseType, CoType > &rhs)

  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.*

6.41.1 Detailed Description

template<class BaseType, class CoType> class ABA_ACTIVE< BaseType, CoType >

template class implements the sets of act ive constraints and variables which are associated w ith each subproblem

Definition at line 62 of file active.h.

6.41.2 Constructor & Destructor Documentation

6.41.2.1 template<class BaseType, class CoType> ABA_ACTIVE< BaseType, CoType >::ABA_ACTIVE (ABA_MASTER ∗ *master*, int *max*)

The constructor.

Parameters:

master A pointer to the corresponding master of the optimization.

max The maximal number of active constraints/variables.

6.41.2.2 template<class BaseType, class CoType> ABA_ACTIVE< BaseType, CoType >::ABA_ACTIVE (ABA_MASTER ∗ *master*, ABA_ACTIVE< BaseType, CoType > ∗ *a*, int *max*)

In addition to the previous constructor, this constructor initializes the active set.

Parameters:

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.

6.41.2.3 template<class BaseType, class CoType> ABA_ACTIVE< BaseType, CoType >::ABA_ACTIVE (const ABA_ACTIVE< BaseType, CoType > & *rhs*)

The copy constructor.

Parameters:

rhs The active set that is copied.

6.41.2.4 template<class BaseType, class CoType> ABA_ACTIVE< BaseType, CoType >::∼ABA_ACTIVE ()

The destructor.

6.41.3 Member Function Documentation

6.41.3.1 template<class BaseType, class CoType> void ABA_ACTIVE< BaseType, CoType >::incrementRedundantAge (int *i*)

Increments the number of iterations the item *i* is already redundant by 1.

Parameters:

i The index of the constraint/variable.

6.41.3.2 template<class BaseType, class CoType> void ABA_ACTIVE< BaseType, CoType >::insert (ABA_BUFFER< ABA_POOLSLOT< BaseType, CoType > ∗ > & *ps*)

Is overloaded that also several items can be added at the same time.

Parameters:

ps The buffer storing the pool slots of all constraints/variables that are added.

6.41.3.3 template< class BaseType, class CoType > void ABA_ACTIVE< BaseType, CoType >::insert (ABA_POOLSLOT< BaseType, CoType > ∗ *ps*)

Adds a constraint/variable to the active items.

Parameters:
ps The pool slot storing the constraint/variable being added.

6.41.3.4 template< class BaseType, class CoType > int ABA_ACTIVE< BaseType, CoType >::max () const

Returns:
The maximum number of storable active items (without reallocation).

6.41.3.5 template< class BaseType, class CoType > int ABA_ACTIVE< BaseType, CoType >::number () const

Returns:
The current number of active items.

6.41.3.6 template< class BaseType, class CoType > const ABA_ACTIVE<BaseType, CoType>& ABA_ACTIVE< BaseType, CoType >::operator= (const ABA_ACTIVE< BaseType, CoType > & *rhs*) `[private]`

6.41.3.7]

template< class BaseType, class CoType > BaseType∗ ABA_ACTIVE< BaseType, CoType >::operator[ ] (int *i*)

The operator [].

Returns:
A pointer to the *i-th* active item or
0 if this item has been removed in the meantime.

Parameters:
i The number of the active item.

6.41.3.8 template<class BaseType, class CoType> ABA_POOLSLOTREF<BaseType, CoType>∗ ABA_ACTIVE< BaseType, CoType >::poolSlotRef (int *i*)

Returns:

The *i-th* entry in the ABA_ARRAY *active*.

Parameters:

i The number of the active item.

6.41.3.9 template<class BaseType, class CoType> void ABA_ACTIVE< BaseType, CoType >::realloc (int *newSize*)

Changes the maximum number of active items which can be stored in an object of this class.

Parameters:

newSize The new maximal number of active items.

6.41.3.10 template<class BaseType, class CoType> int ABA_ACTIVE< BaseType, CoType >::redundantAge (int *i*) const

Returns:

The number of iterations a constraint/variable is already redundant.

6.41.3.11 template<class BaseType, class CoType> void ABA_ACTIVE< BaseType, CoType >::remove (ABA_BUFFER< int > & *del*)

Removes items from the list of active items.

Parameters:

del The numbers of the items that should be removed. These numbers must be upward sorted.

6.41.3.12 template<class BaseType, class CoType> void ABA_ACTIVE< BaseType, CoType >::resetRedundantAge (int *i*)

the number of iterations item *i* is redundant to 0.

Parameters:

i The index of the constraint/variable.

6.41.4 Friends And Related Function Documentation

6.41.4.1 template<class BaseType, class CoType> ostream& operator<< (ostream & *out*, const ABA_ACTIVE< BaseType, CoType > & *rhs*) `[friend]`

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.

Returns:
A reference to the output stream.

Parameters:
out The output stream.

rhs The active set being output.

6.41.5 Member Data Documentation

6.41.5.1 template<class BaseType, class CoType> ABA_ARRAY<ABA_POOLSLOTREF<BaseType, CoType> *> ABA_ACTIVE< BaseType, CoType >::active_ `[private]`

The array storing references to the pool slots of the active items.

Definition at line 263 of file active.h.

6.41.5.2 template<class BaseType, class CoType> ABA_MASTER* ABA_ACTIVE< BaseType, CoType >::master_ `[private]`

A pointer to corresponding master of the optimization.

Definition at line 256 of file active.h.

6.41.5.3 template<class BaseType, class CoType> int ABA_ACTIVE< BaseType, CoType >::n_ `[private]`

The number of active items.

Definition at line 259 of file active.h.

6.41.5.4 template<class BaseType, class CoType> ABA_ARRAY<int> ABA_ACTIVE< BaseType, CoType >::redundantAge_ `[private]`

The number of iterations a constraint is already redundant.

Definition at line 267 of file active.h.

The documentation for this class was generated from the following file:

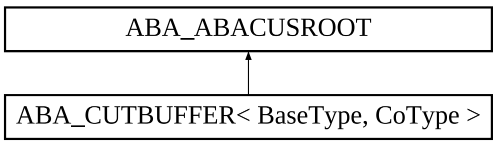
- Include/abacus/active.h

6.42 ABA_CUTBUFFER< BaseType, CoType > Class Template Reference

template class implements a buffer for constraints and variables which are generated during the cutting plane or column generation phase.

```
#include <cutbuffer.h>
```

Inheritance diagram for ABA_CUTBUFFER< BaseType, CoType >::



Public Member Functions

- 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)

  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().

- void remove (ABA_BUFFER< int > &index)
- ABA_POOLSLOT< BaseType, CoType > ∗ slot (int i)

Private Member Functions

- void extract (int max, ABA_BUFFER< ABA_POOLSLOT< BaseType, CoType > ∗ > &newSlots)
- void sort (int threshold)
- ABA_CUTBUFFER (const ABA_CUTBUFFER< BaseType, CoType > &rhs)
- const ABA_CUTBUFFER< BaseType, CoType > & operator= (const ABA_CUTBUFFER< BaseType, CoType > &rhs)

Private Attributes

- ABA_MASTER ∗ master_
- int n_
- ABA_ARRAY< ABA_POOLSLOTREF< BaseType, CoType > ∗ > psRef_
- ABA_ARRAY< bool > keepInPool_

  If keepInPool_*[i] is* true *for a buffered constraint/variables, then it is not removed from its pool although it might be discarded in* extract().

- ABA_ARRAY< double > rank_

- bool ranking_

  This flag is true *if a rank for each buffered item is available. As soon as an item without rank is inserted it becomes* false.

Friends

- class ABA_SUB

6.42.1 Detailed Description

template<class BaseType, class CoType> class ABA_CUTBUFFER< BaseType, CoType >

template class implements a buffer for constraints and variables which are generated during the cutting plane or column generation phase.

Definition at line 49 of file cutbuffer.h.

6.42.2 Constructor & Destructor Documentation

6.42.2.1 template<class BaseType, class CoType> ABA_CUTBUFFER< BaseType, CoType >::ABA_CUTBUFFER (ABA_MASTER ∗ *master*, int *size*)

The constructor.

Parameters:

master A pointer to the corresponding master of the optimization.

size The maximal number of constraints/variables which can be buffered.

6.42.2.2 template<class BaseType, class CoType> ABA_CUTBUFFER< BaseType, CoType >::∼ABA_CUTBUFFER ()

The destructor.

If there are still items buffered when this object is destructed then we have to unset the locks of the buffered items. This can happen if in the feasibility test constraints are generated but for some reason (e.g., due to tailing off) the optimization of the subproblem is terminated.

6.42.2.3 template<class BaseType, class CoType> ABA_CUTBUFFER< BaseType, CoType >::ABA_CUTBUFFER (const ABA_CUTBUFFER< BaseType, CoType > & *rhs*) `[private]`

6.42.3 Member Function Documentation

6.42.3.1 template<class BaseType, class CoType> void ABA_CUTBUFFER< BaseType, CoType >::extract (int *max*, ABA_BUFFER< ABA_POOLSLOT< BaseType, CoType > * > & *newSlots*) `[private]`

Takes the first *max* items from the buffer and clears the buffer.

Constraints or variables stored in slots which are not extracted are also removed from their pools if *keepInPool* has not been set to *true* at insertion time.

Parameters:

max The maximal number of extracted items.

newSlots The extracted items are inserted into this buffer.

6.42.3.2 template<class BaseType, class CoType> int ABA_CUTBUFFER< BaseType, CoType >::insert (ABA_POOLSLOT< BaseType, CoType > * *slot*, bool *keepInPool*, double *rank*)

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()*.

Returns:

0 If the item can be inserted.
1 If the buffer is already full.

Parameters:

rank A rank associated with the constraint/variable.

6.42.3.3 template<class BaseType, class CoType> int ABA_CUTBUFFER< BaseType, CoType >::insert (ABA_POOLSLOT< BaseType, CoType > * *slot*, bool *keepInPool*)

Adds a slot to the buffer.

The member *ranking_* has to be set to *false*, because since no rank is added together with this item a ranking of all items is impossible. Such that newly generated items cannot be removed immediately in a cleaning up process of the pool we set a lock which will be removed in the function *extract()*.

Returns:

0 If the item can be inserted.
1 If the buffer is already full.

Parameters:

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*.

6.42.3.4 template<class BaseType, class CoType> int ABA_CUTBUFFER< BaseType, CoType >::number () const

Returns:

The number of buffered items.

6.42.3.5 template<class BaseType, class CoType> const ABA_CUTBUFFER<BaseType, CoType>& ABA_CUTBUFFER< BaseType, CoType >::operator= (const ABA_CUTBUFFER< BaseType, CoType > & *rhs*) `[private]`

6.42.3.6 template<class BaseType, class CoType> void ABA_CUTBUFFER< BaseType, CoType >::remove (ABA_BUFFER< int > & *index*)

Removes the specified elements from the buffer.

Parameters:

index The numbers of the elements which should be removed.

6.42.3.7 template<class BaseType, class CoType> int ABA_CUTBUFFER< BaseType, CoType >::size () const

Returns:

The maximal number of items that can be buffered.

6.42.3.8 template<class BaseType, class CoType> ABA_POOLSLOT<BaseType, CoType>* ABA_CUTBUFFER< BaseType, CoType >::slot (int *i*)

Returns:

A pointer to the *i-th* ABA_POOLSLOT that is buffered.

6.42.3.9 template<class BaseType, class CoType> void ABA_CUTBUFFER< BaseType, CoType >::sort (int *threshold*) `[private]`

Sorts the items according to their ranks.

Parameters:

threshold Only if more than *threshold* items are buffered, the sorting is performed.

6.42.3.10 template<class BaseType, class CoType> int ABA_CUTBUFFER< BaseType, CoType >::space () const

Returns:

The number of items which can still be inserted into the buffer.

6.42.4 Friends And Related Function Documentation

6.42.4.1 template<class BaseType, class CoType> friend class ABA_SUB `[friend]`

Definition at line 50 of file cutbuffer.h.

6.42.5 Member Data Documentation

6.42.5.1 template<class BaseType, class CoType> ABA_ARRAY<bool> ABA_CUTBUFFER< BaseType, CoType >::keepInPool_ `[private]`

If *keepInPool_*[i] is *true* for a buffered constraint/variables, then it is not removed from its pool although it might be discarded in *extract()*.

Definition at line 164 of file cutbuffer.h.

6.42.5.2 template<class BaseType, class CoType> ABA_MASTER* ABA_CUTBUFFER< BaseType, CoType >::master_ `[private]`

A pointer to the corresponding master of the optimization.

Definition at line 150 of file cutbuffer.h.

6.42.5.3 template<class BaseType, class CoType> int ABA_CUTBUFFER< BaseType, CoType >::n_ `[private]`

The number of buffered items.

Definition at line 154 of file cutbuffer.h.

6.42.5.4 template<class BaseType, class CoType> ABA_ARRAY<ABA_POOLSLOTREF<BaseType, CoType>*> ABA_CUTBUFFER< BaseType, CoType >::psRef_ `[private]`

References to the pool slots of the buffered constraints/variables.

Definition at line 158 of file cutbuffer.h.

6.42.5.5 template<class BaseType, class CoType> ABA_ARRAY<double> ABA_CUTBUFFER< BaseType, CoType >::rank_ `[private]`

This array stores optionally the rank of the buffered items.

Definition at line 168 of file cutbuffer.h.

6.42.5.6 template<class BaseType, class CoType> bool ABA_CUTBUFFER< BaseType, CoType >::ranking_ `[private]`

This flag is *true* if a rank for each buffered item is available. As soon as an item without rank is inserted it becomes *false*.

Definition at line 173 of file cutbuffer.h.

The documentation for this class was generated from the following file:

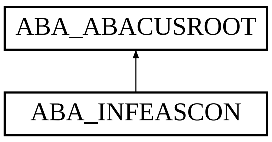
- Include/abacus/cutbuffer.h

6.43 ABA_INFEASCON Class Reference

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.

`#include <infeascon.h>`

Inheritance diagram for ABA_INFEASCON::



Public Types

- enum INFEAS { TooSmall = -1, Feasible, TooLarge }

Public Member Functions

- ABA_INFEASCON (ABA_MASTER ∗master, ABA_CONSTRAINT ∗con, INFEAS inf)
- ABA_CONSTRAINT ∗ constraint () const
- INFEAS infeas () const
- bool goodVar (ABA_VARIABLE ∗v)

Private Attributes

- ABA_MASTER ∗ master_
- ABA_CONSTRAINT ∗ constraint_
- INFEAS infeas_

6.43.1 Detailed Description

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.

Definition at line 48 of file infeascon.h.

6.43.2 Member Enumeration Documentation

6.43.2.1 enum ABA_INFEASCON::INFEAS

The different ways of infeasibility of a constraint.

Parameters:

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.

Enumeration values:

TooSmall

Feasible

TooLarge

Definition at line 57 of file infeascon.h.

6.43.3 Constructor & Destructor Documentation

6.43.3.1 ABA_INFEASCON::ABA_INFEASCON (ABA_MASTER ∗ *master*, ABA_CONSTRAINT ∗ *con*, INFEAS *inf*)

The constructor.

Parameters:

master A pointer to the corresponding master of the optimization.

con The infeasible constraint.

inf The way of infeasibility.

6.43.4 Member Function Documentation

6.43.4.1 ABA_CONSTRAINT ∗ ABA_INFEASCON::constraint () const `[inline]`

Returns:
A pointer to the infeasible constraint.

Definition at line 99 of file infeascon.h.

6.43.4.2 bool ABA_INFEASCON::goodVar (ABA_VARIABLE ∗ *v*)

Returns:
true If the variable *v* might reduce the infeasibility,
false otherwise.

Parameters:
v A variable for which we test if its addition might reduce infeasibility.

6.43.4.3 ABA_INFEASCON::INFEAS ABA_INFEASCON::infeas () const `[inline]`

Returns:
The way of infeasibility of the constraint.

Definition at line 104 of file infeascon.h.

6.43.5 Member Data Documentation

6.43.5.1 ABA_CONSTRAINT∗ ABA_INFEASCON::constraint_ `[private]`

A pointer to the infeasible constraint.

Definition at line 91 of file infeascon.h.

6.43.5.2 INFEAS ABA_INFEASCON::infeas_ `[private]`

The way of infeasibility.

Definition at line 95 of file infeascon.h.

6.43.5.3 ABA_MASTER∗ ABA_INFEASCON::master_ `[private]`

A pointer to the corresponding master of the optimization.

Definition at line 87 of file infeascon.h.

The documentation for this class was generated from the following file:

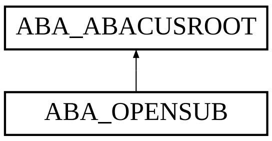
- Include/abacus/infeascon.h

6.44 ABA_OPENSUB Class Reference

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.

`#include <opensub.h>`

Inheritance diagram for ABA_OPENSUB::



Public Member Functions

- ABA_OPENSUB (ABA_MASTER ∗master)

  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.*

- int number () const
- bool empty () const
- double dualBound () const

Private Member Functions

- ABA_SUB ∗ select ()

  Selects a subproblem according to the strategy in master *and removes it from the list of open subproblems.*

- void insert (ABA_SUB ∗sub)
- void remove (ABA_SUB ∗sub)
- void prune ()
- void updateDualBound ()

  Updates the member dualBound_ *according to the dual bounds of the subproblems contained in this set.*

- ABA_OPENSUB (const ABA_OPENSUB &rhs)
- const ABA_OPENSUB & operator= (const ABA_OPENSUB &rhs)

Private Attributes

- ABA_MASTER ∗ master_
- ABA_DLIST< ABA_SUB ∗ > list_
- int n_
- double dualBound_

Friends

- class ABA_SUB
- class ABA_MASTER

6.44.1 Detailed Description

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.

Definition at line 50 of file opensub.h.

6.44.2 Constructor & Destructor Documentation

6.44.2.1 ABA_OPENSUB::ABA_OPENSUB (ABA_MASTER ∗ *master*)

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.

Parameters:

master A pointer to the corresponding master of the optimization.

6.44.2.2 ABA_OPENSUB::ABA_OPENSUB (const ABA_OPENSUB & *rhs*) `[private]`

6.44.3 Member Function Documentation

6.44.3.1 double ABA_OPENSUB::dualBound () const

Returns:

The value of the dual bound of all subproblems in the list.

6.44.3.2 bool ABA_OPENSUB::empty () const `[inline]`

Returns:

true If there is no subproblem in the set of open subproblems,
false otherwise.

Definition at line 179 of file opensub.h.

6.44.3.3 void ABA_OPENSUB::insert (ABA_SUB ∗ *sub*) `[private]`

Adds a subproblem to the set of open subproblems.

Parameters:

sub The subproblem that is inserted.

6.44.3.4 int ABA_OPENSUB::number () const `[inline]`

Returns:

The current number of open subproblems contained in this set.

Definition at line 174 of file opensub.h.

6.44.3.5 const ABA_OPENSUB& ABA_OPENSUB::operator= (const ABA_OPENSUB & *rhs*) `[private]`

6.44.3.6 void ABA_OPENSUB::prune () `[private]`

Removes all elements from the set of opens subproblems.

6.44.3.7 void ABA_OPENSUB::remove (ABA_SUB ∗ *sub*) `[private]`

Removes subproblem from the set of open subproblems.

Parameters:

sub The subproblem that is removed.

6.44.3.8 ABA_SUB∗ ABA_OPENSUB::select () `[private]`

Selects a subproblem according to the strategy in *master* and removes it from the list of open subproblems.

The function *select()* scans the list of open subproblems, and selects the subproblem with highest priority from the set of open subproblems. Dormant subproblems are ignored if possible.

Returns:

The selected subproblem. If the set of open subproblems is empty, 0 is returned.

6.44.3.9 void ABA_OPENSUB::updateDualBound () `[private]`

Updates the member *dualBound_* according to the dual bounds of the subproblems contained in this set.

6.44.4 Friends And Related Function Documentation

6.44.4.1 friend class ABA_MASTER `[friend]`

Definition at line 52 of file opensub.h.

6.44.4.2 friend class ABA_SUB `[friend]`

Definition at line 51 of file opensub.h.

6.44.5 Member Data Documentation

6.44.5.1 double ABA_OPENSUB::dualBound_ `[private]`

The dual bound of all open subproblems.

Definition at line 167 of file opensub.h.

6.44.5.2 ABA_DLIST<ABA_SUB*> ABA_OPENSUB::list_ `[private]`

The doubly linked list storing the open subproblems.

Definition at line 159 of file opensub.h.

6.44.5.3 ABA_MASTER* ABA_OPENSUB::master_ `[private]`

A pointer to corresponding master of the optimization.

Definition at line 137 of file opensub.h.

6.44.5.4 int ABA_OPENSUB::n_ `[private]`

The number of open subproblems.

Definition at line 163 of file opensub.h.

The documentation for this class was generated from the following file:

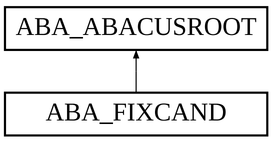
- Include/abacus/opensub.h

6.45 ABA_FIXCAND Class Reference

candidates for fixing

`#include <fixcand.h>`

Inheritance diagram for ABA_FIXCAND::



Public Member Functions

- ABA_FIXCAND (ABA_MASTER ∗master)
- ∼ABA_FIXCAND ()

  The destructor.

Private Member Functions

- void saveCandidates (ABA_SUB ∗sub)
- void fixByRedCost (ABA_CUTBUFFER< ABA_VARIABLE, ABA_CONSTRAINT > ∗addVarBuffer)
- void deleteAll ()
- void allocate (int nCand)
- ABA_FIXCAND (const ABA_FIXCAND &rhs)
- const ABA_FIXCAND & operator= (const ABA_FIXCAND &rhs)

Private Attributes

- ABA_MASTER ∗ master_
- ABA_BUFFER< ABA_POOLSLOTREF< ABA_VARIABLE, ABA_CONSTRAINT > ∗ > ∗ candidates_
- ABA_BUFFER< ABA_FSVARSTAT ∗ > ∗ fsVarStat_
- ABA_BUFFER< double > ∗ lhs_

Friends

- class ABA_SUB
- class ABA_MASTER

6.45.1 Detailed Description

candidates for fixing

Definition at line 60 of file fixcand.h.

6.45.2 Constructor & Destructor Documentation

6.45.2.1 ABA_FIXCAND::ABA_FIXCAND (ABA_MASTER ∗ *master*)

The constructor.

Parameters:

master A pointer to the corresponding master of the optimization.

6.45.2.2 ABA_FIXCAND::∼ABA_FIXCAND ()

The destructor.

6.45.2.3 ABA_FIXCAND::ABA_FIXCAND (const ABA_FIXCAND & *rhs*) `[private]`

6.45.3 Member Function Documentation

6.45.3.1 void ABA_FIXCAND::allocate (int *nCand*) `[private]`

Allocates memory to store *nCand* candidates for fixing.

6.45.3.2 void ABA_FIXCAND::deleteAll () `[private]`

Deletes all allocated memory of members.

The member pointers are set to 0 that multiple deletion cannot cause any error.

6.45.3.3 void ABA_FIXCAND::fixByRedCost (ABA_CUTBUFFER< ABA_VARIABLE, ABA_CONSTRAINT > ∗ *addVarBuffer*) `[private]`

Tries to fix as many candidates as possible.

The new variable status is both stored in the global variable status of the class ABA_MASTER and in the local variable status of ABA_SUB. Candidates which are fixed are removed from the candidate set.

Returns:

1 If contradictions to the variables statuses of *sub* are detected.
0 otherwise.

Parameters:

addVarBuffer Inactive variables which are fixed to a nonzero value are added to *addVarBuffer* to be activated in the next iteration.

We do not used the function ABA_MASTER::primalViolated() for checking of a variable can be fixed, because here we also have to be careful for integer objective function.

6.45.3.4 const ABA_FIXCAND& ABA_FIXCAND::operator= (const ABA_FIXCAND & *rhs*) `[private]`

6.45.3.5 void ABA_FIXCAND::saveCandidates (ABA_SUB $*$ *sub*) `[private]`

Memorizes suitable variables for fixing.

Parameters:

sub A pointer to the root node of the remaining $\backslash$ tree.

6.45.4 Friends And Related Function Documentation

6.45.4.1 friend class ABA_MASTER `[friend]`

Definition at line 62 of file fixcand.h.

6.45.4.2 friend class ABA_SUB `[friend]`

Definition at line 61 of file fixcand.h.

6.45.5 Member Data Documentation

6.45.5.1 ABA_BUFFER<ABA_POOLSLOTREF<ABA_VARIABLE, ABA_CONSTRAINT>$*$>$*$ ABA_FIXCAND::candidates_ `[private]`

The candidates for fixing.

Definition at line 119 of file fixcand.h.

6.45.5.2 ABA_BUFFER<ABA_FSVARSTAT∗>∗ ABA_FIXCAND::fsVarStat_ `[private]`

The fixing status of the candidates.

Definition at line 123 of file fixcand.h.

6.45.5.3 ABA_BUFFER<double>∗ ABA_FIXCAND::lhs_ `[private]`

The left hand side of the expression evaluated for fixing.

Definition at line 127 of file fixcand.h.

6.45.5.4 ABA_MASTER∗ ABA_FIXCAND::master_ `[private]`

A pointer to the corresponding master of the optimization.

Definition at line 115 of file fixcand.h.

The documentation for this class was generated from the following file:

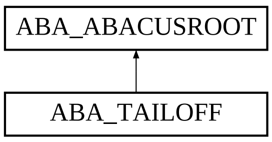
- Include/abacus/fixcand.h

6.46 ABA_TAILOFF Class Reference

This class stores the history of the values of the last LP-solutions and implements all functions to control tailing-off effect.

`#include <tailoff.h>`

Inheritance diagram for ABA_TAILOFF::



Public Member Functions

- ABA_TAILOFF (ABA_MASTER ∗master)

  The constructor takes the length of the tailing off history from ABA_MASTER::tailOffNLp().

- ABA_TAILOFF (ABA_MASTER ∗master, int NLp)

  An alternative constructor takes the length of the tailing off history from the parameter NLp.

- ∼ABA_TAILOFF ()

  The destructor.

- bool tailOff () const

- int diff (int nLps, double &d) const

  Can be used to retrieve the difference between the last and a previous LP-solution in percent.

Private Member Functions

- void update (double value)
- void reset ()

Private Attributes

- ABA_MASTER $*$ master_
- ABA_RING$<$ double $>$ $*$ lpHistory_

Friends

- class ABA_SUB
- ostream & operator$<<$ (ostream &out, const ABA_TAILOFF &rhs)

  The output operator writes the memorized LP-values on an output stream.

6.46.1 Detailed Description

This class stores the history of the values of the last LP-solutions and implements all functions to control tailing-off effect.

Definition at line 53 of file tailoff.h.

6.46.2 Constructor & Destructor Documentation

6.46.2.1 ABA_TAILOFF::ABA_TAILOFF (ABA_MASTER $*$ *master*)

The constructor takes the length of the tailing off history from ABA_MASTER::tailOffNLp().

Parameters:

master A pointer to the corresponding master of the optimization.

6.46.2.2 ABA_TAILOFF::ABA_TAILOFF (ABA_MASTER $*$ *master*, int *NLp*)

An alternative constructor takes the length of the tailing off history from the parameter NLp.

Parameters:

master A pointer to the corresponding master of the optimization.

NLp The length of the tailing off history.

6.46.2.3 ABA_TAILOFF::∼ABA_TAILOFF ()

The destructor.

6.46.3 Member Function Documentation

6.46.3.1 int ABA_TAILOFF::diff (int *nLps*, double & *d*) const

Can be used to retrieve the difference between the last and a previous LP-solution in percent.

Returns:

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.

Parameters:

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.

6.46.3.2 void ABA_TAILOFF::reset () `[private]`

Clears the solution history.

This function should be called if variables are added, because normally the solution value of the LP-relaxation gets worse after the addition of variables. Such a change could falsely indicate a tailing-off effect if the history of LP-values is not reset.

6.46.3.3 bool ABA_TAILOFF::tailOff () const

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.

Returns:

true If a tailing off effect is observed,
false otherwise.

6.46.3.4 void ABA_TAILOFF::update (double *value*) `[private]`

A new LP-solution value can be stored by calling the function *update()*.

This update should be performed after every solution of an LP in the cutting plane generation phase of the subproblem optimization process.

Parameters:

value The LP-solution value.

6.46.4 Friends And Related Function Documentation

6.46.4.1 friend class ABA_SUB `[friend]`

Definition at line 54 of file tailoff.h.

6.46.4.2 ostream& operator$<<$ (ostream & *out*, const ABA_TAILOFF & *rhs*) `[friend]`

The output operator writes the memorized LP-values on an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The tailing-off manager being output.

6.46.5 Member Data Documentation

6.46.5.1 ABA_RING$<$double$>*$ ABA_TAILOFF::lpHistory_ `[private]`

The LP-values considered in the tailing off analysis.

Definition at line 143 of file tailoff.h.

6.46.5.2 ABA_MASTER$*$ ABA_TAILOFF::master_ `[private]`

A pointer to the corresponding master of the optimization.

Definition at line 139 of file tailoff.h.

The documentation for this class was generated from the following file:

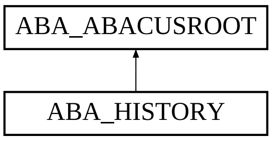
- Include/abacus/tailoff.h

6.47 ABA_HISTORY Class Reference

class implements the storage of the solution history.

`#include <history.h>`

Inheritance diagram for ABA_HISTORY::



Public Member Functions

- ABA_HISTORY (ABA_MASTER ∗master)
- virtual ∼ABA_HISTORY ()

  The destructor.

- void 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.

Private Member Functions

- int size () const
- void realloc ()

  The function realloc() enlarges the history table by 100 components.

Private Attributes

- ABA_MASTER ∗ master_
- ABA_ARRAY< double > primalBound_
- ABA_ARRAY< double > dualBound_
- ABA_ARRAY< long > time_
- int n_

Friends

- ostream & operator<< (ostream &out, const ABA_HISTORY &rhs)

6.47.1 Detailed Description

class implements the storage of the solution history.

Definition at line 43 of file history.h.

6.47.2 Constructor & Destructor Documentation

6.47.2.1 ABA_HISTORY::ABA_HISTORY (ABA_MASTER $*$ *master*)

The constructor initializes a history table with 100 possible entries.

If this number is exceeded an automatic reallocation is performed.

Parameters:

master A pointer to the corresponding master of the optimization.

6.47.2.2 virtual ABA_HISTORY::∼ABA_HISTORY () `[virtual]`

The destructor.

6.47.3 Member Function Documentation

6.47.3.1 void ABA_HISTORY::realloc () `[private]`

The function *realloc()* enlarges the history table by 100 components.

6.47.3.2 int ABA_HISTORY::size () const `[inline, private]`

Returns the length of the history table.

Definition at line 107 of file history.h.

6.47.3.3 void ABA_HISTORY::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.

6.47.4 Friends And Related Function Documentation

6.47.4.1 ostream& operator<< (ostream & *out*, const ABA_HISTORY & *rhs*) `[friend]`

The output operator.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The solution history being output.

6.47.5 Member Data Documentation

6.47.5.1 ABA_ARRAY<double> ABA_HISTORY::dualBound_ `[private]`

The array storing the value of the best dual solution.

Definition at line 95 of file history.h.

6.47.5.2 ABA_MASTER* ABA_HISTORY::master_ `[private]`

A pointer to corresponding master of the optimization.

Definition at line 87 of file history.h.

6.47.5.3 int ABA_HISTORY::n_ `[private]`

The number of entries in the history table.

Definition at line 103 of file history.h.

6.47.5.4 ABA_ARRAY<double> ABA_HISTORY::primalBound_ `[private]`

The array storing the value of the best primal solution.

Definition at line 91 of file history.h.

6.47.5.5 ABA_ARRAY<long> ABA_HISTORY::time_ `[private]`

The CPU time in seconds, when the entry in the table was made.

Definition at line 99 of file history.h.

The documentation for this class was generated from the following file:

- Include/abacus/history.h

6.48 Basic Data Structures

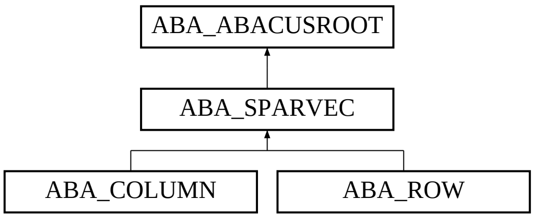
This subsection documents various basic data structures which we have used within ABACUS. They can also be used within an application. The templated basic data structures are documented in Section 6.53.

6.49 ABA_SPARVEC Class Reference

Since other classes, e.g., the class ABA_RO are derived from this class, all data members are protected in order to provide efficient access also in these derived classes.

`#include <sparvec.h>`

Inheritance diagram for ABA_SPARVEC::



Public Member Functions

- 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)

  A constructor with initialization of the support and coefficients of the sparse vector.

- ABA_SPARVEC (ABA_GLOBAL ∗glob, int size, int ∗s, double ∗c, double reallocFac=10.0)

  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 (const ABA_SPARVEC &rhs)
- ∼ABA_SPARVEC ()

  The destructor.

- const ABA_SPARVEC & operator= (const ABA_SPARVEC &rhs)

  The assignment operator requires that the left hand and the right hand side have the same length (otherwise use the function copy()).

- int support (int i) const

  A range check is performed if the function is compiled with -DABACUSSAFE.

- double coeff (int i) const

  A range check is performed if the function is compiled with -DABACUSSAFE.

- double origCoeff (int i) const

- void insert (int s, double c)
- void leftShift (ABA_BUFFER< int > &del)
- void copy (const ABA_SPARVEC &vec)

    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 clear ()
- void rename (ABA_ARRAY< int > &newName)
- int size () const
- int nnz () const
- double norm ()
- void realloc ()
- void realloc (int newSize)

    This other version of realloc() *reallocates the sparse vector to a given length.*

Protected Member Functions

- void rangeCheck (int i) const

    Terminates the program with an error message if i *is negative or greater or equal than the number of nonzero elements.*

Protected Attributes

- ABA_GLOBAL ∗ glob_
- int size_

    The maximal number of nonzero coefficients which can be stored without reallocation.

- int nnz_
- double reallocFac_

    If a new element is inserted but the sparse vector is full, then its size is increased by reallocFac_ *percent.*

- int ∗ support_
- double ∗ coeff_

Friends

- ostream & operator<< (ostream &out, const ABA_SPARVEC &rhs)

    The output operator writes the elements of the support and their coefficients line by line on an output stream.

6.49.1 Detailed Description

Since other classes, e.g., the class ABA_RO are derived from this class, all data members are protected in order to provide efficient access also in these derived classes.

Definition at line 50 of file sparvec.h.

6.49.2 Constructor & Destructor Documentation

6.49.2.1 ABA_SPARVEC::ABA_SPARVEC (ABA_GLOBAL ∗ *glob*, int *size*, double *reallocFac* = `10.0`)

The constructor for an empty sparse vector.

Parameters:

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.

If no memory for *support_* and *coeff_* is allocated then an automatic allocation will be performed when the function *insert()* is called the first time.

6.49.2.2 ABA_SPARVEC::ABA_SPARVEC (ABA_GLOBAL ∗ *glob*, int *size*, const ABA_ARRAY< int > & *s*, const ABA_ARRAY< double > & *c*, double *reallocFac* = `10.0`)

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.

Parameters:

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.

If *size* is 0, then also no elements are copied in the *for-loop* since *nnz_* will be also 0.

6.49.2.3 ABA_SPARVEC::ABA_SPARVEC (ABA_GLOBAL ∗ *glob*, int *size*, int ∗ *s*, double ∗ *c*, double *reallocFac* = `10.0`)

This constructor is equivalent to the previous one except that it is using C-style arrays for the initialization of the sparse vector.

6.49.2.4 ABA_SPARVEC::ABA_SPARVEC (const ABA_SPARVEC & *rhs*)

The copy constructor.

Parameters:

rhs The sparse vector that is copied.

6.49.2.5 ABA_SPARVEC::∼ABA_SPARVEC ()

The destructor.

6.49.3 Member Function Documentation

6.49.3.1 void ABA_SPARVEC::clear () `[inline]`

Removes all nonzeros from the sparse vector.

Definition at line 327 of file sparvec.h.

6.49.3.2 double ABA_SPARVEC::coeff (int *i*) const `[inline]`

A range check is performed if the function is compiled with *-DABACUSSAFE*.

Returns:

The coefficient of the *i-th* nonzero element.

Parameters:

i The number of the nonzero element.

Definition at line 308 of file sparvec.h.

6.49.3.3 void ABA_SPARVEC::copy (const ABA_SPARVEC & *vec*)

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.

Parameters:

vec The sparse vector that is copied.

6.49.3.4 void ABA_SPARVEC::insert (int *s*, double *c*) `[inline]`

Adds a new support/coefficient pair to the vector.

If necessary a reallocation of the member data is performed automatically.

Parameters:

s The new support.

c The new coefficient.

Definition at line 316 of file sparvec.h.

6.49.3.5 void ABA_SPARVEC::leftShift (ABA_BUFFER< int > & *del*)

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.

Parameters:

del The numbers of the elements removed from the sparse vector.

6.49.3.6 int ABA_SPARVEC::nnz () const `[inline]`

Returns:

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.

Definition at line 337 of file sparvec.h.

6.49.3.7 double ABA_SPARVEC::norm ()

Returns:

The Euclidean norm of the sparse vector.

6.49.3.8 const ABA_SPARVEC& ABA_SPARVEC::operator= (const ABA_SPARVEC & *rhs*)

The assignment operator requires that the left hand and the right hand side have the same length (otherwise use the function *copy()*).

Returns:

A reference to the left hand side.

Parameters:

rhs The right hand side of the assignment.

6.49.3.9 double ABA_SPARVEC::origCoeff (int *i*) const

Returns:

The coefficient having support *i*.

Parameters:

i The number of the original coefficient.

6.49.3.10 void ABA_SPARVEC::rangeCheck (int *i*) const `[protected]`

Terminates the program with an error message if *i* is negative or greater or equal than the number of nonzero elements.

If the class ABA_SPARVEC is compiled with the flag *-DABACUSSAFE*, then before each access operation on element *i* of the sparse vector the function *rangeCheck()* is called.

Parameters:

i An integer that should be checked if it is in the range of the sparse vector.

6.49.3.11 void ABA_SPARVEC::realloc (int *newSize*)

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.

Parameters:

newSize The new maximal number of nonzeroes that can be stored in the sparse vector.

6.49.3.12 void ABA_SPARVEC::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.

6.49.3.13 void ABA_SPARVEC::rename (ABA_ARRAY< int > & *newName*)

Replaces the index of the support by new names.

Parameters:

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.

6.49.3.14 int ABA_SPARVEC::size () const `[inline]`

Returns:

The maximal length of the sparse vector.

Definition at line 332 of file sparvec.h.

6.49.3.15 int ABA_SPARVEC::support (int *i*) const `[inline]`

A range check is performed if the function is compiled with *-DABACUSSAFE*.

Returns:

The support of the *i-th* nonzero element.

Parameters:

i The number of the nonzero element.

Definition at line 300 of file sparvec.h.

6.49.4 Friends And Related Function Documentation

6.49.4.1 ostream& operator<< (ostream & *out*, const ABA_SPARVEC & *rhs*) `[friend]`

The output operator writes the elements of the support and their coefficients line by line on an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The sparse vector being output.

6.49.5 Member Data Documentation

6.49.5.1 double* ABA_SPARVEC::coeff_ `[protected]`

The array storing the corresponding nonzero coefficients.

Definition at line 296 of file sparvec.h.

6.49.5.2 ABA_GLOBAL* ABA_SPARVEC::glob_ `[protected]`

A pointer to the corresponding global object.

Definition at line 273 of file sparvec.h.

6.49.5.3 int ABA_SPARVEC::nnz_ `[protected]`

The number of stored elements (“nonzeros”).

Definition at line 282 of file sparvec.h.

6.49.5.4 double ABA_SPARVEC::reallocFac_ `[protected]`

If a new element is inserted but the sparse vector is full, then its size is increased by *reallocFac_* percent.

Definition at line 288 of file sparvec.h.

6.49.5.5 int ABA_SPARVEC::size_ `[protected]`

The maximal number of nonzero coefficients which can be stored without reallocation.

Definition at line 278 of file sparvec.h.

6.49.5.6 int∗ ABA_SPARVEC::support_ `[protected]`

The array storing the nonzero variables.

Definition at line 292 of file sparvec.h.

The documentation for this class was generated from the following file:

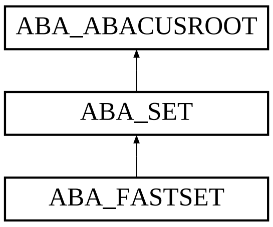
- Include/abacus/sparvec.h

6.50 ABA_SET Class Reference

class implements a data structure for collections of dynamic disjoint sets of integers

`#include <set.h>`

Inheritance diagram for ABA_SET::



Public Member Functions

- ABA_SET (ABA_GLOBAL ∗glob, int size)
- virtual ∼ABA_SET ()

  The destructor.

- void makeSet (int x)
- bool unionSets (int x, int y)
- int findSet (int x)

Protected Attributes

- ABA_GLOBAL $*$ glob_
- 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.*

6.50.1 Detailed Description

class implements a data structure for collections of dynamic disjoint sets of integers

Definition at line 41 of file set.h.

6.50.2 Constructor & Destructor Documentation

6.50.2.1 ABA_SET::ABA_SET (ABA_GLOBAL $*$ *glob*, int *size*)

The constructor.

Parameters:

glob A pointer to the corresponding global object.

size Only integers between 0 and *size-1* can be inserted in the set.

6.50.2.2 virtual ABA_SET::~ABA_SET () `[virtual]`

The destructor.

6.50.3 Member Function Documentation

6.50.3.1 int ABA_SET::findSet (int *x*)

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()*.

A path compression is performed, i.e., all nodes of the tree on the path from *x* to the root are directly attached to the root of the tree.

Returns:

The representative of the set containing *x*.

Parameters:

x An element of the searched set.

6.50.3.2 void ABA_SET::makeSet (int *x*)

Creates a set storing only one element and adds it to the collection of sets.

Parameters:

x The single element of the new set.

6.50.3.3 bool ABA_SET::unionSets (int *x*, int *y*)

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.

Returns:

true If both sets have been disjoint before the function call,
false otherwise.

Parameters:

x An element of the first set of the union operation.

y An element in the second set of the union operation.

Reimplemented in ABA_FASTSET.

6.50.4 Member Data Documentation

6.50.4.1 ABA_GLOBAL∗ ABA_SET::glob_ `[protected]`

A pointer to the corresponding global object.

Definition at line 101 of file set.h.

6.50.4.2 ABA_ARRAY<int> ABA_SET::parent_ `[protected]`

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.

Definition at line 108 of file set.h.

The documentation for this class was generated from the following file:

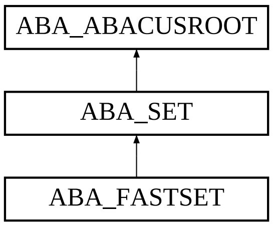
- Include/abacus/set.h

6.51 ABA_FASTSET Class Reference

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.

`#include <fastset.h>`

Inheritance diagram for ABA_FASTSET::



Public Member Functions

- ABA_FASTSET (ABA_GLOBAL ∗glob, int size)
- bool unionSets (int x, int y)

Private Attributes

- ABA_ARRAY< int > rank_

6.51.1 Detailed Description

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.

Parameters:

rank_ The rank of each set.

Definition at line 42 of file fastset.h.

6.51.2 Constructor & Destructor Documentation

6.51.2.1 ABA_FASTSET::ABA_FASTSET (ABA_GLOBAL $*$ *glob*, int *size*)

The constructor.

At the beginning each possible set receives the rank 0.

Parameters:

glob A pointer to the corresponding global object.

size Only integers between 0 and *size-1* can be inserted in the set.

6.51.3 Member Function Documentation

6.51.3.1 bool ABA_FASTSET::unionSets (int *x*, int *y*)

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.

Returns:

true If both sets have been disjoint before the function call,
false otherwise.

Parameters:

x An element of the first set of the union operation.

y An element in the second set of the union operation.

Reimplemented from ABA_SET.

6.51.4 Member Data Documentation

6.51.4.1 ABA_ARRAY<int> ABA_FASTSET::rank_ `[private]`

Definition at line 71 of file fastset.h.

The documentation for this class was generated from the following file:

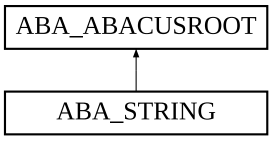
- Include/abacus/fastset.h

6.52 ABA_STRING Class Reference

class ABA_STRING implements are very simple class for the representation of character strings.

`#include <string.h>`

Inheritance diagram for ABA_STRING::



Public Member Functions

- 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 ()

  The destructor.

- const ABA_STRING & operator= (const ABA_STRING &rhs)

  The assignment operator makes a copy of the right hand side and reallocates memory if required.

- const ABA_STRING & operator= (const char ∗rhs)

  The assignment operator is overloaded for character strings.

- char & operator[ ] (int i)

  With the subscript operator a single character of the string can be accessed or modified.

- const char & operator[ ] (int i) const

  The subscript operator is overloaded for constant use.

- int size () const
- int ascii2int (int i=0) const
- unsigned int ascii2unsignedint () const
- double ascii2double () const

  Emulates the function atof() *of the standard C library and converts the string to a floating point number.*

- bool ascii2bool () const
- bool ending (const char ∗end) const
- char ∗ string () const

Private Member Functions

- void rangeCheck (int i) const

  Terminates the program with an error message if i *is not the position of a character of the string.*

Private Attributes

- ABA_GLOBAL ∗ glob_
- char ∗ string_

Friends

- int operator== (const ABA_STRING &lhs, const ABA_STRING &rhs)
- int operator== (const ABA_STRING &lhs, const char ∗rhs)

  The comparison operator is overloaded for character strings on the right hand side.

- int operator!= (const ABA_STRING &lhs, const ABA_STRING &rhs)
- int operator!= (const ABA_STRING &lhs, const char ∗rhs)

  The not-equal operator is overloaded for character strings on the right hand side.

- ostream & operator<< (ostream &out, const ABA_STRING &rhs)

6.52.1 Detailed Description

class ABA_STRING implements are very simple class for the representation of character strings.

Definition at line 45 of file string.h.

6.52.2 Constructor & Destructor Documentation

6.52.2.1 ABA_STRING::ABA_STRING (ABA_GLOBAL ∗ *glob*, const char ∗ *cString* = "")

The constructor.

Parameters:

glob A pointer to the corresponding global object.

cString The initializing string, by default the empty string.

6.52.2.2 ABA_STRING::ABA_STRING (ABA_GLOBAL ∗ *glob*, const char ∗ *cString*, int *index*)

A constructor building a string from a string and an integer.

This constructor is especially useful for building variable or constraint names like { con18}.

Parameters:

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}).

6.52.2.3 ABA_STRING::ABA_STRING (const ABA_STRING & *rhs*)

The copy constructor.

Parameters:
rhs The string that is copied.

6.52.2.4 ABA_STRING::∼ABA_STRING ()

The destructor.

6.52.3 Member Function Documentation

6.52.3.1 bool ABA_STRING::ascii2bool () const

Converts the string to a boolean value.

This is only possible for the strings *"true"* and *"false"*.

Returns:
The string converted to *true* or *false*.

6.52.3.2 double ABA_STRING::ascii2double () const

Emulates the function *atof()* of the standard C library and converts the string to a floating point number.

Returns:
The string converted to a floating point number

6.52.3.3 int ABA_STRING::ascii2int (int *i* = `0`) const

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.

Returns:
The string converted to an integer value.

Parameters:
i The number of the character at which the conversion should start. The default value of *i* is 0.

6.52.3.4 unsigned int ABA_STRING::ascii2unsignedint () const

The function *ascii2unsignedint()* converts the string to an *unsigned* int value.

Returns:

The string converted to an unsigned integer.

6.52.3.5 bool ABA_STRING::ending (const char $*$ *end*) const

Returns:

true If the string ends with the string *end*,
false otherwise.

Parameters:

end The string with which the ending of the string is compared.

6.52.3.6 const ABA_STRING& ABA_STRING::operator= (const char $*$ *rhs*)

The assignment operator is overloaded for character strings.

6.52.3.7 const ABA_STRING& ABA_STRING::operator= (const ABA_STRING & *rhs*)

The assignment operator makes a copy of the right hand side and reallocates memory if required.

Returns:

A reference to the object.

Parameters:

rhs The right hand side of the assignment.

6.52.3.8]

const char& ABA_STRING::operator[ ] (int *i*) const

The subscript operator is overloaded for constant use.

6.52.3.9]

char& ABA_STRING::operator[ ] (int *i*)

With the subscript operator a single character of the string can be accessed or modified.

If the class is compiled with the preprocessor flag *-DABACUSSAFE*, then a range check is performed.

Returns:
A reference to the *i-th* character of the string.

Parameters:
i The number of the character that should be accessed or modified. The first character has number 0.

6.52.3.10 void ABA_STRING::rangeCheck (int *i*) const `[private]`

Terminates the program with an error message if *i* is not the position of a character of the string.

The ' ' at the end of the string is not a valid character in this sense.

6.52.3.11 int ABA_STRING::size () const

Returns:
The length of the string, not including the { '\ 0'} terminating the string.

6.52.3.12 char∗ ABA_STRING::string () const

Returns:
The *char∗* representing the string to make it accessible for C-functions.

6.52.4 Friends And Related Function Documentation

6.52.4.1 int operator!= (const ABA_STRING & *lhs*, const char ∗ *rhs*) `[friend]`

The not-equal operator is overloaded for character strings on the right hand side.

6.52.4.2 int operator!= (const ABA_STRING & *lhs*, const ABA_STRING & *rhs*) `[friend]`

The not-equal operator.

Note:
the C-library function *strcmp()* returns 0 if both strings equal.

Returns:
0 If both strings are equal,
1 otherwise.

Parameters:
lhs The left hand side of the comparison.
rhs The right hand side of the comparison.

6.52.4.3 ostream& operator$<<$ (ostream & *out*, const ABA_STRING & *rhs*) `[friend]`

The output operator.

Returns:
A reference to the output stream.

Parameters:
out The output stream.
rhs The string being output.

6.52.4.4 int operator== (const ABA_STRING & *lhs*, const char $*$ *rhs*) `[friend]`

The comparison operator is overloaded for character strings on the right hand side.

6.52.4.5 int operator== (const ABA_STRING & *lhs*, const ABA_STRING & *rhs*) `[friend]`

The comparison operator.

Note:
the C-library function *strcmp()* returns 0 if both strings equal.

Returns:
0 If both strings are not equal,
1 otherwise.

Parameters:
lhs The left hand side of the comparison.
rhs The right hand side of the comparison.

6.52.5 Member Data Documentation

6.52.5.1 ABA_GLOBAL$*$ ABA_STRING::glob_ `[private]`

A pointer to the corresponding master of the optimization.

Definition at line 239 of file string.h.

6.52.5.2 char$*$ ABA_STRING::string_ `[private]`

An array storing the character of the string. This array must be terminated with a ' '.

Definition at line 244 of file string.h.

The documentation for this class was generated from the following file:

- Include/abacus/string.h

6.53 Templates

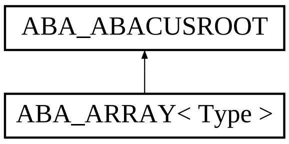
Various basic data structures are available as templates within ABACUS. For the instantiation of templates we refer to Section 5.3.

6.54 ABA_ARRAY< Type > Class Template Reference

it is a template for arrays. It can be used like a "normal" C-style array

`#include <array.h>`

Inheritance diagram for ABA_ARRAY< Type >::



Public Member Functions

- 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 ()

  The destructor.

- const ABA_ARRAY< Type > & operator= (const ABA_ARRAY< Type > &rhs)
- const ABA_ARRAY< Type > & operator= (const ABA_BUFFER< Type > &rhs)

  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.*

- 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)

  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.*

- void leftShift (ABA_BUFFER< int > &ind)

  Removes the components listed in ind *by shifting the remaining components to the left.*

- void leftShift (ABA_ARRAY< bool > &remove)

  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 set (int l, int r, Type val)

- void set (Type val)

  This version of the function set() initializes all components of the array with the same value.

- int size () const
- void realloc (int newSize)

  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 realloc (int newSize, Type init)

  Is overloaded such that also an initialization with a new value of the elements of the array after reallocation is possible.

Private Member Functions

- void rangeCheck (int i) const

  Stops the program with an error message if the index i *is not within the bounds of the array.*

Private Attributes

- ABA_GLOBAL ∗ glob_
- int n_
- Type ∗ a_

Friends

- ostream & operator<< (ostream &out, const ABA_ARRAY< Type > &array)

  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.

6.54.1 Detailed Description

template<class Type> class ABA_ARRAY< Type >

it is a template for arrays. It can be used like a "normal" C-style array

Definition at line 53 of file array.h.

6.54.2 Constructor & Destructor Documentation

6.54.2.1 template<class Type> ABA_ARRAY< Type >::ABA_ARRAY (ABA_GLOBAL ∗ *glob*, int *size*)

A constructor without initialization.

Parameters:

glob A pointer to the corresponding global object.

size The length of the array.

6.54.2.2 template<class Type> ABA_ARRAY< Type >::ABA_ARRAY (ABA_GLOBAL ∗ *glob*, int *size*, Type *init*)

A constructor with initialization.

Parameters:

glob A pointer to the corresponding global object.

size The length of the array.

init The initial value of all elements of the array.

6.54.2.3 template<class Type> ABA_ARRAY< Type >::ABA_ARRAY (ABA_GLOBAL ∗ *glob*, const ABA_BUFFER< Type > & *buf*)

A constructor.

Parameters:

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.

6.54.2.4 template<class Type> ABA_ARRAY< Type >::ABA_ARRAY (const ABA_ARRAY< Type > & *rhs*)

The copy constructor.

Parameters:

rhs The array being copied.

6.54.2.5 template<class Type> ABA_ARRAY< Type >::∼ABA_ARRAY ()

The destructor.

6.54.3 Member Function Documentation

6.54.3.1 template<class Type> void ABA_ARRAY< Type >::copy (const ABA_ARRAY< Type > & *rhs*, int *l*, int *r*)

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.

Parameters:

rhs The array that is partially copied.

l The first element being copied.

r the last element being copied.

6.54.3.2 template<class Type> void ABA_ARRAY< Type >::copy (const ABA_ARRAY< Type > & *rhs*)

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.

Parameters:

rhs The array being copied.

6.54.3.3 template<class Type> void ABA_ARRAY< Type >::leftShift (ABA_ARRAY< bool > & *remove*)

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.

Parameters:

remove The marked components are removed from the array.

6.54.3.4 template<class Type> void ABA_ARRAY< Type >::leftShift (ABA_BUFFER< int > & *ind*)

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.

Parameters:

ind The compenents being removed from the array.

6.54.3.5 template<class Type> const ABA_ARRAY<Type>& ABA_ARRAY< Type >::operator= (const ABA_BUFFER< Type > & *rhs*)

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.

Returns:
A reference to the array on the left hand side.

Parameters:
rhs The buffer being assigned.

6.54.3.6 template<class Type> const ABA_ARRAY<Type>& ABA_ARRAY< Type >::operator= (const ABA_ARRAY< Type > & *rhs*)

The assignment operator can only be used for arrays with equal length.

Returns:
A reference to the array on the left hand side.

Parameters:
rhs The array being assigned.

6.54.3.7]

template<class Type> const Type& ABA_ARRAY< Type >::operator[ ] (int *i*) const

The operator [] is overloaded for constant use.

6.54.3.8]

template<class Type> Type& ABA_ARRAY< Type >::operator[ ] (int *i*)

The operator [].

Returns:
The *i-th* element of the array.

Parameters:
i The element being accessed.

6.54.3.9 template<class Type> void ABA_ARRAY< Type >::rangeCheck (int *i*) const `[private]`

Stops the program with an error message if the index *i* is not within the bounds of the array.

6.54.3.10 template<class Type> void ABA_ARRAY< Type >::realloc (int *newSize*, Type *init*)

Is overloaded such that also an initialization with a new value of the elements of the array after reallocation is possible.

Parameters:

newSize The new length of the array.

init The new value of all components of the array.

6.54.3.11 template<class Type> void ABA_ARRAY< Type >::realloc (int *newSize*)

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.

Parameters:

newSize The new length of the array.

6.54.3.12 template<class Type> void ABA_ARRAY< Type >::set (Type *val*)

This version of the function *set()* initializes all components of the array with the same value.

Parameters:

val The new value of all components.

6.54.3.13 template<class Type> void ABA_ARRAY< Type >::set (int *l*, int *r*, Type *val*)

Assigns the same value to a subset of the components of the array.

Parameters:

l The first component the value is assigned.

r The last component the value is assigned.

val The new value of these components.

6.54.3.14 template<class Type> int ABA_ARRAY< Type >::size () const

Returns:

The length of the array.

6.54.4 Friends And Related Function Documentation

6.54.4.1 template<class Type> ostream& operator<< (ostream & *out*, const ABA_ARRAY< Type > & *array*) [friend]

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*.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

array The array being output.

6.54.5 Member Data Documentation

6.54.5.1 template<class Type> Type* ABA_ARRAY< Type >::a_ [private]

The /-style array storing the elements of the *ABA_ARRAY*.

Definition at line 265 of file array.h.

6.54.5.2 template<class Type> ABA_GLOBAL* ABA_ARRAY< Type >::glob_ [private]

A pointer to the corresponding global object.

Definition at line 257 of file array.h.

6.54.5.3 template<class Type> int ABA_ARRAY< Type >::n_ [private]

The length of the array.

Definition at line 261 of file array.h.

The documentation for this class was generated from the following file:

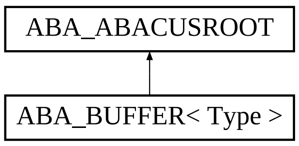
- Include/abacus/array.h

6.55 ABA_BUFFER< Type > Class Template Reference

class implements a buffer by an array and storing the number of already buffered elements.

`#include <buffer.h>`

Inheritance diagram for ABA_BUFFER< Type >::



Public Member Functions

- ABA_BUFFER (ABA_GLOBAL ∗glob, int size)
- ABA_BUFFER (const ABA_BUFFER< Type > &rhs)
- ∼ABA_BUFFER ()

    The destructor.

- const ABA_BUFFER< Type > & operator= (const ABA_BUFFER< Type > &rhs)
- 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)

    Removes the components listed in the buffer ind *by shifting the remaining components to the left.*

- void realloc (int newSize)

    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.*

Private Attributes

- ABA_GLOBAL ∗ glob_
- int size_
- int n_
- Type ∗ buf_

Friends

- ostream & operator<< (ostream &out, const ABA_BUFFER< Type > &buffer)

    The output operator writes all buffered elements line by line to an output stream in the format { number\/}{ : }{ value\/}.

6.55.1 Detailed Description

template<class Type> class ABA_BUFFER< Type >

class implements a buffer by an array and storing the number of already buffered elements.

Definition at line 61 of file buffer.h.

6.55.2 Constructor & Destructor Documentation

6.55.2.1 template<class Type> ABA_BUFFER< Type >::ABA_BUFFER (ABA_GLOBAL ∗ *glob*, int *size*)

The constructor generates an empty buffer.

Parameters:

glob	The corresponding global object.
size	The size of the buffer.

6.55.2.2 template<class Type> ABA_BUFFER< Type >::ABA_BUFFER (const ABA_BUFFER< Type > & *rhs*)

The copy constructor.

Parameters:

rhs	The buffer being copied.

6.55.2.3 template<class Type> ABA_BUFFER< Type >::~ABA_BUFFER ()

The destructor.

6.55.3 Member Function Documentation

6.55.3.1 template<class Type> void ABA_BUFFER< Type >::clear ()

Sets the number of buffered items to 0 such that the buffer is empty.

6.55.3.2 template<class Type> bool ABA_BUFFER< Type >::empty () const

Returns:

true If no items are buffered,
false otherwise.

6.55.3.3 template<class Type> bool ABA_BUFFER< Type >::full () const

Returns:

true If no more elements can be inserted into the buffer,
false otherwise.

6.55.3.4 template<class Type> void ABA_BUFFER< Type >::leftShift (ABA_BUFFER< int > & *ind*)

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.

If this function is compiled with *-DABACUSSAFE* then it is checked if each value of *ind* is in the range 0,, *number()-1*.

Parameters:

ind The numbers of the components being removed.

6.55.3.5 template<class Type> int ABA_BUFFER< Type >::number () const

Returns:

The number of buffered elements.

6.55.3.6 template<class Type> const ABA_BUFFER<Type>& ABA_BUFFER< Type >::operator= (const ABA_BUFFER< Type > & *rhs*)

The assignment operator is only allowed between buffers having equal size.

Returns:

A reference to the buffer on the left hand side of the assignment operator.

Parameters:

rhs The buffer being assigned.

6.55.3.7]

template<class Type> const Type& ABA_BUFFER< Type >::operator[ ] (int *i*) const

The operator [] is overloaded that it can be also used to access elements of constant buffers.

6.55.3.8]

template<class Type> Type& ABA_BUFFER< Type >::operator[ ] (int *i*)

The operator [] can be used to access an element of the buffer.

It is only allowed to access buffered elements. Otherwise, if the function is compiled with -*DABACUSSAFE* the program stops with an error message.

Returns:

The *i-th* element of the buffer.

Parameters:

i The number of the component which should be returned.

6.55.3.9 template<class Type> Type ABA_BUFFER< Type >::pop ()

Removes and returns the last inserted item from the buffer.

It is a fatal error to perform this operation on an empty buffer.

In this case the program stops with an error message if this function is compiled with -*DABACUSSAFE*.

Returns:

The last item that has been inserted into the buffer.

6.55.3.10 template<class Type> void ABA_BUFFER< Type >::push (Type *item*)

Inserts an item into the buffer.

It is a fatal error to perform this operation if the buffer is full.

In this case the program stops with an error message if this function is compiled with -*DABACUSSAFE*.

Parameters:

item The item that should be inserted into the buffer.

6.55.3.11 template<class Type> void ABA_BUFFER< Type >::realloc (int *newSize*)

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.

Parameters:

newSize The new length of the buffer.

6.55.3.12 template<class Type> int ABA_BUFFER< Type >::size () const

Returns:

The maximal number of elements which can be stored in the buffer.

6.55.4 Friends And Related Function Documentation

6.55.4.1 template<class Type> ostream& operator<< (ostream & *out*, const ABA_BUFFER< Type > & *buffer*) [friend]

The output operator writes all buffered elements line by line to an output stream in the format { number\/}{ : }{ value\/}.

Returns:

A reference to the stream the buffer is written to.

Parameters:

out The output stream.

buffer The buffer being output.

6.55.5 Member Data Documentation

6.55.5.1 template<class Type> Type∗ ABA_BUFFER< Type >::buf_ [private]

The /-style array storing the buffered elements.

Definition at line 231 of file buffer.h.

6.55.5.2 template<class Type> ABA_GLOBAL∗ ABA_BUFFER< Type >::glob_ [private]

A pointer to the corresponding global object.

Definition at line 219 of file buffer.h.

6.55.5.3 template<class Type> int ABA_BUFFER< Type >::n_ [private]

The number of buffered elements.

Definition at line 227 of file buffer.h.

6.55.5.4 template<class Type> int ABA_BUFFER< Type >::size_ [private]

The maximal number of elements which can be stored in the buffer.

Definition at line 223 of file buffer.h.

The documentation for this class was generated from the following file:

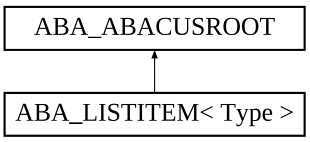
- Include/abacus/buffer.h

6.56 ABA_LISTITEM< Type > Class Template Reference

We call the basic building block of a linked list an { item\/} that is implemented by the class ABA_LISTITEM.

`#include <listitem.h>`

Inheritance diagram for ABA_LISTITEM< Type >::



Public Member Functions

- ABA_LISTITEM (const Type &elem, ABA_LISTITEM< Type > ∗succ)
- Type elem () const
- ABA_LISTITEM< Type > ∗ succ () const

Private Attributes

- Type elem_
- ABA_LISTITEM< Type > ∗ succ_

Friends

- class ABA_LIST< Type >
- ostream & operator<< (ostream &out, const ABA_LISTITEM< Type > &item)

6.56.1 Detailed Description

template<class Type> class ABA_LISTITEM< Type >

We call the basic building block of a linked list an { item\/} that is implemented by the class ABA_LISTITEM.

Parameters:

Type elem_ The element of the item.

ABA_LISTITEM$<$*Type*$>$ $*$succ_ A pointer to the successor of the item in the list. The successor of the last item is *0*.

Definition at line 55 of file listitem.h.

6.56.2 Constructor & Destructor Documentation

6.56.2.1 template$<$class Type$>$ ABA_LISTITEM$<$ Type $>$::ABA_LISTITEM (const Type & *elem*, ABA_LISTITEM$<$ Type $>$ $*$ *succ*)

The constructor.

Parameters:

elem A copy of *elem* becomes the element of the list item.

succ A pointer to the successor of the item in the list.

6.56.3 Member Function Documentation

6.56.3.1 template$<$class Type$>$ Type ABA_LISTITEM$<$ Type $>$::elem () const

Returns:

The element of the item.

6.56.3.2 template$<$class Type$>$ ABA_LISTITEM$<$Type$>$$*$ ABA_LISTITEM$<$ Type $>$::succ () const

Returns:

The successor of the item in the list.

6.56.4 Friends And Related Function Documentation

6.56.4.1 template$<$class Type$>$ friend class ABA_LIST$<$ Type $>$ `[friend]`

Definition at line 56 of file listitem.h.

6.56.4.2 template$<$class Type$>$ ostream& operator$<<$ (ostream & *out*, const ABA_LISTITEM$<$ Type $>$ & *item*) `[friend]`

The output operator.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

item The list item being output.

6.56.5 Member Data Documentation

6.56.5.1 template<class Type> Type ABA_LISTITEM< Type >::elem_ `[private]`

Definition at line 95 of file listitem.h.

6.56.5.2 template<class Type> ABA_LISTITEM<Type>∗ ABA_LISTITEM< Type >::succ_ `[private]`

Definition at line 96 of file listitem.h.

The documentation for this class was generated from the following file:

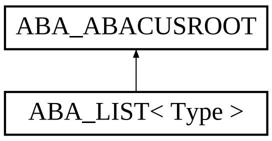
- Include/abacus/listitem.h

6.57 ABA_LIST< Type > Class Template Reference

class ABA_LIST

`#include <list.h>`

Inheritance diagram for ABA_LIST< Type >::



Public Member Functions

- ABA_LIST (const ABA_GLOBAL ∗glob)
- ~ABA_LIST ()
- void appendHead (const Type &elem)
- void appendTail (const Type &elem)
- int extractHead (Type &elem)
- int firstElem (Type &elem) const

Assign elem *the first element as the function* extractHead() *but does not remove this element from the list.*

- bool empty () const

Private Member Functions

- ABA_LISTITEM< Type > ∗ first () const
- ABA_LISTITEM< Type > ∗ last () const
- void appendHead (ABA_LISTITEM< Type > ∗item)

    This version of the function appendHead() *adds* item *at the front of the list.*

- void appendTail (ABA_LISTITEM< Type > ∗item)

    This version of the function appendTail() *adds* item *at the end of the list.*

- ABA_LIST (const ABA_LIST &rhs)
- const ABA_LIST< Type > & operator= (const ABA_LIST< Type > &rhs)

Private Attributes

- const ABA_GLOBAL ∗ glob_
- ABA_LISTITEM< Type > ∗ first_
- ABA_LISTITEM< Type > ∗ last_

Friends

- class ABA_LISTITEM< Type >
- ostream & operator<< (ostream &, const ABA_LIST< Type > &list)

6.57.1 Detailed Description

template<class Type> class ABA_LIST< Type >

class ABA_LIST

Definition at line 56 of file list.h.

6.57.2 Constructor & Destructor Documentation

6.57.2.1 template<class Type> ABA_LIST< Type >::ABA_LIST (const ABA_GLOBAL ∗ *glob*)

The constructor initializes the list with the empty list.

This is done by assigning *first_* and *last_* to the *0-pointer*.

Parameters:

glob A pointer to the corresponding global object.

6.57.2.2 template<class Type> ABA_LIST< Type >::~ABA_LIST ()

The destructor deallocates the memory of all items in the list.

6.57.2.3 template<class Type> ABA_LIST< Type >::ABA_LIST (const ABA_LIST< Type > & *rhs*) `[private]`

6.57.3 Member Function Documentation

6.57.3.1 template<class Type> void ABA_LIST< Type >::appendHead (ABA_LISTITEM< Type > * *item*) `[private]`

This version of the function *appendHead()* adds *item* at the front of the list.

6.57.3.2 template<class Type> void ABA_LIST< Type >::appendHead (const Type & *elem*)

Adds an element at the front of the list.

Parameters:

elem The element being appended.

6.57.3.3 template<class Type> void ABA_LIST< Type >::appendTail (ABA_LISTITEM< Type > * *item*) `[private]`

This version of the function *appendTail()* adds *item* at the end of the list.

6.57.3.4 template<class Type> void ABA_LIST< Type >::appendTail (const Type & *elem*)

Adds an element at the end of the list.

Parameters:

elem The element being appended.

6.57.3.5 template<class Type> bool ABA_LIST< Type >::empty () const

Returns:

true If no element is contained in the list,
false otherwise.

6.57.3.6 template<class Type> int ABA_LIST< Type >::extractHead (Type & *elem*)

Assigns to *elem* the first element in the list and removes it from the list.

Returns:
0 If the operation can be be executed successfully.
1 If the list is empty.

Parameters:
elem If the list is nonemty, the first element is assigned to *elem*.

6.57.3.7 template<class Type> ABA_LISTITEM<Type>∗ ABA_LIST< Type >::first () const [private]

Returns a pointer to the first item in the list.

6.57.3.8 template<class Type> int ABA_LIST< Type >::firstElem (Type & *elem*) const

Assign *elem* the first element as the function *extractHead()* but does not remove this element from the list.

Returns:
0 If the operation can be be executed successfully.
1 If the list is empty.

Parameters:
elem If the list is nonemty, the first element is assigned to *elem*.

6.57.3.9 template<class Type> ABA_LISTITEM<Type>∗ ABA_LIST< Type >::last () const [private]

Returns a pointer to the last item in the list.

6.57.3.10 template<class Type> const ABA_LIST<Type>& ABA_LIST< Type >::operator= (const ABA_LIST< Type > & *rhs*) [private]

6.57.4 Friends And Related Function Documentation

6.57.4.1 template<class Type> friend class ABA_LISTITEM< Type > [friend]

Definition at line 57 of file list.h.

6.57.4.2 template<class Type> ostream& operator<< (ostream &, const ABA_LIST< Type > & *list*) `[friend]`

The output operator writes all items of the list on an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

list The list being output.

6.57.5 Member Data Documentation

6.57.5.1 template<class Type> ABA_LISTITEM<Type>∗ ABA_LIST< Type >::first_ `[private]`

A pointer to the first item of the list.

Definition at line 158 of file list.h.

6.57.5.2 template<class Type> const ABA_GLOBAL∗ ABA_LIST< Type >::glob_ `[private]`

A pointer to the corresponding global object.

Definition at line 154 of file list.h.

6.57.5.3 template<class Type> ABA_LISTITEM<Type>∗ ABA_LIST< Type >::last_ `[private]`

Definition at line 162 of file list.h.

The documentation for this class was generated from the following file:

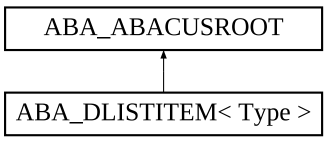
- Include/abacus/list.h

6.58 ABA_DLISTITEM< Type > Class Template Reference

A ABA_DLISTITEM stores a copy of the inserted element and has pointers to its predecessor and its successor.

`#include <dlistitem.h>`

Inheritance diagram for ABA_DLISTITEM< Type >::



Public Member Functions

- ABA_DLISTITEM (const Type &elem, ABA_DLISTITEM< Type > ∗pred, ABA_DLISTITEM< Type > ∗succ)
- Type elem () const
- ABA_DLISTITEM< Type > ∗ succ () const
- ABA_DLISTITEM< Type > ∗ pred () const

Private Attributes

- Type elem_
- ABA_DLISTITEM< Type > ∗ pred_
- ABA_DLISTITEM< Type > ∗ succ_

Friends

- class ABA_DLIST< Type >
- ostream & operator<< (ostream &out, const ABA_DLISTITEM< Type > &item)

6.58.1 Detailed Description

template<class Type> class ABA_DLISTITEM< Type >

A ABA_DLISTITEM stores a copy of the inserted element and has pointers to its predecessor and its successor.

Parameters:

Type elem_ The element stored in the item.

ABA_DLISTITEM<Type> ∗pred_ A pointer to predecessor of the item in the list.

ABA_DLISTITEM<Type> ∗succ_ A pointer to the successor of the item in the list.

Definition at line 54 of file dlistitem.h.

6.58.2 Constructor & Destructor Documentation

6.58.2.1 template<class Type> ABA_DLISTITEM< Type >::ABA_DLISTITEM (const Type & *elem*, ABA_DLISTITEM< Type > ∗ *pred*, ABA_DLISTITEM< Type > ∗ *succ*)

The constructor.

Parameters:

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.

6.58.3 Member Function Documentation

6.58.3.1 template<class Type> Type ABA_DLISTITEM< Type >::elem () const

Returns:
The element stored in the item.

6.58.3.2 template<class Type> ABA_DLISTITEM<Type>∗ ABA_DLISTITEM< Type >::pred () const

Returns:
A pointer to the predecessor of the item in the list.

6.58.3.3 template<class Type> ABA_DLISTITEM<Type>∗ ABA_DLISTITEM< Type >::succ () const

Returns:
A pointer to the successor of the item in the list.

6.58.4 Friends And Related Function Documentation

6.58.4.1 template<class Type> friend class ABA_DLIST< Type > `[friend]`

Definition at line 55 of file dlistitem.h.

6.58.4.2 template<class Type> ostream& operator<< (ostream & *out*, const ABA_DLISTITEM< Type > & *item*) `[friend]`

The output operator.

Returns:
A reference to the output stream.

Parameters:
out The output stream.
item The list item being output.

6.58.5 Member Data Documentation

6.58.5.1 template<class Type> Type ABA_DLISTITEM< Type >::elem_ `[private]`

Definition at line 102 of file dlistitem.h.

6.58.5.2 template<class Type> ABA_DLISTITEM<Type>∗ ABA_DLISTITEM< Type >::pred_ `[private]`

Definition at line 103 of file dlistitem.h.

6.58.5.3 template<class Type> ABA_DLISTITEM<Type>∗ ABA_DLISTITEM< Type >::succ_ `[private]`

Definition at line 104 of file dlistitem.h.

The documentation for this class was generated from the following file:

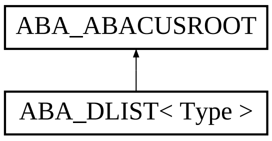
- Include/abacus/dlistitem.h

6.59 ABA_DLIST< Type > Class Template Reference

class ABA_DLIST implements a doubly linked linear list. The list is implemented by a doubly linked list of ABA_DLISTITEMs.

`#include <dlist.h>`

Inheritance diagram for ABA_DLIST< Type >::



Public Member Functions

- ABA_DLIST (ABA_GLOBAL ∗glob)
- ~ABA_DLIST ()
- void append (const Type &elem)
- int extractHead (Type &elem)
- int removeHead ()

  If the list is non-empty, the function removeHead() removes the head of the list.

- void remove (const Type &elem)
- void remove (ABA_DLISTITEM< Type > ∗item)

  This version of the function remove() scans the list for an item with element elem *beginning at the first element of the list.*

- ABA_DLISTITEM< Type > ∗ first () const
- ABA_DLISTITEM< Type > ∗ last () const
- bool empty () const
- int firstElem (Type &elem) const

Private Member Functions

- ABA_DLIST (const ABA_DLIST &rhs)
- const ABA_DLIST< Type > & operator= (const ABA_DLIST< Type > &rhs)

Private Attributes

- ABA_GLOBAL ∗ glob_
- ABA_DLISTITEM< Type > ∗ first_
- ABA_DLISTITEM< Type > ∗ last_

Friends

- ostream & operator<< (ostream &, const ABA_DLIST< Type > &list)

6.59.1 Detailed Description

template<class Type> class ABA_DLIST< Type >

class ABA_DLIST implements a doubly linked linear list. The list is implemented by a doubly linked list of ABA_DLISTITEMs.

Definition at line 62 of file dlist.h.

6.59.2 Constructor & Destructor Documentation

6.59.2.1 template<class Type> ABA_DLIST< Type >::ABA_DLIST (ABA_GLOBAL ∗ *glob*)

The constructor for an empty list.

Parameters:
glob A pointer to the corresponding global object.

6.59.2.2 template<class Type> ABA_DLIST< Type >::∼ABA_DLIST ()

The destructor deallocates the memory of all items in the list.

6.59.2.3 template<class Type> ABA_DLIST< Type >::ABA_DLIST (const ABA_DLIST< Type > & *rhs*) `[private]`

6.59.3 Member Function Documentation

6.59.3.1 template<class Type> void ABA_DLIST< Type >::append (const Type & *elem*)

Adds an element at the end of the list.

Parameters:

elem The element being appended.

6.59.3.2 template<class Type> bool ABA_DLIST< Type >::empty () const

Returns:

true If no element is contained in the list,
false otherwise.

6.59.3.3 template<class Type> int ABA_DLIST< Type >::extractHead (Type & *elem*)

Assigns to *elem* the first element in the list and removes it from the list.

Returns:

0 If the operation can be executed successfully.
1 If the list is empty.

Parameters:

elem If the list is nonemty, the first element is assigned to *elem*.

6.59.3.4 template<class Type> ABA_DLISTITEM<Type>* ABA_DLIST< Type >::first () const

Returns a pointer to the first item of the list.

6.59.3.5 template<class Type> int ABA_DLIST< Type >::firstElem (Type & *elem*) const

Retrieves the first element of the list.

Returns:

0 If the list is not empty,
1 otherwise.

Parameters:

elem Stores the first element of the list after the function call if the list is not empty.

6.59.3.6 template<class Type> ABA_DLISTITEM<Type>* ABA_DLIST< Type >::last () const

Returns a pointer to the last item of the list.

6.59.3.7 template<class Type> const ABA_DLIST<Type>& ABA_DLIST< Type >::operator= (const ABA_DLIST< Type > & *rhs*) `[private]`

6.59.3.8 template<class Type> void ABA_DLIST< Type >::remove (ABA_DLISTITEM< Type > ∗ *item*)

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.

Parameters:

elem The element which should be removed.

6.59.3.9 template<class Type> void ABA_DLIST< Type >::remove (const Type & *elem*)

The function *remove()* removes *item* from the list.

6.59.3.10 template<class Type> int ABA_DLIST< Type >::removeHead ()

If the list is non-empty, the function *removeHead()* removes the head of the list.

Returns:

0 If the list is non-empty before the function is called,
1 otherwise.

6.59.4 Friends And Related Function Documentation

6.59.4.1 template<class Type> ostream& operator<< (ostream &, const ABA_DLIST< Type > & *list*) `[friend]`

The output operator writes all elements of the *list* on an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

list The list being output.

6.59.5 Member Data Documentation

6.59.5.1 template<class Type> ABA_DLISTITEM<Type>* ABA_DLIST< Type >::first_ `[private]`

A pointer to the first item of the list.

Definition at line 166 of file dlist.h.

6.59.5.2 template<class Type> ABA_GLOBAL* ABA_DLIST< Type >::glob_ `[private]`

A pointer to corresponding global object.

Definition at line 162 of file dlist.h.

6.59.5.3 template<class Type> ABA_DLISTITEM<Type>* ABA_DLIST< Type >::last_ `[private]`

A pointer to the last item in the list.

Definition at line 170 of file dlist.h.

The documentation for this class was generated from the following file:

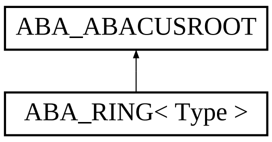
- Include/abacus/dlist.h

6.60 ABA_RING< Type > Class Template Reference

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

`#include <ring.h>`

Inheritance diagram for ABA_RING< Type >::



Public Member Functions

- ABA_RING (ABA_GLOBAL ∗glob, int size)
- virtual ∼ABA_RING ()

  The destructor.

- Type & operator[ ] (int i)
- const Type & operator[ ] (int i) const

  The operator [] is overloaded for constant use.

- 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)

Private Attributes

- ABA_GLOBAL ∗ glob_
- ABA_ARRAY< Type > ring_
- int head_
- bool filled_

Friends

- ostream & operator<< (ostream &out, const ABA_RING< Type > &ring)

  The output operator writes the elements of the ring to an output stream starting with the oldest element in the ring.

6.60.1 Detailed Description

template<class Type> class ABA_RING< Type >

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

Definition at line 49 of file ring.h.

6.60.2 Constructor & Destructor Documentation

6.60.2.1 template<class Type> ABA_RING< Type >::ABA_RING (ABA_GLOBAL ∗ *glob*, int *size*)

The constructor.

Parameters:

glob A pointer to the corresponding global object.

size The length of the ring.

6.60.2.2 template<class Type> virtual ABA_RING< Type >::~ABA_RING () `[virtual]`

The destructor.

6.60.3 Member Function Documentation

6.60.3.1 template<class Type> void ABA_RING< Type >::clear ()

Empties the ring.

6.60.3.2 template<class Type> bool ABA_RING< Type >::empty () const

Returns:

true If no element is contained in the ring,
false otherwise.

6.60.3.3 template<class Type> bool ABA_RING< Type >::filled () const

Returns:

true If the ABA_RING is completely filled up,
false otherwise.

6.60.3.4 template<class Type> void ABA_RING< Type >::insert (Type *elem*)

Inserts a new element into the ring.

If the ring is already full, this operation overwrites the oldest element in the ring.

Parameters:

elem The element being inserted.

6.60.3.5 template<class Type> Type ABA_RING< Type >::newest () const

Returns:

The newest element in the ring.
The result is undefined if the ring is empty.

6.60.3.6 template<class Type> int ABA_RING< Type >::newestIndex () const

Returns:

The index of the newest element in the ring.
The result is undefined if the ring is empty.

6.60.3.7 template<class Type> int ABA_RING< Type >::number () const

Returns:

The current number of elements in the ring.

6.60.3.8 template<class Type> Type ABA_RING< Type >::oldest () const

Returns:

The oldest element in the ring.
The result is undefined, if the ring is empty.

6.60.3.9 template<class Type> int ABA_RING< Type >::oldestIndex () const

Returns:

The index of the oldest element in the ring.
The result is undefined, if the ring is empty.

6.60.3.10]

template< class Type> const Type& ABA_RING< Type >::operator[ ] (int *i*) const

The operator [] is overloaded for constant use.

6.60.3.11]

template<class Type> Type& ABA_RING< Type >::operator[ ] (int *i*)

Returns:

The *i-th* element of the ring. The operation is undefined if no element has been inserted in the *i-th* position so far.

Parameters:

i The element being accessed.

6.60.3.12 template<class Type> int ABA_RING< Type >::previous (int *i*, Type & *p*) const

Can be used to access any element between the oldest and newest inserted element.

Returns:

0 If there are enough elements in the ring such that the element *i* entries before the newest one could be accessed,
1 otherwise.

Parameters:

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.

6.60.3.13 template<class Type> void ABA_RING< Type >::realloc (int *newSize*)

Changes the length of the ring.

Parameters:

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.

6.60.3.14 template<class Type> int ABA_RING< Type >::size () const

Returns:

The size of the ring.

6.60.4 Friends And Related Function Documentation

6.60.4.1 template<class Type> ostream& operator<< (ostream & *out*, const ABA_RING< Type > & *ring*) `[friend]`

The output operator writes the elements of the ring to an output stream starting with the oldest element in the ring.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The ring being output.

6.60.5 Member Data Documentation

6.60.5.1 template<class Type> bool ABA_RING< Type >::filled_ `[private]`

This member becomes *true* if ring is completely filled up.

Definition at line 189 of file ring.h.

6.60.5.2 template<class Type> ABA_GLOBAL∗ ABA_RING< Type >::glob_ `[private]`

A pointer to the corresponding global object.

Definition at line 177 of file ring.h.

6.60.5.3 template<class Type> int ABA_RING< Type >::head_ `[private]`

The position in the array *ring_* where the next element will be inserted.

Definition at line 185 of file ring.h.

6.60.5.4 template<class Type> ABA_ARRAY<Type> ABA_RING< Type >::ring_ `[private]`

{An array storing the elements of the ring.

Definition at line 181 of file ring.h.

The documentation for this class was generated from the following file:

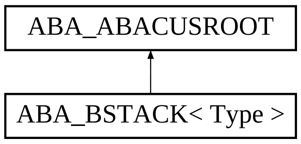
- Include/abacus/ring.h

6.61 ABA_BSTACK< Type > Class Template Reference

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.

`#include <bstack.h>`

Inheritance diagram for ABA_BSTACK< Type >::



Public Member Functions

- ABA_BSTACK (ABA_GLOBAL ∗glob, int size)
- int size () const
- int tos () const
- bool empty () const
- bool full () const
- void push (Type item)
- Type top () const
- Type pop ()

  Accesses like top() the last element pushed on the stack and removes in addition this item from the stack.

- void realloc (int newSize)

Private Attributes

- ABA_GLOBAL ∗ glob_
- ABA_ARRAY< Type > stack_
- int tos_

Friends

- ostream & operator<< (ostream &out, const ABA_BSTACK< Type > &rhs)

  The output operator writes the numbers of all stacked elements and the elements line by line on an output stream.

6.61.1 Detailed Description

template<class Type> class ABA_BSTACK< Type >

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.

Definition at line 56 of file bstack.h.

6.61.2 Constructor & Destructor Documentation

6.61.2.1 template<class Type> ABA_BSTACK< Type >::ABA_BSTACK (ABA_GLOBAL $*$ *glob*, int *size*)

The constructor initializes an empty stack.

Parameters:

glob A pointer to the corresponding global object.

size The maximal number of elements the stack can store.

6.61.3 Member Function Documentation

6.61.3.1 template<class Type> bool ABA_BSTACK< Type >::empty () const

Returns:

true If there is no element in the stack,
false otherwise.

6.61.3.2 template<class Type> bool ABA_BSTACK< Type >::full () const

Returns:

true If the maximal number of elements has been inserted in the stack,
false otherwise.

6.61.3.3 template<class Type> Type ABA_BSTACK< Type >::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. If this function is compiled with -*DABACUSSAFE*, then the program terminates if this error occurs.

Returns:

The last element pushed on the stack.

6.61.3.4 template<class Type> void ABA_BSTACK< Type >::push (Type *item*)

Adds an element to the stack.

It is a fatal error to insert an element if the stack is full. If this function is compiled with -*DABACUSSAFE*, then the program terminates if this error occurs.

Parameters:

item The element added to the stack.

6.61.3.5 template<class Type> void ABA_BSTACK< Type >::realloc (int *newSize*)

Changes the maximal number of elements of the stack.

Parameters:

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.

6.61.3.6 template<class Type> int ABA_BSTACK< Type >::size () const

Returns:

The maximal number of elements which can be inserted into the stack.

6.61.3.7 template<class Type> Type ABA_BSTACK< Type >::top () const

Accesses the last element pushed on the stack without removing it.

It is an error to perform this operation on an empty stack. If this function is compiled with *-DABACUSSAFE*, then the program terminates if this error occurs.

Returns:

The last element pushed on the stack.

6.61.3.8 template<class Type> int ABA_BSTACK< Type >::tos () const

Returns:

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.

6.61.4 Friends And Related Function Documentation

6.61.4.1 template<class Type> ostream& operator<< (ostream & *out*, const ABA_BSTACK< Type > & *rhs*) `[friend]`

The output operator writes the numbers of all stacked elements and the elements line by line on an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The stack being output.

6.61.5 Member Data Documentation

6.61.5.1 template<class Type> ABA_GLOBAL∗ ABA_BSTACK< Type >::glob_ [private]

A pointer to the corresponding global object.

Definition at line 157 of file bstack.h.

6.61.5.2 template<class Type> ABA_ARRAY<Type> ABA_BSTACK< Type >::stack_ [private]

The array storing the elements of the stack.

Definition at line 161 of file bstack.h.

6.61.5.3 template<class Type> int ABA_BSTACK< Type >::tos_ [private]

The top of stack (next free component).

Definition at line 165 of file bstack.h.

The documentation for this class was generated from the following file:

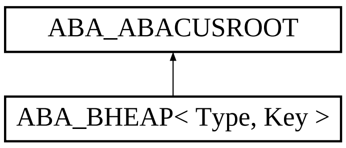
- Include/abacus/bstack.h

6.62 ABA_BHEAP< Type, Key > Class Template Reference

This template class implements a heap with a fixed maximal size, however a reallocation can be performed if required.

`#include <bheap.h>`

Inheritance diagram for ABA_BHEAP< Type, Key >::



Public Member Functions

- ABA_BHEAP (ABA_GLOBAL ∗glob, int size)
- ABA_BHEAP (ABA_GLOBAL ∗glob, const ABA_BUFFER< Type > &elems, const ABA_BUFFER< Key > &keys)
- 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)
- void check () const

Private Member Functions

- int leftSon (int i) const
- int rightSon (int i) const
- int father (int i) const
- void heapify (int i)

Private Attributes

- ABA_GLOBAL ∗ glob_
- ABA_ARRAY< Type > heap_
- ABA_ARRAY< Key > keys_
- int n_

Friends

- ostream & operator<< (ostream &out, const ABA_BHEAP< Type, Key > &rhs)

  The output operator writes the elements of the heap together with their keys on an output stream.

6.62.1 Detailed Description

template<class Type, class Key> class ABA_BHEAP< Type, Key >

This template class implements a heap with a fixed maximal size, however a reallocation can be performed if required.

Definition at line 74 of file bheap.h.

6.62.2 Constructor & Destructor Documentation

6.62.2.1 template<class Type, class Key> ABA_BHEAP< Type, Key >::ABA_BHEAP (ABA_GLOBAL ∗ *glob*, int *size*)

A constructor.

Parameters:

glob A pointer to the corresponding global object.

size The maximal number of elements which can be stored.

6.62.2.2 template<class Type, class Key> ABA_BHEAP< Type, Key >::ABA_BHEAP (ABA_GLOBAL ∗ *glob*, const ABA_BUFFER< Type > & *elems*, const ABA_BUFFER< Key > & *keys*)

A constructor with initialization.

The heap is initialized from an ABA_BUFFER of elements and a corresponding ABA_BUFFER of keys. The running time is O(n) for n elements.

Parameters:

glob A pointer to the corresponding global object.

elem A ABA_BUFFER wich contains the elements.

elem A ABA_BUFFER wich contains the keys.

6.62.3 Member Function Documentation

6.62.3.1 template<class Type, class Key> void ABA_BHEAP< Type, Key >::check () const

Stops with an error message if the heap properties are violated.

This function is used for debugging this class.

6.62.3.2 template<class Type, class Key> void ABA_BHEAP< Type, Key >::clear ()

Empties the heap.

6.62.3.3 template<class Type, class Key> bool ABA_BHEAP< Type, Key >::empty () const

Returns:

true If there are no elements in the heap,
false otherwise.

6.62.3.4 template<class Type, class Key> Type ABA_BHEAP< Type, Key >::extractMin ()

Accesses and removes the minimum element from the heap.

The minimum element is stored in the root of the tree, i.e., in *heap_*[0]. If the heap does not become empty by removing the minimal element, we move the last element stored in *heap_* to the root (*heap_*[0]). Whereas this operation can destroy the heap property, the two subtrees rooted at nodes 1 and 2 still satisfy the heap property. Hence calling *heapify(0)* will restore the overall heap property.

Returns:

The minimum element of the heap.

6.62.3.5 template<class Type, class Key> int ABA_BHEAP< Type, Key >::father (int *i*) const `[private]`

Returns the index of the father of element *i*.

6.62.3.6 template<class Type, class Key> Type ABA_BHEAP< Type, Key >::getMin () const

Returns:

The minimum element of the heap. This operation must not be performed if the heap is empty. Since the heap property holds the element having minimal key is always stored in *heap*_[0].

6.62.3.7 template<class Type, class Key> Key ABA_BHEAP< Type, Key >::getMinKey () const

Returns:

The key of the minimum element of the heap. This operation must not be performed if the heap is empty. Since the heap property holds the element having minimal key is always stored in *heap*_[0] and its key in *key*_[0].

6.62.3.8 template<class Type, class Key> void ABA_BHEAP< Type, Key >::heapify (int *i*) `[private]`

Is the central function to maintain the heap property.

The function assumes that the two trees rooted at *left(i)* and *right(i)* fulfil already the heap property and restores the heap property of the (sub-) tree rooted at *i*.

6.62.3.9 template<class Type, class Key> void ABA_BHEAP< Type, Key >::insert (Type *elem*, Key *key*)

Inserts an item with a key into the heap.

It is a fatal error to perform this operation if the heap is full. If the precompiler flag *-DABACUSSAFE* is set, we check if the heap is not already full.

Parameters:

elem The element being inserted into the heap.

key The key of this element.

6.62.3.10 template<class Type, class Key> int ABA_BHEAP< Type, Key >::leftSon (int *i*) const `[private]`

Returns the index of the left son of node *i*.

6.62.3.11 template<class Type, class Key> int ABA_BHEAP< Type, Key >::number () const

Returns:
The number of elements in the heap.

6.62.3.12 template<class Type, class Key> void ABA_BHEAP< Type, Key >::realloc (int *newSize*)

Changes the size of the heap.

Parameters:
newSize The new maximal number of elements in the heap.

6.62.3.13 template<class Type, class Key> int ABA_BHEAP< Type, Key >::rightSon (int *i*) const `[private]`

Returns the index of the right son of node *i*.

6.62.3.14 template<class Type, class Key> int ABA_BHEAP< Type, Key >::size () const

Returns:
The maximal number of elements which can be stored in the heap.

6.62.4 Friends And Related Function Documentation

6.62.4.1 template<class Type, class Key> ostream& operator<< (ostream & *out*, const ABA_BHEAP< Type, Key > & *rhs*) `[friend]`

The output operator writes the elements of the heap together with their keys on an output stream.

Returns:
A reference to the output stream.

Parameters:
out The output stream.
rhs The heap being output.

6.62.5 Member Data Documentation

6.62.5.1 template<class Type, class Key> ABA_GLOBAL∗ ABA_BHEAP< Type, Key >::glob_ `[private]`

Definition at line 216 of file bheap.h.

6.62.5.2 template<class Type, class Key> ABA_ARRAY<Type> ABA_BHEAP< Type, Key >::heap_ `[private]`

Definition at line 217 of file bheap.h.

6.62.5.3 template<class Type, class Key> ABA_ARRAY<Key> ABA_BHEAP< Type, Key >::keys_ `[private]`

Definition at line 218 of file bheap.h.

6.62.5.4 template<class Type, class Key> int ABA_BHEAP< Type, Key >::n_ `[private]`

Definition at line 219 of file bheap.h.

The documentation for this class was generated from the following file:

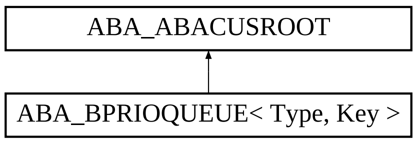
- Include/abacus/bheap.h

6.63 ABA_BPRIOQUEUE< Type, Key > Class Template Reference

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.

`#include <bprioqueue.h>`

Inheritance diagram for ABA_BPRIOQUEUE< Type, Key >::



Public Member Functions

- 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)

  Extends the function getMin(min) *in the way that the minimal element is also removed from the priority queue.*

- void clear ()
- int size () const
- int number () const
- void realloc (int newSize)

Private Attributes

- ABA_GLOBAL * glob_
- ABA_BHEAP< Type, Key > heap_

6.63.1 Detailed Description

template<class Type, class Key> class ABA_BPRIOQUEUE< Type, 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.

Definition at line 57 of file bprioqueue.h.

6.63.2 Constructor & Destructor Documentation

6.63.2.1 template<class Type, class Key> ABA_BPRIOQUEUE< Type, Key >::ABA_BPRIOQUEUE (ABA_GLOBAL * *glob*, int *size*)

The constructor of an empty priority queue.

Parameters:

glob A pointer to the corresponding object.

size The maximal number of elements the priority queue can hold without reallocation.

6.63.3 Member Function Documentation

6.63.3.1 template<class Type, class Key> void ABA_BPRIOQUEUE< Type, Key >::clear ()

Makes the priority queue empty.

6.63.3.2 template<class Type, class Key> int ABA_BPRIOQUEUE< Type, Key >::extractMin (Type & *min*)

Extends the function *getMin(min)* in the way that the minimal element is also removed from the priority queue.

Returns:

0 If the priority queue is non-empty,
1 otherwise.

Parameters:

min If the priority queue is non-empty the minimal element is assigned to *min*.

6.63.3.3 template<class Type, class Key> int ABA_BPRIOQUEUE< Type, Key >::getMin (Type & *min*) const

Retrieves the element with minimal key from the priority queue.

Returns:

0 If the priority queue is non-empty,
1 otherwise.

Parameters:

min If the priority queue is non-empty the minimal element is assigned to *min*.

6.63.3.4 template<class Type, class Key> int ABA_BPRIOQUEUE< Type, Key >::getMinKey (Key & *minKey*) const

Retrieves the key of the minimal element in the priority queue.

Returns:

0 If the priority queue is non-empty,
1 otherwise.

Parameters:

minKey Holds after the call the key of the minimal element in the priority queue, if the queue is non-emtpy.

6.63.3.5 template<class Type, class Key> void ABA_BPRIOQUEUE< Type, Key >::insert (Type *elem*, Key *key*)

Inserts an element in the priority queue.

Parameters:

elem The element being inserted.

key The key of the element.

6.63.3.6 template<class Type, class Key> int ABA_BPRIOQUEUE< Type, Key >::number () const

Returns:

The number of elements stored in the priority queue.

6.63.3.7 template<class Type, class Key> void ABA_BPRIOQUEUE< Type, Key >::realloc (int *newSize*)

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.

Parameters:

newSize The new size of the priority queue.

6.63.3.8 template<class Type, class Key> int ABA_BPRIOQUEUE< Type, Key >::size () const

Returns:

The maximal number of elements which can be stored in the priority queue.

6.63.4 Member Data Documentation

6.63.4.1 template<class Type, class Key> ABA_GLOBAL∗ ABA_BPRIOQUEUE< Type, Key >::glob_ `[private]`

A pointer to the corresponding global object.

Definition at line 131 of file bprioqueue.h.

6.63.4.2 template<class Type, class Key> ABA_BHEAP<Type, Key> ABA_BPRIOQUEUE< Type, Key >::heap_ `[private]`

The heap implementing the priority queue.

Definition at line 135 of file bprioqueue.h.

The documentation for this class was generated from the following file:

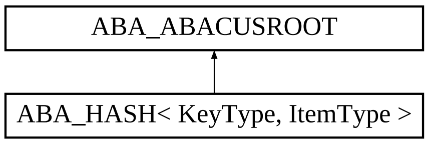
- Include/abacus/bprioqueue.h

6.64 ABA_HASH< KeyType, ItemType > Class Template Reference

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.

`#include <hash.h>`

Inheritance diagram for ABA_HASH< KeyType, ItemType >::



Public Member Functions

- ABA_HASH (ABA_GLOBAL ∗glob, int size)

    Initializes each slot with a 0-pointer to indicate that the linked list of hash items of this slot is empty.

- ∼ABA_HASH ()

    The destructor deletes for each hash item by going through all non-empty lists of hash items.

- void insert (const KeyType &newKey, const ItemType &newItem)
- void overWrite (const KeyType &newKey, const ItemType &newItem)

    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.

- ItemType ∗ find (const KeyType &key)
- bool find (const KeyType &key, const ItemType &item)

    This version of the function find() checks if a prespecified item with a prespecified key is contained in the hash table.

- int remove (const KeyType &key)
- int remove (const KeyType &key, const ItemType &item)

    This version of the function remove() removes the first item with a given key and a prespecified element from the hash table.

- int size () const
- int nCollisions () const
- void resize (int newSize)

The functions initializeIteration() *and* next() *can be used to iterate through all items stored in the hash table having the same key.*

- ItemType ∗ initializeIteration (const KeyType &key)
- ItemType ∗ next (const KeyType &key)

    The function next() can be used to go to the next item in the hash table with key key.

Private Member Functions

- int hf (int key)
- int hf (unsigned key)

    This version of hf() implements a Fibonacci hash function for keys of type unsigned.

- int hf (const ABA_STRING &string)
- ABA_HASH (const ABA_HASH &rhs)
- ABA_HASH & operator= (const ABA_HASH &rhs)

Private Attributes

- ABA_GLOBAL ∗ glob_
- ABA_HASHITEM< KeyType, ItemType > ∗∗ table_
- int size_
- int nCollisions_
- ABA_HASHITEM< KeyType, ItemType > ∗ iter_

Friends

- ostream & operator<< (ostream &out, const ABA_HASH< KeyType, ItemType > &hash)

  The output operator writes row by row all elements stored in the list associated with a slot on an output stream.

6.64.1 Detailed Description

template<class KeyType, class ItemType> class ABA_HASH< KeyType, ItemType >

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.

Definition at line 137 of file hash.h.

6.64.2 Constructor & Destructor Documentation

6.64.2.1 template<class KeyType, class ItemType> ABA_HASH< KeyType, ItemType >::ABA_HASH (ABA_GLOBAL ∗ *glob*, int *size*)

Initializes each slot with a 0-pointer to indicate that the linked list of hash items of this slot is empty.

Parameters:

glob A pointer to the corresponding global object.

size The size of the hash table.

6.64.2.2 template<class KeyType, class ItemType> ABA_HASH< KeyType, ItemType >::∼ABA_HASH ()

The destructor deletes for each hash item by going through all non-empty lists of hash items.

6.64.2.3 template<class KeyType, class ItemType> ABA_HASH< KeyType, ItemType >::ABA_HASH (const ABA_HASH< KeyType, ItemType > & *rhs*) `[private]`

6.64.3 Member Function Documentation

6.64.3.1 template<class KeyType, class ItemType> bool ABA_HASH< KeyType, ItemType >::find (const KeyType & *key*, const ItemType & *item*)

This version of the function *find()* checks if a prespecified item with a prespecified key is contained in the hash table.

Returns:
true If there is an element (key, item) in the hash table,
false otherwise.

Parameters:
key The key of the item.
item The searched item.

6.64.3.2 template<class KeyType, class ItemType> ItemType∗ ABA_HASH< KeyType, ItemType >::find (const KeyType & *key*)

Looks for an item in the hash table with a given key.

Returns:
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.

Parameters:
key The key of the searched item.

6.64.3.3 template<class KeyType, class ItemType> int ABA_HASH< KeyType, ItemType >::hf (const ABA_STRING & *string*) `[private]`

This is a hash function for character strings.

It is taken from Knu93a}, page 300.

6.64.3.4 template<class KeyType, class ItemType> int ABA_HASH< KeyType, ItemType >::hf (unsigned *key*) `[private]`

This version of *hf()* implements a Fibonacci hash function for keys of type *unsigned*.

6.64.3.5 template<class KeyType, class ItemType> int ABA_HASH< KeyType, ItemType >::hf (int *key*) `[private]`

Computes the hash value of *key*.

It must be overloaded for all key types, which are used together with this template.

This following version of *hf()* implements a Fibonacci hash function for keys of type *int*.

6.64.3.6 template<class KeyType, class ItemType> ItemType* ABA_HASH< KeyType, ItemType >::initializeIteration (const KeyType & *key*)

The function *initializeIteration()* retrieves the first item.

Returns:
A pointer to the first item found in the hash table having key *key*, or 0 if there is no such item.

Parameters:
key The key of the items through which we want to iterate.

6.64.3.7 template<class KeyType, class ItemType> void ABA_HASH< KeyType, ItemType >::insert (const KeyType & *newKey*, const ItemType & *newItem*)

Adds an item to the hash table.

The new item is inserted at the head of the list in the corresponding slot. It is possible to insert several items with the same key into the hash table.

Parameters:
key The key of the new item.

item The item being inserted.

6.64.3.8 template<class KeyType, class ItemType> int ABA_HASH< KeyType, ItemType >::nCollisions () const

Returns:
The number of collisions which occurred during all previous calls of the functions *insert()* and *overWrite()*.

6.64.3.9 template<class KeyType, class ItemType> ItemType* ABA_HASH< KeyType, ItemType >::next (const KeyType & *key*)

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.

Returns:

A pointer to the next item having key *key*, or 0 if there is no more item with this key in the hash table.

Parameters:

key The key of the items through which we want to iterate.

6.64.3.10 template<class KeyType, class ItemType> ABA_HASH& ABA_HASH< KeyType, ItemType >::operator= (const ABA_HASH< KeyType, ItemType > & *rhs*) `[private]`

6.64.3.11 template<class KeyType, class ItemType> void ABA_HASH< KeyType, ItemType >::overWrite (const KeyType & *newKey*, const ItemType & *newItem*)

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.

Parameters:

key The key of the new item.

item The item being inserted.

6.64.3.12 template<class KeyType, class ItemType> int ABA_HASH< KeyType, ItemType >::remove (const KeyType & *key*, const ItemType & *item*)

This version of the function *remove()* removes the first item with a given key and a prespecified element from the hash table.

Returns:

0 If an item with the key is found.
1 If there is no item with this key.

Parameters:

key The key of the item that should be removed.

item The item which is searched.

6.64.3.13 template<class KeyType, class ItemType> int ABA_HASH< KeyType, ItemType >::remove (const KeyType & *key*)

Removes the first item with a given key from the hash table.

Returns:

0 If an item with the key is found.
1 If there is no item with this key.

Parameters:

key The key of the item that should be removed.

6.64.3.14 template<class KeyType, class ItemType> void ABA_HASH< KeyType, ItemType >::resize (int *newSize*)

Can be used to change the size of the hash table.

Parameters:

newSize The new size of the hash table (must be positive).

6.64.3.15 template<class KeyType, class ItemType> int ABA_HASH< KeyType, ItemType >::size () const

Returns:

The length of the hash table.

6.64.4 Friends And Related Function Documentation

6.64.4.1 template<class KeyType, class ItemType> ostream& operator<< (ostream & *out*, const ABA_HASH< KeyType, ItemType > & *hash*) `[friend]`

The output operator writes row by row all elements stored in the list associated with a slot on an output stream.

The output of an empty slot is suppressed.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The hash table being output.

6.64.5 Member Data Documentation

6.64.5.1 template<class KeyType, class ItemType> ABA_GLOBAL* ABA_HASH< KeyType, ItemType >::glob_ [private]

A pointer to the corresponding global object.

Definition at line 331 of file hash.h.

6.64.5.2 template<class KeyType, class ItemType> ABA_HASHITEM<KeyType, ItemType>* ABA_HASH< KeyType, ItemType >::iter_ [private]

An iterator for all items stored in a slot.

This variable is initialized by calling *initializeIteration()* and incremented by the function *next()*.

Definition at line 355 of file hash.h.

6.64.5.3 template<class KeyType, class ItemType> int ABA_HASH< KeyType, ItemType >::nCollisions_ [private]

The number of collisions on calls of *insert()* and *overWrite()*.

Definition at line 347 of file hash.h.

6.64.5.4 template<class KeyType, class ItemType> int ABA_HASH< KeyType, ItemType >::size_ [private]

The length of the hash table.

Definition at line 343 of file hash.h.

6.64.5.5 template<class KeyType, class ItemType> ABA_HASHITEM<KeyType, ItemType>** ABA_HASH< KeyType, ItemType >::table_ [private]

The hash table storing a linked list of hash items in each slot.

*table*_[i] is initialized with a 0-pointer in order to indicate that it is empty. The linked lists of each slot are terminated with a 0-pointer, too.

Definition at line 339 of file hash.h.

The documentation for this class was generated from the following file:

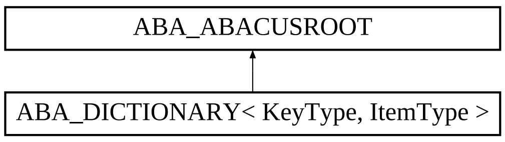
- Include/abacus/hash.h

6.65 ABA_DICTIONARY< KeyType, ItemType > Class Template Reference

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.

`#include <dictionary.h>`

Inheritance diagram for ABA_DICTIONARY< KeyType, ItemType >::



Public Member Functions

- void insert (const KeyType &key, const ItemType &item)
- ItemType ∗ lookUp (const KeyType &key)

Public Attributes

- ABA_DICTIONARYABA_GLOBAL ∗ glob
- ABA_DICTIONARYABA_GLOBAL int size

Private Member Functions

- ABA_DICTIONARY (const ABA_DICTIONARY< KeyType, ItemType > &rhs)
- const ABA_DICTIONARY & operator= (const ABA_DICTIONARY< KeyType, ItemType > &rhs)

Private Attributes

- ABA_GLOBAL ∗ glob_
- ABA_HASH< KeyType, ItemType > hash_

Friends

- ostream & operator<< (ostream &out, const ABA_DICTIONARY< KeyType, ItemType > &rhs)

  The output operator writes the hash table implementing the dictionary on an output stream.

6.65.1 Detailed Description

template<class KeyType, class ItemType> class ABA_DICTIONARY< KeyType, ItemType >

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.

Definition at line 47 of file dictionary.h.

6.65.2 Constructor & Destructor Documentation

6.65.2.1 template<class KeyType, class ItemType> ABA_DICTIONARY< KeyType, ItemType >::ABA_DICTIONARY (const ABA_DICTIONARY< KeyType, ItemType > & *rhs*) `[private]`

6.65.3 Member Function Documentation

6.65.3.1 template<class KeyType, class ItemType> void ABA_DICTIONARY< KeyType, ItemType >::insert (const KeyType & *key*, const ItemType & *item*)

Adds the item together with a key to the dictionary.

Parameters:

key The key of the new item.

item The new item.

6.65.3.2 template<class KeyType, class ItemType> ItemType∗ ABA_DICTIONARY< KeyType, ItemType >::lookUp (const KeyType & *key*)

Returns:

A pointer to the item associated with *key* in the ABA_DICTIONARY, or 0 if there is no such item.

Parameters:

key The key of the searched item.

6.65.3.3 template<class KeyType, class ItemType> const ABA_DICTIONARY& ABA_DICTIONARY< KeyType, ItemType >::operator= (const ABA_DICTIONARY< KeyType, ItemType > & *rhs*) `[private]`

6.65.4 Friends And Related Function Documentation

6.65.4.1 template<class KeyType, class ItemType> ostream& operator<< (ostream & *out*, const ABA_DICTIONARY< KeyType, ItemType > & *rhs*) `[friend]`

The output operator writes the hash table implementing the dictionary on an output stream.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The hash table being output.

6.65.5 Member Data Documentation

6.65.5.1 template<class KeyType, class ItemType> ABA_DICTIONARYABA_GLOBAL∗ ABA_DICTIONARY< KeyType, ItemType >::glob

The constructor.

Parameters:

glob A pointer to the corresponding global object.

size The size of the hash table implementing the dictionary.

Definition at line 55 of file dictionary.h.

6.65.5.2 template<class KeyType, class ItemType> ABA_GLOBAL∗ ABA_DICTIONARY< KeyType, ItemType >::glob_ `[private]`

A pointer to the corresponding global object.

Definition at line 86 of file dictionary.h.

6.65.5.3 template<class KeyType, class ItemType> ABA_HASH<KeyType, ItemType> ABA_DICTIONARY< KeyType, ItemType >::hash_ `[private]`

The hash table implementing the dictionary.

Definition at line 90 of file dictionary.h.

6.65.5.4 template<class KeyType, class ItemType> ABA_DICTIONARYABA_GLOBAL int ABA_DICTIONARY< KeyType, ItemType >::size

The constructor.

Parameters:

glob A pointer to the corresponding global object.

size The size of the hash table implementing the dictionary.

Definition at line 55 of file dictionary.h.

The documentation for this class was generated from the following file:

- Include/abacus/dictionary.h

6.66 Tools

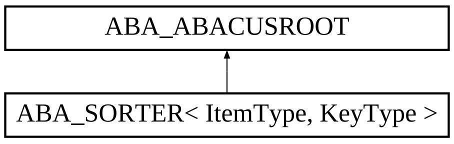
This section documents some tools for sorting objects, measuring time, and generating output.

6.67 ABA_SORTER< ItemType, KeyType > Class Template Reference

This class implements several functions for sorting arrays according to increasing keys.

`#include <sorter.h>`

Inheritance diagram for ABA_SORTER< ItemType, KeyType >::



Public Member Functions

- ABA_SORTER (ABA_GLOBAL ∗glob)
- void quickSort (int n, ABA_ARRAY< ItemType > &items, ABA_ARRAY< KeyType > &keys)

  Sorts the elements of an array of n *items according to their 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)

Private Member Functions

- int partition (ABA_ARRAY< ItemType > &items, ABA_ARRAY< KeyType > &keys, int left, int right)

  Returns a number q *({{left <= q <= right)} and guarantees that all elements* i *with {key[i] <= key[q]}} are stored in the left part of the array, i.e., in* items*[left], ,* items*[q], and all elements* j *with* key*[j] > key[q] are stored in the right part of the array, i.e., in* items*[q+1], . . . ,* items*[right].*

- void buildHeap (int n, ABA_ARRAY< ItemType > &items, ABA_ARRAY< KeyType > &keys)

  Resorts the elements if items *and* keys *such that the heap property holds, i.e.,* keys*[i] >= keys[2∗i+1] and* keys*[i] >= keys[2∗i+2].*

- void heapify (int n, ABA_ARRAY< ItemType > &items, ABA_ARRAY< KeyType > &keys, int root)

  Assumes that the heap property holds for the subtrees rooted at the sons of root *and restores the heap property for the subtree rooted at* root*.*

- void check (int n, ABA_ARRAY< ItemType > &items, ABA_ARRAY< KeyType > &keys)

  Is a debugging function and terminates the program with an error message if the elements of items *are not sorted by increasing keys.*

Private Attributes

- ABA_GLOBAL ∗ glob_
- ItemType itemSwap_
- KeyType keySwap_

6.67.1 Detailed Description

template<class ItemType, class KeyType> class ABA_SORTER< ItemType, KeyType >

This class implements several functions for sorting arrays according to increasing keys.

Definition at line 46 of file sorter.h.

6.67.2 Constructor & Destructor Documentation

6.67.2.1 template<class ItemType, class KeyType> ABA_SORTER< ItemType, KeyType >::ABA_SORTER (ABA_GLOBAL ∗ *glob*)

The constructor.

Parameters:

glob A pointer to the corresponding global object.

6.67.3 Member Function Documentation

6.67.3.1 template<class ItemType, class KeyType> void ABA_SORTER< ItemType, KeyType >::buildHeap (int *n*, ABA_ARRAY< ItemType > & *items*, ABA_ARRAY< KeyType > & *keys*) `[private]`

Resorts the elements if *items* and *keys* such that the heap property holds, i.e., *keys*[i] >= keys[2∗i+1] and *keys*[i] >= keys[2∗i+2].

Parameters:

n The number of elements of the following arrays.

items The items being sorted.

keys The keys for sorting the items.

The function *heapify()* is called for each node of the tree which is not necessarily a leaf. First nodes on higher level in the tree processed.

6.67.3.2 template<class ItemType, class KeyType> void ABA_SORTER< ItemType, KeyType >::check (int *n*, ABA_ARRAY< ItemType > & *items*, ABA_ARRAY< KeyType > & *keys*) `[private]`

Is a debugging function and terminates the program with an error message if the elements of *items* are not sorted by increasing keys.

Parameters:

n The number of elements of the following arrays.

items The items being sorted.

keys The keys for sorting the items.

6.67.3.3 template<class ItemType, class KeyType> void ABA_SORTER< ItemType, KeyType >::heapify (int *n*, ABA_ARRAY< ItemType > & *items*, ABA_ARRAY< KeyType > & *keys*, int *root*) `[private]`

Assumes that the heap property holds for the subtrees rooted at the sons of *root* and restores the heap property for the subtree rooted at *root*.

Parameters:

n The number of elements of the following arrays.

items The items being sorted.

keys The keys for sorting the items.

root The index where the heaps property has to be restored.

The function *heapify()* checks if the heap property holds for *root*. This is not the case if the *largest* element of *l*, *r*, and *root* is not *root*. In this case the elements of *root* and *largest* are swapped and we iterate. Otherwise, the heap property is restored.

6.67.3.4 template<class ItemType, class KeyType> void ABA_SORTER< ItemType, KeyType >::heapSort (int *n*, ABA_ARRAY< ItemType > & *items*, ABA_ARRAY< KeyType > & *keys*)

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$(n \log n)$.

The function *heapSort()* generates a heap. This guarantees that the largest element is stored in *items*[0]. So it is obvious that if we want to sort the items by increasing keys, this element will finally be stored in *items*[n-1]. Hence we swap the *items* and *keys* of 0 and *n-1* and restore the heap property for the elements 0,, *n-2*. This can be done by *heapify()* since the subtree rooted at 1 and 2 are still heaps (the last element is not considered anymore). This process is iterated until the elements are sorted.

Parameters:

n The number of items being sorted.

items The items being sorted.

keys The keys of the items.

6.67.3.5 template<class ItemType, class KeyType> int ABA_SORTER< ItemType, KeyType >::partition (ABA_ARRAY< ItemType > & *items*, ABA_ARRAY< KeyType > & *keys*, int *left*, int *right*) `[private]`

Returns a number *q* ({{left <= q <= right)} and guarantees that all elements *i* with {key[i] <= key[q]}} are stored in the left part of the array, i.e., in *items*[left], , *items*[q], and all elements *j* with *key*[j] > key[q] are stored in the right part of the array, i.e., in *items*[q+1], . . . , *items*[right].

Parameters:

items The items being sorted.

keys The keys for sorting the items.

left The left border of the partial array being considered.

right The right border ot the partial array being considered.

First, we determine a pivot element *k*. The *while* loop stops by returning *r* as soon as the elements are partitioned in two subsets such all elements *i* in *left* <= i <= r have a smaller key than the elements i with *r+1* <= right.

The *do-loops* stop as soon as a pair of elements is found violating the partition property. This pair of elements is the swapped together with their keys.

6.67.3.6 template<class ItemType, class KeyType> void ABA_SORTER< ItemType, KeyType >::quickSort (ABA_ARRAY< ItemType > & *items*, ABA_ARRAY< KeyType > & *keys*, int *left*, int *right*)

Sorts an partial array.

The function *quickSort()* uses the divide-and-conquer technique. First the function *partition()* puts the small elements to the left part and all big elements to the right part of the array being sorted. More precisely, it holds then, *keys*[i] <= keys[q] for all *i* in *left*, *q* and *keys*[q] < keys[i] for all *i* in *q+1*, *right*. Hence, it is sufficient to sort these two subarrays recursively.

Parameters:

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.

6.67.3.7 template<class ItemType, class KeyType> void ABA_SORTER< ItemType, KeyType >::quickSort (int *n*, ABA_ARRAY< ItemType > & *items*, ABA_ARRAY< KeyType > & *keys*)

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) .

Parameters:

n The number of elements being sorted.

items The items being sorted.

keys The keys of the sorted items.

6.67.4 Member Data Documentation

6.67.4.1 template<class ItemType, class KeyType> ABA_GLOBAL∗ ABA_SORTER< ItemType, KeyType >::glob_ `[private]`

A pointer to the corresponding global object.

Definition at line 183 of file sorter.h.

6.67.4.2 template<class ItemType, class KeyType> ItemType ABA_SORTER< ItemType, KeyType >::itemSwap_ `[private]`

An auxiliary variable for swapping items.

Definition at line 187 of file sorter.h.

6.67.4.3 template<class ItemType, class KeyType> KeyType ABA_SORTER< ItemType, KeyType >::keySwap_ `[private]`

An auxiliary variable for swapping keys.

Definition at line 191 of file sorter.h.

The documentation for this class was generated from the following file:

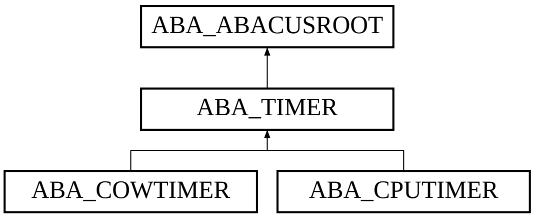
- Include/abacus/sorter.h

6.68 ABA_TIMER Class Reference

class implements a base class for timers measuring the CPU time and the wall-clock time

`#include <timer.h>`

Inheritance diagram for ABA_TIMER::



Public Member Functions

- ABA_TIMER (ABA_GLOBAL ∗glob)

  The constructor for a timer with a pointer to the global object glob*.*

- ABA_TIMER (ABA_GLOBAL ∗glob, long centiSeconds)

  This constructor initializes the total time of the timer with centiSeconds *and the pointer to the corresponding global object* glob*. The timer is not running, too.*

- virtual ∼ABA_TIMER ()

  The destructor.

- void start (bool reset=false)

  The timer is started with the function start()*. For safety starting a running timer is an error.*

- void stop ()

  Stops the timer and adds the difference between the current time and the starting time to the total time.

- void reset ()
- bool running () const
- long centiSeconds () const
- long seconds () const
- long minutes () const
- long hours () const
- bool exceeds (const ABA_STRING &maxTime) const
- void addCentiSeconds (long centiSeconds)

Protected Member Functions

- virtual long theTime () const =0

  Is required for measuring the time difference between the time of the call and some base point (e.g., the program start).

Protected Attributes

- ABA_GLOBAL ∗ glob_

Private Attributes

- long startTime_
- long totalTime_
- bool running_

Friends

- ostream & operator<< (ostream &out, const ABA_TIMER &rhs)

  The output operator writes the time in the format { hours:minutes:seconds.seconds/100} on an output stream.

6.68.1 Detailed Description

class implements a base class for timers measuring the CPU time and the wall-clock time

Definition at line 47 of file timer.h.

6.68.2 Constructor & Destructor Documentation

6.68.2.1 ABA_TIMER::ABA_TIMER (ABA_GLOBAL $*$ *glob*)

The constructor for a timer with a pointer to the global object *glob*.

After the application of the constructor the timer is not running, i.e., to measure time it has to be started explicitly.

6.68.2.2 ABA_TIMER::ABA_TIMER (ABA_GLOBAL $*$ *glob*, long *centiSeconds*)

This constructor initializes the total time of the timer with *centiSeconds* and the pointer to the corresponding global object *glob*. The timer is not running, too.

6.68.2.3 virtual ABA_TIMER::∼ABA_TIMER () `[virtual]`

The destructor.

6.68.3 Member Function Documentation

6.68.3.1 void ABA_TIMER::addCentiSeconds (long *centiSeconds*)

Parameters:

centiSeconds The number of centiseconds to be added.

6.68.3.2 long ABA_TIMER::centiSeconds () const

Returns:

The currently spent time in $\frac{1}{100}$ -seconds. It is not necessary to stop the timer to get the correct time.

6.68.3.3 bool ABA_TIMER::exceeds (const ABA_STRING & *maxTime*) const

Returns:

true If the currently spent time exceeds *maxTime*,
false otherwise.

Parameters:

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).

6.68.3.4 long ABA_TIMER::hours () const

Returns:

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.

6.68.3.5 long ABA_TIMER::minutes () const

Returns:

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.

6.68.3.6 void ABA_TIMER::reset ()

Stops the timer and sets the *totalTime* to 0.

6.68.3.7 bool ABA_TIMER::running () const

Returns:

true If the timer is running,
false otherwise.

6.68.3.8 long ABA_TIMER::seconds () const

Returns:

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.

6.68.3.9 void ABA_TIMER::start (bool *reset* = `false`)

The timer is started with the function *start()*. For safety starting a running timer is an error.

Parameters:

reset If this flag is set to true, the timer is reset before it is started (default=false)}

6.68.3.10 void ABA_TIMER::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.

6.68.3.11 virtual long ABA_TIMER::theTime () const `[protected, pure virtual]`

Is required for measuring the time difference between the time of the call and some base point (e.g., the program start).

We measure time according to the following principle.

The pure virtual function *theTime()* returns the CPU time since the start of the program for the class CPUABA_-TIMER or the elapsed time since some point in the past for the class ABA_COWTIMER. When the timer is started *startTime_* is initialized with the a value returned by *theTime()* and when it is stopped the difference between *theTime()* and *startTime_* is added to the total time.

Implemented in ABA_COWTIMER, and ABA_CPUTIMER.

6.68.4 Friends And Related Function Documentation

6.68.4.1 ostream& operator<< (ostream & *out*, const ABA_TIMER & *rhs*) `[friend]`

The output operator writes the time in the format { hours:minutes:seconds.seconds/100} on an output stream.

After the time has been divided in hours, minutes and seconds we have to take care that an additional leading zero is output if minutes or seconds have a value less than ten.

Returns:

A reference to the output stream.

Parameters:

out The output stream.

rhs The timer being output.

6.68.5 Member Data Documentation

6.68.5.1 ABA_GLOBAL$*$ ABA_TIMER::glob_ `[protected]`

A pointer to the corresponding global object.

Definition at line 176 of file timer.h.

6.68.5.2 bool ABA_TIMER::running_ `[private]`

true, if the timer is running.

Definition at line 190 of file timer.h.

6.68.5.3 long ABA_TIMER::startTime_ `[private]`

The start time of the timer in $\frac{1}{100}$ -seconds.

Definition at line 182 of file timer.h.

6.68.5.4 long ABA_TIMER::totalTime_ `[private]`

The total time in $\frac{1}{100}$ -seconds.

Definition at line 186 of file timer.h.

The documentation for this class was generated from the following file:

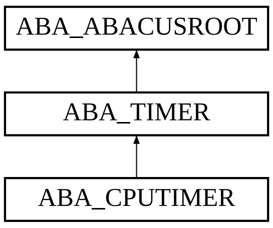
- Include/abacus/timer.h

6.69 ABA_CPUTIMER Class Reference

This class derived from ABA_TIMER implements a timer measuring the cpu time of parts of a program.

`#include <cputimer.h>`

Inheritance diagram for ABA_CPUTIMER::



Public Member Functions

- ABA_CPUTIMER (ABA_GLOBAL $*$glob)
  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_GLOBAL ∗glob, long centiSeconds)
- virtual ∼ABA_CPUTIMER ()

  The destructor.

Private Member Functions

- virtual long theTime () const

Static Private Attributes

- static long clk_tck_

6.69.1 Detailed Description

This class derived from ABA_TIMER implements a timer measuring the cpu time of parts of a program.

Definition at line 38 of file cputimer.h.

6.69.2 Constructor & Destructor Documentation

6.69.2.1 ABA_CPUTIMER::ABA_CPUTIMER (ABA_GLOBAL ∗ *glob*)

After the application of the constructor the timer is not running, i.e., to measure time it has to be started explicitly.

Parameters:

glob A pointer to a global object.

6.69.2.2 ABA_CPUTIMER::ABA_CPUTIMER (ABA_GLOBAL ∗ *glob*, long *centiSeconds*)

This constructor initializes the total time of the timer.

The timer is not running, too.

Parameters:

glob A pointer to a global object.

centiSeconds The intial value of the total time in $\frac{1}{100}$ seconds.

6.69.2.3 virtual ABA_CPUTIMER::∼ABA_CPUTIMER () `[virtual]`

The destructor.

6.69.3 Member Function Documentation

6.69.3.1 virtual long ABA_CPUTIMER::theTime () const `[private, virtual]`

Returns the used cpu time in $\frac{1}{100}$ seconds since the start of the program.

This function redefines the pure virtual function of the base class ABA_TIMER.

> Since *CLOCKS_PER_SEC* can be 1000000 the standard library function *clock()* returns negative values after about than 35 minutes. Hence we measure the cpu time with the function *times* which is common on / systems, although not defined in the /-ANSI-standard.

Implements ABA_TIMER.

6.69.4 Member Data Documentation

6.69.4.1 long ABA_CPUTIMER::clk_tck_ `[static, private]`

Definition at line 62 of file cputimer.h.

The documentation for this class was generated from the following file:

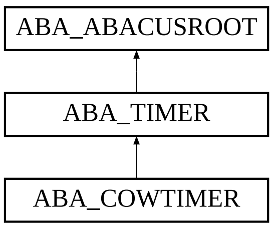
- Include/abacus/cputimer.h

6.70 ABA_COWTIMER Class Reference

class derived from ABA_TIMER implements a timer measuring the elpased time (clock-of-the-wall time) of parts of the program.

`#include <cowtimer.h>`

Inheritance diagram for ABA_COWTIMER::



Public Member Functions

- ABA_COWTIMER (ABA_GLOBAL ∗glob)

  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_GLOBAL ∗glob, long secs)
- virtual ∼ABA_COWTIMER ()

Private Member Functions

- virtual long theTime () const

  Returns the wall clock time since the initialization of the timer in $\frac{1}{100}$ *seconds.*

Private Attributes

- long baseTime_

  Stores the result of a call to the function time(NULL) *at construction time.*

6.70.1 Detailed Description

class derived from ABA_TIMER implements a timer measuring the elpased time (clock-of-the-wall time) of parts of the program.

Definition at line 37 of file cowtimer.h.

6.70.2 Constructor & Destructor Documentation

6.70.2.1 ABA_COWTIMER::ABA_COWTIMER (ABA_GLOBAL ∗ *glob*)

After the application of the constructor the timer is not running, i.e., to measure time it has to be started explicitly.

We initialize base time with the current time, such that later we can convert the time to $\frac{1}{100}$ seconds without arithmetic overflow. The function *time()* is defined in the standard /-library.

Parameters:

glob A pointer to a global object.

6.70.2.2 ABA_COWTIMER::ABA_COWTIMER (ABA_GLOBAL ∗ *glob*, long *secs*)

This constructor initializes the total time of the timer.

The timer is not running, too.

Parameters:

glob A pointer to a global object.

centiSeconds The initial value of the timer in $\frac{1}{100}$ seconds.

6.70.2.3 virtual ABA_COWTIMER::∼ABA_COWTIMER () `[virtual]`

The destructor.

6.70.3 Member Function Documentation

6.70.3.1 virtual long ABA_COWTIMER::theTime () const `[private, virtual]`

Returns the wall clock time since the initialization of the timer in $\frac{1}{100}$ seconds.

This function redefines the pure virtual function of the base class ABA_TIMER.

The function *theTime()* uses the function *times()*, which returns the elapsed real time in clock ticks.

Implements ABA_TIMER.

6.70.4 Member Data Documentation

6.70.4.1 long ABA_COWTIMER::baseTime_ `[private]`

Stores the result of a call to the function *time(NULL)* at construction time.

We require this member such that we can return the time in centiseconds correctly in the function *theTime()*. Otherwise, an arithmetic overflow can occur.

Definition at line 85 of file cowtimer.h.

The documentation for this class was generated from the following file:

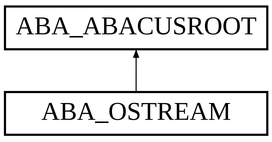
- Include/abacus/cowtimer.h

6.71 ABA_OSTREAM Class Reference

Class implements an output stream which can be turned on and off at run time, i.e., if the output stream is turned off, then no messages written by the operator << reach the associated "real" output stream.

`#include <ostream.h>`

Inheritance diagram for ABA_OSTREAM::



Public Member Functions

- ABA_OSTREAM (ostream &out, const char ∗logStreamName=0)
- ∼ABA_OSTREAM ()

  The destructor.

- ABA_OSTREAM & operator<< (char o)

  Reimplementation for all fundamental types, for const char ∗*, and for some other classes listed below.*

- ABA_OSTREAM & operator<< (unsigned char o)
- ABA_OSTREAM & operator<< (signed char o)
- ABA_OSTREAM & operator<< (short o)
- ABA_OSTREAM & operator<< (unsigned short o)
- ABA_OSTREAM & operator<< (int o)
- ABA_OSTREAM & operator<< (unsigned int o)
- ABA_OSTREAM & operator<< (long o)
- ABA_OSTREAM & operator<< (unsigned long o)
- ABA_OSTREAM & operator<< (float o)
- ABA_OSTREAM & operator<< (double o)
- ABA_OSTREAM & operator<< (const char ∗o)
- ABA_OSTREAM & operator<< (ABA_OSTREAM &(∗pf)(ABA_OSTREAM &))
- ABA_OSTREAM & operator<< (const ABA_STRING &o)

  A manipulator is a function having as argument a reference to an ABA_OSTREAM and returning an ABA_-OSTREAM.

- ABA_OSTREAM & operator<< (const ABA_TIMER &o)
- ABA_OSTREAM & operator<< (const ABA_HISTORY &o)
- ABA_OSTREAM & operator<< (const ABA_LPVARSTAT &o)
- ABA_OSTREAM & operator<< (const ABA_CSENSE &o)
- ABA_OSTREAM & operator<< (const ABA_LP &o)
- void off ()
- void on ()
- void logOn ()
- void logOn (const char ∗logStreamName)

  This version of logOn() *turns the output to the logfile on and sets the log-file to* logStreamName.

- void logOff ()
- bool isOn () const
- bool isLogOn () const
- ofstream ∗ log () const
- void setFormatFlag (fmtflags flag)

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.*

Private Attributes

- ostream & out_

  The "real" stream associated with our output stream (usually cout *or* cerr*).*

- bool on_

  If true*, then output is written to the stream* out_*, otherwise it is suppressed.*

- bool logOn_

  If true_*, then output is also written to the log stream* ∗log*.*

- ofstream ∗ log_

Friends

- ABA_OSTREAM & flush (ABA_OSTREAM &o)

  Flushes the output and the log stream buffers of the stream o*. This function can be called via the manipulator* o << *flush;.*

- ABA_OSTREAM & endl (ABA_OSTREAM &o)
- ABA_OSTREAM & _setWidth (ABA_OSTREAM &o, int w)
- ABA_OSTREAM & _setPrecision (ABA_OSTREAM &o, int p)

6.71.1 Detailed Description

Class implements an output stream which can be turned on and off at run time, i.e., if the output stream is turned off, then no messages written by the operator << reach the associated "real" output stream.

Definition at line 64 of file ostream.h.

6.71.2 Constructor & Destructor Documentation

6.71.2.1 ABA_OSTREAM::ABA_OSTREAM (ostream & *out*, const char ∗ *logStreamName* = 0)

The constructor turns the output on and associates it with a "real" stream.

Parameters:

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.

6.71.2.2 ABA_OSTREAM::∼ABA_OSTREAM ()

The destructor.

6.71.3 Member Function Documentation

6.71.3.1 bool ABA_OSTREAM::isLogOn () const

Returns:
true If the output to the logfile is turned on,
false otherwise.

6.71.3.2 bool ABA_OSTREAM::isOn () const

Returns:
true If the output is turned on,
false otherwise.

6.71.3.3 ofstream$*$ ABA_OSTREAM::log () const

Returns:
A pointer to the stream associated with the log-file.

6.71.3.4 void ABA_OSTREAM::logOff ()

Turns the output to the logfile off.

6.71.3.5 void ABA_OSTREAM::logOn (const char $*$ *logStreamName*)

This version of *logOn()* turns the output to the logfile on and sets the log-file to *logStreamName*.

Parameters:
logStreamName The name of the log-file.

6.71.3.6 void ABA_OSTREAM::logOn ()

Turns the output to the logfile on.

6.71.3.7 void ABA_OSTREAM::off ()

Turns the output off.

6.71.3.8 void ABA_OSTREAM::on ()

Turns the output on.

6.71.3.9 ABA_OSTREAM& ABA_OSTREAM::operator<< (const ABA_LP & *o*)

6.71.3.10 ABA_OSTREAM& ABA_OSTREAM::operator<< (const ABA_CSENSE & *o*)

6.71.3.11 ABA_OSTREAM& ABA_OSTREAM::operator<< (const ABA_LPVARSTAT & *o*)

6.71.3.12 ABA_OSTREAM& ABA_OSTREAM::operator<< (const ABA_HISTORY & *o*)

6.71.3.13 ABA_OSTREAM& ABA_OSTREAM::operator<< (const ABA_TIMER & *o*)

6.71.3.14 ABA_OSTREAM& ABA_OSTREAM::operator<< (const ABA_STRING & *o*)

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.

Returns:

A reference to the output stream.

Parameters:

m An output stream manipulator.

6.71.3.15 ABA_OSTREAM& ABA_OSTREAM::operator<< (ABA_OSTREAM &(∗)(ABA_OSTREAM &) *pf*) `[inline]`

Definition at line 105 of file ostream.h.

6.71.3.16 ABA_OSTREAM& ABA_OSTREAM::operator<< (const char $*$ *o*)

6.71.3.17 ABA_OSTREAM& ABA_OSTREAM::operator<< (double *o*)

6.71.3.18 ABA_OSTREAM& ABA_OSTREAM::operator<< (float *o*)

6.71.3.19 ABA_OSTREAM& ABA_OSTREAM::operator<< (unsigned long *o*)

6.71.3.20 ABA_OSTREAM& ABA_OSTREAM::operator<< (long *o*)

6.71.3.21 ABA_OSTREAM& ABA_OSTREAM::operator<< (unsigned int *o*)

6.71.3.22 ABA_OSTREAM& ABA_OSTREAM::operator<< (int *o*)

6.71.3.23 ABA_OSTREAM& ABA_OSTREAM::operator<< (unsigned short *o*)

6.71.3.24 ABA_OSTREAM& ABA_OSTREAM::operator<< (short *o*)

6.71.3.25 ABA_OSTREAM& ABA_OSTREAM::operator<< (signed char *o*)

6.71.3.26 ABA_OSTREAM& ABA_OSTREAM::operator<< (unsigned char *o*)

6.71.3.27 ABA_OSTREAM& ABA_OSTREAM::operator<< (char *o*)

Reimplementation for all fundamental types, for *const* char $*$, and for some other classes listed below.

output operator $<<$ 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.

return A reference to the output stream.

Parameters:

o The item being output.

6.71.3.28 void ABA_OSTREAM::setFormatFlag (fmtflags *flag*)

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 / iostream Library.

Parameters:

flag The flag being set.

6.71.4 Friends And Related Function Documentation

6.71.4.1 ABA_OSTREAM& _setPrecision (ABA_OSTREAM & *o*, int *p*) `[friend]`

Sets the precision for the output stream.

In most cases the manipulator *setPrecision* is more convenient to use.

Returns:

A reference to the output stream.

Parameters:

o An output stream.

p The precision.

6.71.4.2 ABA_OSTREAM& _setWidth (ABA_OSTREAM & *o*, int *w*) `[friend]`

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.

Returns:

A reference to the output stream.

Parameters:

o An output stream.

w The width of the field.

6.71.4.3 ABA_OSTREAM& endl (ABA_OSTREAM & *o*) `[friend]`

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;.

Returns:

A reference to the output stream.

Parameters:

o An output stream.

6.71.4.4 ABA_OSTREAM& flush (ABA_OSTREAM & *o*) `[friend]`

Flushes the output and the log stream buffers of the stream *o*. This function can be called via the manipulator *o* $<<$ flush;.

Returns:

A reference to the output stream.

Parameters:

o An output stream.

6.71.5 Member Data Documentation

6.71.5.1 ofstream$*$ ABA_OSTREAM::log_ `[private]`

A pointer to a stream associated with the log file.

Definition at line 262 of file ostream.h.

6.71.5.2 bool ABA_OSTREAM::logOn_ `[private]`

If *true_*, then output is also written to the log stream $*$*log*.

Definition at line 258 of file ostream.h.

6.71.5.3 bool ABA_OSTREAM::on_ `[private]`

If *true*, then output is written to the stream *out_*, otherwise it is suppressed.

Definition at line 253 of file ostream.h.

6.71.5.4 ostream& ABA_OSTREAM::out_ `[private]`

The “real” stream associated with our output stream (usually *cout* or *cerr*).

Definition at line 248 of file ostream.h.

The documentation for this class was generated from the following file:

- Include/abacus/ostream.h

6.72 Preprocessor Flags

Table 6.1 summarizes all preprocessors flags that are relevant for ABACUS-users.

Flag	Description	See Section
`ABACUS_COMPILER_GCC41`	GNU C++ compiler 4.1.x	2.4
`ABACUS_COMPILER_GCC34`	GNU C++ compiler 3.4.x	2.4
`ABACUS_COMPILER_GCC33`	GNU C++ compiler 3.3.x	2.4
`ABACUS_COMPILER_SUN`	SUN C++ compiler	2.4

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

Bibliography

[ASC95] INFORMATION PROCESSING SYSTEM Accredited Standards Committee, X3. *The ISO/ANSI C++ Draft*, 1995. http://www.cygnus.com/misc/wp/. 3

[Bay72] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972. 38

[BCC93a] Egon Balas, Sebastian Ceria, and Gerard Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58:295–324, 1993. 36

[BCC93b] Egon Balas, Sebastian Ceria, and Gerard Cornuéjols. 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. 36

[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. 3

[Boo94] G. Booch. *Object-oriented analysis and design with applications*. The Benjamin Cummings Publishing Company, Redwood City, California, 1994. 3

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. MIT Press, Cambridge, 1990. 40

[ES92] M.A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison Wesley, Reading, Massachusetts, 1992. 3

[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. 38

[HP93] Karla Hoffman and Manfred W. Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39:657–682, 1993. 36

[JRT94] Michael Jünger, Gerhard Reinelt, and Stefan Thienel. Provably good solutions for the traveling salesman problem. *Zeitschrift für Operations Research*, 40:183–217, 1994. 32

[JRT95] Michael Jünger, Gerhard 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. 3

[KM90] T. Korson and J.D. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9):40–60, 1990. 3

[Knu93] Donald E. Knuth. *The Stanford GraphBase: a platform for combinatorial computing*. Addison-Wesley, Reading, Massachusetts, 1993. 42

[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. 83

[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. 31

[RF81] David 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. 39

[Sav94] Martin W.P. Savelsbergh. Preprocessing and probing for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994. 36

[Str93] B. Stroustrup. *The C++ programming language—2nd edition*. Addison-Wesley, Reading, Massachusetts, 1993. 3

[Thi95] Stefan Thienel. *ABACUS—A Branch-And-CUt System*. PhD thesis, Universität zu Köln, 1995. 3, 27, 59

[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. 67

[Wun97] Roland Wunderling. *SoPlex, The Sequential object-oriented simplex class library*, 1997. http://www.zib.de/Optimization/Software/Soplex/. 16

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.