Bcp

trunk

Generated by Doxygen 1.8.1.2

Mon Mar 16 2015 20:13:38

Contents

1 Class Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

- CoinDoubleArrayWithLength [external]
- CoinFactorizationDoubleArrayWithLength [external]
- CoinFactorizationLongDoubleArrayWithLength [external]
- CoinIntArrayWithLength [external]
- CoinUnsignedIntArrayWithLength [external]
- CoinVoidStarArrayWithLength [external]

CoinBaseModel [external]

- CoinModel [external]
- CoinStructuredModel [external]

CoinBuild [external]
CoinDenseVector< T > [external]
CoinError [external]
CoinExternalVectorFirstGreater_2< class, class, class > [external]
CoinExternalVectorFirstGreater_3< class, class, class, class > [external]
CoinExternalVectorFirstLess_2< class, class, class > [external]
CoinExternalVectorFirstLess_3< class, class, class, class > [external]
CoinFactorization [external]
CoinFileIOBase [external]

- CoinFileInput [external]
- CoinFileOutput [external]

CoinFirstAbsGreater_2< class, class > [external]
CoinFirstAbsGreater_3< class, class, class > [external]
CoinFirstAbsLess_2< class, class > [external]
CoinFirstAbsLess_3< class, class, class > [external]
CoinFirstGreater_2< class, class > [external]
CoinFirstGreater_3< class, class, class > [external]
CoinFirstLess_2< class, class > [external]
CoinFirstLess_3< class, class, class > [external]
CoinLpIO::CoinHashLink [external]
CoinMpsIO::CoinHashLink [external]
CoinIndexedVector [external]

- CoinPartitionedVector [external]

CoinLpIO [external]
CoinMessageHandler [external]
CoinMessages [external]

- CoinMessage [external]

CoinModelHash [external]
CoinModelHash2 [external]
CoinModelHashLink [external]
CoinModelInfo2 [external]
CoinModelLink [external]
CoinModelLinkedList [external]
CoinModelTriple [external]
CoinMpsCardReader [external]
CoinMpsIO [external]
CoinOneMessage [external]
CoinOtherFactorization [external]

- CoinDenseFactorization [external]
- CoinOslFactorization [external]
- CoinSimpFactorization [external]

CoinPackedMatrix [external]

- **BCP_lp_relax** 70

CoinPackedVectorBase [external]

2 Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

3 File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

4 Class Documentation

4.1 BCP_buffer Class Reference

This class describes the message buffer used for all processes of BCP.

`#include <BCP_buffer.hpp>`

Public Member Functions

Query methods

- BCP_message_tag msgtag () const
  Return the message tag of the message in the buffer.
- int sender () const
  Return a const pointer to the process id of the sender of the message in the buffer.
- int size () const
  Return the size of the current message in the buffer.
- const char ∗ data () const
  Return a const pointer to the data stored in the buffer.

Modifying methods

- void set_position (const int pos) throw (BCP_fatal_error)
  Position the read head in the buffer.
- void set_size (const int s) throw (BCP_fatal_error)
  Cut off the end of the buffer.
- void set_msgtag (const BCP_message_tag tag)
  Set the message tag on the buffer.
- void set_content (const char ∗data, const size_t size, int sender, BCP_message_tag msgtag)
  Set the buffer to be a copy of the given data.
- BCP_buffer & operator= (const BCP_buffer &buf)
  Make an exact replica of the other buffer.
- void make_fit (const int add_size)
  Reallocate the buffer if necessary so that at least `add_size` number of additional bytes will fit into the buffer.
- void clear ()
  Completely clear the buffer.
- template<class T >
  BCP_buffer & pack (const T &value)
  Pack a single object of type `T`.
- template<class T >
  BCP_buffer & unpack (T &value)
  Unpack a single object of type `T`.
- template<class T >
  BCP_buffer & pack (const T ∗const values, const int length)
  Pack a C style array of objects of type `T`.
- template<class T >
  BCP_buffer & unpack (T ∗&values, int &length, bool allocate=true) throw (BCP_fatal_error)
  *Unpack an array of objects of type `T`, where T **must** be a built-in type (ar at least something that can be copied with memcpy).*
- BCP_buffer & pack (const BCP_string &value)
  Pack a `BCP_string` into the buffer.
- BCP_buffer & pack (BCP_string &value)
  Pack a `BCP_string` into the buffer.
- BCP_buffer & unpack (BCP_string &value)
  Unpack a `BCP_string` from the buffer.
- template<class T >
  BCP_buffer & pack (const BCP_vec< T > &vec)
  Pack a `BCP_vec` into the buffer.
- template<class T >
  BCP_buffer & pack (const std::vector< T > &vec)
  Pack a `std::vector` into the buffer.

- template<class T >
  BCP_buffer & unpack (BCP_vec< T > &vec)

  Unpack a `BCP_vec` from the buffer.
- template<class T >
  BCP_buffer & unpack (std::vector< T > &vec)

  Unpack a `std::vector` from the buffer.

Constructors and destructor

- BCP_buffer ()

  The default constructor creates a buffer of size 16 Kbytes with no message in it.
- BCP_buffer (const BCP_buffer &buf)

  The copy constructor makes an exact replica of the other buffer.
- ∼BCP_buffer ()

  The desctructor deletes all data members (including freeing the buffer).

Public Attributes

Data members

The data members are public for efficiency reasons.

Access these fields sparingly.

THINK: maybe it's not that inefficient to access the fields thru functions...

Note that:

1. *The max size of the buffer never decreases. The buffer max size increases when the current max size is not sufficient for the message.*
2. *With normal usage for incoming messages `_size` stays constant while reading out the message and `_pos` moves forward in the buffer.*
3. *With normal usage for outgoing messages `_size` and `_pos` moves forward together as the buffer is filled with the message.*

- BCP_message_tag _msgtag

  The message tag of the last received message.
- int _sender

  The process id of the sender of the last received message.
- size_t _pos

  The next read position in the buffer.
- size_t _max_size

  The amount of memory allocated for the buffer.
- size_t _size

  The current size of the message (the first `_size` bytes of the buffer).
- char ∗ _data

  Pointer to the buffer itself.

4.1.1 Detailed Description

This class describes the message buffer used for all processes of BCP.

This buffer is a character array; the components of a message are simply copied into this array one after the other. Note that each process has only one buffer, which serves both for outgoing and incoming messages. This can be done since when a message arrives it is completely unpacked before anything else is done and conversely, once a message is started to be packed together it will be sent out before another message is received.

NOTE: Only the following type of objects can be packed with the various `pack()` member methods:

- Objects that can be assigned with `memcpy()`, i.e., built-in non-pointer types, and structs recursively built from such types;
- `BCP_vec`s of the above and
- `BCP_string`s.

Everything else that needs to be packed at any time must have a `pack()` member method.

Definition at line 39 of file BCP_buffer.hpp.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 BCP_buffer::BCP_buffer ( ) `[inline]`

The default constructor creates a buffer of size 16 Kbytes with no message in it.

Definition at line 381 of file BCP_buffer.hpp.

4.1.2.2 BCP_buffer::BCP_buffer (const BCP_buffer & *buf*) `[inline]`

The copy constructor makes an exact replica of the other buffer.

Definition at line 385 of file BCP_buffer.hpp.

4.1.2.3 BCP_buffer::∼BCP_buffer ( ) `[inline]`

The desctructor deletes all data members (including freeing the buffer).

Definition at line 392 of file BCP_buffer.hpp.

4.1.3 Member Function Documentation

4.1.3.1 BCP_message_tag BCP_buffer::msgtag ( ) const `[inline]`

Return the message tag of the message in the buffer.

Definition at line 90 of file BCP_buffer.hpp.

4.1.3.2 int BCP_buffer::sender ( ) const `[inline]`

Return a const pointer to the process id of the sender of the message in the buffer.

Definition at line 93 of file BCP_buffer.hpp.

4.1.3.3 int BCP_buffer::size ( ) const `[inline]`

Return the size of the current message in the buffer.

Definition at line 95 of file BCP_buffer.hpp.

4.1.3.4 const char∗ BCP_buffer::data ( ) const `[inline]`

Return a const pointer to the data stored in the buffer.

Definition at line 97 of file BCP_buffer.hpp.

4.1.3.5 void BCP_buffer::set_position (const int *pos*) throw (BCP_fatal_error) `[inline]`

Position the read head in the buffer.

Must be between 0 and size().

Definition at line 104 of file BCP_buffer.hpp.

4.1.3.6 void BCP_buffer::set_size (const int *s*) throw (**BCP_fatal_error**) `[inline]`

Cut off the end of the buffer.

Must be between 0 and size().

Definition at line 110 of file BCP_buffer.hpp.

4.1.3.7 void BCP_buffer::set_content (const char ∗ *data,* const size_t *size,* int *sender,* BCP_message_tag *msgtag*) `[inline]`

Set the buffer to be a copy of the given data.

Use this with care!

Definition at line 119 of file BCP_buffer.hpp.

4.1.3.8 **BCP_buffer**& BCP_buffer::operator= (const **BCP_buffer** & *buf*) `[inline]`

Make an exact replica of the other buffer.

Definition at line 136 of file BCP_buffer.hpp.

4.1.3.9 void BCP_buffer::make_fit (const int *add_size*) `[inline]`

Reallocate the buffer if necessary so that at least `add_size` number of additional bytes will fit into the buffer.

Definition at line 153 of file BCP_buffer.hpp.

4.1.3.10 void BCP_buffer::clear ( ) `[inline]`

Completely clear the buffer.

Delete and zero out `_msgtag, _size, _pos` and `_sender`.

Definition at line 168 of file BCP_buffer.hpp.

4.1.3.11 template<class T > **BCP_buffer**& BCP_buffer::pack (const T & *value*) `[inline]`

Pack a single object of type `T`.

Copies `sizeof(T)` bytes from the address of the object.

Definition at line 177 of file BCP_buffer.hpp.

4.1.3.12 template<class T > **BCP_buffer**& BCP_buffer::unpack (T & *value*) `[inline]`

Unpack a single object of type `T`.

Copies `sizeof(T)` bytes to the address of the object.

Definition at line 186 of file BCP_buffer.hpp.

4.1.3.13 template<class T > **BCP_buffer**& BCP_buffer::pack (const T ∗const *values,* const int *length*) `[inline]`

Pack a C style array of objects of type `T`.

Definition at line 197 of file BCP_buffer.hpp.

4.1.3.14 template< class T > BCP_buffer& BCP_buffer::unpack (T *& *values*, int & *length*, bool *allocate* = `true`) throw (BCP_fatal_error) `[inline]`

Unpack an array of objects of type `T`, where T **must** be a built-in type (ar at least something that can be copied with memcpy).

If the third argument is true then memory is allocated for the array and the array pointer and the length of the array are returned in the arguments.

If the third argument is false then the arriving array's length is compared to `length` and an exception is thrown if they are not the same. Also, the array passed as the first argument will be filled with the arriving array.

Definition at line 222 of file BCP_buffer.hpp.

4.1.3.15 BCP_buffer& BCP_buffer::pack (const BCP_string & *value*) `[inline]`

Pack a `BCP_string` into the buffer.

Definition at line 268 of file BCP_buffer.hpp.

4.1.3.16 BCP_buffer& BCP_buffer::pack (BCP_string & *value*) `[inline]`

Pack a `BCP_string` into the buffer.

Definition at line 281 of file BCP_buffer.hpp.

4.1.3.17 BCP_buffer& BCP_buffer::unpack (BCP_string & *value*) `[inline]`

Unpack a `BCP_string` from the buffer.

Definition at line 294 of file BCP_buffer.hpp.

4.1.3.18 template< class T > BCP_buffer& BCP_buffer::pack (const BCP_vec< T > & *vec*) `[inline]`

Pack a `BCP_vec` into the buffer.

Definition at line 304 of file BCP_buffer.hpp.

4.1.3.19 template< class T > BCP_buffer& BCP_buffer::pack (const std::vector< T > & *vec*) `[inline]`

Pack a `std::vector` into the buffer.

Definition at line 318 of file BCP_buffer.hpp.

4.1.3.20 template< class T > BCP_buffer& BCP_buffer::unpack (BCP_vec< T > & *vec*) `[inline]`

Unpack a `BCP_vec` from the buffer.

Definition at line 332 of file BCP_buffer.hpp.

4.1.3.21 template< class T > BCP_buffer& BCP_buffer::unpack (std::vector< T > & *vec*) `[inline]`

Unpack a `std::vector` from the buffer.

Definition at line 354 of file BCP_buffer.hpp.

4.1.4 Member Data Documentation

4.1.4.1 BCP_message_tag BCP_buffer::_msgtag

The message tag of the last *received* message.

This member has no meaning if the buffer holds an outgoing message.

Definition at line 70 of file BCP_buffer.hpp.

4.1.4.2 int BCP_buffer::_sender

The process id of the sender of the last *received* message.

This member has no meaning if the buffer holds an outgoing message.

Definition at line 73 of file BCP_buffer.hpp.

4.1.4.3 size_t BCP_buffer::_pos

The next read position in the buffer.

Definition at line 75 of file BCP_buffer.hpp.

4.1.4.4 size_t BCP_buffer::_max_size

The amount of memory allocated for the buffer.

Definition at line 77 of file BCP_buffer.hpp.

4.1.4.5 size_t BCP_buffer::_size

The current size of the message (the first `_size` bytes of the buffer).

Definition at line 80 of file BCP_buffer.hpp.

4.1.4.6 char∗ BCP_buffer::_data

Pointer to the buffer itself.

Definition at line 82 of file BCP_buffer.hpp.

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

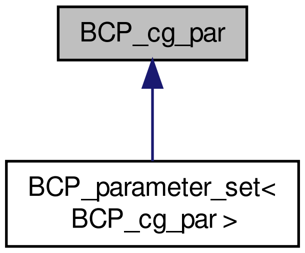
- BCP_buffer.hpp

4.2 BCP_cg_par Struct Reference

Parameters used in the Cut Generator process.

```
#include <BCP_cg_param.hpp>
```

Inheritance diagram for BCP_cg_par:



Public Types

- enum chr_params { MessagePassingIsSerial, ReportWhenDefaultIsExecuted, CgVerb_First, CgVerb_Last }

  Character parameters.
- enum int_params { NiceLevel }

  Integer parameters.
- enum dbl_params

  There are no double parameters.
- enum str_params { LogFileName }

  String parameters.
- enum str_array_params

  There are no string array parameters.

4.2.1 Detailed Description

Parameters used in the Cut Generator process.

These parameters can be set in the original parameter file by including the following line:

`BCP_{parameter name} {parameter value}.`

Definition at line 12 of file BCP_cg_param.hpp.

4.2.2 Member Enumeration Documentation

4.2.2.1 enum BCP_cg_par::chr_params

Character parameters.

All of these variables are used as booleans (true = 1, false = 0).

Enumerator:

MessagePassingIsSerial Indicates whether message passing is serial (all processes are on the same processor) or not.
Values: true (1), false (0). Default: 1.

ReportWhenDefaultIsExecuted Print out a message when the default version of an overridable method is executed. Default: 1.

CgVerb_First Just a marker for the first CgVerb.

CgVerb_Last Just a marker for the last CgVerb.

Definition at line 15 of file BCP_cg_param.hpp.

4.2.2.2 enum BCP_cg_par::int_params

Integer parameters.

Enumerator:

NiceLevel The "nice" level the process should run at. On Linux and on AIX this value should be between -20 and 20. The higher this value the less resource the process will receive from the system. Note that

1. The process starts with 0 priority and it can only be increased.
2. If the load is low on the machine (e.g., when all other processes are interactive like netscape or text editors) then even with 20 priority the process will use close to 100% of the cpu, and the interactive processes will be noticably more responsive than they would be if this process ran with 0 priority.

See Also

the man page of the `setpriority` system function.

Definition at line 32 of file BCP_cg_param.hpp.

4.2.2.3 enum BCP_cg_par::dbl_params

There are no double parameters.

Definition at line 54 of file BCP_cg_param.hpp.

4.2.2.4 enum BCP_cg_par::str_params

String parameters.

Enumerator:

LogFileName The file where the output from this process should be logged. To distinguish the output of this CG process from that of the others, the string "-cg-<process_id>" is appended to the given logfile name to form the real filename.

Definition at line 61 of file BCP_cg_param.hpp.

4.2.2.5 enum BCP_cg_par::str_array_params

There are no string array parameters.

Definition at line 72 of file BCP_cg_param.hpp.

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

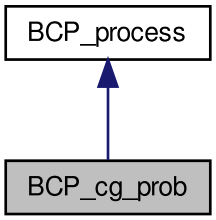
- BCP_cg_param.hpp

4.3 BCP_cg_prob Class Reference

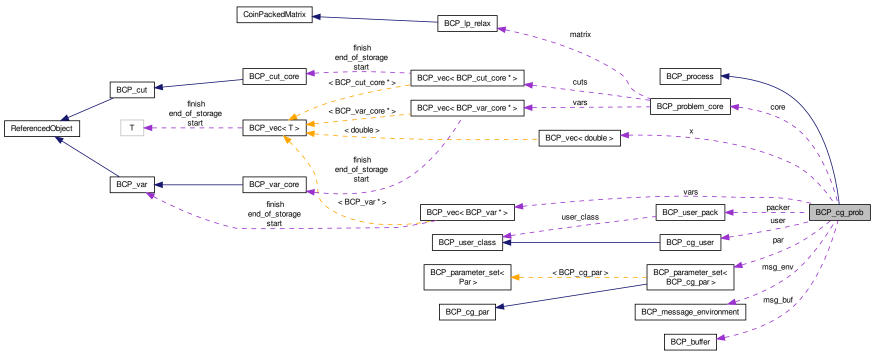
This class is the central class of the Cut Generator process.

```
#include <BCP_cg.hpp>
```

Inheritance diagram for BCP_cg_prob:



Collaboration diagram for BCP_cg_prob:



Public Member Functions

- BCP_var ∗ unpack_var ()

  Unpack a variable.

Constructor and destructor

- BCP_cg_prob (int my_id, int parent)

  The default constructor.
- virtual ∼BCP_cg_prob ()

  The destructor deletes everything.

Query methods

- bool has_ub () const

  Return true/false indicating whether any upper bound has been found.
- double ub () const

Return the current upper bound (`BCP_DBL_MAX` if there's no upper bound found yet.)

Modifying methods

- void ub (const double bd)

  Set the upper bound equal to the argument.
- bool probe_messages ()

  Test if there is a message in the message queue waiting to be processed.

Public Attributes

Data members

- BCP_cg_user ∗ user

  The user object holding the user's data.
- BCP_user_pack ∗ packer

  A class that holds the methods about how to pack things.
- BCP_message_environment ∗ msg_env

  The message passing environment.
- BCP_buffer msg_buf

  The message buffer of the Cut Generator process.
- BCP_parameter_set< BCP_cg_par > par

  The parameters controlling the Cut Generator process.
- BCP_problem_core ∗ core

  The description of the core of the problem.
- double upper_bound

  The proc id of the tree manager.
- BCP_vec< BCP_var ∗ > vars

  Cuts are to be generated for the LP solution given by these variables and their values (next member).
- BCP_vec< double > x

  The primal values corresponding to the variables above.
- int sender

  The process id of the LP process that sent the solution.
- int phase

  The phase the algorithm is in.
- int node_level

  The level of search tree node where the solution was generated.
- int node_index

  The index of search tree node where the solution was generated.
- int node_iteration

  The iteration within the search tree node where the solution was generated.

4.3.1 Detailed Description

This class is the central class of the Cut Generator process.

Only one object of this type is created and that holds all the data in the CG process. A reference to that object is passed to (almost) every function (or member method) that's invoked within the CG process.

Definition at line 32 of file BCP_cg.hpp.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 BCP_cg_prob::BCP_cg_prob (int *my_id*, int *parent*)

The default constructor.

Initializes every data member to a natural state.

4.3.2.2 virtual BCP_cg_prob::∼BCP_cg_prob ( ) [virtual]

The destructor deletes everything.

4.3.3 Member Function Documentation

4.3.3.1 bool BCP_cg_prob::has_ub ( ) const [inline]

Return true/false indicating whether any upper bound has been found.

Definition at line 115 of file BCP_cg.hpp.

4.3.3.2 void BCP_cg_prob::ub (const double *bd*) [inline]

Set the upper bound equal to the argument.

Definition at line 124 of file BCP_cg.hpp.

4.3.3.3 bool BCP_cg_prob::probe_messages ( )

Test if there is a message in the message queue waiting to be processed.

4.3.3.4 BCP_var∗ BCP_cg_prob::unpack_var ( )

Unpack a variable.

Invoked from the built-in BCP_cg_user::unpack_primal_solution().

4.3.4 Member Data Documentation

4.3.4.1 BCP_cg_user∗ BCP_cg_prob::user

The user object holding the user's data.

This object is created by a call to the appropriate member of [USER_initialize]{USER_initialize.html}.

Definition at line 50 of file BCP_cg.hpp.

4.3.4.2 BCP_user_pack∗ BCP_cg_prob::packer

A class that holds the methods about how to pack things.

Definition at line 53 of file BCP_cg.hpp.

4.3.4.3 BCP_message_environment∗ BCP_cg_prob::msg_env

The message passing environment.

This object is created by a call to the appropriate member of [USER_initialize]{USER_initialize.html}.

Definition at line 58 of file BCP_cg.hpp.

4.3.4.4 BCP_buffer BCP_cg_prob::msg_buf

The message buffer of the Cut Generator process.

Definition at line 61 of file BCP_cg.hpp.

4.3.4.5 BCP_parameter_set<BCP_cg_par> BCP_cg_prob::par

The parameters controlling the Cut Generator process.

Definition at line 64 of file BCP_cg.hpp.

4.3.4.6 BCP_problem_core* BCP_cg_prob::core

The description of the core of the problem.

Definition at line 67 of file BCP_cg.hpp.

4.3.4.7 double BCP_cg_prob::upper_bound

The proc id of the tree manager.

The best currently known upper bound.

Definition at line 73 of file BCP_cg.hpp.

4.3.4.8 BCP_vec<BCP_var*> BCP_cg_prob::vars

Cuts are to be generated for the LP solution given by these variables and their values (next member).

Not all variables need to be listed (e.g., list only those that have fractional values in current LP solution).

See Also

BCP_lp_user::pack_primal_solution()

Definition at line 83 of file BCP_cg.hpp.

4.3.4.9 BCP_vec<double> BCP_cg_prob::x

The primal values corresponding to the variables above.

Definition at line 85 of file BCP_cg.hpp.

4.3.4.10 int BCP_cg_prob::sender

The process id of the LP process that sent the solution.

Definition at line 87 of file BCP_cg.hpp.

4.3.4.11 int BCP_cg_prob::phase

The phase the algorithm is in.

Definition at line 91 of file BCP_cg.hpp.

4.3.4.12 int BCP_cg_prob::node_level

The level of search tree node where the solution was generated.

Definition at line 93 of file BCP_cg.hpp.

4.3.4.13 int BCP_cg_prob::node_index

The index of search tree node where the solution was generated.

Definition at line 95 of file BCP_cg.hpp.

4.3.4.14 int BCP_cg_prob::node_iteration

The iteration within the search tree node where the solution was generated.

Definition at line 98 of file BCP_cg.hpp.

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

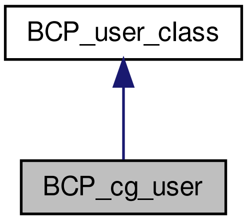
- BCP_cg.hpp

4.4 BCP_cg_user Class Reference

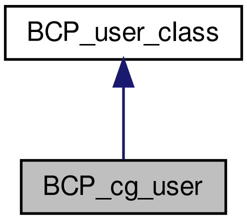
The BCP_cg_user class is the base class from which the user can derive a problem specific class to be used in the Cut Generator process.

```
#include <BCP_cg_user.hpp>
```

Inheritance diagram for BCP_cg_user:



Collaboration diagram for BCP_cg_user:



Public Member Functions

- void send_cut (const BCP_cut &cut)

  Pack the argument into the message buffer and send it to the sender of the LP solution.
- virtual void unpack_module_data (BCP_buffer &buf)

  Unpack the initial information sent to the Cut Generator process by the Tree Manager.
- virtual void unpack_primal_solution (BCP_buffer &buf)

  Unpack the LP solution arriving from the LP process.
- virtual void generate_cuts (BCP_vec< BCP_var ∗ > &vars, BCP_vec< double > &x)

  Perform the actual cut generation.

Methods to set and get the pointer to the BCP_cg_prob

object.

It is unlikely that the users would want to muck around with these (especially with the set method!) but they are here to provide total control.

- void setCgProblemPointer (BCP_cg_prob ∗ptr)

  Set the pointer.
- BCP_cg_prob ∗ getCgProblemPointer ()

  Get the pointer.

Informational methods for the user.

- double upper_bound () const

  Return what is the best known upper bound (might be BCP_DBL_MAX)
- int current_phase () const

  Return the phase the algorithm is in.
- int current_level () const

  Return the level of the search tree node for which cuts are being generated.
- int current_index () const

  Return the internal index of the search tree node for which cuts are being generated.
- int current_iteration () const

  Return the iteration count within the search tree node for which cuts are being generated.

Methods to get/set BCP parameters on the fly

- char **get_param** (const BCP_cg_par::chr_params key) const
- int **get_param** (const BCP_cg_par::int_params key) const
- double **get_param** (const BCP_cg_par::dbl_params key) const
- const BCP_string & **get_param** (const BCP_cg_par::str_params key) const
- void **set_param** (const BCP_cg_par::chr_params key, const bool val)
- void **set_param** (const BCP_cg_par::chr_params key, const char val)
- void **set_param** (const BCP_cg_par::int_params key, const int val)
- void **set_param** (const BCP_cg_par::dbl_params key, const double val)
- void **set_param** (const BCP_cg_par::str_params key, const char ∗val)

Constructor, Destructor

- **BCP_cg_user** ()
- virtual ∼BCP_cg_user ()

  Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

4.4.1 Detailed Description

The BCP_cg_user class is the base class from which the user can derive a problem specific class to be used in the Cut Generator process.

In that derived class the user can store data to be used in the methods she overrides. Also that is the object the user must return in the USER_initialize::cg_init() method.

There are two kind of methods in the class. The non-virtual methods are helper functions for the built-in defaults, but the user can use them as well. The virtual methods execute steps in the BCP algorithm where the user might want to override the default behavior.

The default implementations fall into three major categories.

- Empty; doesn't do anything and immediately returns (e.g., unpack_module_data()).
- There is no reasonable default, so throw an exception. This happens if the parameter settings drive the flow of in a way that BCP can't perform the necessary function. This behavior is correct since such methods are invoked only if the parameter settings drive the flow of the algorithm that way, in which case the user better implement those methods. (At the momemnt there is no such method in CG.)
- A default is given. Frequently there are multiple defaults and parameters govern which one is selected (e.g., unpack_primal_solution()).

Definition at line 50 of file BCP_cg_user.hpp.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 virtual BCP_cg_user::∼BCP_cg_user ( ) `[inline]`, `[virtual]`

Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Definition at line 118 of file BCP_cg_user.hpp.

4.4.3 Member Function Documentation

4.4.3.1 void BCP_cg_user::send_cut (const **BCP_cut** & *cut*)

Pack the argument into the message buffer and send it to the sender of the LP solution.

Whenever the user generates a cut in the generate_cuts() method she should invoke this method to immediately send off the cut to the LP process.

4.4.3.2 virtual void BCP_cg_user::unpack_module_data (**BCP_buffer** & *buf*) `[virtual]`

Unpack the initial information sent to the Cut Generator process by the Tree Manager.

This information was packed by the method BCP_tm_user::pack_module_data() invoked with `BCP_ProcessType-_CG` as the third (target process type) argument.

Default: empty method.

4.4.3.3 virtual void BCP_cg_user::unpack_primal_solution (**BCP_buffer** & *buf*) `[virtual]`

Unpack the LP solution arriving from the LP process.

This method is invoked only if the user packs the info necessary for cut generation by herself, i.e., she overrides the BCP_lp_user::pack_primal_solution(). If that's the case the user has to unpack the same info she has packed in the LP process.

4.4.3.4 virtual void BCP_cg_user::generate_cuts (BCP_vec< BCP_var * > & *vars,* BCP_vec< double > & *x*) [virtual]

Perform the actual cut generation.

Whenever a cut is generated, the user should invoke the send_cut() method to send the generated cut back to the LP process.

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

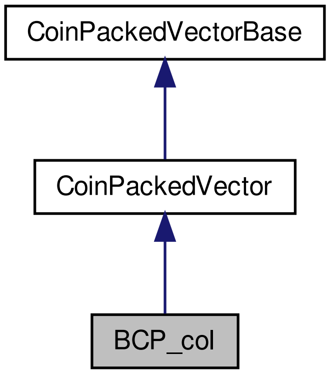
- BCP_cg_user.hpp

4.5 BCP_col Class Reference

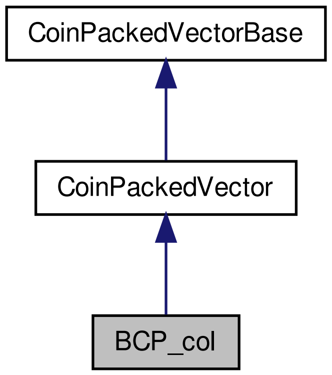
This class holds a column in a compressed form.

`#include <BCP_matrix.hpp>`

Inheritance diagram for BCP_col:



Collaboration diagram for BCP_col:



Public Member Functions

Query methods

- double Objective () const

  Return the objective coefficient.
- double LowerBound () const

  Return the lower bound.
- double UpperBound () const

  Return the upper bound.

General modifying methods

- void Objective (const double obj)

  Set the objective coefficient to the given value.
- void LowerBound (const double lb)

  Set the lower bound to the given value.
- void UpperBound (const double ub)

  Set the upper bound to the given value.
- BCP_col & operator= (const BCP_col &x)

  Assignment operator: copy over the contents of x.
- void assign (const int size, int ∗&ElementIndices, double ∗&ElementValues, const double Obj, const double LB, const double UB)

  Set the objective coefficient, lower and upper bounds to the given values.
- void copy (const int size, const int ∗ElementIndices, const double ∗ElementValues, const double Obj, const double LB, const double UB)

  Copy the arguments into the appropriate data members.
- void copy (BCP_vec< int >::const_iterator firstind, BCP_vec< int >::const_iterator lastind, BCP_vec< double >::const_iterator firstval, BCP_vec< double >::const_iterator lastval, const double Obj, const double LB, const double UB)

  Same as the other `copy()` method, except that instead of using vectors the indices (values) are given in `[firstind,lastind)` (`[firstval,lastval)`).

Constructors / Destructor

- BCP_col ()

  The default constructor creates an empty column with 0 as objective coefficient, 0.0 as lower and +infinity as upper bound.
- BCP_col (const BCP_col &x)

  The copy constructor makes a copy of `x`*.*
- BCP_col (BCP_vec< int >::const_iterator firstind, BCP_vec< int >::const_iterator lastind, BCP_vec< double >::const_iterator firstval, BCP_vec< double >::const_iterator lastval, const double Obj, const double LB, const double UB)

  This constructor acts exactly like the `copy` *method with the same argument list.*
- BCP_col (const int size, int ∗&ElementIndices, double ∗&ElementValues, const double Obj, const double LB, const double UB)

  This constructor acts exactly like the `assign` *method with the same argument list.*
- **BCP_col** (const **CoinPackedVectorBase** &vec, const double Obj, const double LB, const double UB)
- ∼BCP_col ()

  The destructor deletes all data members.

Protected Attributes

Data members

- double _Objective

  The objective function coefficient corresponding to the column.
- double _LowerBound

  The lower bound corresponding to the column.
- double _UpperBound

  The upper bound corresponding to the column.

4.5.1 Detailed Description

This class holds a column in a compressed form.

That is, it is a packed vector with an objective coefficient, lower and upper bound.

Definition at line 26 of file BCP_matrix.hpp.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 BCP_col::BCP_col ( ) `[inline]`

The default constructor creates an empty column with 0 as objective coefficient, 0.0 as lower and +infinity as upper bound.

Definition at line 113 of file BCP_matrix.hpp.

4.5.2.2 BCP_col::BCP_col (const BCP_col & *x*) `[inline]`

The copy constructor makes a copy of `x`.

Definition at line 116 of file BCP_matrix.hpp.

4.5.2.3 BCP_col::BCP_col (BCP_vec< int >::const_iterator *firstind*, BCP_vec< int >::const_iterator *lastind*, BCP_vec< double >::const_iterator *firstval*, BCP_vec< double >::const_iterator *lastval*, const double *Obj*, const double *LB*, const double *UB*) `[inline]`

This constructor acts exactly like the `copy` method with the same argument list.

Definition at line 121 of file BCP_matrix.hpp.

4.5.2.4 BCP_col::BCP_col (const int *size*, int ∗& *ElementIndices*, double ∗& *ElementValues*, const double *Obj*, const double *LB*, const double *UB*) `[inline]`

This constructor acts exactly like the `assign` method with the same argument list.

Definition at line 131 of file BCP_matrix.hpp.

4.5.2.5 BCP_col::∼BCP_col ( ) `[inline]`

The destructor deletes all data members.

Definition at line 143 of file BCP_matrix.hpp.

4.5.3 Member Function Documentation

4.5.3.1 double BCP_col::Objective ( ) const `[inline]`

Return the objective coefficient.

Definition at line 44 of file BCP_matrix.hpp.

4.5.3.2 double BCP_col::LowerBound ( ) const `[inline]`

Return the lower bound.

Definition at line 46 of file BCP_matrix.hpp.

4.5.3.3 double BCP_col::UpperBound ( ) const `[inline]`

Return the upper bound.

Definition at line 48 of file BCP_matrix.hpp.

4.5.3.4 void BCP_col::Objective (const double *obj*) `[inline]`

Set the objective coefficient to the given value.

Definition at line 55 of file BCP_matrix.hpp.

4.5.3.5 void BCP_col::LowerBound (const double *lb*) `[inline]`

Set the lower bound to the given value.

Definition at line 57 of file BCP_matrix.hpp.

4.5.3.6 void BCP_col::UpperBound (const double *ub*) `[inline]`

Set the upper bound to the given value.

Definition at line 59 of file BCP_matrix.hpp.

4.5.3.7 BCP_col& BCP_col::operator= (const BCP_col & *x*) [inline]

Assignment operator: copy over the contents of `x`.

Definition at line 62 of file BCP_matrix.hpp.

4.5.3.8 void BCP_col::assign (const int *size,* int ∗& *ElementIndices,* double ∗& *ElementValues,* const double *Obj,* const double *LB,* const double *UB*) [inline]

Set the objective coefficient, lower and upper bounds to the given values.

Also invokes the assign method of the underlying packed vector.

Definition at line 73 of file BCP_matrix.hpp.

4.5.3.9 void BCP_col::copy (const int *size,* const int ∗ *ElementIndices,* const double ∗ *ElementValues,* const double *Obj,* const double *LB,* const double *UB*) [inline]

Copy the arguments into the appropriate data members.

Definition at line 83 of file BCP_matrix.hpp.

4.5.3.10 void BCP_col::copy (BCP_vec< int >::const_iterator *firstind,* BCP_vec< int >::const_iterator *lastind,* BCP_vec< double >::const_iterator *firstval,* BCP_vec< double >::const_iterator *lastval,* const double *Obj,* const double *LB,* const double *UB*) [inline]

Same as the other `copy()` method, except that instead of using vectors the indices (values) are given in `[firstind,lastind)` (`[firstval,lastval)`).

Definition at line 95 of file BCP_matrix.hpp.

4.5.4 Member Data Documentation

4.5.4.1 double BCP_col::_Objective [protected]

The objective function coefficient corresponding to the column.

Definition at line 32 of file BCP_matrix.hpp.

4.5.4.2 double BCP_col::_LowerBound [protected]

The lower bound corresponding to the column.

Definition at line 34 of file BCP_matrix.hpp.

4.5.4.3 double BCP_col::_UpperBound [protected]

The upper bound corresponding to the column.

Definition at line 36 of file BCP_matrix.hpp.

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

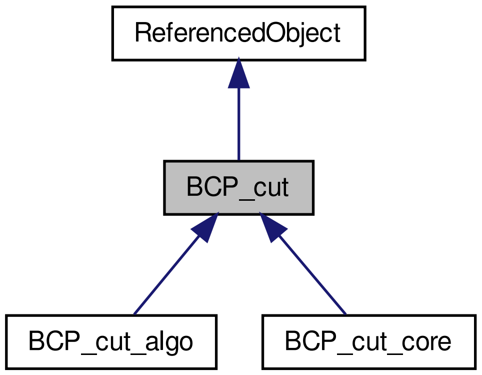
- BCP_matrix.hpp

4.6 BCP_cut Class Reference

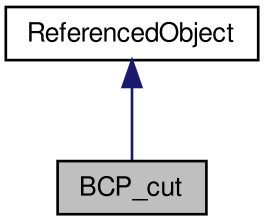
Abstract base class that defines members common to all types of cuts.

`#include <BCP_cut.hpp>`

Inheritance diagram for BCP_cut:



Collaboration diagram for BCP_cut:



Public Member Functions

Constructor and destructor

Note that there is no default constructor.

There is no such thing as "default cut".

- BCP_cut (const double lb, const double ub)

  The constructor sets the internal index of the cut to zero and the other data members to the given arguments.
- virtual ∼BCP_cut ()

  The destructor is virtual so that the appropriate destructor is invoked for every cut.

Query methods

- virtual BCP_object_t obj_type () const =0

  Return the type of the variable.
- int effective_count () const

  Return the effectiveness count of the cut (only in LP process).
- double lb () const

  Return the lower bound on the cut.
- double ub () const

  Return the upper bound on the cut.
- int bcpind () const

  Return the internal index of the cut.
- BCP_obj_status status () const

  Return the status of the cut.
- bool dont_send_to_pool () const

  Return whether the cut should be sent to the Cut Pool process.
- bool is_non_removable () const

  Return whether the cut marked as NotRemovable.
- bool is_to_be_removed () const

  Return whether the cut must be removed from the formulation.

Modifying methods

- void set_effective_count (const int cnt)

  Set the effectiveness count to the given value.
- int increase_effective_count ()

  Increase the effectiveness count by 1 (or to 1 if it was negative).
- int decrease_effective_count ()

  Decrease the effectiveness count by 1 (or to -1 if it was positive).
- void set_lb (const double lb)

  Set the lower bound on the cut.
- void set_ub (const double ub)

  Set the upper bound on the cut.
- void change_lb_ub_st (const BCP_obj_change &change)

  Set the lower/upper bounds and the status of the cut simultaneously to the values given in the data members of the argument.
- void change_bounds (const double lb, const double ub)

  Change just the lower/upper bounds.
- void set_bcpind (const int bcpind)

  Set the internal index of the cut.
- void set_bcpind_flip ()

  Flip the internal index of the variable to its negative.
- void set_status (const BCP_obj_status stat)

  Set the status of the cut.
- void dont_send_to_pool (bool flag)

  Set/unset the flag controlling whether the cut could be sent to the Cut Pool process.
- void make_active ()

  Mark the cut as active.
- void make_non_removable ()

  Mark the cut as NotRemovable.
- void make_to_be_removed ()

  Mark the cut as ToBeRemoved.

Protected Attributes

- double _lb

  Lower bound of the cut.
- double _ub

  Upper bound of the cut.

4.6.1 Detailed Description

Abstract base class that defines members common to all types of cuts.

Classes describing the three types of cuts (core and algorithmic) are derived from this class. No object of type BCP_cut can exist (having purely virtual members in the class enforces this restriction).

Definition at line 29 of file BCP_cut.hpp.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 BCP_cut::BCP_cut (const double *lb*, const double *ub*) `[inline]`

The constructor sets the internal index of the cut to zero and the other data members to the given arguments.

Definition at line 68 of file BCP_cut.hpp.

4.6.2.2 virtual BCP_cut::∼BCP_cut ( ) `[inline]`, `[virtual]`

The destructor is virtual so that the appropriate destructor is invoked for every cut.

Definition at line 72 of file BCP_cut.hpp.

4.6.3 Member Function Documentation

4.6.3.1 virtual BCP_object_t BCP_cut::obj_type ( ) const `[pure virtual]`

Return the type of the variable.

Implemented in BCP_cut_algo, and BCP_cut_core.

4.6.3.2 int BCP_cut::effective_count ( ) const `[inline]`

Return the effectiveness count of the cut (only in LP process).

Definition at line 80 of file BCP_cut.hpp.

4.6.3.3 double BCP_cut::lb ( ) const `[inline]`

Return the lower bound on the cut.

Definition at line 82 of file BCP_cut.hpp.

4.6.3.4 double BCP_cut::ub ( ) const `[inline]`

Return the upper bound on the cut.

Definition at line 84 of file BCP_cut.hpp.

4.6.3.5 int BCP_cut::bcpind ( ) const `[inline]`

Return the internal index of the cut.

Definition at line 86 of file BCP_cut.hpp.

4.6.3.6 BCP_obj_status BCP_cut::status ( ) const `[inline]`

Return the status of the cut.

Definition at line 91 of file BCP_cut.hpp.

4.6.3.7 bool BCP_cut::dont_send_to_pool ( ) const [inline]

Return whether the cut should be sent to the Cut Pool process.

(Assuming that it stays in the formulation long enough to qualify to be sent to the Cut Pool at all.

Definition at line 95 of file BCP_cut.hpp.

4.6.3.8 bool BCP_cut::is_non_removable ( ) const [inline]

Return whether the cut marked as NotRemovable.

Such cuts are, e.g., the branching cuts.

Definition at line 100 of file BCP_cut.hpp.

4.6.3.9 bool BCP_cut::is_to_be_removed ( ) const [inline]

Return whether the cut must be removed from the formulation.

There are very few circumstances when this flag is set; all of them are completely internal to BCP.

Definition at line 106 of file BCP_cut.hpp.

4.6.3.10 void BCP_cut::set_effective_count (const int *cnt*) [inline]

Set the effectiveness count to the given value.

Definition at line 115 of file BCP_cut.hpp.

4.6.3.11 int BCP_cut::increase_effective_count ( ) [inline]

Increase the effectiveness count by 1 (or to 1 if it was negative).

Return the new effectiveness count.

Definition at line 118 of file BCP_cut.hpp.

4.6.3.12 int BCP_cut::decrease_effective_count ( ) [inline]

Decrease the effectiveness count by 1 (or to -1 if it was positive).

Return the new effectiveness count.

Definition at line 124 of file BCP_cut.hpp.

4.6.3.13 void BCP_cut::set_lb (const double *lb*) [inline]

Set the lower bound on the cut.

Definition at line 129 of file BCP_cut.hpp.

4.6.3.14 void BCP_cut::set_ub (const double *ub*) [inline]

Set the upper bound on the cut.

Definition at line 131 of file BCP_cut.hpp.

4.6.3.15 void BCP_cut::change_lb_ub_st (const BCP_obj_change & *change*) [inline]

Set the lower/upper bounds and the status of the cut simultaneously to the values given in the data members of the argument.

Definition at line 134 of file BCP_cut.hpp.

4.6.3.16 void BCP_cut::change_bounds (const double *lb*, const double *ub*) [inline]

Change just the lower/upper bounds.

Definition at line 142 of file BCP_cut.hpp.

4.6.3.17 void BCP_cut::set_bcpind (const int *bcpind*) [inline]

Set the internal index of the cut.

Definition at line 149 of file BCP_cut.hpp.

4.6.3.18 void BCP_cut::set_status (const BCP_obj_status *stat*) [inline]

Set the status of the cut.

Definition at line 156 of file BCP_cut.hpp.

4.6.3.19 void BCP_cut::dont_send_to_pool (bool *flag*) [inline]

Set/unset the flag controlling whether the cut could be sent to the Cut Pool process.

Definition at line 159 of file BCP_cut.hpp.

4.6.3.20 void BCP_cut::make_active ( ) [inline]

Mark the cut as active.

Note that when this method is invoked the lp formulation must be modified as well: the original bounds of the cut must be reset.

Definition at line 167 of file BCP_cut.hpp.

4.6.3.21 void BCP_cut::make_non_removable ( ) [inline]

Mark the cut as NotRemovable.

Definition at line 171 of file BCP_cut.hpp.

4.6.3.22 void BCP_cut::make_to_be_removed ( ) [inline]

Mark the cut as ToBeRemoved.

It will actually be removed immediately after all cuts that have to be marked this way are marked.

Definition at line 178 of file BCP_cut.hpp.

4.6.4 Member Data Documentation

4.6.4.1 double BCP_cut::_lb [protected]

Lower bound of the cut.

Definition at line 56 of file BCP_cut.hpp.

4.6.4.2 double BCP_cut::_ub [protected]

Upper bound of the cut.

Definition at line 58 of file BCP_cut.hpp.

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

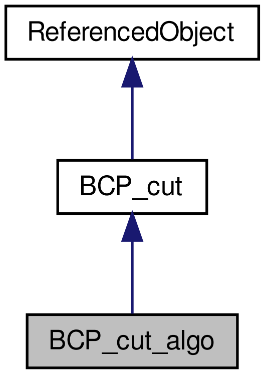
- BCP_cut.hpp

4.7 BCP_cut_algo Class Reference

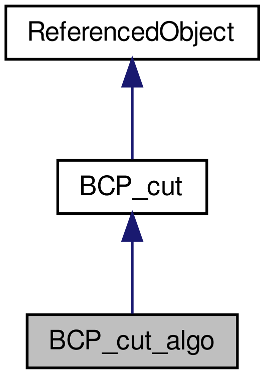
This is the class from which the user should derive her own algorithmic cuts.

`#include <BCP_cut.hpp>`

Inheritance diagram for BCP_cut_algo:



Collaboration diagram for BCP_cut_algo:



Public Member Functions

Constructor and destructor

- BCP_cut_algo (const double lb, const double ub)

    This constructor just sets the data members to the given values.
- virtual ∼BCP_cut_algo ()=0

    The destructor deletes the object.

Query methods

- BCP_object_t obj_type () const

    Return `BCP_AlgoObj` indicating that the object is an algorithmic cut.

Additional Inherited Members

4.7.1 Detailed Description

This is the class from which the user should derive her own algorithmic cuts.

Note that such an object cannot be constructed (it has pure virtual methods), only objects with types derived from BCP-_cut_algo can be created. Such objects are constructed either directly by the user or by the unpacking functions of the `BCP_xx_user` classes.

Definition at line 242 of file BCP_cut.hpp.

4.7.2 Constructor & Destructor Documentation

4.7.2.1 BCP_cut_algo::BCP_cut_algo (const double *lb*, const double *ub*) `[inline]`

This constructor just sets the data members to the given values.

See also the constructor of BCP_cut.

Definition at line 259 of file BCP_cut.hpp.

4.7.2.2 virtual BCP_cut_algo::∼BCP_cut_algo ( ) `[pure virtual]`

The destructor deletes the object.

4.7.3 Member Function Documentation

4.7.3.1 BCP_object_t BCP_cut_algo::obj_type ( ) const `[inline]`, `[virtual]`

Return `BCP_AlgoObj` indicating that the object is an algorithmic cut.

Implements BCP_cut.

Definition at line 268 of file BCP_cut.hpp.

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

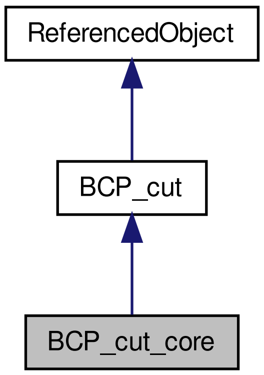
- BCP_cut.hpp

4.8 BCP_cut_core Class Reference

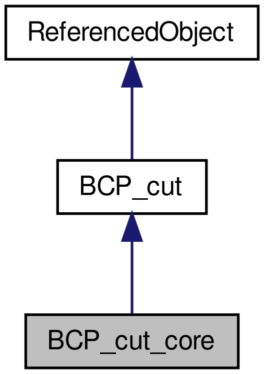
Core cuts are the cuts that always stay in the LP formulation.

`#include <BCP_cut.hpp>`

Inheritance diagram for BCP_cut_core:



Collaboration diagram for BCP_cut_core:



Public Member Functions

Constructors and desctructor

- BCP_cut_core (const BCP_cut_core &x)

  The copy constructor makes a replica of the argument.
- BCP_cut_core (const double lb, const double ub)

  This constructor just sets the data members to the given values.
- ∼BCP_cut_core ()

  The destructor deletes the object.

Query methods

- BCP_object_t obj_type () const

  Return `BCP_CoreObj` indicating that the object is a core cut.

Additional Inherited Members

4.8.1 Detailed Description

Core cuts are the cuts that always stay in the LP formulation.

Therefore the data members in the base class are quite sufficient to describe the cut. The only thing that has to be done here is overriding the pure virtual method obj_type().

Definition at line 195 of file BCP_cut.hpp.

4.8.2 Constructor & Destructor Documentation

4.8.2.1 BCP_cut_core::BCP_cut_core (const BCP_cut_core & *x*) `[inline]`

The copy constructor makes a replica of the argument.

Definition at line 210 of file BCP_cut.hpp.

4.8.2.2 BCP_cut_core::BCP_cut_core (const double *lb,* const double *ub*) `[inline]`

This constructor just sets the data members to the given values.

See also the constructor of BCP_cut.

Definition at line 217 of file BCP_cut.hpp.

4.8.2.3 BCP_cut_core::∼BCP_cut_core ( ) `[inline]`

The destructor deletes the object.

Definition at line 219 of file BCP_cut.hpp.

4.8.3 Member Function Documentation

4.8.3.1 BCP_object_t BCP_cut_core::obj_type ( ) const `[inline]`, `[virtual]`

Return `BCP_CoreObj` indicating that the object is a core cut.

Implements BCP_cut.

Definition at line 226 of file BCP_cut.hpp.

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

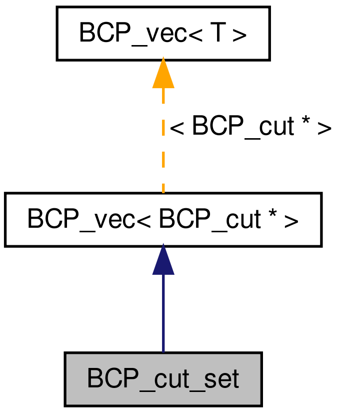
- BCP_cut.hpp

4.9 BCP_cut_set Class Reference

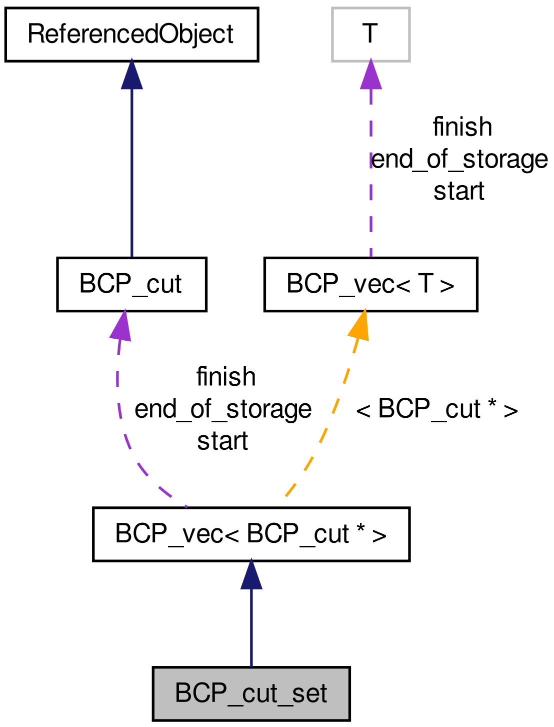
This class is just a collection of pointers to cuts with a number of methods to manipulate these cuts and/or select certain entries.

`#include <BCP_cut.hpp>`

Inheritance diagram for BCP_cut_set:



Collaboration diagram for BCP_cut_set:



Public Member Functions

Constructor and destructor

- BCP_cut_set ()

  The default constructor creates a cut set with no cuts in it.
- ∼BCP_cut_set ()

  The destructor empties the cut set.

Modifying methods

- void append (const BCP_vec< BCP_cut $*$ > &x)

  Append the cuts in the vector x to the end of the cut set.
- void append (BCP_cut_set::const_iterator first, BCP_cut_set::const_iterator last)

  Append the cuts in `[first, last)` to the end of the cut set.
- void set_lb_ub (const BCP_vec< int > &pos, BCP_vec< double >::const_iterator bounds)

  Set the lower/upper bound pairs of the entries given by the contents of `pos` to the values in `[bounds, bounds + pos`.
- void set_lb_ub_st (const BCP_vec< BCP_obj_change > &cc)

  Set the lower/upper bound pairs and the stati of the first `cc`.
- void set_lb_ub_st (BCP_vec< int >::const_iterator pos, const BCP_vec< BCP_obj_change > &cc)

Set the lower/upper bound pairs and the stati of the entries given by the content of `[pos, pos + cc`.

Methods related to deleting cuts from the cut set

- void move_deletable_to_pool (const BCP_vec< int > &deletable_cuts, BCP_vec< BCP_cut ∗ > &pool)
  Move the cut pointers whose indices are listed in `deletable_cuts` *into the* `pool`.

Additional Inherited Members

4.9.1 Detailed Description

This class is just a collection of pointers to cuts with a number of methods to manipulate these cuts and/or select certain entries.

Definition at line 279 of file BCP_cut.hpp.

4.9.2 Constructor & Destructor Documentation

4.9.2.1 BCP_cut_set::BCP_cut_set ( ) `[inline]`

The default constructor creates a cut set with no cuts in it.

Definition at line 293 of file BCP_cut.hpp.

4.9.2.2 BCP_cut_set::∼BCP_cut_set ( ) `[inline]`

The destructor empties the cut set.

NOTE: the destructor does NOT delete the cuts the members of the cut set point to.

Definition at line 296 of file BCP_cut.hpp.

4.9.3 Member Function Documentation

4.9.3.1 void BCP_cut_set::append (const **BCP_vec**< **BCP_cut** ∗ > & *x*) `[inline]`

Append the cuts in the vector `x` to the end of the cut set.

Reimplemented from BCP_vec< BCP_cut ∗ >.

Definition at line 304 of file BCP_cut.hpp.

4.9.3.2 void BCP_cut_set::append (**BCP_cut_set::const_iterator** *first*, **BCP_cut_set::const_iterator** *last*) `[inline]`

Append the cuts in `[first, last)` to the end of the cut set.

Reimplemented from BCP_vec< BCP_cut ∗ >.

Definition at line 309 of file BCP_cut.hpp.

4.9.3.3 void BCP_cut_set::set_lb_ub (const **BCP_vec**< int > & *pos*, **BCP_vec**< double >**::const_iterator** *bounds*)

Set the lower/upper bound pairs of the entries given by the contents of `pos` to the values in `[bounds, bounds + pos.`

`size()).`

4.9.3.4 void BCP_cut_set::set_lb_ub_st (const **BCP_vec**< **BCP_obj_change** > & *cc*)

Set the lower/upper bound pairs and the stati of the first `cc`.

`size()` entries to the triplets given in the vector. This method is invoked when the cut set is all the cuts in the current formulation and we want to change the triplets for the core cuts, which are at the beginning of that cut set.

4.9.3.5 void BCP_cut_set::set_lb_ub_st (**BCP_vec**< int >**::const_iterator** *pos,* const **BCP_vec**< **BCP_obj_change** > & *cc*)

Set the lower/upper bound pairs and the stati of the entries given by the content of `[pos, pos + cc`.

`size())` to the triplets contained in `cc`.

4.9.3.6 void BCP_cut_set::move_deletable_to_pool (const **BCP_vec**< int > & *deletable_cuts,* **BCP_vec**< **BCP_cut** ∗ > & *pool*)

Move the cut pointers whose indices are listed in `deletable_cuts` into the `pool`.

Note that this method does NOT compress the cut set, it merely replaces the cut pointers with 0 pointers.

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

- BCP_cut.hpp

4.10 BCP_fatal_error Class Reference

Currently there isn't any error handling in BCP.

`#include <BCP_error.hpp>`

Public Member Functions

- BCP_fatal_error (const std::string &str)

  The constructor prints out the error message, flushes the stdout buffer and aborts execution.
- ∼BCP_fatal_error ()

  The destructor exists only because it must.

4.10.1 Detailed Description

Currently there isn't any error handling in BCP.

When an object of this type is created, the string stored in the character array of the argument is printed out and execution is aborted with `abort()` (thus a core dump is created).

Definition at line 20 of file BCP_error.hpp.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 BCP_fatal_error::BCP_fatal_error (const std::string & *str*) `[inline]`

The constructor prints out the error message, flushes the stdout buffer and aborts execution.

Definition at line 34 of file BCP_error.hpp.

4.10.2.2 BCP_fatal_error::∼BCP_fatal_error ( ) `[inline]`

The destructor exists only because it must.

Definition at line 50 of file BCP_error.hpp.

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

- BCP_error.hpp

4.11 BCP_internal_brobj Class Reference

This class is the internal representation of a branching object.

`#include <BCP_branch.hpp>`

Public Member Functions

Constructors and destructor

- BCP_internal_brobj ()
  The default constructor creates an empty internal branching object (which can be filled later by unpacking a buffer).
- BCP_internal_brobj (BCP_lp_branching_object &candidate)
  This constructor sets the number of children and copies the contents of the positions and bounds of the forced changes to the positions and bounds of the newly created internal branching object.
- ∼BCP_internal_brobj ()
  The desctructor deletes all data members.

Query methods

- int child_num () const
  Return the number of children.
- int affected_varnum () const
  Return the number of affected variables.
- int affected_cutnum () const
  Return the number of affected cuts.
- const BCP_vec< int > & var_positions () const
  Return a `const` reference to the vector of positions of variables affected by the branching.
- const BCP_vec< int > & cut_positions () const
  Return a `const` reference to the vector of positions of cuts affected by the branching.
- BCP_vec< double >::const_iterator var_bounds_child (const int index) const
  Return a `const` iterator within `_var_bounds` to the location where the bound pairs for the `index`-th child start.
- BCP_vec< double >::const_iterator cut_bounds_child (const int index) const
  Return a `const` iterator within `_cut_bounds` to the location where the bound pairs for the `index`-th child start.

Interaction with the LP solver

- void apply_child_bounds (OsiSolverInterface ∗lp, int child_ind) const
  Modify the bounds in the LP solver by applying the changes specified for the `child_ind`-th child.

Packing and unpacking

- void pack (BCP_buffer &buf) const
  Pack the internal branching object into the buffer.
- void unpack (BCP_buffer &buf)
  Unpack an internal branching object from the buffer.

4.11.1 Detailed Description

This class is the internal representation of a branching object.

We document it only for the sake of completness, the user need not worry about it.

An internal branching object is created AFTER all the cuts/variables the branching object wanted to add the relaxation are already added, thus only the bound changes on affected variables are specified.

NOTE: There are only two ways to set up an internal branching object. One is through a constructor that passes on the data members, the second is to set it up with the default constructor then unpack its content from a buffer.

Definition at line 31 of file BCP_branch.hpp.

4.11.2 Constructor & Destructor Documentation

4.11.2.1 BCP_internal_brobj::BCP_internal_brobj ( ) `[inline]`

The default constructor creates an empty internal branching object (which can be filled later by unpacking a buffer).

Definition at line 66 of file BCP_branch.hpp.

4.11.2.2 BCP_internal_brobj::BCP_internal_brobj (BCP_lp_branching_object & *candidate*)

This constructor sets the number of children and copies the contents of the positions and bounds of the forced changes to the positions and bounds of the newly created internal branching object.

4.11.2.3 BCP_internal_brobj::∼BCP_internal_brobj ( ) `[inline]`

The desctructor deletes all data members.

Definition at line 73 of file BCP_branch.hpp.

4.11.3 Member Function Documentation

4.11.3.1 int BCP_internal_brobj::child_num ( ) const `[inline]`

Return the number of children.

Definition at line 79 of file BCP_branch.hpp.

4.11.3.2 int BCP_internal_brobj::affected_varnum ( ) const `[inline]`

Return the number of affected variables.

Definition at line 81 of file BCP_branch.hpp.

4.11.3.3 int BCP_internal_brobj::affected_cutnum ( ) const `[inline]`

Return the number of affected cuts.

Definition at line 83 of file BCP_branch.hpp.

4.11.3.4 const BCP_vec<int>& BCP_internal_brobj::var_positions ( ) const `[inline]`

Return a `const` reference to the vector of positions of variables affected by the branching.

Definition at line 87 of file BCP_branch.hpp.

4.11.3.5 const BCP_vec<int>& BCP_internal_brobj::cut_positions ( ) const [inline]

Return a `const` reference to the vector of positions of cuts affected by the branching.

Definition at line 90 of file BCP_branch.hpp.

4.11.3.6 BCP_vec<double>::const_iterator BCP_internal_brobj::var_bounds_child (const int *index*) const [inline]

Return a `const` iterator within `_var_bounds` to the location where the bound pairs for the `index`-th child start.

Definition at line 96 of file BCP_branch.hpp.

4.11.3.7 BCP_vec<double>::const_iterator BCP_internal_brobj::cut_bounds_child (const int *index*) const [inline]

Return a `const` iterator within `_cut_bounds` to the location where the bound pairs for the `index`-th child start.

Definition at line 103 of file BCP_branch.hpp.

4.11.3.8 void BCP_internal_brobj::apply_child_bounds (OsiSolverInterface * *lp,* int *child_ind*) const

Modify the bounds in the LP solver by applying the changes specified for the `child_ind`-th child.

4.11.3.9 void BCP_internal_brobj::pack (BCP_buffer & *buf*) const

Pack the internal branching object into the buffer.

4.11.3.10 void BCP_internal_brobj::unpack (BCP_buffer & *buf*)

Unpack an internal branching object from the buffer.

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

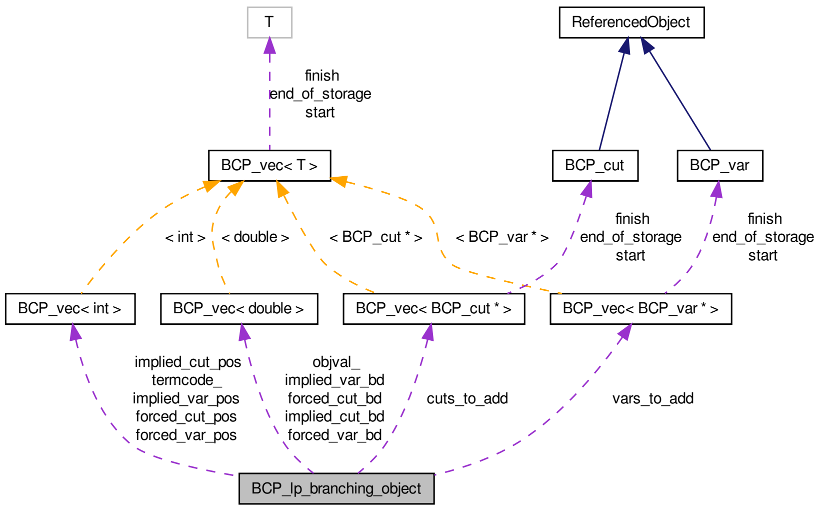
- BCP_branch.hpp

4.12 BCP_lp_branching_object Class Reference

This class describes a generic branching object.

`#include <BCP_lp_branch.hpp>`

Collaboration diagram for BCP_lp_branching_object:



Public Member Functions

Constructor and destructor

- BCP_lp_branching_object (const int children, BCP_vec< BCP_var ∗ > ∗const new_vars, BCP_vec< BCP_cut ∗ > ∗const new_cuts, const BCP_vec< int > ∗const fvp, const BCP_vec< int > ∗const fcp, const BCP_vec< double > ∗const fvb, const BCP_vec< double > ∗const fcb, const BCP_vec< int > ∗const ivp, const BCP_vec< int > ∗const icp, const BCP_vec< double > ∗const ivb, const BCP_vec< double > ∗const icb)

  The constructor makes a copy of each vector passed to it.
- **BCP_lp_branching_object** (const BCP_lp_integer_branching_object &o, const int ∗order)
- **BCP_lp_branching_object** (const OsiSolverInterface ∗osi, const BCP_lp_sos_branching_object &o, const int ∗order)
- void **set_presolve_result** (const BCP_vec< double > &objval, const BCP_vec< int > &termcode)
- ∼BCP_lp_branching_object ()

  The destructor deletes each vector.

Query methods

- int vars_affected () const

  Return the number of variables whose bounds are affected by the branching.
- int cuts_affected () const

  Return the number of cuts whose bounds are affected by the branching.
- int vars_added () const

  Return the number of variables added in the branching.
- int cuts_added () const

  Return the number of cuts added in the branching.
- BCP_vec< double >::const_iterator forced_var_bd_child (const int index) const

  Return a const iterator to the position in the forced variable bound changes where the new bounds for the `index`*-th child start.*
- BCP_vec< double >::const_iterator forced_cut_bd_child (const int index) const

Return a const iterator to the position in the forced cut bound changes where the new bounds for the `index`-th child start.

- BCP_vec< double >::const_iterator implied_var_bd_child (const int index) const

  Return a const iterator to the position in the implied variable bound changes where the new bounds for the `index`-th child start.
- BCP_vec< double >::const_iterator implied_cut_bd_child (const int index) const

  Return a const iterator to the position in the implied cut bound changes where the new bounds for the `index`-th child start.

Modifying methods

- void init_pos_for_added (const int added_vars_start, const int added_cuts_start)

  This method "cleans up" the positions and bounds.
- void apply_child_bd (OsiSolverInterface ∗lp, const int child_ind) const

  This method invokes the appropriate methods of `lp` to apply the forced and implied bound changes of the `child_-ind`-th child.
- void print_branching_info (const int orig_varnum, const double ∗x, const double ∗obj) const

  This method prints out some information about the branching object.

Public Attributes

Data members

- int child_num

  The number of children for this branching object.
- BCP_vec< BCP_var ∗ > ∗ vars_to_add

  Variables to be added to the formulation.
- BCP_vec< BCP_cut ∗ > ∗ cuts_to_add

  Cuts to be added to the formulation.

Data members referring to forced bound changes. "Forced" means

that the branching rule specifies this change.

- BCP_vec< int > ∗ forced_var_pos

  Positions of variables whose bounds change ("forcibly", by branching) in the children.
- BCP_vec< int > ∗ forced_cut_pos

  Positions of cuts whose bounds change ("forcibly", by branching) in the children.
- BCP_vec< double > ∗ forced_var_bd

  Contains the actual bounds for the variables indexed by `forced_var_pos`.
- BCP_vec< double > ∗ forced_cut_bd

  Contains the actual bounds for the cuts indexed by `forced_cut_pos`.

Data members referring to implied bound changes. "Implied" means

that these changes could be made by applying the forced changes and then doing logical fixing.

Therefore this information is not recorded in the Tree Manager when it stores the branching rule. However, if there are implied changes, it is useful to specify them, because strong branching may be more effecive then.

The interpretation of these data members is identical to their forced counterparts.

- BCP_vec< int > ∗ **implied_var_pos**
- BCP_vec< int > ∗ **implied_cut_pos**
- BCP_vec< double > ∗ **implied_var_bd**
- BCP_vec< double > ∗ **implied_cut_bd**

Data members referring to presolved values. The user may have

presolved the candidate.

Then she may store the termcodes and objvals of the children here.

- BCP_vec< double > ∗ **objval_**
- BCP_vec< int > ∗ **termcode_**

4.12.1 Detailed Description

This class describes a generic branching object.

The object may contain variables and cuts to be added to the formulation and it contains the bound changes for each child.

Note that it is unlikely that the user would need any of the member functions. She should simply call its constructor to create the object. The rest is internal to BCP.

Definition at line 70 of file BCP_lp_branch.hpp.

4.12.2 Constructor & Destructor Documentation

4.12.2.1 BCP_lp_branching_object::BCP_lp_branching_object (const int *children*, BCP_vec< BCP_var ∗ > ∗const *new_vars*, BCP_vec< BCP_cut ∗ > ∗const *new_cuts*, const BCP_vec< int > ∗const *fvp*, const BCP_vec< int > ∗const *fcp*, const BCP_vec< double > ∗const *fvb*, const BCP_vec< double > ∗const *fcb*, const BCP_vec< int > ∗const *ivp*, const BCP_vec< int > ∗const *icp*, const BCP_vec< double > ∗const *ivb*, const BCP_vec< double > ∗const *icb*) `[inline]`, `[explicit]`

The constructor makes a copy of each vector passed to it.

If a 0 pointer is passed for one of the arguments that means that the vector is empty. `new_{vars,cuts}` contains the variables/cuts to be added and `[fi][vc][pb]` contains the forced/implied variable/cut positions/bounds.

Definition at line 153 of file BCP_lp_branch.hpp.

4.12.2.2 BCP_lp_branching_object::∼BCP_lp_branching_object ( ) `[inline]`

The destructor deletes each vector.

Definition at line 218 of file BCP_lp_branch.hpp.

4.12.3 Member Function Documentation

4.12.3.1 int BCP_lp_branching_object::vars_affected ( ) const `[inline]`

Return the number of variables whose bounds are affected by the branching.

(Including both forced and implied changes.)

Definition at line 232 of file BCP_lp_branch.hpp.

4.12.3.2 int BCP_lp_branching_object::cuts_affected ( ) const `[inline]`

Return the number of cuts whose bounds are affected by the branching.

(Including both forced and implied changes.)

Definition at line 239 of file BCP_lp_branch.hpp.

4.12.3.3 int BCP_lp_branching_object::vars_added ( ) const [inline]

Return the number of variables added in the branching.

Definition at line 245 of file BCP_lp_branch.hpp.

4.12.3.4 int BCP_lp_branching_object::cuts_added ( ) const [inline]

Return the number of cuts added in the branching.

Definition at line 249 of file BCP_lp_branch.hpp.

4.12.3.5 BCP_vec<double>::const_iterator BCP_lp_branching_object::forced_var_bd_child (const int *index*) const [inline]

Return a const iterator to the position in the forced variable bound changes where the new bounds for the `index`-th child start.

This method should be invoked only if the appropriate data member is non-0.

Definition at line 257 of file BCP_lp_branch.hpp.

4.12.3.6 BCP_vec<double>::const_iterator BCP_lp_branching_object::forced_cut_bd_child (const int *index*) const [inline]

Return a const iterator to the position in the forced cut bound changes where the new bounds for the `index`-th child start.

This method should be invoked only if the appropriate data member is non-0.

Definition at line 265 of file BCP_lp_branch.hpp.

4.12.3.7 BCP_vec<double>::const_iterator BCP_lp_branching_object::implied_var_bd_child (const int *index*) const [inline]

Return a const iterator to the position in the implied variable bound changes where the new bounds for the `index`-th child start.

This method should be invoked only if the appropriate data member is non-0.

Definition at line 273 of file BCP_lp_branch.hpp.

4.12.3.8 BCP_vec<double>::const_iterator BCP_lp_branching_object::implied_cut_bd_child (const int *index*) const [inline]

Return a const iterator to the position in the implied cut bound changes where the new bounds for the `index`-th child start.

This method should be invoked only if the appropriate data member is non-0.

Definition at line 281 of file BCP_lp_branch.hpp.

4.12.3.9 void BCP_lp_branching_object::init_pos_for_added (const int *added_vars_start,* const int *added_cuts_start*)

This method "cleans up" the positions and bounds.

First it re-indexes the negative positions starting from `added_vars_start` for the variables (and from `added_-cuts_start` for the cuts). Then it reorders the positions (and follows that ordering with the bounds) so that the positions will be in increasing order.

4.12.3.10 void BCP_lp_branching_object::apply_child_bd (OsiSolverInterface ∗ *lp,* const int *child_ind*) const

This method invokes the appropriate methods of `lp` to apply the forced and implied bound changes of the `child_-ind`-th child.

4.12.3.11 void BCP_lp_branching_object::print_branching_info (const int *orig_varnum,* const double ∗ *x,* const double ∗ *obj*) const

This method prints out some information about the branching object.

(The positions, new bounds, the primal value of the variables.)

4.12.4 Member Data Documentation

4.12.4.1 int BCP_lp_branching_object::child_num

The number of children for this branching object.

Definition at line 84 of file BCP_lp_branch.hpp.

4.12.4.2 BCP_vec<BCP_var∗>∗ BCP_lp_branching_object::vars_to_add

Variables to be added to the formulation.

Definition at line 86 of file BCP_lp_branch.hpp.

4.12.4.3 BCP_vec<BCP_cut∗>∗ BCP_lp_branching_object::cuts_to_add

Cuts to be added to the formulation.

Definition at line 88 of file BCP_lp_branch.hpp.

4.12.4.4 BCP_vec<int>∗ BCP_lp_branching_object::forced_var_pos

Positions of variables whose bounds change ("forcibly", by branching) in the children.

If a position is non-negative, it refers to a variable in the current LP formulation. If a position is negative (-i), it refers to an added variable (i-1st).

Definition at line 97 of file BCP_lp_branch.hpp.

4.12.4.5 BCP_vec<int>∗ BCP_lp_branching_object::forced_cut_pos

Positions of cuts whose bounds change ("forcibly", by branching) in the children.

If a position is non-negative, it refers to a cut in the current LP formulation. If a position is negative (-i), it refers to an added cut (i-1st).

Definition at line 102 of file BCP_lp_branch.hpp.

4.12.4.6 BCP_vec<double>∗ BCP_lp_branching_object::forced_var_bd

Contains the actual bounds for the variables indexed by `forced_var_pos`.

List the lower/upper bound pairs for each of the variables, for each child in turn. The length of this vector is thus `child_num * 2 * forced_var_pos.size()`.

Definition at line 107 of file BCP_lp_branch.hpp.

4.12.4.7 BCP_vec<double>∗ BCP_lp_branching_object::forced_cut_bd

Contains the actual bounds for the cuts indexed by `forced_cut_pos`.

List the lower/upper bound pairs for each of the cuts, for each child in turn. The length of this vector is thus `child_num * 2 * forced_cut_pos.size()`.

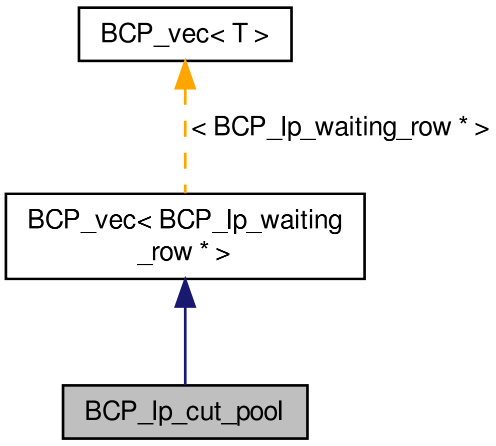
Definition at line 112 of file BCP_lp_branch.hpp.

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

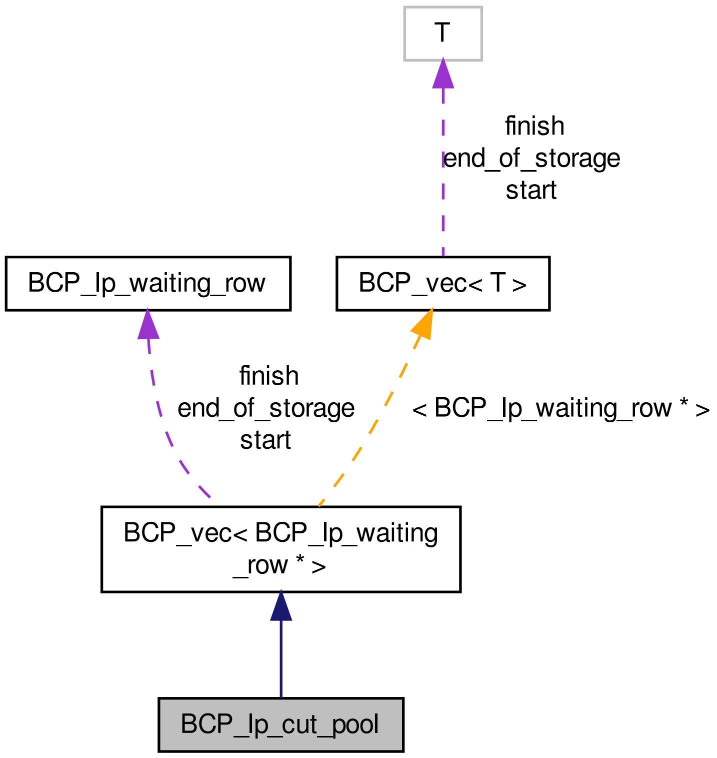
- BCP_lp_branch.hpp

4.13 BCP_lp_cut_pool Class Reference

Inheritance diagram for BCP_lp_cut_pool:



Collaboration diagram for BCP_lp_cut_pool:



Additional Inherited Members

4.13.1 Detailed Description

Definition at line 47 of file BCP_lp_pool.hpp.

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

- BCP_lp_pool.hpp

4.14 BCP_lp_integer_branching_object Class Reference

This class exist only so that we can extract information from OsiIntegerBranchingObject.

```
#include <BCP_lp_branch.hpp>
```

4.14.1 Detailed Description

This class exist only so that we can extract information from OsiIntegerBranchingObject.

Definition at line 25 of file BCP_lp_branch.hpp.

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

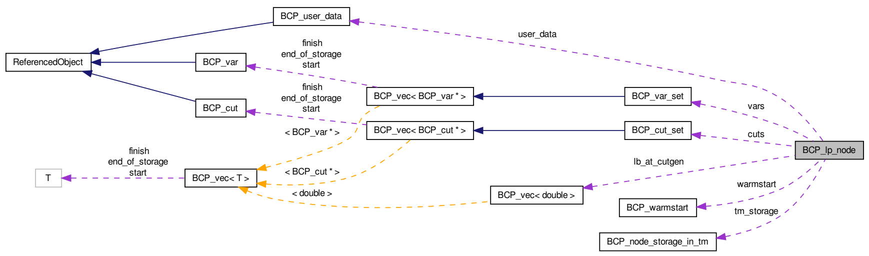
- BCP_lp_branch.hpp

4.15 BCP_lp_node Class Reference

NO OLD DOC.

`#include <BCP_lp_node.hpp>`

Collaboration diagram for BCP_lp_node:



Public Member Functions

Constructor and destructor

- **BCP_lp_node** ()
- ∼**BCP_lp_node** ()

Query methods

- size_t **varnum** () const
- size_t **cutnum** () const

Modifying methods

- void **clean** ()

Public Attributes

- BCP_user_data ∗ user_data

  Data the user wants to pass along with the search tree node.

Data members

- BCP_node_storage_in_tm **tm_storage**

Process id's ??

- int **cg**
- int **cp**
- int **vg**
- int **vp**

4.15.1 Detailed Description

NO OLD DOC.

This class holds the description of the current node itself.

Definition at line 92 of file BCP_lp_node.hpp.

4.15.2 Member Data Documentation

4.15.2.1 BCP_user_data∗ BCP_lp_node::user_data

Data the user wants to pass along with the search tree node.

For now it cannot be stored wrt. the parent.

Definition at line 147 of file BCP_lp_node.hpp.

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

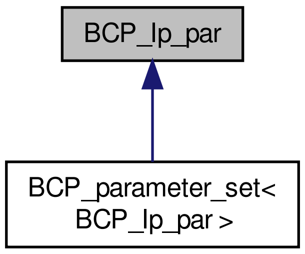
- BCP_lp_node.hpp

4.16 BCP_lp_par Struct Reference

Parameters used in the LP process.

`#include <BCP_lp_param.hpp>`

Inheritance diagram for BCP_lp_par:



Public Types

- enum chr_params {
  BranchOnCuts, CompareNewCutsToOldOnes, CompareNewVarsToOldOnes, DoReducedCostFixingAtZero, DoReducedCostFixingAtAnything, MessagePassingIsSerial, ReportWhenDefaultIsExecuted, SendFathomed-NodeDesc,
  NoCompressionAtFathom , LpVerb_First, LpVerb_AddedCutCount, LpVerb_AddedVarCount,
  LpVerb_ChildrenInfo, LpVerb_ColumnGenerationInfo, LpVerb_CutsToCutPoolCount, LpVerb_VarsToVarPool-

Count,
LpVerb_FathomInfo, LpVerb_IterationCount, LpVerb_RelaxedSolution, LpVerb_FinalRelaxedSolution,
LpVerb_LpMatrixSize, LpVerb_LpSolutionValue, LpVerb_MatrixCompression, LpVerb_NodeTime,
LpVerb_PresolvePositions, LpVerb_PresolveResult, LpVerb_ProcessedNodeIndex, LpVerb_ReportCutGen-Timeout,
LpVerb_ReportVarGenTimeout, LpVerb_ReportLocalCutPoolSize, LpVerb_ReportLocalVarPoolSize, LpVerb_-RepricingResult,
LpVerb_VarTightening, LpVerb_RowEffectivenessCount, LpVerb_StrongBranchPositions, LpVerb_StrongBranch-Result,
LpVerb_GeneratedCutCount, LpVerb_GeneratedVarCount, LpVerb_Last }

Character parameters.

- enum int_params {
NiceLevel, ScaleMatrix, SlackCutDiscardingStrategy, CutEffectiveCountBeforePool,
CutPoolCheckFrequency, VarPoolCheckFrequency, IneffectiveConstraints, IneffectiveBeforeDelete,
MaxNonDualFeasToAdd_Min, MaxNonDualFeasToAdd_Max, CutViolationNorm, MaxCutsAddedPerIteration,
MaxVarsAddedPerIteration, MaxLeftoverCutNum, DeletedColToCompress_Min, DeletedRowToCompress_Min,
MaxPresolveIter, StrongBranchNum, StrongBranch_CloseToHalfNum, BranchingObjectComparison,
ChildPreference, FeasibilityTest, WarmstartInfo, InfoForCG,
InfoForVG }

  Integer parameters.
- enum dbl_params {
Granularity, DeletedColToCompress_Frac, DeletedRowToCompress_Frac, MaxNonDualFeasToAdd_Frac,
MaxLeftoverCutFrac, IntegerTolerance, FirstLP_FirstCutTimeout, LaterLP_FirstCutTimeout,
FirstLP_AllCutsTimeout, LaterLP_AllCutsTimeout, FirstLP_FirstVarTimeout, LaterLP_FirstVarTimeout,
FirstLP_AllVarsTimeout, LaterLP_AllVarsTimeout, MaxRunTime }

  Double parameters.
- enum str_params { LogFileName }

  String parameters.
- enum str_array_params

  There are no string array parameters.

Verbosity Parameters.

```
These are all true/false parameters; if the parameter value is set
then the corresponding information is printed.
```

4.16.1 Detailed Description

Parameters used in the LP process.

These parameters can be set in the original parameter file by including the following line:

`BCP_{parameter name} {parameter value}.`

Definition at line 14 of file BCP_lp_param.hpp.

4.16.2 Member Enumeration Documentation

4.16.2.1 enum BCP_lp_par::chr_params

Character parameters.

All of these variables are used as booleans (true = 1, false = 0).

Enumerator:

BranchOnCuts If true, BCP supports branching on cuts by providing potential branching candidates for the user. These are cuts that were added to the formulation at some point but became slack later in subsequent LP relaxations.

Values: true (1), false (0). Default: 0.

CompareNewCutsToOldOnes If true then the LP process will check each newly received cut whether it already exists in the local cut pool or not.

Values: true (1), false (0). Default: 1.

CompareNewVarsToOldOnes If true then the LP process will check each newly arrived variable whether it already exists in the local variable pool or not.

Values: true (1), false (0). Default: 1.

DoReducedCostFixingAtZero If true the BCP will attempt to do reduced cost fixing only for variables currently at zero.

Values: true (1), false (0). Default: 1.

DoReducedCostFixingAtAnything If true the BCP will attempt to do reduced cost fixing for any variable, no matter what is their current value.

Values: true (1), false (0). Default: 1.

MessagePassingIsSerial Indicates whether message passing is serial (all processes are on the same processor) or not.

Values: true (1), false (0). Default: 1.

ReportWhenDefaultIsExecuted Print out a message when the default version of an overridable method is executed. Default: 1.

SendFathomedNodeDesc Whether to send back the description of fathomed search tree nodes to the Tree Manager. (Might be needed to write proof of optimality.)

Values: true (1), false (0). Default: 1.

NoCompressionAtFathom Whether we should refrain from compressing the problem description right before a fathomed node's description is sent back to the tree manager.

Values: true (1), false (0). Default: 0.

LpVerb_First Just a marker for the first LpVerb.

LpVerb_AddedCutCount Print the number of cuts added from the local cut pool in the current iteration. (BCP_-lp_main_loop)

LpVerb_AddedVarCount Print the number of variables added from the local variable pool in the curent iteration. (BCP_lp_main_loop)

LpVerb_ChildrenInfo After a branching object is selected print what happens to the presolved children (e.g., fathomed). (BCP_print_brobj_stat)

LpVerb_ColumnGenerationInfo Print the number of variables generated before resolving the Lp ir fathoming a node. (BCP_lp_fathom)

LpVerb_CutsToCutPoolCount Print the number of cuts sent from the LP to the cut pool. (BCP_lp_send_cuts_-to_cp)

LpVerb_VarsToVarPoolCount $****$ This parameter is not used anywhere!!! $****$

LpVerb_FathomInfo Print information related to fathoming. (BCP_lp_main_loop, BCP_lp_perform_fathom, BCP-_lp_branch) (BCP_lp_fathom)

LpVerb_IterationCount Print the "Starting iteration x" line. (BCP_lp_main_loop)

LpVerb_RelaxedSolution Turn on the user hook "display_lp_solution". (BCP_lp_main_loop)

LpVerb_FinalRelaxedSolution Turn on the user hook "display_lp_solution" for the last LP relaxation solved at a search tree node. (BCP_lp_main_loop)

LpVerb_LpMatrixSize $****$ This parameter is not used anywhere!!! $****$

LpVerb_LpSolutionValue Print the size of the problem matrix and the LP solution value after resolving the LP. (BCP_lp_main_loop)

LpVerb_MatrixCompression Print the number of columns and rows that were deleted during matrix compression. (BCP_lp_delete_cols_and_rows)

LpVerb_NodeTime For each tree node print out how much time was spent on it.

LpVerb_PresolvePositions Print detailed information about all the branching candidates during strong branching. LpVerb_PresolveResult must be set for this parameter to have an effect. (BCP_lp_perform_strong_branching)

LpVerb_PresolveResult Print information on the presolved branching candidates during strong branching. (BC-P_lp_perform_strong_branching)

LpVerb_ProcessedNodeIndex Print the "Processing NODE x on LEVEL y" line. (BCP_lp-main_loop)

LpVerb_ReportCutGenTimeout Print information if receiving cuts is timed out. (BCP_lp_generate_cuts)

LpVerb_ReportVarGenTimeout Print information if receiving variables is timed out. (BCP_lp_generate_vars)

LpVerb_ReportLocalCutPoolSize Print the current number of cuts in the cut pool. This number is printed several times: before and after generating columns at the current iteration, after removing non-essential cuts, etc. (BCP_lp_generate_cuts)

LpVerb_ReportLocalVarPoolSize Similar as above for variables. (BCP_lp_generate_vars)

LpVerb_RepricingResult **** This parameter is not used anywhere!!! ****

LpVerb_VarTightening Print the number of variables whose bounds have been changed by reduced cost fixing or logical fixing. (BCP_lp_fix_vars)

LpVerb_RowEffectivenessCount Print the number of ineffective rows in the current problem. The definition of what rows are considered ineffective is determined by the paramter IneffectiveConstraints. (BCP_lp_adjust_-row_effectiveness)

LpVerb_StrongBranchPositions Print detailed information on the branching candidate selected by strong branching. LpVerb_StrongBranchResult must be set fo this parameter to have an effect. (BCP_print_brobj_-stat)

LpVerb_StrongBranchResult Print information on the branching candidate selected by strong branching. (BCP-_print_brobj_stat)

LpVerb_GeneratedCutCount Print the number of cuts generated during this iteration (since the LP was resolved last time). (BCP_lp_main_loop)

LpVerb_GeneratedVarCount Print the number of variables generated during this iteration. (BCP_lp_main_loop)

LpVerb_Last Just a marker for the last LpVerb.

Definition at line 17 of file BCP_lp_param.hpp.

4.16.2.2 enum BCP_lp_par::int_params

Integer parameters.

Enumerator:

NiceLevel What should be the "niceness" of the LP process. In a *nix environment the LP process will be reniced to this level.

Values: whatever the operating system accepts. Default: 0.

ScaleMatrix Indicates how matrix scaling should be performed. This parameter is directly passed to the LP solver's `load_lp` member function.

Values: whatever is valid for the LP solver that the user has passed on to BCP. Default: 0.

SlackCutDiscardingStrategy The slack cut discarding strategy used in the default version of the function `purge_slack_pool()`.

Values: BCP_slack_cut_discarding. Default: `BCP_DiscardSlackCutsAtNewIteration`

CutEffectiveCountBeforePool A cut has to remain effective through this many iterations in the LP before it is sent to the Cut Pool process.

The default 1000 effectively says that only those cuts are sent to the pool which are effective at branching.

Values: any positive number. Default: 1000.

CutPoolCheckFrequency The Cut Pool is queried for violated valid inequalities after the first LP relaxation is solved and then after every this many iterations.

Values: any positive number. Default: 10.

VarPoolCheckFrequency The Variable Pool is queried for columns that improve the formulation after the first LP realxation is solved, and then after every this many iterations.

Values: any positive number. Default: 10.

IneffectiveConstraints Indicates which constraints should be considered ineffective.

Values: BCP_IneffectiveConstraints. Default: `BCP_IneffConstr_ZeroDualValue`.

IneffectiveBeforeDelete How many times in a row a constraint must be found ineffective before it is marked for deletion.

Values: any positive number. Default: 1.

MaxNonDualFeasToAdd_Min The number of non dual-feasible colums that can be added at a time to the formulation is a certain fraction (`MaxNonDualFeasToAdd_Frac`) of the number of columns in the current formulation. However, if the computed number is outside of the range `[MaxNonDualFeasToAdd_Min, MaxNonDualFeasToAdd_Max]` then it will be set to the appropriate endpoint of the range.

Values: . Default: 5.

MaxNonDualFeasToAdd_Max See the description of the previous parameter.

Values: . Default: 200.

CutViolationNorm How cut violation should be computed.

Values: BCP_CutViolationNorm. Default: `BCP_CutViolationNorm_Plain` (violation in the usual sense).

(The other option is `BCP_CutViolationNorm_Distance`: the distance of the fractional point from the cut.

MaxCutsAddedPerIteration The maximum number of violated valid inequalities that can be added per iteration.

Values: . Default: 100,000.

MaxVarsAddedPerIteration The maximum number of variables that can be added per iteration.

Values: . Default: 100,000.

MaxLeftoverCutNum The maximum number of violated but not added cuts to be kept from one iteration to the next. Also see the MaxLeftoverCutFrac parameter.

DeletedColToCompress_Min The number of columns that must be marked for deletion before matrix compression can occur. Matrix compressions also subject to a minimum number of marked columns as a fraction of the current number of columns (see `DeletedColToCompress_Frac`).

Values: positive integer. Default: 10.

DeletedRowToCompress_Min The number of rows that must be marked for deletion before matrix compression can occur. Matrix compressionis also subject to a minimum number of marked columns as a fraction of the current number of columns (see `DeletedRowToCompress_Frac`).

Values: positive integer. Default: 10.

MaxPresolveIter Upper limit on the number of iterations performed in each of the children of the search tree node when presolving branching candidates. This parameter is passed onto the LP solver. If the parameter is set to -1 then the branching candidates are not presolved and the first branching candidate is chosen (if there is any).

Note that for different LP solvers (e.g., simplex based algorithm or the Volume Algorithm) the meaning of an iteration is different, thus this parameter will be set differently.

Values: . Default: 100,000.

StrongBranchNum Specifies how many branching variables with values close to half between two integers should be chosen by the built-in branching variable selection routine `select_branching_candidates()` (if this routine is used at all).

Values: . Default: 3.

StrongBranch_CloseToHalfNum <bf>Deprecated parameter. Use StrongBranchNum</bf>

BranchingObjectComparison Specifies the rule used for built-in branching object comparison (if the buit-in routine is used at all).

Values: BCP_branching_object_comparison. Default: `BCP_HighestLowObjval`.

ChildPreference Specifies the rule used for selecting one of the children of the search tree node for diving.

Values: BCP_child_preference. Default: `BCP_PreferChild_LowBound`.

FeasibilityTest Specifies which built-in MIP feasibility testing routine should be invoked (if a buit-in routine is used at all).

Values: BCP_feasibility_test. Default: `BCP_FullTest_Feasible`

WarmstartInfo Specifies how warmstart information should be stored in the TM. Possible valueas: BCP_-WarmstartNone, BCP_WarmstartRoot, BCP_WarmstartParent. The first is obvious, the second means the root node's warmstart info should be sent out along with each node, the last means the parent's warmstart info should be sent out. Default: BCP_WarmstartParent

InfoForCG Indicates what part of the primal solution is sent to the Cut Generator process if the BCP_lp_user-::pack_primal_solution() method is not overridden.

Values: See BCP_primal_solution_description. Default: `BCP_PrimalSolution_Nonzeros`.

InfoForVG Indicates what part of the dual solution is sent to the Variable Generator process if the BCP_lp_user-::pack_dual_solution() method is not overridden.

Values: See BCP_dual_solution_description. Default: `BCP_DualSolution_Full`.

Definition at line 163 of file BCP_lp_param.hpp.

4.16.2.3 enum BCP_lp_par::dbl_params

Double parameters.

Enumerator:

Granularity The minimum difference between the objective value of any two feasible solution (with different objective values).

Values: . Default: .

DeletedColToCompress_Frac The fraction of columns that must be marked for deletion before matrix compression can occur. Matrix compression is also subject to a minimum number of marked columns (see `DeletedColToCompress_Min`).

Values: . Default: `1e-8`.

DeletedRowToCompress_Frac The fraction of rows that must be marked for deletion before matrix compression can occur. Matrix compression is also subject to a minimum number of marked rows (see `DeletedRowToCompress_Min`).

Values: . Default: `1e-8`.

MaxNonDualFeasToAdd_Frac The number of non dual-feasible colums that can be added to the formulation at a time cannot exceed this fraction of the the number of columns in the curent formulation. However, this limit can be overruled by the hard bounds `MaxNonDualFeasToAdd_Min` and `MaxNonDualFeasToAdd_Max` (see above).

Default: 0.05.

MaxLeftoverCutFrac The maximum fraction of the violated but not added cuts to be kept from one iteration to the next. Also see the MaxLeftoverCutNum parameter.

IntegerTolerance Values not further from an integer value than the value of this parameter are considered to be integer.

Values: . Default: .

FirstLP_FirstCutTimeout This and the following three parameters control how long the LP process waits for generated cuts. The parameters specify waiting times (in seconds) separately for the first and later LP relaxations at a search tree node, and also the time to receive the first cut vs all the cuts.

Note that it might make sense to set `AllCutsTimeout` for a *shorter* time than the first cut time out: "We are willing to wait 5 seconds to receive a cut, but if we do receive a cut the total time we wait is only 2 seconds."

This parameter specifies the time to wait for the first generated cut at the first LP relaxation at a search tree node.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

LaterLP_FirstCutTimeout This parameter specifies the time to wait for the first generated cut at iterations that are not the first at a search tree node. See the remarks at `FirstLP_FirstCutTimeout` as well.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

FirstLP_AllCutsTimeout This parameter specifies the time to wait for cuts at the first LP relaxation at a search tree node. See the remarks at `FirstLP_FirstCutTimeout` as well.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

LaterLP_AllCutsTimeout This parameter specifies the time to wait for cuts at iterations that are not the first at a search tree node. See the remarks at `FirstLP_FirstCutTimeout` as well.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

FirstLP_FirstVarTimeout This and the following three parameters control how long the LP process waits for generated variables. These parameters are analogous to the ones that control the waiting time for generated cuts. See the remarks at `FirstLP_FirstCutTimeout` as well.

This parameter specifies the time to wait for the first generated variable at the first LP relaxation at a search tree node.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

LaterLP_FirstVarTimeout This parameter specifies the time to wait for the first generated variable at iterations that are not the first at a search tree node. See the remarks at `FirstLP_FirstVarTimeout` as well.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

FirstLP_AllVarsTimeout This parameter specifies the time to wait for variables at the first LP relaxation at a search tree node. See the remarks at `FirstLP_FirstVarTimeout` as well.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

LaterLP_AllVarsTimeout This parameter specifies the time to wait for variables at iterations that are not the first at a search tree node. See the remarks at `FirstLP_FirstVarTimeout` as well.

Values: Any non-negative number represents a waiting time in seconds. Negative numbers indicate no time limit on waiting. Default: -1.

MaxRunTime Maximum allowed running time.

Definition at line 313 of file BCP_lp_param.hpp.

4.16.2.4 enum BCP_lp_par::str_params

String parameters.

Enumerator:

LogFileName The filename all the output should go to.

Definition at line 424 of file BCP_lp_param.hpp.

4.16.2.5 enum BCP_lp_par::str_array_params

There are no string array parameters.

Definition at line 432 of file BCP_lp_param.hpp.

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

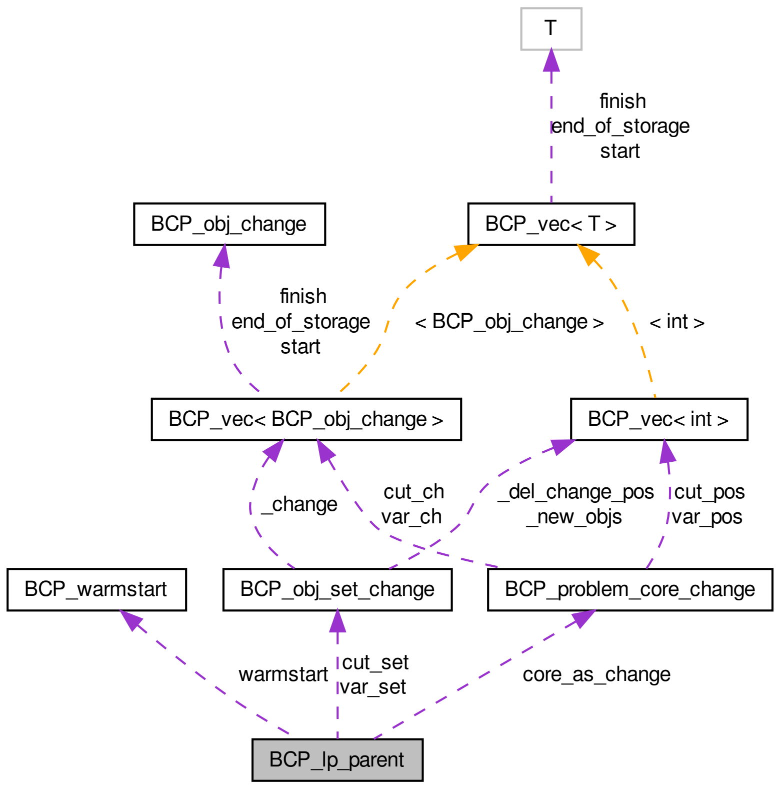
- BCP_lp_param.hpp

4.17 BCP_lp_parent Class Reference

NO OLD DOC.

```
#include <BCP_lp_node.hpp>
```

Collaboration diagram for BCP_lp_parent:



Public Member Functions

Constructor and destructor

- **BCP_lp_parent** ()
- ∼**BCP_lp_parent** ()

Modifying methods

- void **clean** ()

Public Attributes

Data members

- BCP_problem_core_change **core_as_change**
- BCP_obj_set_change **var_set**

  this is always explicit, it's just that coding is simpler if we reuse the BCP_obj_set_change object
- BCP_obj_set_change **cut_set**

  this is always explicit, it's just that coding is simpler if we reuse the BCP_obj_set_change object
- BCP_warmstart ∗ **warmstart**
- int **index**

4.17.1 Detailed Description

NO OLD DOC.

This class holds the description of the parent of the current node.

Definition at line 42 of file BCP_lp_node.hpp.

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

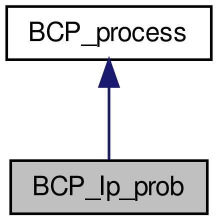
- BCP_lp_node.hpp

4.18 BCP_lp_prob Class Reference

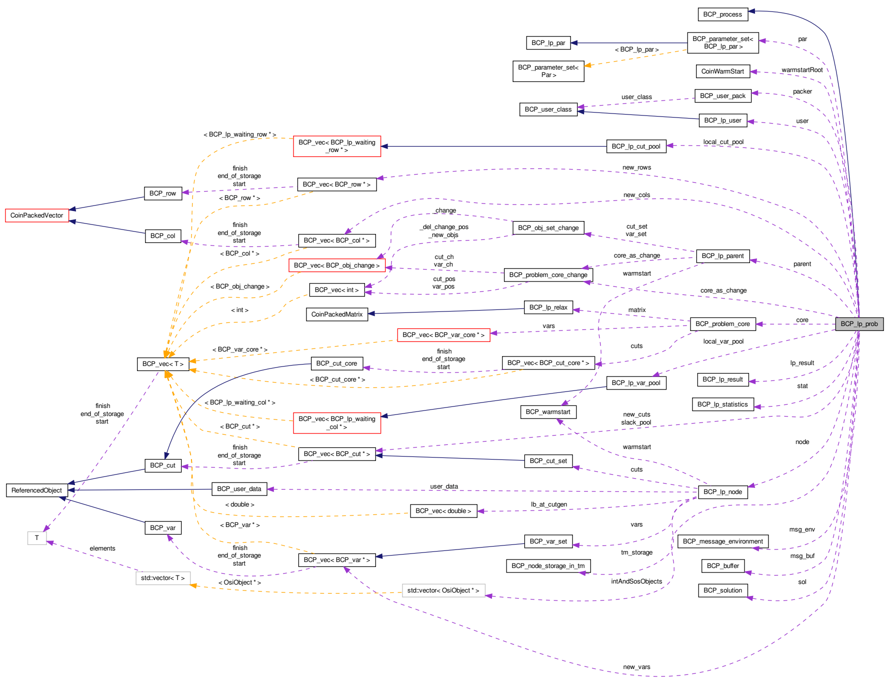
NO OLD DOC.

`#include <BCP_lp.hpp>`

Inheritance diagram for BCP_lp_prob:



Collaboration diagram for BCP_lp_prob:



Public Member Functions

Constructor and destructor

- **BCP_lp_prob** (int my_id, int parent)
- virtual ∼**BCP_lp_prob** ()

Methods to pack/unpack objects

- void **pack_var** (const BCP_var &var)
- BCP_var ∗ **unpack_var** ()
- void **pack_cut** (const BCP_cut &cut)
- BCP_cut ∗ **unpack_cut** ()

Acessing parameters

- char **param** (BCP_lp_par::chr_params key) const
- int **param** (BCP_lp_par::int_params key) const
- double **param** (BCP_lp_par::dbl_params key) const
- const BCP_string & **param** (BCP_lp_par::str_params key) const
- const BCP_vec< BCP_string > & **param** (BCP_lp_par::str_array_params key) const
- double **granularity** () const

Accessing bounds

- bool **has_ub** () const
- double **ub** () const
- bool **ub** (double new_ub)
- bool **over_ub** (double lb) const

Public Attributes

- std::vector< OsiObject ∗ > intAndSosObjects

    Things that can be branched on.

User provided members

- BCP_lp_user ∗ **user**
- BCP_user_pack ∗ packer

    A class that holds the methods about how to pack things.
- OsiSolverInterface ∗ **master_lp**
- OsiSolverInterface ∗ **lp_solver**
- BCP_message_environment ∗ **msg_env**

Parameters

- BCP_parameter_set< BCP_lp_par > **par**

Description of the core of the problem

- BCP_problem_core ∗ **core**
- BCP_problem_core_change ∗ **core_as_change**

Current search tree node and its parent

- BCP_lp_node ∗ node

    Description he current search tree node.
- BCP_lp_parent ∗ parent

    Description of the parent of the current node.
- **CoinWarmStart** ∗ warmstartRoot

    Description of the warmstart info from the end of the root node.

Information needed for processing a node

Need to be updated when starting a new node.

- BCP_lp_result ∗ **lp_result**
- int **var_bound_changes_since_logical_fixing**
- BCP_vec< BCP_cut ∗ > **slack_pool**
- BCP_lp_var_pool ∗ **local_var_pool**
- BCP_lp_cut_pool ∗ **local_cut_pool**
- int **next_var_index**
- int **last_var_index**
- int **next_cut_index**
- int **last_cut_index**

Time measurement

- double **start_time**
- BCP_lp_statistics **stat**

Internal data members

- double **upper_bound**
- int **phase**
- int **no_more_cuts_cnt**
- int **no_more_vars_cnt**

Message passing related fields

- BCP_buffer **msg_buf**

4.18.1 Detailed Description

NO OLD DOC.

Definition at line 102 of file BCP_lp.hpp.

4.18.2 Member Data Documentation

4.18.2.1 BCP_user_pack∗ BCP_lp_prob::packer

A class that holds the methods about how to pack things.

Definition at line 133 of file BCP_lp.hpp.

4.18.2.2 std::vector<OsiObject ∗> BCP_lp_prob::intAndSosObjects

Things that can be branched on.

If not filled out then BCP scans for them every time a new node is processed.

Definition at line 161 of file BCP_lp.hpp.

4.18.2.3 BCP_lp_node∗ BCP_lp_prob::node

Description he current search tree node.

Definition at line 168 of file BCP_lp.hpp.

4.18.2.4 BCP_lp_parent∗ BCP_lp_prob::parent

Description of the parent of the current node.

Definition at line 170 of file BCP_lp.hpp.

4.18.2.5 CoinWarmStart∗ BCP_lp_prob::warmstartRoot

Description of the warmstart info from the end of the root node.

Used only if the BCP_lp_par::WarmstartInfo parameter is set to BCP_WarmstartRoot.

Definition at line 174 of file BCP_lp.hpp.

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

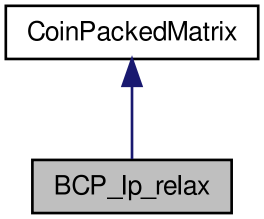
- BCP_lp.hpp

4.19 BCP_lp_relax Class Reference

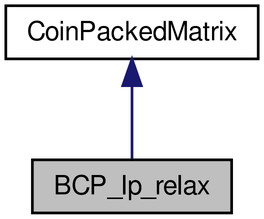
An object of type `BCP_lp_relax` holds the description of an lp relaxation.

`#include <BCP_matrix.hpp>`

Inheritance diagram for BCP_lp_relax:



Collaboration diagram for BCP_lp_relax:



Public Member Functions

Query methods

- size_t colnum () const

  The number of columns.
- size_t rownum () const

  The number of rows.
- const BCP_vec< double > & Objective () const

  A const reference to the vector of objective coefficients.
- const BCP_vec< double > & ColLowerBound () const

  A const reference to the vector of lower bounds on the variables.
- const BCP_vec< double > & ColUpperBound () const

  A const reference to the vector of upper bounds on the variables.
- const BCP_vec< double > & RowLowerBound () const

  A const reference to the vector of lower bounds on the cuts.
- const BCP_vec< double > & RowUpperBound () const

  A const reference to the vector of upper bounds on the cuts.

Methods modifying the whole LP relaxation.

- BCP_lp_relax & operator= (const BCP_lp_relax &mat)

  Copy the content of `x` *into the LP relaxation.*
- void reserve (const int MaxColNum, const int MaxRowNum, const int MaxNonzeros)

  Reserve space in the LP relaxation for at least `MaxColNum` *columns,* `MaxRowNum` *rows and* `MaxNonzeros` *nonzero entries.*
- void clear ()

  Clear the LP relaxation.
- void copyOf (const **CoinPackedMatrix** &m, const double ∗OBJ, const double ∗CLB, const double ∗CUB, const double ∗RLB, const double ∗RUB)

  Set up the LP relaxation by making a copy of the arguments.
- void assign (**CoinPackedMatrix** &m, double ∗&OBJ, double ∗&CLB, double ∗&CUB, double ∗&RLB, double ∗&RUB)

  Set up the LP relaxation by taking over the pointers in the arguments.

Methods for expanding/shrinking the LP relaxation.

- void erase_col_set (const BCP_vec< int > &pos)

  Remove the columns whose indices are listed in `pos` *from the LP relaxation.*
- void erase_row_set (const BCP_vec< int > &pos)

  Remove the rows whose indices are listed in `pos` *from the LP relaxation.*

Packing/unpacking

- void pack (BCP_buffer &buf) const

  Pack the LP relaxation into the buffer.
- void unpack (BCP_buffer &buf)

  Unpack the LP relaxation from the buffer.

Constructors and destructor

- BCP_lp_relax (const bool colordered=true)

  Create an empty LP relaxation with given ordering.
- BCP_lp_relax (const BCP_lp_relax &mat)

  The copy constructor makes a copy of the argument LP relaxation.
- BCP_lp_relax (BCP_vec< BCP_row ∗ > &rows, BCP_vec< double > &CLB, BCP_vec< double > &CUB, BCP_vec< double > &OBJ)

  Create a row major ordered LP relaxation by assigning the content of the arguments to the LP relaxation.
- BCP_lp_relax (BCP_vec< BCP_row ∗ > &rows, BCP_vec< double > &CLB, BCP_vec< double > &CUB, BCP_vec< double > &OBJ, double extra_gap, double extra_major)

  Same as the previous method except that this method allows extra_gap and extra_major to be specified.
- BCP_lp_relax (BCP_vec< BCP_col ∗ > &cols, BCP_vec< double > &RLB, BCP_vec< double > &RUB)

  Create a column major ordered LP relaxation by assigning the content of the arguments to the LP relaxation.
- BCP_lp_relax (BCP_vec< BCP_col ∗ > &cols, BCP_vec< double > &RLB, BCP_vec< double > &RUB, double extra_gap, double extra_major)

  Same as the previous method except that this method allows extra_gap and extra_major to be specified.
- BCP_lp_relax (const bool colordered, const BCP_vec< int > &VB, const BCP_vec< int > &EI, const BCP_-vec< double > &EV, const BCP_vec< double > &OBJ, const BCP_vec< double > &CLB, const BCP_vec< double > &CUB, const BCP_vec< double > &RLB, const BCP_vec< double > &RUB)

  Create an LP relaxation of the given ordering by copying the content of the arguments to the LP relaxation.
- BCP_lp_relax (const bool colordered, const int rownum, const int colnum, const int nznum, int ∗&VB, int ∗&EI, double ∗&EV, double ∗&OBJ, double ∗&CLB, double ∗&CUB, double ∗&RLB, double ∗&RUB)

  Create an LP relaxation of the given ordering by assigning the content of the arguments to the LP relaxation.
- ∼BCP_lp_relax ()

  The destructor deletes the data members.

4.19.1 Detailed Description

An object of type `BCP_lp_relax` holds the description of an lp relaxation.

The matrix, lower/upper bounds on the variables and cuts and objective coefficients for the variables.

Definition at line 267 of file BCP_matrix.hpp.

4.19.2 Constructor & Destructor Documentation

4.19.2.1 BCP_lp_relax::BCP_lp_relax (const bool *colordered* = `true`) `[inline]`

Create an empty LP relaxation with given ordering.

Definition at line 377 of file BCP_matrix.hpp.

4.19.2.2 BCP_lp_relax::BCP_lp_relax (const **BCP_lp_relax** & *mat*)

The copy constructor makes a copy of the argument LP relaxation.

4.19.2.3 BCP_lp_relax::BCP_lp_relax (**BCP_vec**< **BCP_row** * > & *rows*, **BCP_vec**< double > & *CLB*, **BCP_vec**< double > & *CUB*, **BCP_vec**< double > & *OBJ*)

Create a row major ordered LP relaxation by assigning the content of the arguments to the LP relaxation.

When the constructor returns the content of 'rows' doesn't change while that of CLB, CUB and OBJ will be whatever the default constructor for those arguments create.

4.19.2.4 BCP_lp_relax::BCP_lp_relax (**BCP_vec**< **BCP_row** * > & *rows*, **BCP_vec**< double > & *CLB*, **BCP_vec**< double > & *CUB*, **BCP_vec**< double > & *OBJ*, double *extra_gap*, double *extra_major*)

Same as the previous method except that this method allows extra_gap and extra_major to be specified.

For the description of those see the documentation of **CoinPackedMatrix**. (extra_gap will be used for adding columns, while extra major for adding rows)

4.19.2.5 BCP_lp_relax::BCP_lp_relax (**BCP_vec**< **BCP_col** * > & *cols*, **BCP_vec**< double > & *RLB*, **BCP_vec**< double > & *RUB*)

Create a column major ordered LP relaxation by assigning the content of the arguments to the LP relaxation.

When the constructor returns the content of 'cols' doesn't change while that of RLB and RUB will be whatever the default constructor for those arguments create.

4.19.2.6 BCP_lp_relax::BCP_lp_relax (**BCP_vec**< **BCP_col** * > & *cols*, **BCP_vec**< double > & *RLB*, **BCP_vec**< double > & *RUB*, double *extra_gap*, double *extra_major*)

Same as the previous method except that this method allows extra_gap and extra_major to be specified.

For the description of those see the documentation of **CoinPackedMatrix**. (extra_gap will be used for adding rows, while extra major for adding columns)

4.19.2.7 BCP_lp_relax::BCP_lp_relax (const bool *colordered*, const **BCP_vec**< int > & *VB*, const **BCP_vec**< int > & *EI*, const **BCP_vec**< double > & *EV*, const **BCP_vec**< double > & *OBJ*, const **BCP_vec**< double > & *CLB*, const **BCP_vec**< double > & *CUB*, const **BCP_vec**< double > & *RLB*, const **BCP_vec**< double > & *RUB*)

Create an LP relaxation of the given ordering by copying the content of the arguments to the LP relaxation.

4.19.2.8 BCP_lp_relax::BCP_lp_relax (const bool *colordered,* const int *rownum,* const int *colnum,* const int *nznum,* int ∗& *VB,* int ∗& *EI,* double ∗& *EV,* double ∗& *OBJ,* double ∗& *CLB,* double ∗& *CUB,* double ∗& *RLB,* double ∗& *RUB*)

Create an LP relaxation of the given ordering by assigning the content of the arguments to the LP relaxation.

When the constructor returns the content of the arguments will be NULL pointers.

4.19.3 Member Function Documentation

4.19.3.1 size_t BCP_lp_relax::colnum ( ) const [inline]

The number of columns.

Definition at line 288 of file BCP_matrix.hpp.

4.19.3.2 size_t BCP_lp_relax::rownum ( ) const [inline]

The number of rows.

Definition at line 290 of file BCP_matrix.hpp.

4.19.3.3 const BCP_vec<double>& BCP_lp_relax::Objective ( ) const [inline]

A const reference to the vector of objective coefficients.

Definition at line 292 of file BCP_matrix.hpp.

4.19.3.4 const BCP_vec<double>& BCP_lp_relax::ColLowerBound ( ) const [inline]

A const reference to the vector of lower bounds on the variables.

Definition at line 294 of file BCP_matrix.hpp.

4.19.3.5 const BCP_vec<double>& BCP_lp_relax::ColUpperBound ( ) const [inline]

A const reference to the vector of upper bounds on the variables.

Definition at line 296 of file BCP_matrix.hpp.

4.19.3.6 const BCP_vec<double>& BCP_lp_relax::RowLowerBound ( ) const [inline]

A const reference to the vector of lower bounds on the cuts.

Definition at line 298 of file BCP_matrix.hpp.

4.19.3.7 const BCP_vec<double>& BCP_lp_relax::RowUpperBound ( ) const [inline]

A const reference to the vector of upper bounds on the cuts.

Definition at line 300 of file BCP_matrix.hpp.

4.19.3.8 BCP_lp_relax& BCP_lp_relax::operator= (const BCP_lp_relax & *mat*)

Copy the content of `x` into the LP relaxation.

4.19.3.9 void BCP_lp_relax::reserve (const int *MaxColNum,* const int *MaxRowNum,* const int *MaxNonzeros*)

Reserve space in the LP relaxation for at least `MaxColNum` columns, `MaxRowNum` rows and `MaxNonzeros` nonzero entries.

This is useful when columns/rows are added to the LP relaxation in small chunks and to avoid a series of reallocation we reserve sufficient space up front.

4.19.3.10 void BCP_lp_relax::clear ( )

Clear the LP relaxation.

Reimplemented from **CoinPackedMatrix**.

4.19.3.11 void BCP_lp_relax::erase_col_set (const BCP_vec< int > & *pos*)

Remove the columns whose indices are listed in `pos` from the LP relaxation.

4.19.3.12 void BCP_lp_relax::erase_row_set (const BCP_vec< int > & *pos*)

Remove the rows whose indices are listed in `pos` from the LP relaxation.

4.19.3.13 void BCP_lp_relax::pack (BCP_buffer & *buf*) const

Pack the LP relaxation into the buffer.

4.19.3.14 void BCP_lp_relax::unpack (BCP_buffer & *buf*)

Unpack the LP relaxation from the buffer.

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

- BCP_matrix.hpp

4.20 BCP_lp_result Class Reference

This class holds the results after solving an LP relaxation.

`#include <BCP_lp_result.hpp>`

Public Member Functions

Constructor and destructor

- BCP_lp_result ()
  The default constructor initializes an empty solution, i.e., one which holds neither an exact nor an approximate solution.
- ∼BCP_lp_result ()
  The destructor deletes the data members if they are private copies.

Query methods for the solution. These methods (except for

the first) just return the value of the queried member (in case of the vector members a reference to the vector is returned instead of the pointer.

- const std::string & **solvername** () const
- int **termcode** () const
- int **iternum** () const
- double **objval** () const
- const double ∗ **x** () const
- const double ∗ **pi** () const
- const double ∗ **dj** () const
- const double ∗ **lhs** () const

Query methods for general solver information.

- double primalTolerance () const
  Return the primal tolerance of the solver.
- double dualTolerance () const
  Return the dual tolerance of the solver.

Modifying methods

- void get_results (OsiSolverInterface &lp_solver)
  Get the result from the LP solver.
- void fake_objective_value (const double val)
  Set the lower bound and the exact and approximate objective values to the value given in the argument.

4.20.1 Detailed Description

This class holds the results after solving an LP relaxation.

There may be an exact and/or an approximate solution.

Definition at line 39 of file BCP_lp_result.hpp.

4.20.2 Constructor & Destructor Documentation

4.20.2.1 BCP_lp_result::BCP_lp_result ( ) [inline]

The default constructor initializes an empty solution, i.e., one which holds neither an exact nor an approximate solution.

Definition at line 87 of file BCP_lp_result.hpp.

4.20.2.2 BCP_lp_result::∼BCP_lp_result ( ) [inline]

The destructor deletes the data members if they are private copies.

Definition at line 94 of file BCP_lp_result.hpp.

4.20.3 Member Function Documentation

4.20.3.1 double BCP_lp_result::primalTolerance ( ) const [inline]

Return the primal tolerance of the solver.

Definition at line 135 of file BCP_lp_result.hpp.

4.20.3.2 double BCP_lp_result::dualTolerance ( ) const [inline]

Return the dual tolerance of the solver.

Definition at line 137 of file BCP_lp_result.hpp.

4.20.3.3 void BCP_lp_result::get_results (OsiSolverInterface & *lp_solver*)

Get the result from the LP solver.

Non-vector members will get their values from the LP solver. Vector members are copied out from the LP solver.

4.20.3.4 void BCP_lp_result::fake_objective_value (const double *val*) [inline]

Set the lower bound and the exact and approximate objective values to the value given in the argument.

Definition at line 148 of file BCP_lp_result.hpp.

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

- BCP_lp_result.hpp

4.21 BCP_lp_sos_branching_object Class Reference

This class exist only so that we can extract information from OsiIntegerBranchingObject.

`#include <BCP_lp_branch.hpp>`

4.21.1 Detailed Description

This class exist only so that we can extract information from OsiIntegerBranchingObject.

Definition at line 38 of file BCP_lp_branch.hpp.

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

- BCP_lp_branch.hpp

4.22 BCP_lp_statistics Class Reference

NO OLD DOC.

`#include <BCP_lp.hpp>`

Public Member Functions

- BCP_lp_statistics ()

    The contsructor just zeros out every timing data.
- void display () const

    Print out the statistics.
- void add (const BCP_lp_statistics &stat)

    Add the argument statistics to this one.

Packing and unpacking

- void **pack** (BCP_buffer &buf)
- void **unpack** (BCP_buffer &buf)

4.22.1 Detailed Description

NO OLD DOC.

Definition at line 56 of file BCP_lp.hpp.

4.22.2 Member Function Documentation

4.22.2.1 void BCP_lp_statistics::add (const BCP_lp_statistics & *stat*)

Add the argument statistics to this one.

This method is used when multiple LP processes are running and their stats need to be combined.

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

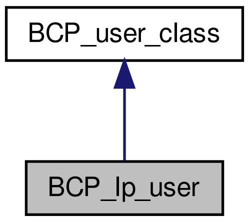
- BCP_lp.hpp

4.23 BCP_lp_user Class Reference

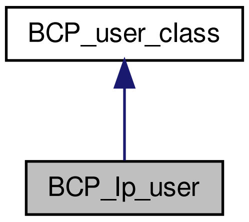
The BCP_lp_user class is the base class from which the user can derive a problem specific class to be used in the LP process.

```
#include <BCP_lp_user.hpp>
```

Inheritance diagram for BCP_lp_user:



Collaboration diagram for BCP_lp_user:



Public Member Functions

- void print (const bool ifprint, const char ∗format,...) const

  A method to print a message with the process id.
- int process_id () const

  What is the process id of the current process.
- int parent () const

  the process id of the parent
- void send_message (const int target, const BCP_buffer &buf, BCP_message_tag tag=BCP_Msg_User)

  Send a message to a particular process.
- void receive_message (const int sender, BCP_buffer &buf, BCP_message_tag tag=BCP_Msg_User)

  Wait for a message and receive it.
- void broadcast_message (const BCP_process_t proc_type, const BCP_buffer &buf)

  Broadcast the message to all processes of the given type.
- virtual void process_message (BCP_buffer &buf)

  Process a message that has been sent by another process' user part to this process' user part.
- virtual OsiSolverInterface ∗ initialize_solver_interface ()

  Create LP solver environment.
- virtual void initialize_int_and_sos_list (std::vector< OsiObject ∗ > &intAndSosObjects)

  Create the list of objects that can be used for branching (simple integer vars and SOS sets).
- virtual void initialize_new_search_tree_node (const BCP_vec< BCP_var ∗ > &vars, const BCP_vec< BCP_cut ∗ > &cuts, const BCP_vec< BCP_obj_status > &var_status, const BCP_vec< BCP_obj_status > &cut_status, BCP_vec< int > &var_changed_pos, BCP_vec< double > &var_new_bd, BCP_vec< int > &cut_changed_pos, BCP_vec< double > &cut_new_bd)

  Initializing a new search tree node.
- virtual void load_problem (OsiSolverInterface &osi, BCP_problem_core ∗core, BCP_var_set &vars, BCP_cut_set &cuts)

  Load the problem specified by core, vars, and cuts into the solver interface.
- virtual void modify_lp_parameters (OsiSolverInterface ∗lp, const int changeType, bool in_strong_branching)

  Modify parameters of the LP solver before optimization.
- virtual void process_lp_result (const BCP_lp_result &lpres, const BCP_vec< BCP_var ∗ > &vars, const BCP-_vec< BCP_cut ∗ > &cuts, const double old_lower_bound, double &true_lower_bound, BCP_solution ∗&sol, BCP_vec< BCP_cut ∗ > &new_cuts, BCP_vec< BCP_row ∗ > &new_rows, BCP_vec< BCP_var ∗ > &new_-vars, BCP_vec< BCP_col ∗ > &new_cols)

  Process the result of an iteration.
- virtual double compute_lower_bound (const double old_lower_bound, const BCP_lp_result &lpres, const BCP_-vec< BCP_var ∗ > &vars, const BCP_vec< BCP_cut ∗ > &cuts)

  Compute a true lower bound for the subproblem.
- virtual BCP_solution ∗ generate_heuristic_solution (const BCP_lp_result &lpres, const BCP_vec< BCP_var ∗ > &vars, const BCP_vec< BCP_cut ∗ > &cuts)

  Try to generate a heuristic solution (or return one generated during cut/variable generation.
- virtual void restore_feasibility (const BCP_lp_result &lpres, const std::vector< double ∗ > dual_rays, const BC-P_vec< BCP_var ∗ > &vars, const BCP_vec< BCP_cut ∗ > &cuts, BCP_vec< BCP_var ∗ > &vars_to_add, BCP_vec< BCP_col ∗ > &cols_to_add)

  Restoring feasibility.
- void reduced_cost_fixing (const double ∗dj, const double ∗x, const double gap, BCP_vec< BCP_var ∗ > &vars, int &newly_changed)

  Reduced cost fixing.

- virtual
  BCP_branching_object_relation compare_branching_candidates (BCP_presolved_lp_brobj ∗new_solved, BCP_-presolved_lp_brobj ∗old_solved)

  Decide which branching object is preferred for branching.
- virtual void set_actions_for_children (BCP_presolved_lp_brobj ∗best)

  Decide what to do with the children of the selected branching object.
- virtual void set_user_data_for_children (BCP_presolved_lp_brobj ∗best, const int selected)

  For each child create a user data object and put it into the appropriate entry in `best->user_data()`.
- virtual void set_user_data_for_children (BCP_presolved_lp_brobj ∗best)

  Deprecated version of the previos method (it does not pass the index of the selected branching candidate).

Methods to set and get the pointer to the BCP_lp_prob

object.

It is unlikely that the users would want to muck around with these (especially with the set method!) but they are here to provide total control.

- void setLpProblemPointer (BCP_lp_prob ∗ptr)

  Set the pointer.
- BCP_lp_prob ∗ getLpProblemPointer ()

  Get the pointer.

Informational methods for the user.

- double upper_bound () const

  Return what is the best known upper bound (might be BCP_DBL_MAX)
- bool over_ub (double lb) const

  Return true / false depending on whether the lb argument is over the current upper bound or not.
- int current_phase () const

  Return the phase the algorithm is in.
- int current_level () const

  Return the level of the search tree node being processed.
- int current_index () const

  Return the internal index of the search tree node being processed.
- int current_iteration () const

  Return the iteration count within the search tree node being processed.
- double start_time () const

  Return when the LP process started.
- BCP_user_data ∗ get_user_data ()

  Return a pointer to the BCP_user_data structure the user (may have) stored in this node.

Methods to get/set BCP parameters on the fly

- char **get_param** (const BCP_lp_par::chr_params key) const
- int **get_param** (const BCP_lp_par::int_params key) const
- double **get_param** (const BCP_lp_par::dbl_params key) const
- const BCP_string & **get_param** (const BCP_lp_par::str_params key) const
- void **set_param** (const BCP_lp_par::chr_params key, const bool val)
- void **set_param** (const BCP_lp_par::chr_params key, const char val)
- void **set_param** (const BCP_lp_par::int_params key, const int val)
- void **set_param** (const BCP_lp_par::dbl_params key, const double val)
- void **set_param** (const BCP_lp_par::str_params key, const char ∗val)

A methods to send a solution to the Tree Manager. The user can

invoke this method at any time to send off a solution.

- void **send_feasible_solution** (const BCP_solution ∗sol)

Constructor, Destructor

- **BCP_lp_user** ()
- virtual ∼BCP_lp_user ()

  Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Helper functions for selecting subset of entries from a double

vector.

The indices (their position with respect to `first`) of the variables satisfying the criteria are returned in the last argument.

- void select_nonzeros (const double ∗first, const double ∗last, const double etol, BCP_vec< int > &nonzeros) const

  Select all nonzero entries.
- void select_zeros (const double ∗first, const double ∗last, const double etol, BCP_vec< int > &zeros) const

  Select all zero entries.
- void select_positives (const double ∗first, const double ∗last, const double etol, BCP_vec< int > &positives) const

  Select all positive entries.
- void select_fractions (const double ∗first, const double ∗last, const double etol, BCP_vec< int > &fractions) const

  Select all fractional entries.

Packing and unpacking methods

- virtual void unpack_module_data (BCP_buffer &buf)

  Unpack the initial information sent to the LP process by the Tree Manager.

MIP feasibility testing of LP solutions and heuristics

- virtual BCP_solution ∗ test_feasibility (const BCP_lp_result &lp_result, const BCP_vec< BCP_var ∗ > &vars, const BCP_vec< BCP_cut ∗ > &cuts)

  Evaluate and return MIP feasibility of the current solution.

Helper functions for `test_feasibility`.

*If the solution is feasible a pointer to a BCP_solution_generic object is returned. Note that the solutions generated by these helper functions **DO NOT OWN** the pointers in the `_vars` member of the solution. Also note that all of these functions assume that the specified integer tolerance in larger than the LP primal tolerance extracted from `lpres` and that the solution in `lpres` do not violate the bounds by more than the LP tolerance.*

- BCP_solution_generic ∗ test_binary (const BCP_lp_result &lpres, const BCP_vec< BCP_var ∗ > &vars, const double etol) const

  Test whether all variables are 0/1.
- BCP_solution_generic ∗ test_integral (const BCP_lp_result &lpres, const BCP_vec< BCP_var ∗ > &vars, const double etol) const

  Test whether all variables are integer.

- BCP_solution_generic ∗ test_full (const BCP_lp_result &lpres, const BCP_vec< BCP_var ∗ > &vars, const double etol) const

    Test whether the variables specified as integers are really integer.

Packing of solutions

- virtual void pack_feasible_solution (BCP_buffer &buf, const BCP_solution ∗sol)

    Pack a MIP feasible solution into a buffer.
- virtual void pack_primal_solution (BCP_buffer &buf, const BCP_lp_result &lp_result, const BCP_vec< BCP_var ∗ > &vars, const BCP_vec< BCP_cut ∗ > &cuts)

    Pack the information necessary for cut generation into the buffer.
- virtual void pack_dual_solution (BCP_buffer &buf, const BCP_lp_result &lp_result, const BCP_vec< BCP_var ∗ > &vars, const BCP_vec< BCP_cut ∗ > &cuts)

    Pack the information necessary for variable generation into the buffer.

Displaying of LP solutions

- virtual void display_lp_solution (const BCP_lp_result &lp_result, const BCP_vec< BCP_var ∗ > &vars, const BCP_vec< BCP_cut ∗ > &cuts, const bool final_lp_solution)

    Display the result of most recent LP optimization.

Converting cuts and variables into rows and columns

- virtual void cuts_to_rows (const BCP_vec< BCP_var ∗ > &vars, BCP_vec< BCP_cut ∗ > &cuts, BCP_vec< BCP_row ∗ > &rows, const BCP_lp_result &lpres, BCP_object_origin origin, bool allow_multiple)

    Convert (and possibly lift) a set of cuts into corresponding rows for the current LP relaxation.
- virtual void vars_to_cols (const BCP_vec< BCP_cut ∗ > &cuts, BCP_vec< BCP_var ∗ > &vars, BCP_vec< BCP_col ∗ > &cols, const BCP_lp_result &lpres, BCP_object_origin origin, bool allow_multiple)

    Convert a set of variables into corresponding columns for the current LP relaxation.

Generating cuts and variables

- virtual void generate_cuts_in_lp (const BCP_lp_result &lpres, const BCP_vec< BCP_var ∗ > &vars, const BCP_vec< BCP_cut ∗ > &cuts, BCP_vec< BCP_cut ∗ > &new_cuts, BCP_vec< BCP_row ∗ > &new_rows)

    Generate cuts within the LP process.
- virtual void generate_vars_in_lp (const BCP_lp_result &lpres, const BCP_vec< BCP_var ∗ > &vars, const BCP_vec< BCP_cut ∗ > &cuts, const bool before_fathom, BCP_vec< BCP_var ∗ > &new_vars, BCP_vec< BCP_col ∗ > &new_cols)

    Generate variables within the LP process.
- virtual BCP_object_compare_result compare_cuts (const BCP_cut ∗c0, const BCP_cut ∗c1)

    Compare two generated cuts.
- virtual BCP_object_compare_result compare_vars (const BCP_var ∗v0, const BCP_var ∗v1)

    Compare two generated variables.

Logical fixing

- virtual void logical_fixing (const BCP_lp_result &lpres, const BCP_vec< BCP_var ∗ > &vars, const BCP_vec< BCP_cut ∗ > &cuts, const BCP_vec< BCP_obj_status > &var_status, const BCP_vec< BCP_obj_status > &cut_status, const int var_bound_changes_since_logical_fixing, BCP_vec< int > &changed_pos, BCP_vec< double > &new_bd)

    This method provides an opportunity for the user to tighten the bounds of variables.

Branching related methods

- virtual BCP_branching_decision select_branching_candidates (const BCP_lp_result &lpres, const BCP_vec< BCP_var ∗ > &vars, const BCP_vec< BCP_cut ∗ > &cuts, const BCP_lp_var_pool &local_var_pool, const BCP_lp_cut_pool &local_cut_pool, BCP_vec< BCP_lp_branching_object ∗ > &cands, bool force_branch=false)

  Decide whether to branch or not and select a set of branching candidates if branching is decided upon.

Helper functions for select_branching_candidates()

- virtual int **try_to_branch** (OsiBranchingInformation &branchInfo, OsiSolverInterface ∗solver, OsiChooseVariable ∗choose, OsiBranchingObject ∗&branchObject, bool allowVarFix)
- void branch_close_to_half (const BCP_lp_result &lpres, const BCP_vec< BCP_var ∗ > &vars, const int to_be_selected, const double etol, BCP_vec< BCP_lp_branching_object ∗ > &candidates)

  Select the "close-to-half" variables for strong branching.
- void branch_close_to_one (const BCP_lp_result &lpres, const BCP_vec< BCP_var ∗ > &vars, const int to_be_selected, const double etol, BCP_vec< BCP_lp_branching_object ∗ > &candidates)

  Select the "close-to-one" variables for strong branching.
- void append_branching_vars (const double ∗x, const BCP_vec< BCP_var ∗ > &vars, const BCP_vec< int > &select_pos, BCP_vec< BCP_lp_branching_object ∗ > &candidates)

  This helper method creates branching variable candidates and appends them to `cans`.

Purging the slack pool

- virtual void purge_slack_pool (const BCP_vec< BCP_cut ∗ > &slack_pool, BCP_vec< int > &to_be_purged)

  Selectively purge the list of slack cuts.

4.23.1 Detailed Description

The BCP_lp_user class is the base class from which the user can derive a problem specific class to be used in the LP process.

In that derived class the user can store data to be used in the methods she overrides. Also that is the object the user must return in the USER_initialize::lp_init() method.

There are two kind of methods in the class. The non-virtual methods are helper functions for the built-in defaults, but the user can use them as well. The virtual methods execute steps in the BCP algorithm where the user might want to override the default behavior.

The default implementations fall into three major categories.

- Empty; doesn't do anything and immediately returns (e.g., unpack_module_data()).
- There is no reasonable default, so throw an exception. This happens if the parameter settings drive the flow of in a way that BCP can't perform the necessary function. This behavior is correct since such methods are invoked only if the parameter settings drive the flow of the algorithm that way, in which case the user better implement those methods
- A default is given. Frequently there are multiple defaults and parameters govern which one is selected (e.g., test_feasibility()).

Definition at line 75 of file BCP_lp_user.hpp.

4.23.2 Constructor & Destructor Documentation

4.23.2.1 virtual BCP_lp_user::∼BCP_lp_user ( ) [inline], [virtual]

Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Definition at line 157 of file BCP_lp_user.hpp.

4.23.3 Member Function Documentation

4.23.3.1 bool BCP_lp_user::over_ub (double *lb*) const

Return true / false depending on whether the lb argument is over the current upper bound or not.

4.23.3.2 void BCP_lp_user::select_nonzeros (const double ∗ *first,* const double ∗ *last,* const double *etol,* BCP_vec< int > & *nonzeros*) const

Select all nonzero entries.

Those are considered nonzero that have absolute value greater than `etol`.

4.23.3.3 void BCP_lp_user::select_zeros (const double ∗ *first,* const double ∗ *last,* const double *etol,* BCP_vec< int > & *zeros*) const

Select all zero entries.

Those are considered zero that have absolute value less than `etol`.

4.23.3.4 void BCP_lp_user::select_positives (const double ∗ *first,* const double ∗ *last,* const double *etol,* BCP_vec< int > & *positives*) const

Select all positive entries.

Those are considered positive that have value greater than `etol`.

4.23.3.5 void BCP_lp_user::select_fractions (const double ∗ *first,* const double ∗ *last,* const double *etol,* BCP_vec< int > & *fractions*) const

Select all fractional entries.

Those are considered fractional that are further than `etol` away from any integer value.

4.23.3.6 virtual void BCP_lp_user::unpack_module_data (BCP_buffer & *buf*) [virtual]

Unpack the initial information sent to the LP process by the Tree Manager.

This information was packed by the method BCP_tm_user::pack_module_data() invoked with `BCP_ProcessType-_LP` as the third (target process type) argument.

Default: empty method.

4.23.3.7 virtual void BCP_lp_user::process_message (BCP_buffer & *buf*) [virtual]

Process a message that has been sent by another process' user part to this process' user part.

4.23.3.8 virtual OsiSolverInterface∗ BCP_lp_user::initialize_solver_interface ( ) [virtual]

Create LP solver environment.

Create the LP solver class that will be used for solving the LP relaxations. The default implementation picks up which

`COIN_USE_XXX` is defined and initializes an lp solver of that type. This is probably OK for most users. The only reason to override this method is to be able to choose at runtime which lp solver to instantiate (maybe even different solvers on different processors). In this case she should probably also override the pack_warmstart() and unpack_warmstart() methods in this class and in the BCP_tm_user class.

4.23.3.9 virtual void BCP_lp_user::initialize_int_and_sos_list (std::vector< OsiObject * > & *intAndSosObjects*) [virtual]

Create the list of objects that can be used for branching (simple integer vars and SOS sets).

If nothing is done here then for each search tree node (just before starting to process the node) BCP will scan the variables and the matrix for candidates.

4.23.3.10 virtual void BCP_lp_user::initialize_new_search_tree_node (const BCP_vec< BCP_var * > & *vars*, const BCP_vec< BCP_cut * > & *cuts*, const BCP_vec< BCP_obj_status > & *var_status*, const BCP_vec< BCP_obj_status > & *cut_status*, BCP_vec< int > & *var_changed_pos*, BCP_vec< double > & *var_new_bd*, BCP_vec< int > & *cut_changed_pos*, BCP_vec< double > & *cut_new_bd*) [virtual]

Initializing a new search tree node.

This method serves as hook for the user to do some preprocessing on a search tree node before the node is processed. Also, logical fixing results can be returned in the last four parameters. This might be very useful if the branching implies significant tightening.

Default: empty method.

Parameters

vars	(IN) The variables in the current formulation
cuts	(IN) The cuts in the current formulation
var_status	(IN) The stati of the variables
cut_status	(IN) The stati of the cuts
var_changed_-pos	(OUT) The positions of the variables whose bounds should be tightened
var_new_bd	(OUT) The new lb/ub of those variables
cut_changed_-pos	(OUT) The positions of the cuts whose bounds should be tightened
cut_new_bd	(OUT) The new lb/ub of those cuts

4.23.3.11 virtual void BCP_lp_user::load_problem (OsiSolverInterface & *osi*, BCP_problem_core * *core*, BCP_var_set & *vars*, BCP_cut_set & *cuts*) [virtual]

Load the problem specified by core, vars, and cuts into the solver interface.

If the solver is an LP solver then the default is fine. If it's an NLP then the user has to do this herself.

4.23.3.12 virtual void BCP_lp_user::modify_lp_parameters (OsiSolverInterface * *lp*, const int *changeType*, bool *in_strong_branching*) [virtual]

Modify parameters of the LP solver before optimization.

This method provides an opportunity for the user to change parameters of the LP solver before optimization in the LP solver starts. The second argument indicates what has changed in the LP before this method is called. 0: no change; 1: changes that affect primal feasibility (change in column/row bounds, added cuts); 2: changes that affect dual feasibility (added columns); 3: both. The last argument indicates whether the optimization is a "regular" optimization or it will take place in strong branching.

Default: If 1 or 2 then the appropriate simplex method will be hinted to the solver.

4.23.3.13 virtual void BCP_lp_user::process_lp_result (const **BCP_lp_result** & *lpres,* const **BCP_vec**< **BCP_var** ∗ > & *vars,* const **BCP_vec**< **BCP_cut** ∗ > & *cuts,* const double *old_lower_bound,* double & *true_lower_bound,* **BCP_solution** ∗& *sol,* **BCP_vec**< **BCP_cut** ∗ > & *new_cuts,* **BCP_vec**< **BCP_row** ∗ > & *new_rows,* **BCP_vec**< **BCP_var** ∗ > & *new_vars,* **BCP_vec**< **BCP_col** ∗ > & *new_cols*) `[virtual]`

Process the result of an iteration.

This includes:

- computing a true lower bound on the subproblem.

  In case column generation is done the lower bound for the subproblem might not be the same as the objective value of the current LP relaxation. Here the user has an option to return a true lower bound.
- test feasibility of the solution (or generate a heuristic solution)
- generating cuts and/or variables

The reason for the existence of this method is that (especially when column generation is done) these tasks are so intertwined that it is much easier to execute them in one method instead of in several separate methods.

The default behavior is to do nothing and invoke the individual methods one-by-one.

Parameters

lp_result	the result of the most recent LP optimization (IN)
vars	variables currently in the formulation (IN)
cuts	variables currently in the formulation (IN)
old_lower_bound	the previously known best lower bound (IN)
new_cuts	the vector of generated cuts (OUT)
new_rows	the correspontding rows(OUT)
new_vars	the vector of generated variables (OUT)
new_cols	the correspontding columns(OUT)

4.23.3.14 virtual double BCP_lp_user::compute_lower_bound (const double *old_lower_bound,* const **BCP_lp_result** & *lpres,* const **BCP_vec**< **BCP_var** ∗ > & *vars,* const **BCP_vec**< **BCP_cut** ∗ > & *cuts*) `[virtual]`

Compute a true lower bound for the subproblem.

In case column generation is done the lower bound for the subproblem might not be the same as the objective value of the current LP relaxation. Here the user has an option to return a true lower bound.

The default implementation returns the objective value of the current LP relaxation if no column generation is done, otherwise returns the current (somehow previously computed) true lower bound.

4.23.3.15 virtual **BCP_solution**∗ BCP_lp_user::test_feasibility (const **BCP_lp_result** & *lp_result,* const **BCP_vec**< **BCP_var** ∗ > & *vars,* const **BCP_vec**< **BCP_cut** ∗ > & *cuts*) `[virtual]`

Evaluate and return MIP feasibility of the current solution.

If the solution is MIP feasible, return a solution object otherwise return a NULL pointer. The useris also welcome to heuristically generate a solution and return a pointer to that solution (although the user will have another chance (after cuts and variables are generated) to return/create heuristically generated solutions. (After all, it's quite possible that solutions are generated during cut/variable generation.)

Default: test feasibility based on the `FeeasibilityTest` parameter in BCP_lp_par which defults to `BCP_Full-Test_Feasible`.

Parameters

lp_result	the result of the most recent LP optimization
vars	variables currently in the formulation
cuts	variables currently in the formulation

4.23.3.16 BCP_solution_generic∗ BCP_lp_user::test_binary (const BCP_lp_result & *lpres,* const BCP_vec< BCP_var ∗ > & *vars,* const double *etol*) const

Test whether all variables are 0/1.

Note that this method assumes that all variables are binary, i.e., their original lower/upper bounds are 0/1.

4.23.3.17 BCP_solution_generic∗ BCP_lp_user::test_integral (const BCP_lp_result & *lpres,* const BCP_vec< BCP_var ∗ > & *vars,* const double *etol*) const

Test whether all variables are integer.

Note that this method assumes that all variables are integer.

4.23.3.18 BCP_solution_generic∗ BCP_lp_user::test_full (const BCP_lp_result & *lpres,* const BCP_vec< BCP_var ∗ > & *vars,* const double *etol*) const

Test whether the variables specified as integers are really integer.

4.23.3.19 virtual BCP_solution∗ BCP_lp_user::generate_heuristic_solution (const BCP_lp_result & *lpres,* const BCP_vec< BCP_var ∗ > & *vars,* const BCP_vec< BCP_cut ∗ > & *cuts*) [virtual]

Try to generate a heuristic solution (or return one generated during cut/variable generation.

Return a pointer to the generated solution or return a NULL pointer.

Default: Return a NULL pointer

4.23.3.20 virtual void BCP_lp_user::pack_feasible_solution (BCP_buffer & *buf,* const BCP_solution ∗ *sol*) [virtual]

Pack a MIP feasible solution into a buffer.

The solution will be unpacked in the Tree Manager by the BCP_tm_user::unpack_feasible_solution() method.

Default: The default implementation assumes that `sol` is a BCP_solution_generic object (containing variables at nonzero level) and packs it.

Parameters

buf	(OUT) the buffer to pack into
sol	(IN) the solution to be packed

4.23.3.21 virtual void BCP_lp_user::pack_primal_solution (BCP_buffer & *buf,* const BCP_lp_result & *lp_result,* const BCP_vec< BCP_var ∗ > & *vars,* const BCP_vec< BCP_cut ∗ > & *cuts*) [virtual]

Pack the information necessary for cut generation into the buffer.

Note that the name of the method is pack_primal_solution because most likely that (or some part of that) will be needed for cut generation. However, if the user overrides the method she is free to pack anything (of course she'll have to unpack it in CG).

This information will be sent to the Cut Generator (and possibly to the Cut Pool) where the user has to unpack it. If the user uses the built-in method here, then the built-in method will be used in the Cut Generator as well.

Default: The content of the message depends on the value of the `PrimalSolForCG` parameter in BCP_lp_par. By default the variables at nonzero level are packed.

Parameters

buf	(OUT) the buffer to pack into
lp_result	(IN) the result of the most recent LP optimization
vars	(IN) variables currently in the formulation
cuts	(IN) cuts currently in the formulation

4.23.3.22 virtual void BCP_lp_user::pack_dual_solution (**BCP_buffer** & *buf,* const **BCP_lp_result** & *lp_result,* const **BCP_vec**< **BCP_var** ∗ > & *vars,* const **BCP_vec**< **BCP_cut** ∗ > & *cuts*) `[virtual]`

Pack the information necessary for variable generation into the buffer.

Note that the name of the method is pack_dual_solution because most likely that (or some part of that) will be needed for variable generation. However, if the user overrides the method she is free to pack anything (of course she'll have to unpack it in CG).

This information will be sent to the Variable Generator (and possibly to the Variable Pool) where the user has to unpack it. If the user uses the built-in method here, then the built-in method will be used in the Variable Generator as well.

Default: The content of the message depends on the value of the `DualSolForVG` parameter in BCP_lp_par. By default the full dual solution is packed.

Parameters

buf	(OUT) the buffer to pack into
lp_result	(IN) the result of the most recent LP optimization
vars	(IN) variables currently in the formulation
cuts	(IN) cuts currently in the formulation

4.23.3.23 virtual void BCP_lp_user::display_lp_solution (const **BCP_lp_result** & *lp_result,* const **BCP_vec**< **BCP_var** ∗ > & *vars,* const **BCP_vec**< **BCP_cut** ∗ > & *cuts,* const bool *final_lp_solution*) `[virtual]`

Display the result of most recent LP optimization.

This method is invoked every time an LP relaxation is optimized and the user can display (or not display) it.

Note that this method is invoked only if `final_lp_solution` is true (i.e., no cuts/variables were found) and the `Lp-Verb_FinalRelaxedSolution` parameter of BCP_lp_par is set to true (or alternatively, `final_lp_solution` is false and `LpVerb_RelaxedSolution` is true).

Default: display the solution if the appropriate verbosity code entry is set.

Parameters

lp_result	(IN) the result of the most recent LP optimization
vars	(IN) variables currently in the formulation
final_lp_solution	(IN) whether the lp solution is final or not.

4.23.3.24 virtual void BCP_lp_user::restore_feasibility (const **BCP_lp_result** & *lpres,* const std::vector< double ∗ > *dual_rays,* const **BCP_vec**< **BCP_var** ∗ > & *vars,* const **BCP_vec**< **BCP_cut** ∗ > & *cuts,* **BCP_vec**< **BCP_var** ∗ > & *vars_to_add,* **BCP_vec**< **BCP_col** ∗ > & *cols_to_add*) `[virtual]`

Restoring feasibility.

This method is invoked before fathoming a search tree node that has been found infeasible *and* the variable pricing did not generate any new variables.

4.23.3.25 virtual void BCP_lp_user::cuts_to_rows (const **BCP_vec**< **BCP_var** ∗ > & *vars*, **BCP_vec**< **BCP_cut** ∗ > & *cuts*, **BCP_vec**< **BCP_row** ∗ > & *rows*, const **BCP_lp_result** & *lpres*, BCP_object_origin *origin*, bool *allow_multiple*) `[virtual]`

Convert (and possibly lift) a set of cuts into corresponding rows for the current LP relaxation.

Converting means computing for each cut the coefficients corresponding to each variable and creating BCP_row objects that can be added to the formulation.

This method has different purposes depending on the value of the last argument. If multiple expansion is not allowed then the user must generate a unique row for each cut. This unique row must always be the same for any given cut. This kind of operation is needed so that an LP relaxation can be exactly recreated.

On the other hand if multiple expansion is allowed then the user has (almost) free reign over what she returns. She can delete some of the `cuts` or append new ones (e.g., lifted ones) to the end. The result of the LP relaxation and the origin of the cuts are there to help her to make a decision about what to do. For example, she might want to lift cuts coming from the Cut Generator, but not those coming from the Cut Pool. The only requirement is that when this method returns the number of cuts and rows must be the same and the i-th row must be the unique row corresponding to the i-th cut.

Parameters

vars	the variables currently in the relaxation (IN)
cuts	the cuts to be converted (IN/OUT)
rows	the rows into which the cuts are converted (OUT)
lpres	solution to the current LP relaxation (IN)
origin	where the cuts come from (IN)
allow_multiple	whether multiple expansion, i.e., lifting, is allowed (IN)

Default: throw an exception (if this method is invoked then the user must have generated cuts and BCP has no way to know how to convert them).

4.23.3.26 virtual void BCP_lp_user::vars_to_cols (const **BCP_vec**< **BCP_cut** ∗ > & *cuts*, **BCP_vec**< **BCP_var** ∗ > & *vars*, **BCP_vec**< **BCP_col** ∗ > & *cols*, const **BCP_lp_result** & *lpres*, BCP_object_origin *origin*, bool *allow_multiple*) `[virtual]`

Convert a set of variables into corresponding columns for the current LP relaxation.

Converting means to compute for each variable the coefficients corresponding to each cut and create BCP_col objects that can be added to the formulation.

See the documentation of cuts_to_rows() above for the use of this method (just reverse the role of cuts and variables.)

Parameters

cuts	the cuts currently in the relaxation (IN)
vars	the variables to be converted (IN/OUT)
cols	the colums the variables convert into (OUT)
lpres	solution to the current LP relaxation (IN)
origin	where the do the cuts come from (IN)
allow_multiple	whether multiple expansion, i.e., lifting, is allowed (IN)

Default: throw an exception (if this method is invoked then the user must have generated variables and BCP has no way to know how to convert them).

4.23.3.27 virtual void BCP_lp_user::generate_cuts_in_lp (const BCP_lp_result & *lpres,* const BCP_vec< BCP_var * > & *vars,* const BCP_vec< BCP_cut * > & *cuts,* BCP_vec< BCP_cut * > & *new_cuts,* BCP_vec< BCP_row * > & *new_rows*) `[virtual]`

Generate cuts within the LP process.

Sometimes too much information would need to be transmitted for cut generation (e.g., the full tableau for Gomory cuts) or the cut generation is so fast that transmitting the info would take longer than generating the cuts. In such cases it might better to generate the cuts locally. This routine provides the opportunity.

Default: empty for now. To be interfaced to Cgl.

Parameters

lpres	solution to the current LP relaxation (IN)
vars	the variabless currently in the relaxation (IN)
cuts	the cuts currently in the relaxation (IN)
new_cuts	the vector of generated cuts (OUT)
new_rows	the correspontding rows(OUT)

4.23.3.28 virtual void BCP_lp_user::generate_vars_in_lp (const BCP_lp_result & *lpres,* const BCP_vec< BCP_var * > & *vars,* const BCP_vec< BCP_cut * > & *cuts,* const bool *before_fathom,* BCP_vec< BCP_var * > & *new_vars,* BCP_vec< BCP_col * > & *new_cols*) `[virtual]`

Generate variables within the LP process.

Sometimes too much information would need to be transmitted for variable generation or the variable generation is so fast that transmitting the info would take longer than generating the variables. In such cases it might be better to generate the variables locally. This routine provides the opportunity.

Default: empty method.

Parameters

lpres	solution to the current LP relaxation (IN)
vars	the variabless currently in the relaxation (IN)
cuts	the cuts currently in the relaxation (IN)
before_fathom	if true then BCP is about to fathom the node, so spend some extra effort generating variables if you want to avoid that...
new_vars	the vector of generated variables (OUT)
new_cols	the correspontding columns(OUT)

4.23.3.29 virtual BCP_object_compare_result BCP_lp_user::compare_cuts (const BCP_cut * *c0,* const BCP_cut * *c1*) `[virtual]`

Compare two generated cuts.

Cuts are generated in different iterations, they come from the Cut Pool, etc. There is a very real possibility that the LP process receives several cuts that are either identical or one of them is better then another (cuts off everything the other cuts off). This routine is used to decide which one to keep if not both.

Default: Return `BCP_DifferentObjs`.

4.23.3.30 virtual BCP_object_compare_result BCP_lp_user::compare_vars (const BCP_var * *v0,* const BCP_var * *v1*) `[virtual]`

Compare two generated variables.

Variables are generated in different iterations, they come from the Variable Pool, etc. There is a very real possibility that the LP process receives several variables that are either identical or one of them is better then another (e.g., almost identical but has much lower reduced cost). This routine is used to decide which one to keep if not both.

Default: Return `BCP_DifferentObjs`.

4.23.3.31 virtual void BCP_lp_user::logical_fixing (const BCP_lp_result & *lpres*, const BCP_vec< BCP_var ∗ > & *vars*, const BCP_vec< BCP_cut ∗ > & *cuts*, const BCP_vec< BCP_obj_status > & *var_status*, const BCP_vec< BCP_obj_status > & *cut_status*, const int *var_bound_changes_since_logical_fixing*, BCP_vec< int > & *changed_pos*, BCP_vec< double > & *new_bd*) `[virtual]`

This method provides an opportunity for the user to tighten the bounds of variables.

The method is invoked after reduced cost fixing. The results are returned in the last two parameters.

Default: empty method.

Parameters

lpres	the result of the most recent LP optimization,
vars	the variables in the current formulation,
status	the stati of the variables as known to the system,
var_bound_-changes_since_-logical_fixing	the number of variables whose bounds have changed (by reduced cost fixing) since the most recent invocation of this method that has actually forced changes returned something in the last two arguments,
changed_pos	the positions of the variables whose bounds should be changed
new_bd	the new bounds (lb/ub pairs) of these variables.

4.23.3.32 void BCP_lp_user::reduced_cost_fixing (const double ∗ *dj*, const double ∗ *x*, const double *gap*, BCP_vec< BCP_var ∗ > & *vars*, int & *newly_changed*)

Reduced cost fixing.

This is not exactly a helper function, but the user might want to invoke it...

4.23.3.33 virtual BCP_branching_decision BCP_lp_user::select_branching_candidates (const BCP_lp_result & *lpres*, const BCP_vec< BCP_var ∗ > & *vars*, const BCP_vec< BCP_cut ∗ > & *cuts*, const BCP_lp_var_pool & *local_var_pool*, const BCP_lp_cut_pool & *local_cut_pool*, BCP_vec< BCP_lp_branching_object ∗ > & *cands*, bool *force_branch* = `false`) `[virtual]`

Decide whether to branch or not and select a set of branching candidates if branching is decided upon.

The return value indicates what should be done: branching, continuing with the same node or abandoning the node completely.

Default: Branch if both local pools are empty. If branching is done then several (based on the `StrongBranch_-CloseToHalfNum` and `StrongBranch_CloseToOneNum` parameters in BCP_lp_par) variables are selected for strong branching.

"Close-to-half" variables are those that should be integer and are at a fractional level. The measure of their fractionality is their distance from the closest integer. The most fractional variables will be selected, i.e., those that are close to half. If there are too many such variables then those with higher objective value have priority.

"Close-to-on" is interpreted in a more literal sense. It should be used only if the integer variables are binary as it select those fractional variables which are away from 1 but are still close. If there are too many such variables then those with lower objective value have priority.

Parameters

lpres	the result of the most recent LP optimization.
vars	the variables in the current formulation.
cuts	the cuts in the current formulation.
local_var_pool	the local pool that holds variables with negative reduced cost. In case of continuing with the node the best so many variables will be added to the formulation (those with the most negative reduced cost).
local_cut_pool	the local pool that holds violated cuts. In case of continuing with the node the best so many cuts will be added to the formulation (the most violated ones).
cands	the generated branching candidates.
force_branch	indicate whether to force branching regardless of the size of the local cut/var pools

4.23.3.34 void BCP_lp_user::branch_close_to_half (const **BCP_lp_result** & *lpres,* const **BCP_vec**< **BCP_var** ∗ > & *vars,* const int *to_be_selected,* const double *etol,* **BCP_vec**< **BCP_lp_branching_object** ∗ > & *candidates*)

Select the "close-to-half" variables for strong branching.

Variables that are at least `etol` away from integrality are considered and `to_be_selected` of them will be picked up.

4.23.3.35 void BCP_lp_user::branch_close_to_one (const **BCP_lp_result** & *lpres,* const **BCP_vec**< **BCP_var** ∗ > & *vars,* const int *to_be_selected,* const double *etol,* **BCP_vec**< **BCP_lp_branching_object** ∗ > & *candidates*)

Select the "close-to-one" variables for strong branching.

Variables that are at least `etol` away from integrality are considered and `to_be_selected` of them will be picked up.

4.23.3.36 void BCP_lp_user::append_branching_vars (const double ∗ *x,* const **BCP_vec**< **BCP_var** ∗ > & *vars,* const **BCP_vec**< int > & *select_pos,* **BCP_vec**< **BCP_lp_branching_object** ∗ > & *candidates*)

This helper method creates branching variable candidates and appends them to `cans`.

The indices (in the current formulation) of the variables from which candidates should be created are listed in `select-_pos`.

4.23.3.37 virtual BCP_branching_object_relation BCP_lp_user::compare_branching_candidates (**BCP_presolved_lp_brobj** ∗ *new_solved,* **BCP_presolved_lp_brobj** ∗ *old_solved*) [virtual]

Decide which branching object is preferred for branching.

Based on the member fields of the two presolved candidate branching objects decide which one should be preferred for really branching on it. Possible return values are: `BCP_OldPresolvedIsBetter`, `BCP_NewPresolvedIs-Better` and `BCP_NewPresolvedIsBetter_BranchOnIt`. This last value (besides specifying which candidate is preferred) also indicates that no further candidates should be examined, branching should be done on this candidate.

Default: The behavior of this method is governed by the `BranchingObjectComparison` parameter in BCP_lp_-par.

4.23.3.38 virtual void BCP_lp_user::set_actions_for_children (**BCP_presolved_lp_brobj** ∗ *best*) [virtual]

Decide what to do with the children of the selected branching object.

Fill out the `_child_action` field in `best`. This will specify for every child what to do with it. Possible values for each individual child are `BCP_FathomChild`, `BCP_ReturnChild` and `BCP_KeepChild`. There can be at most child with this last action specified. It means that in case of diving this child will be processed by this LP process as the next search tree node.

Default: Every action is `BCP_ReturnChild`. However, if BCP dives then one child will be mark with `BCP_KeepChild`. The decision which child to keep is based on the `ChildPreference` parameter in BCP_lp_par. Also, if a child has a presolved lower bound that is higher than the current upper bound then that child is mark as `BCP_FathomChild`.

THINK: Should those children be sent back for processing in the next phase?

4.23.3.39 virtual void BCP_lp_user::set_user_data_for_children (**BCP_presolved_lp_brobj** ∗ *best,* const int *selected*) `[virtual]`

For each child create a user data object and put it into the appropriate entry in `best->user_data()`.

When this function is called the `best->user_data()` vector is already the right size and is filled will 0 pointers. The second argument is usefule if strong branching was done. It is the index of the branching candidate that was selected for branching (the one that's the source of `best`.

4.23.3.40 virtual void BCP_lp_user::purge_slack_pool (const **BCP_vec**< **BCP_cut** ∗ > & *slack_pool,* **BCP_vec**< int > & *to_be_purged*) `[virtual]`

Selectively purge the list of slack cuts.

When a cut becomes ineffective and is eventually purged from the LP formulation it is moved into `slack_pool`. The user might consider cuts might later for branching. This function enables the user to purge any cut from the slack pool (those she wouldn't consider anyway). Of course, the user is not restricted to these cuts when branching, this is only there to help to collect slack cuts. The user should put the indices of the cuts to be purged into the provided vector.

Default: Purges the slack cut pool according to the `SlackCutDiscardingStrategy` rule in BCP_lp_par (purge everything before every iteration or before a new search tree node).

Parameters

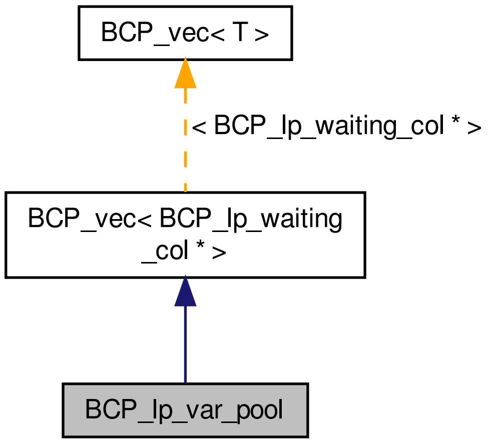
slack_pool	the pool of slacks. (IN)
to_be_purged	the indices of the cuts to be purged. (OUT)

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

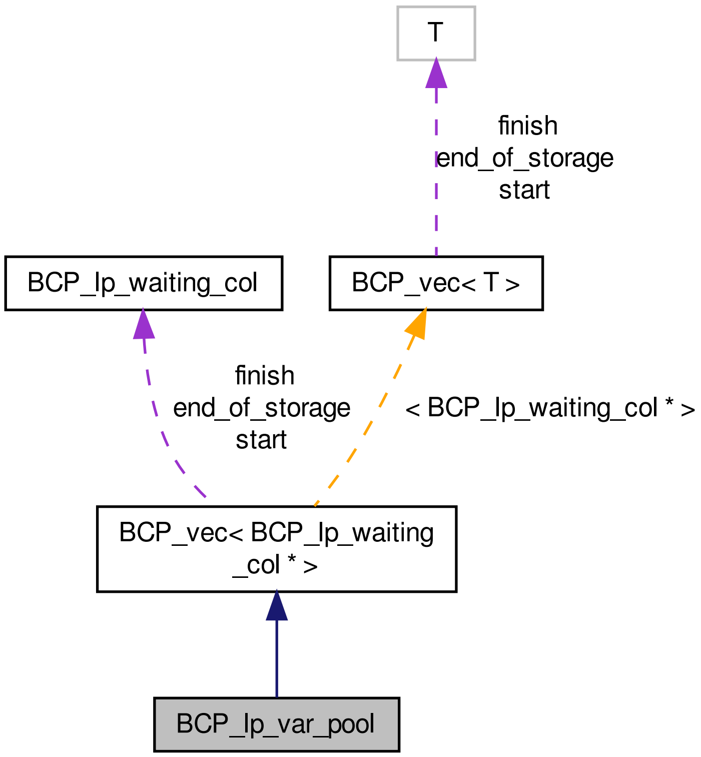
- BCP_lp_user.hpp

4.24 BCP_lp_var_pool Class Reference

Inheritance diagram for BCP_lp_var_pool:



Collaboration diagram for BCP_lp_var_pool:



Additional Inherited Members

4.24.1 Detailed Description

Definition at line 109 of file BCP_lp_pool.hpp.

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

- BCP_lp_pool.hpp

4.25 BCP_lp_waiting_col Class Reference

4.25.1 Detailed Description

Definition at line 78 of file BCP_lp_pool.hpp.

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

- BCP_lp_pool.hpp

4.26 BCP_lp_waiting_row Class Reference

4.26.1 Detailed Description

Definition at line 15 of file BCP_lp_pool.hpp.

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

- BCP_lp_pool.hpp

4.27 BCP_MemPool Class Reference

4.27.1 Detailed Description

Definition at line 8 of file BCP_mempool.hpp.

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

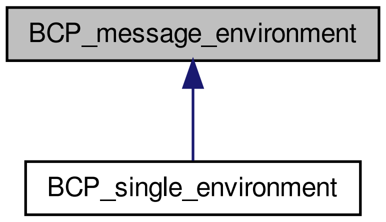
- BCP_mempool.hpp

4.28 BCP_message_environment Class Reference

This is an abstract base class that describes the message passing environment.

`#include <BCP_message.hpp>`

Inheritance diagram for BCP_message_environment:



Public Member Functions

Destructor

- virtual ∼BCP_message_environment ()
  Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Registering a process

- virtual int register_process (USER_initialize ∗user_init)=0

A process can register (receive its process id) with the message passing environment.

Identify parent process

- virtual int parent_process ()=0

  Return the process id of the parent process (the process that spawned the currnet process.

Testing processes

- virtual bool alive (const int pid)=0

  Test if the process given by the argument is alive or not.
- virtual const int ∗ alive (int num, const int ∗pids)=0

  Test if the processes given by the process array in the argument are alive or not.

Send to one process

- virtual void send (const int target, const BCP_message_tag tag)=0

  Send an empty message (message tag only) to the process given by the frist argument.
- virtual void send (const int target, const BCP_message_tag tag, const BCP_buffer &buf)=0

  Send the message in the buffer with the given message tag to the process given by the first argument.

Broadcasting

- virtual void multicast (int num, const int ∗targets, const BCP_message_tag tag)=0

  Send an empty message (message tag only) to all the processes in the process array.
- virtual void multicast (int num, const int ∗targets, const BCP_message_tag tag, const BCP_buffer &buf)=0

  Send the message in the buffer with the given message tag to all processes in the process array.

Receiving

- virtual void receive (const int source, const BCP_message_tag tag, BCP_buffer &buf, const double timeout)=0

  Blocking receive with timeout.
- virtual bool probe (const int source, const BCP_message_tag tag)=0

  Probe if there are any messages from the given process with the given message tag.

Starting a process

- virtual int start_process (const BCP_string &exe, const bool debug)=0

  Spawn a new process.
- virtual int start_process (const BCP_string &exe, const BCP_string &machine, const bool debug)=0

  Spawn a new process on the machine specified by the second argument.
- virtual bool start_processes (const BCP_string &exe, const int proc_num, const bool debug, int ∗ids)=0

  Spawn `proc_num` *processes, all with the same executable.*
- virtual bool start_processes (const BCP_string &exe, const int proc_num, const BCP_vec< BCP_string > &machines, const bool debug, int ∗ids)=0

  Spawn `proc_num` *processes on the machines given by the third argument, all with the same executable.*

Additional function for MPI interface

- virtual int num_procs ()

  Return the number of processes.

4.28.1 Detailed Description

This is an abstract base class that describes the message passing environment.

The implementation of the message passing protocol must implement this class.

All methods are pure virtual, enforcing the correct overriding of the methods.

Definition at line 30 of file BCP_message.hpp.

4.28.2 Constructor & Destructor Documentation

4.28.2.1 virtual BCP_message_environment::∼BCP_message_environment ( ) `[inline], [virtual]`

Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Definition at line 36 of file BCP_message.hpp.

4.28.3 Member Function Documentation

4.28.3.1 virtual int BCP_message_environment::register_process (USER_initialize ∗ *user_init*) `[pure virtual]`

A process can register (receive its process id) with the message passing environment.

Implemented in BCP_single_environment.

4.28.3.2 virtual int BCP_message_environment::parent_process ( ) `[pure virtual]`

Return the process id of the parent process (the process that spawned the currnet process.

Returns null if the process does not have a parent.

Implemented in BCP_single_environment.

4.28.3.3 virtual bool BCP_message_environment::alive (const int *pid*) `[pure virtual]`

Test if the process given by the argument is alive or not.

Return true if alive, false otherwise.

Implemented in BCP_single_environment.

4.28.3.4 virtual const int∗ BCP_message_environment::alive (int *num,* const int ∗ *pids*) `[pure virtual]`

Test if the processes given by the process array in the argument are alive or not.

Return a pointer to the first dead process, or to the end of the array if there are no dead processes.

Implemented in BCP_single_environment.

4.28.3.5 virtual void BCP_message_environment::send (const int *target,* const BCP_message_tag *tag*) `[pure virtual]`

Send an empty message (message tag only) to the process given by the frist argument.

Implemented in BCP_single_environment.

4.28.3.6 virtual void BCP_message_environment::send (const int *target,* const BCP_message_tag *tag,* const BCP_buffer & *buf*) `[pure virtual]`

Send the message in the buffer with the given message tag to the process given by the first argument.

Implemented in BCP_single_environment.

4.28.3.7 virtual void BCP_message_environment::multicast (int *num*, const int ∗ *targets*, const BCP_message_tag *tag*) `[pure virtual]`

Send an empty message (message tag only) to all the processes in the process array.

Implemented in BCP_single_environment.

4.28.3.8 virtual void BCP_message_environment::multicast (int *num*, const int ∗ *targets*, const BCP_message_tag *tag*, const BCP_buffer & *buf*) `[pure virtual]`

Send the message in the buffer with the given message tag to all processes in the process array.

Implemented in BCP_single_environment.

4.28.3.9 virtual void BCP_message_environment::receive (const int *source*, const BCP_message_tag *tag*, BCP_buffer & *buf*, const double *timeout*) `[pure virtual]`

Blocking receive with timeout.

Wait until a message is received from the given process with the given message tag within the given timelimit. If `timeout` is positive then the receive routine times out after this many seconds. If it is negative then the method blocks until a message is received. The received message is saved into the buffer. With a 0 pointer as the first argument (or the predefined `BCP_AnyProcess` process id can be used) a message from any process will be accepted. Messages with any message tag are accepted if the second argument is `BCP_Msg_AnyMessage`.

Implemented in BCP_single_environment.

4.28.3.10 virtual bool BCP_message_environment::probe (const int *source*, const BCP_message_tag *tag*) `[pure virtual]`

Probe if there are any messages from the given process with the given message tag.

Return true if such a message is found, false otherwise. Note that the message is not "read", only its existence is checked. Similarly as above, the wild cards `BCP_AnyProcess` and `BCP_Msg_AnyMessage` can be used.

Implemented in BCP_single_environment.

4.28.3.11 virtual int BCP_message_environment::start_process (const BCP_string & *exe*, const bool *debug*) `[pure virtual]`

Spawn a new process.

The first argument contains the path to the executable to be spawned. If the second argument is set to true, then the executable is spawned under a debugger.

Implemented in BCP_single_environment.

4.28.3.12 virtual int BCP_message_environment::start_process (const BCP_string & *exe*, const BCP_string & *machine*, const bool *debug*) `[pure virtual]`

Spawn a new process on the machine specified by the second argument.

Implemented in BCP_single_environment.

4.28.3.13 virtual bool BCP_message_environment::start_processes (const BCP_string & *exe*, const int *proc_num*, const bool *debug*, int ∗ *ids*) `[pure virtual]`

Spawn `proc_num` processes, all with the same executable.

NOTE: ids must be allocated already and have enough space for `proc_num` entries.

Returns

true/false depending on success.

Implemented in BCP_single_environment.

4.28.3.14 virtual bool BCP_message_environment::start_processes (const BCP_string & *exe*, const int *proc_num*, const BCP_vec< BCP_string > & *machines*, const bool *debug*, int ∗ *ids*) `[pure virtual]`

Spawn `proc_num` processes on the machines given by the third argument, all with the same executable.

NOTE: ids must be allocated already and have enough space for `proc_num` entries.

Returns

true/false depending on success.

Implemented in BCP_single_environment.

4.28.3.15 virtual int BCP_message_environment::num_procs ( ) `[inline]`, `[virtual]`

Return the number of processes.

For non-MPI this just returns 0.

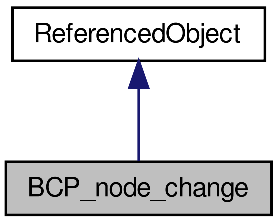
Definition at line 151 of file BCP_message.hpp.

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

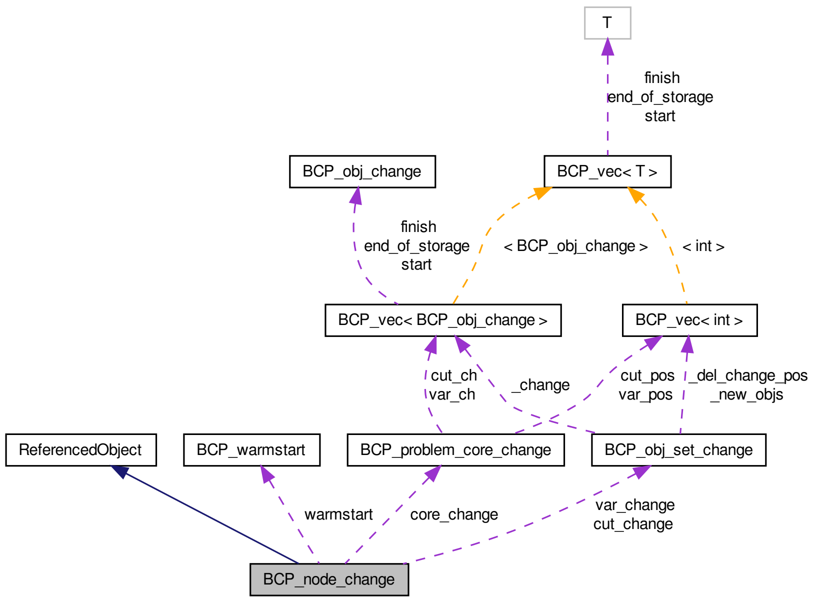
- BCP_message.hpp

4.29 BCP_node_change Class Reference

Inheritance diagram for BCP_node_change:



Collaboration diagram for BCP_node_change:



4.29.1 Detailed Description

Definition at line 19 of file BCP_node_change.hpp.

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

- BCP_node_change.hpp

4.30 BCP_node_storage_in_tm Struct Reference

NO OLD DOC.

```
#include <BCP_lp_node.hpp>
```

4.30.1 Detailed Description

NO OLD DOC.

This structure describes the ways various pieces of the current node's description are stored in the Tree Manager.

Definition at line 25 of file BCP_lp_node.hpp.

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

- BCP_lp_node.hpp

4.31 BCP_obj_change Class Reference

4.31.1 Detailed Description

Definition at line 23 of file BCP_obj_change.hpp.

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

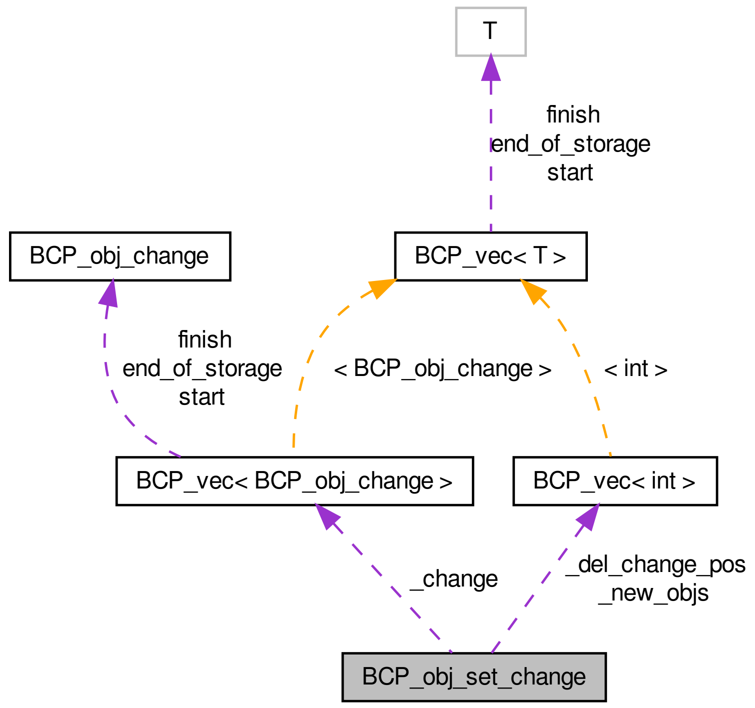
- BCP_obj_change.hpp

4.32 BCP_obj_set_change Class Reference

This class stores data about how an object set (set of vars or set of cuts) changes.

`#include <BCP_obj_change.hpp>`

Collaboration diagram for BCP_obj_set_change:



4.32.1 Detailed Description

This class stores data about how an object set (set of vars or set of cuts) changes.

Definition at line 57 of file BCP_obj_change.hpp.

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

- BCP_obj_change.hpp

4.33 BCP_parameter Class Reference

This parameter indeintifies a single parameter entry.

`#include <BCP_parameters.hpp>`

Public Member Functions

Constructors / Destructor

- BCP_parameter ()
  The default constructor creates a phony parameter.
- BCP_parameter (const BCP_parameter_t t, const int i)
  Constructor where members are specified.
- ~BCP_parameter ()
  The destructor.

Query methods

- BCP_parameter_t type () const
  Return the type of the parameter.
- int index () const
  Return the index of the parameter within all parameters of the same type.

4.33.1 Detailed Description

This parameter indeintifies a single parameter entry.

Definition at line 52 of file BCP_parameters.hpp.

4.33.2 Constructor & Destructor Documentation

4.33.2.1 BCP_parameter::BCP_parameter ( ) `[inline]`

The default constructor creates a phony parameter.

Definition at line 68 of file BCP_parameters.hpp.

4.33.2.2 BCP_parameter::BCP_parameter (const BCP_parameter_t *t*, const int *i*) `[inline]`

Constructor where members are specified.

Definition at line 70 of file BCP_parameters.hpp.

4.33.2.3 BCP_parameter::~BCP_parameter ( ) `[inline]`

The destructor.

Definition at line 73 of file BCP_parameters.hpp.

4.33.3 Member Function Documentation

4.33.3.1 BCP_parameter_t BCP_parameter::type ( ) const [inline]

Return the type of the parameter.

Definition at line 79 of file BCP_parameters.hpp.

4.33.3.2 int BCP_parameter::index ( ) const [inline]

Return the index of the parameter within all parameters of the same type.

Definition at line 82 of file BCP_parameters.hpp.

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

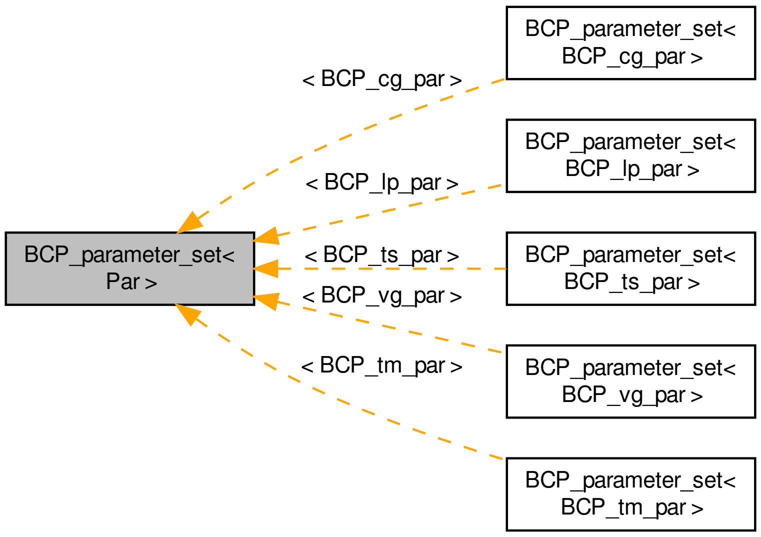
- BCP_parameters.hpp

4.34 BCP_parameter_set< Par > Class Template Reference

This is the class serves as a holder for a set of parameters.

`#include <BCP_parameters.hpp>`

Inheritance diagram for BCP_parameter_set< Par >:


BCP_parameter_set< Par >
< BCP_cg_par >
< BCP_lp_par >
< BCP_ts_par >
< BCP_vg_par >
< BCP_tm_par >
BCP_parameter_set< BCP_cg_par >
BCP_parameter_set< BCP_lp_par >
BCP_parameter_set< BCP_ts_par >
BCP_parameter_set< BCP_vg_par >
BCP_parameter_set< BCP_tm_par >

Public Member Functions

Query methods

The members of the parameter set can be queried for using the overloaded entry() method.

Using the example in the class documentation the user can get a parameter with the "<code>param.entry(USER-_par::parameter_name)</code>" expression.

- char **entry** (const chr_params key) const
- int **entry** (const int_params key) const
- double **entry** (const dbl_params key) const
- const BCP_string & **entry** (const str_params key) const
- const BCP_vec< BCP_string > & **entry** (const str_array_params key) const

Set methods

First, there is the assignment operator that sets the whole parameter set at once.

Individual members of the parameter set can be set for using the overloaded set_entry() method. Using the example in the class documentation the user can set a parameter with the "<code>param.set_entry(USER_par::parameter-_name, param_value)</code>" expression.

- BCP_parameter_set< Par > & **operator=** (const BCP_parameter_set< Par > &x)
- void **set_entry** (const chr_params key, const char val)
- void **set_entry** (const chr_params key, const bool val)
- void **set_entry** (const int_params key, const int val)
- void **set_entry** (const dbl_params key, const double val)
- void **set_entry** (const str_params key, const char ∗val)
- void **set_entry** (const str_array_params key, const char ∗val)
- void **set_entry** (const BCP_parameter key, const char ∗val)

Read parameters from a stream.

- void read_from_stream (std::istream &parstream)
  Read the parameters from the stream specified in the argument.

Expand parameter value (look for environment vars).

- std::string **expand** (const char ∗value)

Read parameters from a file.

- void read_from_file (const char ∗paramfile)
  Simply invoke reading from a stream.

Read parameters from the command line

- void read_from_arglist (const int argnum, const char ∗const ∗arglist)
  Simply invoke reading from a stream.

Write parameters to a stream.

- void write_to_stream (std::ostream &outstream) const
  Write keyword-value pairs to the stream specified in the argument.

Packing/unpacking methods

- void pack (BCP_buffer &buf)
  Pack the parameter set into the buffer.
- void unpack (BCP_buffer &buf)
  Unpack the parameter set from the buffer.

Constructors and destructor.

- BCP_parameter_set ()
  The default constructor creates a parameter set with from the template argument structure.
- ∼BCP_parameter_set ()
  The destructor deletes all data members.

4.34.1 Detailed Description

template<class Par>class BCP_parameter_set< Par >

This is the class serves as a holder for a set of parameters.

For example, BCP stores has a parameter set for each process. Of course, the user can use this class for her own parameters. To use this class the user must

1. first create a `struct` with the names of the parameters (see, e.g., BCP_tm_par. Assume that the structure is named `USER_par`.
2. define the member functions `BCP_parameter_set<USER_par>::create_keyword_list()` and `BCP_parameter_set<USER_par>::set_default_entries()`. For an example look at the file `BC-P-common/TM/BCP_tm_param.cpp`. Essentially, the first method defines what keywords should be looked for in the parameter file, and if one is found which parameter should take the corresponding value; the other method specifies the default values for each parameter.
3. in her source code declare a variable "<code>BCP_parameter_set<USER_par> param;</code>"

After this the user can read in the parameters from a file, she can set/access the parameters in the parameter, etc.

Definition at line 113 of file BCP_parameters.hpp.

4.34.2 Constructor & Destructor Documentation

4.34.2.1 template<class Par> **BCP_parameter_set**< Par >::**BCP_parameter_set** ( ) `[inline]`

The default constructor creates a parameter set with from the template argument structure.

The keyword list is created and the defaults are set.

Definition at line 564 of file BCP_parameters.hpp.

4.34.2.2 template<class Par> **BCP_parameter_set**< Par >::∼**BCP_parameter_set** ( ) `[inline]`

The destructor deletes all data members.

Definition at line 576 of file BCP_parameters.hpp.

4.34.3 Member Function Documentation

4.34.3.1 template<class Par> void **BCP_parameter_set**< Par >::read_from_stream (std::istream & *parstream*) `[inline]`

Read the parameters from the stream specified in the argument.

The stream is interpreted as a lines separated by newline characters. The first word on each line is tested for match with the keywords specified in the create_keyword_list() method. If there is a match then the second word will be interpreted as the value for the corresponding parameter. Any further words on the line are discarded. Every non-matching line is discarded.

If the keyword corresponds to a non-array parameter then the new value simply overwrites the old one. Otherwise, i.e., if it is a StringArrayPar, the value is appended to the list of strings in that array.

Definition at line 262 of file BCP_parameters.hpp.

4.34.3.2 template<class Par> void **BCP_parameter_set**< Par >::read_from_file (const char * *paramfile*) [inline]

Simply invoke reading from a stream.

Definition at line 441 of file BCP_parameters.hpp.

4.34.3.3 template<class Par> void **BCP_parameter_set**< Par >::read_from_arglist (const int *argnum*, const char *const * *arglist*) [inline]

Simply invoke reading from a stream.

Definition at line 454 of file BCP_parameters.hpp.

4.34.3.4 template<class Par> void **BCP_parameter_set**< Par >::write_to_stream (std::ostream & *outstream*) const [inline]

Write keyword-value pairs to the stream specified in the argument.

Each keyword-value pair is separated by a newline character.

Definition at line 476 of file BCP_parameters.hpp.

4.34.3.5 template<class Par> void **BCP_parameter_set**< Par >::pack (**BCP_buffer** & *buf*) [inline]

Pack the parameter set into the buffer.

Definition at line 522 of file BCP_parameters.hpp.

4.34.3.6 template<class Par> void **BCP_parameter_set**< Par >::unpack (**BCP_buffer** & *buf*) [inline]

Unpack the parameter set from the buffer.

Definition at line 535 of file BCP_parameters.hpp.

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

- BCP_parameters.hpp

4.35 BCP_presolved_lp_brobj Class Reference

A presolved branching object candidate.

`#include <BCP_lp_branch.hpp>`

Public Member Functions

Constructor and destructor

- BCP_presolved_lp_brobj (BCP_lp_branching_object *candidate)
  The only one way to construct a presolved branching object is to create it from a regular branching object.
- ~BCP_presolved_lp_brobj ()
  The destructor simply deletes every member (deletes every lpres in the vector).

Query methods

- BCP_lp_branching_object * candidate ()
  Return a pointer to the candidate.
- const BCP_lp_branching_object * candidate () const

Return a const pointer to the candidate.
- const BCP_lp_result & lpres (const int child_ind) const
  Return a const reference to the presolved results of the `child_ind`-th child.
- BCP_vec< BCP_child_action > & action ()
  Return a reference to the actions to be taken.
- const BCP_vec< BCP_child_action > & action () const
  Return a const reference to the actions to be taken.
- BCP_vec< BCP_user_data ∗ > & user_data ()
  Return a reference to the user data vector.
- const BCP_vec< BCP_user_data ∗ > & user_data () const
  Return a const reference to the user data vector.
- bool fathomable (const double objval_limit) const
  Return true if every children can be fathomed.
- bool had_numerical_problems () const
  Return true if at least one child had numerical difficulties while presolving.

Modifying methods

- void **initialize_lower_bound** (const double val)
- void **keep_no_child** ()
- bool **is_pruned** () const
- void get_lower_bounds (BCP_vec< double > &obj)
  Fill up `obj` with the lower bound on each child.
- void set_lower_bounds (const BCP_vec< double > &obj)
  Fill up the lower bounds on the children with the content of `obj`.
- void get_results (OsiSolverInterface &lp, const int child_ind)
  Extract the lp results from the LP solver for the `child_ind`-th child.
- void fake_objective_values (const double itlim_objval)
  Examine the termination codes for the children and for those that do not have a valid lower bound fake the objective value depending on the termination code:
- void set_objective_values (const BCP_vec< double > &obj, const BCP_vec< int > &termcode, const double itlim_objval)
  Set the appropriate fields of all _lpres to the given termcode and objval if the termcode is "normal".
- void swap (BCP_presolved_lp_brobj &rhs)
  swap the two presolved branching object
- BCP_vec< BCP_vec< BCP_cut ∗ > > & **get_new_cuts** ()
- BCP_vec< BCP_vec< BCP_row ∗ > > & **get_new_rows** ()

4.35.1 Detailed Description

A presolved branching object candidate.

During strong branching all children of a branching object (of class [`BCP_lp_branching_object`]{BCP_lp_-branching_object.html}) are *presolved* to gain some insight how good it would be to branch on that particular branching object. The results of the presolved children are stored in an object of this type as well as the instructions what to do with each child in case this object is selected for branching.

Definition at line 321 of file BCP_lp_branch.hpp.

4.35.2 Constructor & Destructor Documentation

4.35.2.1 BCP_presolved_lp_brobj::BCP_presolved_lp_brobj (**BCP_lp_branching_object** ∗ *candidate*) [inline], [explicit]

The only one way to construct a presolved branching object is to create it from a regular branching object.

Definition at line 358 of file BCP_lp_branch.hpp.

4.35.2.2 BCP_presolved_lp_brobj::~BCP_presolved_lp_brobj ( ) [inline]

The destructor simply deletes every member (deletes every lpres in the vector).

Definition at line 373 of file BCP_lp_branch.hpp.

4.35.3 Member Function Documentation

4.35.3.1 BCP_lp_branching_object* BCP_presolved_lp_brobj::candidate ( ) [inline]

Return a pointer to the candidate.

THINK : is it necessary to have a non-const version???

Definition at line 388 of file BCP_lp_branch.hpp.

4.35.3.2 const BCP_lp_branching_object* BCP_presolved_lp_brobj::candidate ( ) const [inline]

Return a const pointer to the candidate.

Definition at line 392 of file BCP_lp_branch.hpp.

4.35.3.3 const BCP_lp_result& BCP_presolved_lp_brobj::lpres (const int *child_ind*) const [inline]

Return a const reference to the presolved results of the `child_ind`-th child.

Definition at line 397 of file BCP_lp_branch.hpp.

4.35.3.4 BCP_vec<BCP_child_action>& BCP_presolved_lp_brobj::action ( ) [inline]

Return a reference to the actions to be taken.

A non-const method is needed, since this is the easiest way to set the entries. Maybe it'd be cleaner to have a separate set method...

Definition at line 403 of file BCP_lp_branch.hpp.

4.35.3.5 const BCP_vec<BCP_child_action>& BCP_presolved_lp_brobj::action ( ) const [inline]

Return a const reference to the actions to be taken.

Definition at line 407 of file BCP_lp_branch.hpp.

4.35.3.6 BCP_vec<BCP_user_data*>& BCP_presolved_lp_brobj::user_data ( ) [inline]

Return a reference to the user data vector.

A non-const method is needed, since this is the easiest way to set the entries. Maybe it'd be cleaner to have a separate set method...

Definition at line 414 of file BCP_lp_branch.hpp.

4.35.3.7 const BCP_vec<BCP_user_data*>& BCP_presolved_lp_brobj::user_data ( ) const [inline]

Return a const reference to the user data vector.

Definition at line 418 of file BCP_lp_branch.hpp.

4.35.3.8 bool BCP_presolved_lp_brobj::fathomable (const double *objval_limit*) const

Return true if every children can be fathomed.

(The lower bound for each is above `objval_limit`.)

4.35.3.9 bool BCP_presolved_lp_brobj::had_numerical_problems ( ) const

Return true if at least one child had numerical difficulties while presolving.

4.35.3.10 void BCP_presolved_lp_brobj::get_lower_bounds (**BCP_vec**< double > & *obj*) `[inline]`

Fill up `obj` with the lower bound on each child.

Definition at line 450 of file BCP_lp_branch.hpp.

4.35.3.11 void BCP_presolved_lp_brobj::set_lower_bounds (const **BCP_vec**< double > & *obj*) `[inline]`

Fill up the lower bounds on the children with the content of `obj`.

Definition at line 459 of file BCP_lp_branch.hpp.

4.35.3.12 void BCP_presolved_lp_brobj::get_results (OsiSolverInterface & *lp*, const int *child_ind*) `[inline]`

Extract the lp results from the LP solver for the `child_ind`-th child.

This is done immediately after presolving the child.

Definition at line 467 of file BCP_lp_branch.hpp.

4.35.3.13 void BCP_presolved_lp_brobj::fake_objective_values (const double *itlim_objval*)

Examine the termination codes for the children and for those that do not have a valid lower bound fake the objective value depending on the termination code:

- primal infeasibility / dual objective limit: `BCP_DBL_MAX`
- iteration limit : maximum of the lower bound at termination and `itlim_objval`
- abandoned : `itlim_objval`

4.35.3.14 void BCP_presolved_lp_brobj::set_objective_values (const **BCP_vec**< double > & *obj*, const **BCP_vec**< int > & *termcode*, const double *itlim_objval*)

Set the appropriate fields of all _lpres to the given termcode and objval if the termcode is "normal".

If not normal (like the cases in fake_objective_values(), then apply the same rules.

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

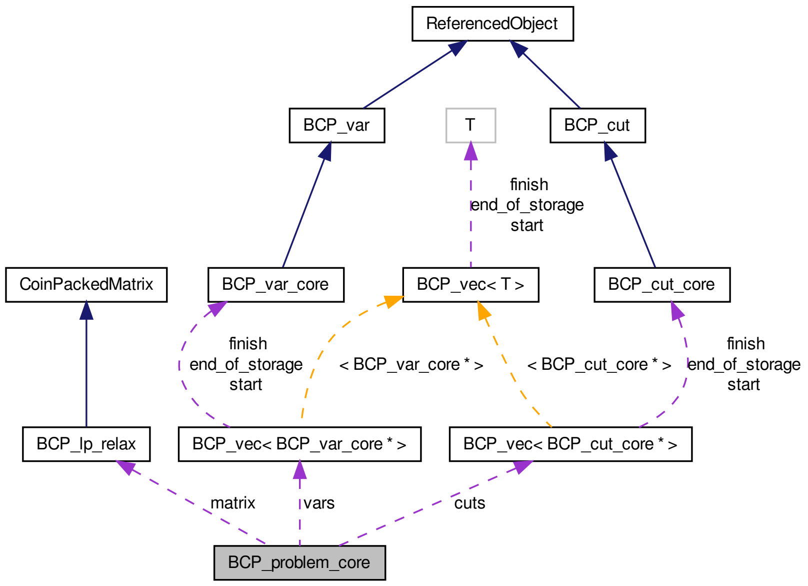
- BCP_lp_branch.hpp

4.36 BCP_problem_core Class Reference

This class describes the core of the MIP problem, the variables/cuts in it as well as the matrix corresponding the core variables and cuts.

```
#include <BCP_problem_core.hpp>
```

Collaboration diagram for BCP_problem_core:



Public Member Functions

Constructors and destructor

- BCP_problem_core ()

  The default constructor creates an empty core description: no variables/cuts and an empty matrix.
- BCP_problem_core (BCP_vec< BCP_var_core ∗ > &v, BCP_vec< BCP_cut_core ∗ > &c, BCP_lp_relax ∗&m)

  This constructor "takes over" the arguments.
- ∼BCP_problem_core ()

  The desctructor deletes all data members.

Query methods

- size_t varnum () const

  Return the number of variables in the core.
- size_t cutnum () const

  Return the number of cuts in the core.

Packing and unpacking methods

- void pack (BCP_buffer &buf) const

  Pack the contents of the core description into the buffer.
- void unpack (BCP_buffer &buf)

  Unpack the contents of the core description from the buffer.

Public Attributes

Data members

- BCP_vec< BCP_var_core ∗ > vars

  A vector of pointers to the variables in the core of the problem.
- BCP_vec< BCP_cut_core ∗ > cuts

  A vector of pointers to the cuts in the core of the problem.
- BCP_lp_relax ∗ matrix

  A pointer to the constraint matrix corresponding to the core variables and cuts.

4.36.1 Detailed Description

This class describes the core of the MIP problem, the variables/cuts in it as well as the matrix corresponding the core variables and cuts.

Core cuts and variables never leave the formulation.

Definition at line 31 of file BCP_problem_core.hpp.

4.36.2 Constructor & Destructor Documentation

4.36.2.1 BCP_problem_core::BCP_problem_core ( )

The default constructor creates an empty core description: no variables/cuts and an empty matrix.

4.36.2.2 BCP_problem_core::BCP_problem_core (**BCP_vec< BCP_var_core ∗ > &** *v*, **BCP_vec< BCP_cut_core ∗ > &** *c*, **BCP_lp_relax ∗&** *m*) [inline]

This constructor "takes over" the arguments.

The created core description will have the content of the arguments in its data members while the arguments lose their content.

Definition at line 65 of file BCP_problem_core.hpp.

4.36.2.3 BCP_problem_core::∼BCP_problem_core ( )

The desctructor deletes all data members.

4.36.3 Member Function Documentation

4.36.3.1 size_t BCP_problem_core::varnum ( ) const [inline]

Return the number of variables in the core.

Definition at line 78 of file BCP_problem_core.hpp.

4.36.3.2 size_t BCP_problem_core::cutnum ( ) const [inline]

Return the number of cuts in the core.

Definition at line 80 of file BCP_problem_core.hpp.

4.36.3.3 void BCP_problem_core::pack (**BCP_buffer &** *buf*) const

Pack the contents of the core description into the buffer.

4.36.3.4 void BCP_problem_core::unpack (**BCP_buffer** & *buf*)

Unpack the contents of the core description from the buffer.

4.36.4 Member Data Documentation

4.36.4.1 **BCP_vec<BCP_var_core*>** BCP_problem_core::vars

A vector of pointers to the variables in the core of the problem.

These are the variables that always stay in the problem formulation.

Definition at line 48 of file BCP_problem_core.hpp.

4.36.4.2 **BCP_vec<BCP_cut_core*>** BCP_problem_core::cuts

A vector of pointers to the cuts in the core of the problem.

These are the cuts that always stay in the problem formulation.

Definition at line 51 of file BCP_problem_core.hpp.

4.36.4.3 **BCP_lp_relax*** BCP_problem_core::matrix

A pointer to the constraint matrix corresponding to the core variables and cuts.

Definition at line 54 of file BCP_problem_core.hpp.

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

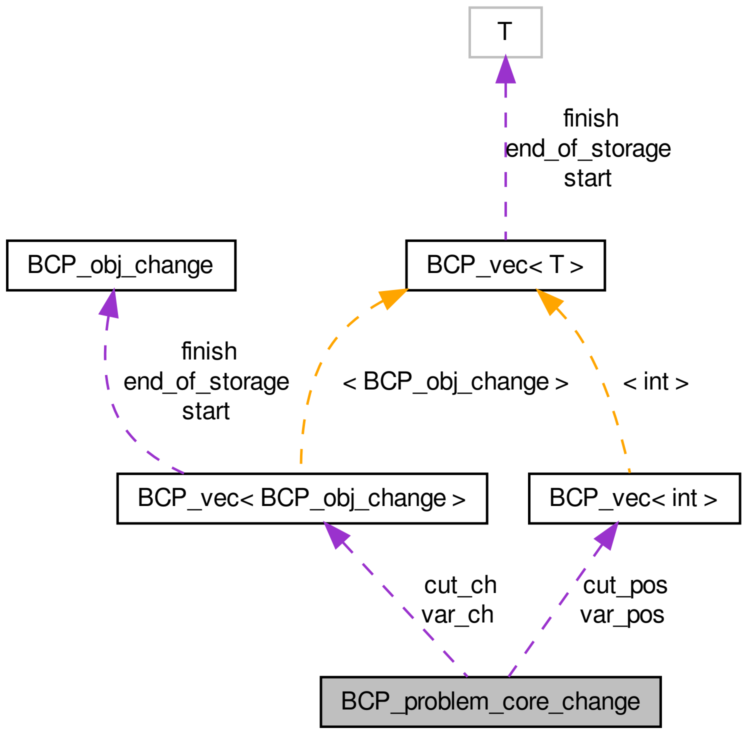
- BCP_problem_core.hpp

4.37 BCP_problem_core_change Class Reference

This class describes changes in the core of the problem.

```
#include <BCP_problem_core.hpp>
```

Collaboration diagram for BCP_problem_core_change:



Public Member Functions

Constructors and destructor

- BCP_problem_core_change (BCP_storage_t store=BCP_Storage_WrtCore)

  This constructor creates a core change with the given storage.
- BCP_problem_core_change (int bvarnum, BCP_var_set &vars, int bcutnum, BCP_cut_set &cuts)

  This constructor creates an Explicit core change description.
- BCP_problem_core_change (BCP_storage_t storage, BCP_problem_core_change &ocore, BCP_problem_-core_change &ncore)

  Create a core change describing the changes from `old_bc</node> to new_bc.`
- ∼BCP_problem_core_change ()

  The destructor deletes all data members.

Query methods

- BCP_storage_t storage () const

  Return the storage type.
- size_t varnum () const

  Return the number of changed variables (the length of the array `var_ch`).
- size_t cutnum () const

Return the number of changed cuts (the length of the array `cut_ch`).

Modifying methods

- BCP_problem_core_change & operator= (const BCP_problem_core &core)

  Set the core change description to be an explicit description (in the form of a change) of the given `core`.
- void ensure_explicit (const BCP_problem_core_change &expl_core)

  If the current storage is not already explicit then replace it with an explicit description of the core generated by applying the currently stored changes to `expl_core` (which of course, must be explicitly stored).
- void make_wrtcore_if_shorter (const BCP_problem_core_change &orig_core)

  Replace the current explicitly stored core change with one stored with respect to the explicitly stored original core change in `orig_core` if the WrtCore description is shorter (requires less space to pack into a buffer).
- void swap (BCP_problem_core_change &other)

  Swap the contents of the current core change with that of `other`.
- void update (const BCP_problem_core_change &expl_core, const BCP_problem_core_change &core_-change)

  Update the current change according to `core_change`.

Packing and unpacking

- int pack_size () const

  Return the buffer size needed to pack the data in the core change.
- void pack (BCP_buffer &buf) const

  Pack the core change into the buffer.
- void unpack (BCP_buffer &buf)

  Unpack the core change data from the buffer.

Public Attributes

Data members

- BCP_storage_t _storage

  Describes how the change is stored.
- BCP_vec< int > var_pos

  The positions of the core variables (in the `vars` member of [`BCP_problem_core`]{BCP_problem_core.html}) whose bounds and/or stati have changed.
- BCP_vec< BCP_obj_change > var_ch

  The new lb/ub/status triplet for each variable for which any of those three have changed.
- BCP_vec< int > cut_pos

  The positions of the core cuts (in the `cuts` member of [`BCP_problem_core`]{BCP_problem_core.html}) whose bounds and/or stati have changed.
- BCP_vec< BCP_obj_change > cut_ch

  The new lb/ub/status triplet for each cut for which any of those three have changed.

4.37.1 Detailed Description

This class describes changes in the core of the problem.

While the set of core variables and cuts always remains the same, their bounds and stati may change during processing. An object of this type may store the description of the bounds and stati of the core of three possible ways:

- explicitly: the bounds and stati of everything in the core is stored in this class. In this case `..._pos` is empty and `..._ch` stores the bounds and stati.

- with respect to parent: this is the case when a sequence of core changes have to be applied to get the current state. Each core change in the sequence describes the changes to be done after the previous change has been applied.
- with respect to core: the current state of the core is given as one set of changes with respect to the very original state of the core.

This class is internal to the framework, the user shouldn't worry about it.

Definition at line 116 of file BCP_problem_core.hpp.

4.37.2 Constructor & Destructor Documentation

4.37.2.1 BCP_problem_core_change::BCP_problem_core_change (BCP_storage_t *store* = `BCP_Storage_WrtCore`) `[inline]`

This constructor creates a core change with the given storage.

The other members are empty, i.e., the created object contains NoData; is Explicit and empty; is WrtCore or WrtParent but there's no change.

Note that the argument defaults to WrtCore, this constructor is the default constructor as well, and as the default constructor it creates a "no change WrtCore" core change.

Definition at line 169 of file BCP_problem_core.hpp.

4.37.2.2 BCP_problem_core_change::BCP_problem_core_change (int *bvarnum*, BCP_var_set & *vars*, int *bcutnum*, BCP_cut_set & *cuts*)

This constructor creates an Explicit core change description.

The first `bvarnum` variables in `vars` are the core variables. Similarly for the cuts.

4.37.2.3 BCP_problem_core_change::BCP_problem_core_change (BCP_storage_t *storage*, BCP_problem_core_change & *ocore*, BCP_problem_core_change & *ncore*)

Create a core change describing the changes from `old_bc</node> to new_bc.`

Both core change arguments must have explicit storage. The only reason for passing `storage` as an argument (and not setting it automatically to WrtParent) is that `old_bc` might be the original core in which case storage must be set to WrtCore.

4.37.2.4 BCP_problem_core_change::~BCP_problem_core_change ( ) `[inline]`

The destructor deletes all data members.

Definition at line 189 of file BCP_problem_core.hpp.

4.37.3 Member Function Documentation

4.37.3.1 BCP_storage_t BCP_problem_core_change::storage ( ) const `[inline]`

Return the storage type.

Definition at line 198 of file BCP_problem_core.hpp.

4.37.3.2 size_t BCP_problem_core_change::varnum ( ) const `[inline]`

Return the number of changed variables (the length of the array `var_ch`).

Definition at line 201 of file BCP_problem_core.hpp.

4.37.3.3 size_t BCP_problem_core_change::cutnum ( ) const [inline]

Return the number of changed cuts (the length of the array `cut_ch`).

Definition at line 204 of file BCP_problem_core.hpp.

4.37.3.4 BCP_problem_core_change& BCP_problem_core_change::operator= (const BCP_problem_core & *core*)

Set the core change description to be an explicit description (in the form of a change) of the given `core`.

4.37.3.5 void BCP_problem_core_change::ensure_explicit (const BCP_problem_core_change & *expl_core*)

If the current storage is not already explicit then replace it with an explicit description of the core generated by applying the currently stored changes to `expl_core` (which of course, must be explicitly stored).

NOTE: An exception is thrown if the currently stored change is not stored as explicit or WrtCore; or the storage of `expl_core` is not explicit.

4.37.3.6 void BCP_problem_core_change::make_wrtcore_if_shorter (const BCP_problem_core_change & *orig_core*)

Replace the current explicitly stored core change with one stored with respect to the explicitly stored original core change in `orig_core` if the WrtCore description is shorter (requires less space to pack into a buffer).

NOTE: An exception is thrown if either the current storage or that of `expl_core` is not explicit.

4.37.3.7 void BCP_problem_core_change::swap (BCP_problem_core_change & *other*)

Swap the contents of the current core change with that of `other`.

4.37.3.8 void BCP_problem_core_change::update (const BCP_problem_core_change & *expl_core,* const BCP_problem_core_change & *core_change*)

Update the current change according to `core_change`.

If the storage type of `core_change` is

- NoData or Explicit then it is simply copied over into this change.
- WrtParent then the current change is supposed to be the parent and this explicitly stored core change will be updated with the changes in `core_change` (an exception is thrown if the current change is not explicitly stored).
- WrtCore storage then the current change will be replaced by `expl_core` updated with `core_change` (the storage of `expl_core` must be explicit or an exception is thrown).

NOTE: When this function exits the stored change will have either explicit or NoData storage.

4.37.3.9 int BCP_problem_core_change::pack_size ( ) const

Return the buffer size needed to pack the data in the core change.

4.37.3.10 void BCP_problem_core_change::pack (BCP_buffer & *buf*) const

Pack the core change into the buffer.

4.37.3.11 void BCP_problem_core_change::unpack (BCP_buffer & *buf*)

Unpack the core change data from the buffer.

4.37.4 Member Data Documentation

4.37.4.1 BCP_storage_t BCP_problem_core_change::_storage

Describes how the change is stored.

The interpretation of the other data members depends on the storage type.

- `BCP_Storage_NoData`: when no data is stored (i.e., the change is not described yet) the other members are undefined.
- `BCP_Storage_WrtCore`: with respect to core storage (as explained in the class description).
- `BCP_Storage_WrtParent`: with respect to parent storage (as explained in the class description).
- `BCP_Storage_Explicit`: Explicit storage (as explained in the class description).

Definition at line 143 of file BCP_problem_core.hpp.

4.37.4.2 BCP_vec<int> BCP_problem_core_change::var_pos

The positions of the core variables (in the `vars` member of [`BCP_problem_core`]{BCP_problem_core.html}) whose bounds and/or stati have changed.

Definition at line 147 of file BCP_problem_core.hpp.

4.37.4.3 BCP_vec<BCP_obj_change> BCP_problem_core_change::var_ch

The new lb/ub/status triplet for each variable for which any of those three have changed.

Definition at line 150 of file BCP_problem_core.hpp.

4.37.4.4 BCP_vec<int> BCP_problem_core_change::cut_pos

The positions of the core cuts (in the `cuts` member of [`BCP_problem_core`]{BCP_problem_core.html}) whose bounds and/or stati have changed.

Definition at line 154 of file BCP_problem_core.hpp.

4.37.4.5 BCP_vec<BCP_obj_change> BCP_problem_core_change::cut_ch

The new lb/ub/status triplet for each cut for which any of those three have changed.

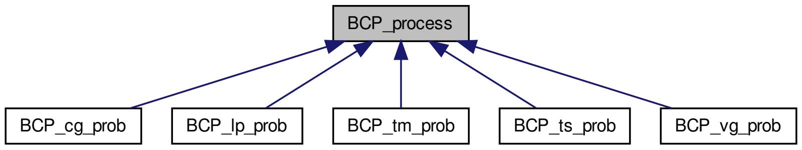
Definition at line 157 of file BCP_problem_core.hpp.

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

- BCP_problem_core.hpp

4.38 BCP_process Class Reference

Inheritance diagram for BCP_process:



4.38.1 Detailed Description

Definition at line 16 of file BCP_process.hpp.

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

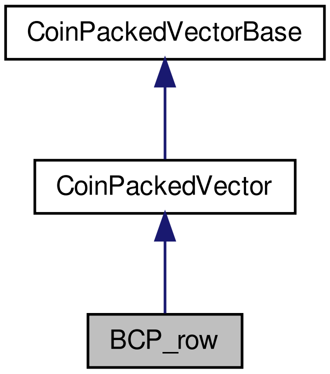
- BCP_process.hpp

4.39 BCP_row Class Reference

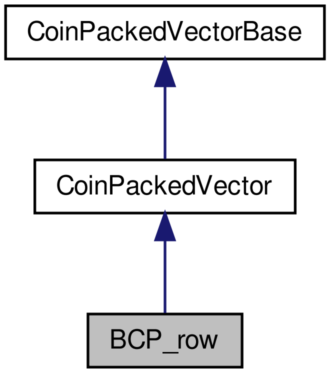
This class holds a row in a compressed form.

`#include <BCP_matrix.hpp>`

Inheritance diagram for BCP_row:



Collaboration diagram for BCP_row:



Public Member Functions

Query methods

- double LowerBound () const

  Return the lower bound.
- double UpperBound () const

  Return the upper bound.

General modifying methods

- void LowerBound (double lb)

  Set the lower bound to the given value.
- void UpperBound (double ub)

  Set the upper bound to the given value.
- BCP_row & operator= (const BCP_row &x)

  Assignment operator: copy over the contents of x.
- void assign (const int size, int ∗&ElementIndices, double ∗&ElementValues, const double LB, const double UB)

  Set the lower and upper bounds to the given values.
- void copy (const int size, const int ∗ElementIndices, const double ∗ElementValues, const double LB, const double UB)

  Copy the arguments into the appropriate data members.
- void copy (BCP_vec< int >::const_iterator firstind, BCP_vec< int >::const_iterator lastind, BCP_vec< double >::const_iterator firstval, BCP_vec< double >::const_iterator lastval, const double LB, const double UB)

  Same as the other `copy()` method, except that instead of using vectors the indices (values) are given in `[firstind,lastind)` (`[firstval,lastval)`).

Constructors / Destructor

- BCP_row ()

The default constructor creates an empty row with -infinity as lower and +infinity as upper bound.
- BCP_row (const BCP_row &x)

  The copy constructor makes a copy of `x`.
- BCP_row (BCP_vec< int >::const_iterator firstind, BCP_vec< int >::const_iterator lastind, BCP_vec< double >::const_iterator firstval, BCP_vec< double >::const_iterator lastval, const double LB, const double UB)

  This constructor acts exactly like the `copy` method with the same argument list.
- BCP_row (const int size, int ∗&ElementIndices, double ∗&ElementValues, const double LB, const double UB)

  This constructor acts exactly like the `assign` method with the same argument list.
- **BCP_row** (const **CoinPackedVectorBase** &vec, const double LB, const double UB)
- ∼BCP_row ()

  The destructor deletes all data members.

Protected Attributes

Data members

- double _LowerBound

  The lower bound corresponding to the row.
- double _UpperBound

  The upper bound corresponding to the row.

4.39.1 Detailed Description

This class holds a row in a compressed form.

That is, it is a packed vector with a lower and upper bound.

Definition at line 152 of file BCP_matrix.hpp.

4.39.2 Constructor & Destructor Documentation

4.39.2.1 BCP_row::BCP_row ( ) [inline]

The default constructor creates an empty row with -infinity as lower and +infinity as upper bound.

Definition at line 228 of file BCP_matrix.hpp.

4.39.2.2 BCP_row::BCP_row (const BCP_row & *x*) [inline]

The copy constructor makes a copy of `x`.

Definition at line 231 of file BCP_matrix.hpp.

4.39.2.3 BCP_row::BCP_row (BCP_vec< int >::const_iterator *firstind*, BCP_vec< int >::const_iterator *lastind*, BCP_vec< double >::const_iterator *firstval*, BCP_vec< double >::const_iterator *lastval*, const double *LB*, const double *UB*) [inline]

This constructor acts exactly like the `copy` method with the same argument list.

Definition at line 236 of file BCP_matrix.hpp.

4.39.2.4 BCP_row::BCP_row (const int *size*, int ∗& *ElementIndices*, double ∗& *ElementValues*, const double *LB*, const double *UB*) [inline]

This constructor acts exactly like the `assign` method with the same argument list.

Definition at line 246 of file BCP_matrix.hpp.

4.39.2.5 BCP_row::∼BCP_row ( ) [inline]

The destructor deletes all data members.

Definition at line 257 of file BCP_matrix.hpp.

4.39.3 Member Function Documentation

4.39.3.1 double BCP_row::LowerBound ( ) const [inline]

Return the lower bound.

Definition at line 167 of file BCP_matrix.hpp.

4.39.3.2 double BCP_row::UpperBound ( ) const [inline]

Return the upper bound.

Definition at line 169 of file BCP_matrix.hpp.

4.39.3.3 void BCP_row::LowerBound (double *lb*) [inline]

Set the lower bound to the given value.

Definition at line 176 of file BCP_matrix.hpp.

4.39.3.4 void BCP_row::UpperBound (double *ub*) [inline]

Set the upper bound to the given value.

Definition at line 178 of file BCP_matrix.hpp.

4.39.3.5 BCP_row& BCP_row::operator= (const BCP_row & *x*) [inline]

Assignment operator: copy over the contents of `x`.

Definition at line 181 of file BCP_matrix.hpp.

4.39.3.6 void BCP_row::assign (const int *size,* int ∗& *ElementIndices,* double ∗& *ElementValues,* const double *LB,* const double *UB*) [inline]

Set the lower and upper bounds to the given values.

Also invokes the assign method of the underlying packed vector.

Definition at line 191 of file BCP_matrix.hpp.

4.39.3.7 void BCP_row::copy (const int *size,* const int ∗ *ElementIndices,* const double ∗ *ElementValues,* const double *LB,* const double *UB*) [inline]

Copy the arguments into the appropriate data members.

Definition at line 200 of file BCP_matrix.hpp.

4.39.3.8 void BCP_row::copy (BCP_vec< int >::const_iterator *firstind,* BCP_vec< int >::const_iterator *lastind,* BCP_vec< double >::const_iterator *firstval,* BCP_vec< double >::const_iterator *lastval,* const double *LB,* const double *UB*) [inline]

Same as the other `copy()` method, except that instead of using vectors the indices (values) are given in `[firstind,lastind)` (`[firstval,lastval)`).

Definition at line 211 of file BCP_matrix.hpp.

4.39.4 Member Data Documentation

4.39.4.1 double BCP_row::_LowerBound [protected]

The lower bound corresponding to the row.

Definition at line 157 of file BCP_matrix.hpp.

4.39.4.2 double BCP_row::_UpperBound [protected]

The upper bound corresponding to the row.

Definition at line 159 of file BCP_matrix.hpp.

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

- BCP_matrix.hpp

4.40 BCP_scheduler Class Reference

Public Member Functions

- BCP_scheduler ()

  Default constructor.
- void setParams (double OverEstimationStatic, double SwitchToRateThreshold, double TimeRootNodeSolve, double FactorTimeHorizon, double OverEstimationRate, double MaxNodeIdRatio, int MaxNodeIdNum, int MaxSbIdNum, int MinSbIdNum)

  Method for setting scheduler parameters.
- void add_free_ids (int numIds, const int ∗ids)

  Pass in a list of freeIds_ to add.
- int request_sb_ids (int numIds, int ∗ids)

  Request for a number of id's to do some strong branching.
- void release_sb_id (int id)

  Gives back to scheduler an id used for strong branching.
- int request_node_id ()

  Request an id for processing nodes.
- void release_node_id (int id)

  Give back an id to scheduler used for processing a node.
- bool has_free_node_id () const

  Decide if there is an id that can be returned for processin a node.
- int numNodeIds () const

  Return the number of busy LP processes.
- int maxNodeIds () const

  Return the maximum possible number of busy LP processes.
- double node_idle (int p)

  Return how much time did process p spent idling as a node process.
- double sb_idle (int p)

  Return how much time did process p spent idling as a SB process.
- void update_idle_times ()

  Update idle times with the last idle time.

4.40.1 Detailed Description

Definition at line 49 of file BCP_process.hpp.

4.40.2 Constructor & Destructor Documentation

4.40.2.1 BCP_scheduler::BCP_scheduler ( )

Default constructor.

4.40.3 Member Function Documentation

4.40.3.1 void BCP_scheduler::setParams (double *OverEstimationStatic,* double *SwitchToRateThreshold,* double *TimeRootNodeSolve,* double *FactorTimeHorizon,* double *OverEstimationRate,* double *MaxNodeIdRatio,* int *MaxNodeIdNum,* int *MaxSbIdNum,* int *MinSbIdNum*)

Method for setting scheduler parameters.

Parameters

OverEstimation-Static,:	Factor for providing more IDs in static strategy.
SwitchToRate-Threshold,:	When more than SwitchToRateThreshold times the number of strong-branching CPUs are busy, which to rate-based strategy.
TimeRootNode-Solve,:	Time to solve root node NLP (in secs)
FactorTime-Horizon,:	This number times TimeRootNodeSolve is used to compute the rates
OverEstimation-Rate,:	Factor for providing more IDs in rate-based strategy.
MaxNodeId-Ratio,:	At most this fraction of the total number of ids can be used as a nodeid.
MaxNodeIdNum,:	At most this many ids can be used as a nodeid.

4.40.3.2 void BCP_scheduler::add_free_ids (int *numIds,* const int $*$ *ids*)

Pass in a list of freeIds_ to add.

4.40.3.3 int BCP_scheduler::request_sb_ids (int *numIds,* int $*$ *ids*)

Request for a number of id's to do some strong branching.

NOTE: ids must be already allocated to be at least `numIds` size.

Parameters

numIds	: number of ids requested
ids	: filled vector with the number of ids served.

Returns

number of ids served.

4.40.3.4 void BCP_scheduler::release_sb_id (int *id*)

Gives back to scheduler an id used for strong branching.

4.40.3.5 int BCP_scheduler::request_node_id ( )

Request an id for processing nodes.

Returns

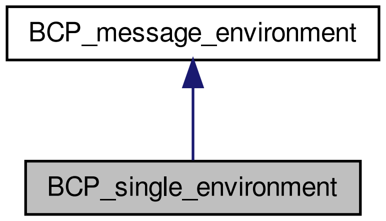
id number or -1 if none is available.

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

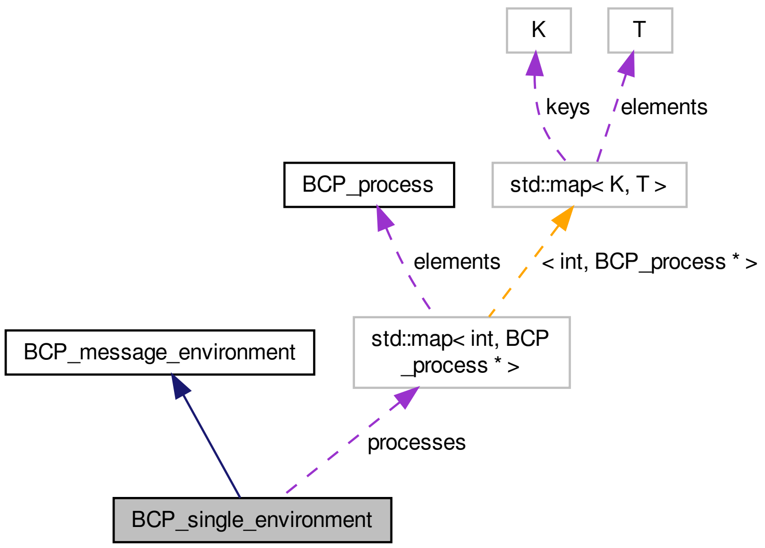
- BCP_process.hpp

4.41 BCP_single_environment Class Reference

Inheritance diagram for BCP_single_environment:



Collaboration diagram for BCP_single_environment:



Public Member Functions

- int register_process (USER_initialize ∗user_init)

  A process can register (receive its process id) with the message passing environment.
- int parent_process ()

  Return the process id of the parent process (the process that spawned the currnet process.
- bool alive (const int pid)

  Test if the process given by the argument is alive or not.
- const int ∗ alive (int num, const int ∗pids)

  Test if the processes given by the process array in the argument are alive or not.
- void send (const int target, const BCP_message_tag tag)

  Send an empty message (message tag only) to the process given by the frist argument.
- void send (const int target, const BCP_message_tag tag, const BCP_buffer &buf)

  Send the message in the buffer with the given message tag to the process given by the first argument.
- void multicast (int num, const int ∗targets, const BCP_message_tag tag)

  Send an empty message (message tag only) to all the processes in the process array.
- void multicast (int num, const int ∗targets, const BCP_message_tag tag, const BCP_buffer &buf)

  Send the message in the buffer with the given message tag to all processes in the process array.
- void receive (const int source, const BCP_message_tag tag, BCP_buffer &buf, const double timeout)

  Blocking receive with timeout.
- bool probe (const int source, const BCP_message_tag tag)

  Probe if there are any messages from the given process with the given message tag.
- int start_process (const BCP_string &exe, const bool debug)

  Spawn a new process.

- int start_process (const BCP_string &exe, const BCP_string &machine, const bool debug)

  Spawn a new process on the machine specified by the second argument.
- bool start_processes (const BCP_string &exe, const int proc_num, const bool debug, int ∗ids)

  Spawn `proc_num` *processes, all with the same executable.*
- bool start_processes (const BCP_string &exe, const int proc_num, const BCP_vec< BCP_string > &machines, const bool debug, int ∗ids)

  Spawn `proc_num` *processes on the machines given by the third argument, all with the same executable.*

4.41.1 Detailed Description

Definition at line 16 of file BCP_message_single.hpp.

4.41.2 Member Function Documentation

4.41.2.1 int BCP_single_environment::register_process (USER_initialize ∗ *user_init*) [virtual]

A process can register (receive its process id) with the message passing environment.

Implements BCP_message_environment.

4.41.2.2 int BCP_single_environment::parent_process ( ) [virtual]

Return the process id of the parent process (the process that spawned the currnet process.

Returns null if the process does not have a parent.

Implements BCP_message_environment.

4.41.2.3 bool BCP_single_environment::alive (const int *pid*) [virtual]

Test if the process given by the argument is alive or not.

Return true if alive, false otherwise.

Implements BCP_message_environment.

4.41.2.4 const int∗ BCP_single_environment::alive (int *num*, const int ∗ *pids*) [virtual]

Test if the processes given by the process array in the argument are alive or not.

Return a pointer to the first dead process, or to the end of the array if there are no dead processes.

Implements BCP_message_environment.

4.41.2.5 void BCP_single_environment::send (const int *target*, const BCP_message_tag *tag*) [virtual]

Send an empty message (message tag only) to the process given by the frist argument.

Implements BCP_message_environment.

4.41.2.6 void BCP_single_environment::send (const int *target*, const BCP_message_tag *tag*, const BCP_buffer & *buf*) [virtual]

Send the message in the buffer with the given message tag to the process given by the first argument.

Implements BCP_message_environment.

4.41.2.7 void BCP_single_environment::multicast (int *num,* const int ∗ *targets,* const BCP_message_tag *tag*) [virtual]

Send an empty message (message tag only) to all the processes in the process array.

Implements BCP_message_environment.

4.41.2.8 void BCP_single_environment::multicast (int *num,* const int ∗ *targets,* const BCP_message_tag *tag,* const BCP_buffer & *buf*) [virtual]

Send the message in the buffer with the given message tag to all processes in the process array.

Implements BCP_message_environment.

4.41.2.9 void BCP_single_environment::receive (const int *source,* const BCP_message_tag *tag,* BCP_buffer & *buf,* const double *timeout*) [virtual]

Blocking receive with timeout.

Wait until a message is received from the given process with the given message tag within the given timelimit. If `timeout` is positive then the receive routine times out after this many seconds. If it is negative then the method blocks until a message is received. The received message is saved into the buffer. With a 0 pointer as the first argument (or the predefined `BCP_AnyProcess` process id can be used) a message from any process will be accepted. Messages with any message tag are accepted if the second argument is `BCP_Msg_AnyMessage`.

Implements BCP_message_environment.

4.41.2.10 bool BCP_single_environment::probe (const int *source,* const BCP_message_tag *tag*) [virtual]

Probe if there are any messages from the given process with the given message tag.

Return true if such a message is found, false otherwise. Note that the message is not "read", only its existence is checked. Similarly as above, the wild cards `BCP_AnyProcess` and `BCP_Msg_AnyMessage` can be used.

Implements BCP_message_environment.

4.41.2.11 int BCP_single_environment::start_process (const BCP_string & *exe,* const bool *debug*) [virtual]

Spawn a new process.

The first argument contains the path to the executable to be spawned. If the second argument is set to true, then the executable is spawned under a debugger.

Implements BCP_message_environment.

4.41.2.12 int BCP_single_environment::start_process (const BCP_string & *exe,* const BCP_string & *machine,* const bool *debug*) [virtual]

Spawn a new process on the machine specified by the second argument.

Implements BCP_message_environment.

4.41.2.13 bool BCP_single_environment::start_processes (const BCP_string & *exe,* const int *proc_num,* const bool *debug,* int ∗ *ids*) [virtual]

Spawn `proc_num` processes, all with the same executable.

NOTE: ids must be allocated already and have enough space for `proc_num` entries.

Returns

true/false depending on success.

Implements BCP_message_environment.

4.41.2.14 bool BCP_single_environment::start_processes (const **BCP_string** & *exe,* const int *proc_num,* const **BCP_vec**< **BCP_string** > & *machines,* const bool *debug,* int ∗ *ids*) `[virtual]`

Spawn `proc_num` processes on the machines given by the third argument, all with the same executable.

NOTE: ids must be allocated already and have enough space for `proc_num` entries.

Returns

true/false depending on success.

Implements BCP_message_environment.

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

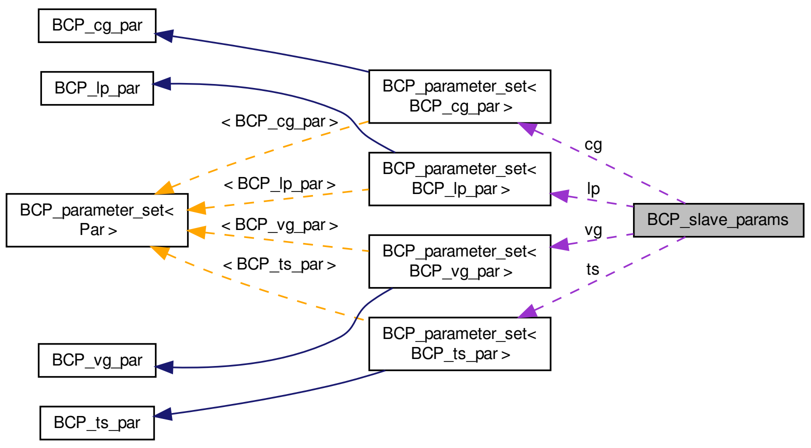
- BCP_message_single.hpp

4.42 BCP_slave_params Struct Reference

NO OLD DOC.

`#include <BCP_tm.hpp>`

Collaboration diagram for BCP_slave_params:



4.42.1 Detailed Description

NO OLD DOC.

Definition at line 57 of file BCP_tm.hpp.

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

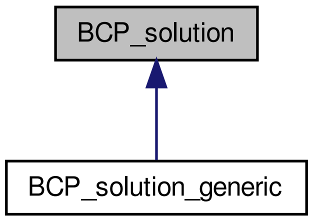
- BCP_tm.hpp

4.43 BCP_solution Class Reference

This is the abstract base class for a solution to a Mixed Integer Programming problem.

`#include <BCP_solution.hpp>`

Inheritance diagram for BCP_solution:



Public Member Functions

- virtual ~BCP_solution ()

  The virtual destructor ensures that the correct destructor is invoked.
- virtual double objective_value () const =0

  The method returning the objective value of the solution.

4.43.1 Detailed Description

This is the abstract base class for a solution to a Mixed Integer Programming problem.

Definition at line 14 of file BCP_solution.hpp.

4.43.2 Constructor & Destructor Documentation

4.43.2.1 virtual BCP_solution::~BCP_solution ( ) `[inline], [virtual]`

The virtual destructor ensures that the correct destructor is invoked.

Definition at line 17 of file BCP_solution.hpp.

4.43.3 Member Function Documentation

4.43.3.1 virtual double BCP_solution::objective_value ( ) const `[pure virtual]`

The method returning the objective value of the solution.

Implemented in BCP_solution_generic.

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

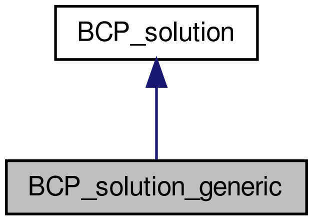
- BCP_solution.hpp

4.44 BCP_solution_generic Class Reference

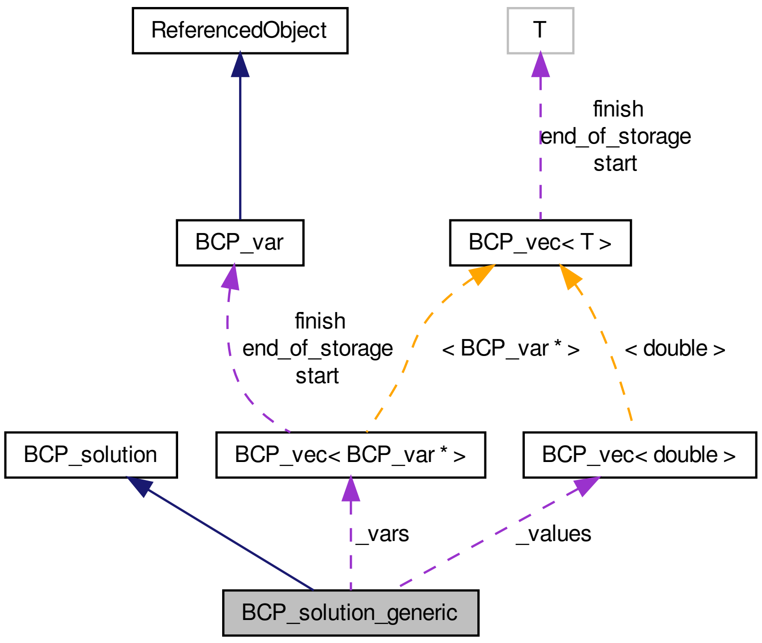
This class holds a MIP feasible primal solution.

`#include <BCP_solution.hpp>`

Inheritance diagram for BCP_solution_generic:



Collaboration diagram for BCP_solution_generic:



Public Member Functions

- BCP_solution_generic (bool delvars=true)

    The default constructor creates a solution with zero objective value.
- virtual ∼BCP_solution_generic ()

    The destructor deletes the data members.
- virtual double objective_value () const

    Return the objective value of the solution.
- void set_objective_value (double v)

    Set the objective value of the solution.
- void display () const

    Display the solution.
- void add_entry (BCP_var ∗var, double value)

    Append a variable and the corresponding value to the end of the appropriate vectors.

Public Attributes

Data members

- double _objective

    The objective value of the solution.
- bool _delete_vars

An indicator to show whether the pointers in `_vars` *should be deleted upon destruction or not.*

- BCP_vec< BCP_var * > _vars

  Vector of variables that are at nonzero level in the solution.
- BCP_vec< double > _values

  Values of these variables in the solution.

4.44.1 Detailed Description

This class holds a MIP feasible primal solution.

The default BCP_lp_user::pack_feasible_solution() uses this class to pack an MIP feasible solution, but the user can use this class (instead of using BCP_solution as a base class to derive a different solution holder. (This might be necessary if, for example, the dual values are of importance as well.) The virtual destructor and virtual member functions make this easy.

Definition at line 33 of file BCP_solution.hpp.

4.44.2 Constructor & Destructor Documentation

4.44.2.1 BCP_solution_generic::BCP_solution_generic (bool *delvars* = `true`) `[inline]`

The default constructor creates a solution with zero objective value.

Definition at line 50 of file BCP_solution.hpp.

4.44.2.2 virtual BCP_solution_generic::~BCP_solution_generic ( ) `[inline]`, `[virtual]`

The destructor deletes the data members.

Note that it purges `_vars` (i.e., deletes the variables the pointers in `_vars` point to) only if the `_delete_vars` member is true.

Definition at line 55 of file BCP_solution.hpp.

4.44.3 Member Function Documentation

4.44.3.1 virtual double BCP_solution_generic::objective_value ( ) const `[inline]`, `[virtual]`

Return the objective value of the solution.

Implements BCP_solution.

Definition at line 61 of file BCP_solution.hpp.

4.44.3.2 void BCP_solution_generic::set_objective_value (double *v*) `[inline]`

Set the objective value of the solution.

Definition at line 64 of file BCP_solution.hpp.

4.44.3.3 void BCP_solution_generic::display ( ) const

Display the solution.

4.44.3.4 void BCP_solution_generic::add_entry (BCP_var * *var*, double *value*) `[inline]`

Append a variable and the corresponding value to the end of the appropriate vectors.

This method is used when unpacking the solution.

Definition at line 71 of file BCP_solution.hpp.

4.44.4 Member Data Documentation

4.44.4.1 double BCP_solution_generic::_objective

The objective value of the solution.

Definition at line 38 of file BCP_solution.hpp.

4.44.4.2 bool BCP_solution_generic::_delete_vars

An indicator to show whether the pointers in `_vars` should be deleted upon destruction or not.

By default they are not.

Definition at line 41 of file BCP_solution.hpp.

4.44.4.3 BCP_vec<BCP_var*> BCP_solution_generic::_vars

Vector of variables that are at nonzero level in the solution.

Definition at line 43 of file BCP_solution.hpp.

4.44.4.4 BCP_vec<double> BCP_solution_generic::_values

Values of these variables in the solution.

Definition at line 45 of file BCP_solution.hpp.

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

- BCP_solution.hpp

4.45 BCP_string Class Reference

This class is a very simple impelementation of a constant length string.

`#include <BCP_string.hpp>`

4.45.1 Detailed Description

This class is a very simple impelementation of a constant length string.

Using it one can avoid some memory errors related to using functions operating on C style strings.

Definition at line 13 of file BCP_string.hpp.

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

- BCP_string.hpp

4.46 BCP_tm_flags Struct Reference

NO OLD DOC.

```
#include <BCP_tm.hpp>
```

Public Attributes

- bool root_pricing_unpacked

  Set to true if the result of root pricing is already unpacked.

4.46.1 Detailed Description

NO OLD DOC.

Definition at line 74 of file BCP_tm.hpp.

4.46.2 Member Data Documentation

4.46.2.1 bool BCP_tm_flags::root_pricing_unpacked

Set to true if the result of root pricing is already unpacked.

Important in a single process environment, so we don't unpack things twice.

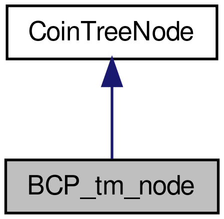
Definition at line 78 of file BCP_tm.hpp.

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

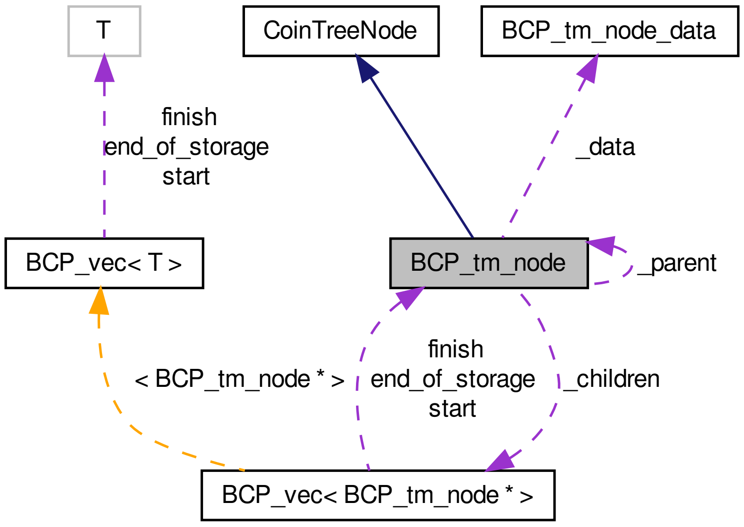
- BCP_tm.hpp

4.47 BCP_tm_node Class Reference

Inheritance diagram for BCP_tm_node:



Collaboration diagram for BCP_tm_node:



Public Member Functions

Constructors and destructor

- **BCP_tm_node** (int level, BCP_node_change ∗desc)
- ∼**BCP_tm_node** ()

Query methods

- int **index** () const
- int **child_num** () const
- int **birth_index** () const
- BCP_tm_node ∗ **child** (int ind)
- BCP_tm_node ∗ **parent** ()
- const BCP_tm_node ∗ **child** (int ind) const
- const BCP_tm_node ∗ **parent** () const

Modifying methods

- int **mark_descendants_for_deletion** ()
- void **remove_child** (BCP_tm_node ∗node)
- void **reserve_child_num** (int num)
- void **new_child** (BCP_tm_node ∗node)

Data members

- static int **num_local_nodes**

- static int **num_remote_nodes**
- BCP_tm_node_status **status**
- int **_index**
- BCP_tm_node ∗ **_parent**
- int **_birth_index**
- BCP_vec< BCP_tm_node ∗ > **_children**
- int **lp**
- int **cg**
- int **cp**
- int **vg**
- int **vp**
- int **_processed_leaf_num**
- int **_pruned_leaf_num**
- int **_tobepriced_leaf_num**
- int **_leaf_num**
- int **_core_storage**:4
- int **_var_storage**:4
- int **_cut_storage**:4
- int **_ws_storage**:4
- int **_locally_stored**:2
- int **_data_location**:30
- BCP_tm_node_data **_data**

4.47.1 Detailed Description

Definition at line 60 of file BCP_tm_node.hpp.

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

- BCP_tm_node.hpp

4.48 BCP_tm_node_data Class Reference

4.48.1 Detailed Description

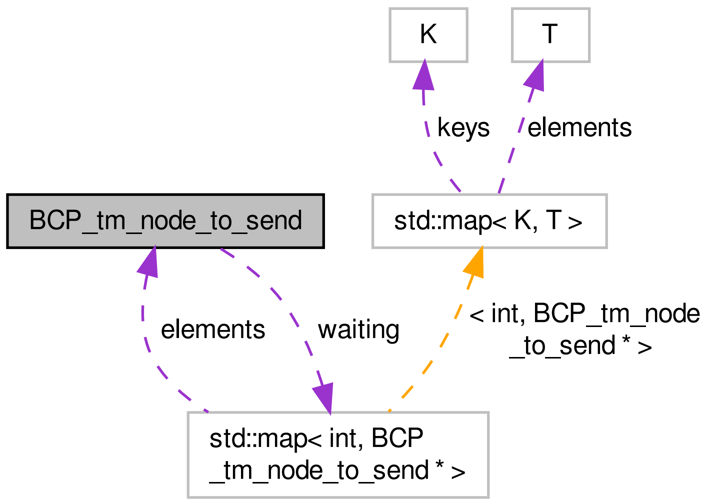
Definition at line 51 of file BCP_tm_node.hpp.

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

- BCP_tm_node.hpp

4.49 BCP_tm_node_to_send Class Reference

Collaboration diagram for BCP_tm_node_to_send:


K
T
keys
elements
BCP_tm_node_to_send
std::map< K, T >
elements
waiting
< int, BCP_tm_node _to_send * >
std::map< int, BCP _tm_node_to_send * >

Public Member Functions

- bool send ()

  return true or false depending on whether the node was really sent out or it's still waiting for some data
- bool receive_node_desc (BCP_buffer &buf)

  return true if has everything to send the thing off to the LP.
- bool receive_vars (BCP_buffer &buf)

  return true if has everything to send the thing off to the LP.
- bool receive_cuts (BCP_buffer &buf)

  return true if has everything to send the thing off to the LP.

4.49.1 Detailed Description

Definition at line 241 of file BCP_tm_node.hpp.

4.49.2 Member Function Documentation

4.49.2.1 bool BCP_tm_node_to_send::receive_node_desc (**BCP_buffer** & *buf*)

return true if has everything to send the thing off to the LP.

Actually, it sends it off, so if this method returns true then then object can be deleted

4.49.2.2 bool BCP_tm_node_to_send::receive_vars (BCP_buffer & *buf*)

return true if has everything to send the thing off to the LP.

Actually, it sends it off, so if this method returns true then then object can be deleted

4.49.2.3 bool BCP_tm_node_to_send::receive_cuts (BCP_buffer & *buf*)

return true if has everything to send the thing off to the LP.

Actually, it sends it off, so if this method returns true then then object can be deleted

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

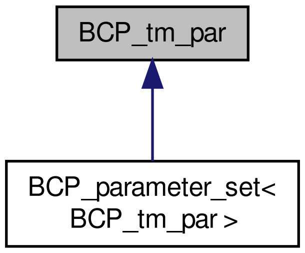
- BCP_tm_node.hpp

4.50 BCP_tm_par Struct Reference

Parameters used in the Tree Manager process.

`#include <BCP_tm_param.hpp>`

Inheritance diagram for BCP_tm_par:



Public Types

- enum chr_params {
DebugLpProcesses, DebugCgProcesses, DebugVgProcesses, DebugCpProcesses,
DebugVpProcesses, GenerateVars, MessagePassingIsSerial, ReportWhenDefaultIsExecuted,
TrimTreeBeforeNewPhase, RemoveExploredBranches, VerbosityShutUp, TmVerb_First,
TmVerb_AllFeasibleSolutionValue, TmVerb_AllFeasibleSolution, TmVerb_BetterFeasibleSolutionValue, TmVerb-_BetterFeasibleSolution,
TmVerb_BestFeasibleSolution, TmVerb_NewPhaseStart, TmVerb_PrunedNodeInfo, TmVerb_TimeOfImproving-Solution,
TmVerb_TrimmedNum, TmVerb_FinalStatistics, TmVerb_ReportDefault, TmVerb_Last }

  Character parameters.
- enum int_params {

WarmstartInfo, MaxHeapSize, TmVerb_SingleLineInfoFrequency, TreeSearchStrategy, NiceLevel, LpProcessNum, CgProcessNum, CpProcessNum, VgProcessNum, VpProcessNum, TmTimeout, LPscheduler_MaxNodeIdNum, LPscheduler_MaxSbIdNum, LPscheduler_MinSbIdNum }

Integer parameters.

- enum dbl_params { UnconditionalDiveProbability, QualityRatioToAllowDiving_HasUB, QualityRatioToAllowDiving_NoUB, Granularity, MaxRunTime, TerminationGap_Absolute, TerminationGap_Relative, UpperBound, LPscheduler_OverEstimationStatic }

  Double parameters.
- enum str_params { SaveRootCutsTo, ReadRootCutsFrom, ExecutableName, LogFileName }

  String parameters.
- enum str_array_params { LpMachines, CgMachines, VgMachines, CpMachines, VpMachines }

  ???

4.50.1 Detailed Description

Parameters used in the Tree Manager process.

These parameters can be set in the original parameter file by including the following line:

```
BCP_{parameter name} {parameter value}
```

Definition at line 10 of file BCP_tm_param.hpp.

4.50.2 Member Enumeration Documentation

4.50.2.1 enum BCP_tm_par::chr_params

Character parameters.

Most of these variables are used as booleans (true = 1, false = 0).

Enumerator:

DebugLpProcesses Indicates whether to debug LP processes or not. Values: 1 (true), 0 (false). Default: 0.

DebugCgProcesses Indicates whether to debug Cut Generator processes or not. Values: 1 (true), 0 (false). Default: 0.

DebugVgProcesses Indicates whether to debug Variable Generator processes or not. Values: 1 (true), 0 (false). Default: 0.

DebugCpProcesses Indicates whether to debug Cut Pool processes or not. Values: 1 (true), 0 (false). Default: 0.

DebugVpProcesses Indicates whether to debug Variable Pool processes or not. Values: 1 (true), 0 (false). Default: 0.

GenerateVars Indicates whether to variable generation will take place or not. Values: 1 (true), 0 (false). Default: 0.

MessagePassingIsSerial Indicates whether message passing is serial (all processes are on the same processor) or not. Values: 1 (true), 0 (false). Default: 0.

ReportWhenDefaultIsExecuted Print out a message when the default version of an overridable method is executed. Default: 1.

TrimTreeBeforeNewPhase Indicates whether to trim the search tree before a new phase. Values: 1 (true), 0 (false). Default: 0.

RemoveExploredBranches Indicates whether that part of the tree that's completely explored should be freed as soon as possible. This conserves memory, but may make it harder to track what's happening in the tree. Values: 1 (true), 0 (false). Default: 1.

VerbosityShutUp A flag that instructs BCP to be (almost) absolutely silent. It zeros out all the XxVerb flags *even if the verbosity flag is set to 1 later in the parameter file*. Exceptions (flags whose status is not changed) are: `TmVerb_SingleLineInfoFrequency`, `TmVerb_FinalStatistics` and `TmVerb_Best-FeasibleSolution`. Default: 0.

TmVerb_First Verbosity flags for the tree manager. Just a marker for the first TmVerb

TmVerb_AllFeasibleSolutionValue Print the value of *any* integer feasible solution found. (BCP_tm_prob-::process_message)

TmVerb_AllFeasibleSolution Invoke the user-written "display_feasible_solution" function if *any* feasible solution is found. (BCP_tm_prob::process_message)

TmVerb_BetterFeasibleSolutionValue Print the value of the integer solution when a solution better than the current best solution is found. (BCP_tm_prob::process_message)

TmVerb_BetterFeasibleSolution Invoke the user-written "display_feasible_solution" function if a better integral feasible solution is found. (BCP_tm_prob::process_message)

TmVerb_BestFeasibleSolution Invoke "display_feasible_solution" user routine for the best feasible solution after the entire tree is processed. (BCP_tm_wrapup)

TmVerb_NewPhaseStart Print the "Starting phase x" line. (BCP_tm_tasks_before_new_phase)

TmVerb_PrunedNodeInfo Print information about nodes that are pruned by bound in the tree manager. These nodes would be returned by the LP if they were sent there, so prune them in the TM instead. (BCP_tm_start-_one_node)

TmVerb_TimeOfImprovingSolution Print the time (and the solution value and solution if the above paramters are set appropriately) when any/better solution is found. (BCP_tm_prob::process_message)

TmVerb_TrimmedNum Print the number of nodes trimmed between phases. (BCP_tm_trim_tree_wrapper)

TmVerb_FinalStatistics Print statistics: running time, tree size, best solution value. (For the best solution set `TmVerb_BestFeasibleSolution`.

TmVerb_ReportDefault Print out a message when the default version of an overridable method is executed. Default: 1.

TmVerb_Last Just a marker for the last TmVerb.

Definition at line 13 of file BCP_tm_param.hpp.

4.50.2.2 enum BCP_tm_par::int_params

Integer parameters.

Enumerator:

WarmstartInfo Specifies how warmstart information should be stored in the TM. Possible valueas: BCP_-WarmstartNone, BCP_WarmstartRoot, BCP_WarmstartParent. The first is obvious, the second means the root node's warmstart info should be sent out along with each node, the last means the parent's warmstart info should be sent out. Default: BCP_WarmstartParent

MaxHeapSize The maximum size of the memory heap the TM can use. If the TM reaches this limit then it converts an LP process into a TS (storage) process). If positive, it's the number of megabytes. 0 indicates the TM should attempt to find this out. -1 indicates that no TS process should be created. Default: -1.

TmVerb_SingleLineInfoFrequency At every this many search tree node provide a single line info on the progress of the search tree. If $<=$ 0 then never. Default: 0.

TreeSearchStrategy Which search tree enumeration strategy should be used. Values: 0 (BCP_BestFirstSearch), 1 (BCP_BreadthFirstSearch), 2 (BCP_DepthFirstSearch). Default: 0

NiceLevel How resource-hog the processes should be. Interpretation is system dependent, and the value is passed directly to the renice() function. Usually the bigger the number the less demanding the processes will be.

LpProcessNum The number of LP processes that should be spawned.

CgProcessNum The number of Cut Generator processes that should be spawned.

CpProcessNum The number of Cut Pool processes that should be spawned. Values:

VgProcessNum The number of Variable Generator processes that should be spawned.

VpProcessNum The number of Variable Pool processes that should be spawned.

TmTimeout ???

LPscheduler_MaxNodeIdNum Parameters related to scheduling the LP processes.

LPscheduler_MaxSbIdNum Max how many SB nodes should the scheduler give to an LP process.

LPscheduler_MinSbIdNum Parameters related to scheduling the LP processes.

Definition at line 103 of file BCP_tm_param.hpp.

4.50.2.3 enum BCP_tm_par::dbl_params

Double parameters.

Enumerator:

UnconditionalDiveProbability The probability with which the LP process is directed to dive. Values: Default:

QualityRatioToAllowDiving_HasUB The LP process is allowed to dive if the ratio between the quality (for now the presolved objective value) of the child to be kept and the best quality among the candidate nodes is not larger the this parameter. This ratio is applied if an upper bound already exists. Values: Default:

QualityRatioToAllowDiving_NoUB Same as above, but this value is used if an upper bound does not exist yet. Values: Default:

Granularity ??? Values: Default:

MaxRunTime Maximum allowed running time.

TerminationGap_Absolute ??? Values: Default:

TerminationGap_Relative ??? Values: Default:

UpperBound ??? Values: Default:

LPscheduler_OverEstimationStatic Parameters related to scheduling the LP processes.

Definition at line 152 of file BCP_tm_param.hpp.

4.50.2.4 enum BCP_tm_par::str_params

String parameters.

Enumerator:

SaveRootCutsTo The name of the file where those cuts should be saved that were in the root node in the 0-th phase at the end of processing the root node.

ReadRootCutsFrom The name of the file where cuts to be added to the root description should be read ot from.

ExecutableName The name of the executable that's running (and that should be spawned on the other processors.

LogFileName ???

Definition at line 186 of file BCP_tm_param.hpp.

4.50.2.5 enum BCP_tm_par::str_array_params

???

Enumerator:

LpMachines ???

CgMachines ???

VgMachines ???

CpMachines ???

VpMachines ???

Definition at line 203 of file BCP_tm_param.hpp.

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

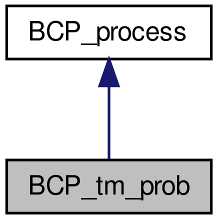
- BCP_tm_param.hpp

4.51 BCP_tm_prob Class Reference

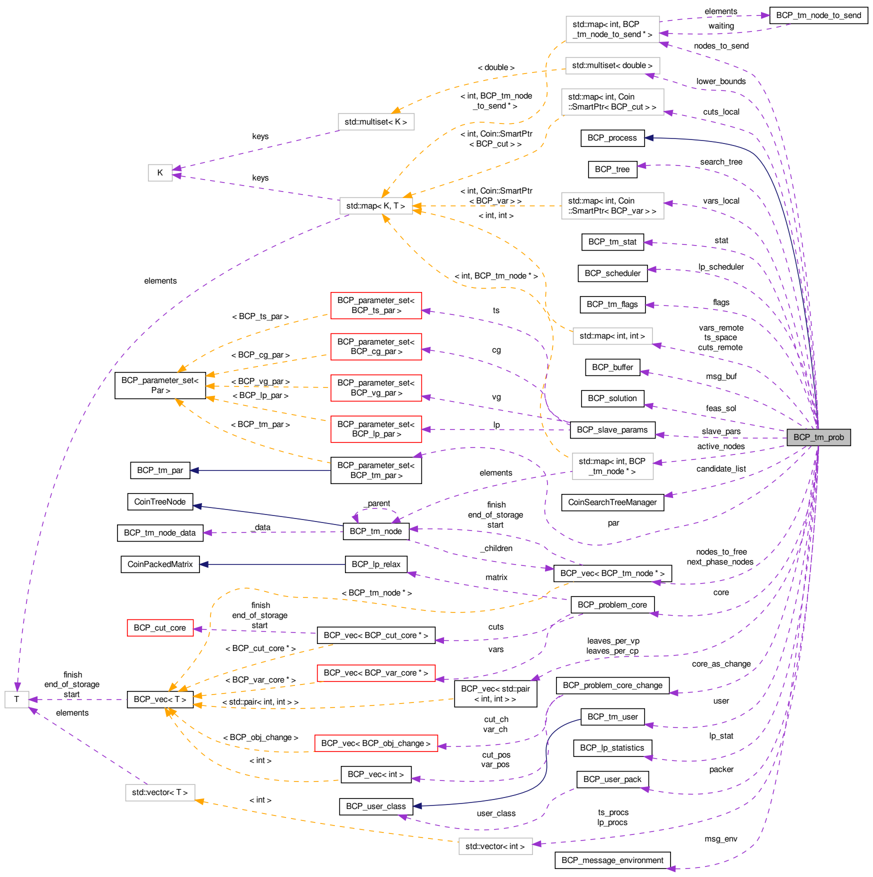
NO OLD DOC.

`#include <BCP_tm.hpp>`

Inheritance diagram for BCP_tm_prob:



Collaboration diagram for BCP_tm_prob:



Public Member Functions

Constructor and destructor

- **BCP_tm_prob** ()
- virtual ∼**BCP_tm_prob** ()

Methods to pack/unpack objects

- void **pack_var** (const BCP_var &var)
- BCP_var ∗ **unpack_var_without_bcpind** (BCP_buffer &buf)
- int **unpack_var** ()
- void **pack_cut** (const BCP_cut &cut)
- BCP_cut ∗ **unpack_cut_without_bcpind** (BCP_buffer &buf)
- int **unpack_cut** ()

Query methods

- char **param** (BCP_tm_par::chr_params key) const
- int **param** (BCP_tm_par::int_params key) const
- double **param** (BCP_tm_par::dbl_params key) const

- const BCP_string & **param** (BCP_tm_par::str_params key) const
- const BCP_vec< BCP_string > & **param** (BCP_tm_par::str_array_params key) const
- double **granularity** () const
- bool **has_ub** () const
- double **ub** () const
- bool **ub** (double new_ub)
- bool **over_ub** (const double lb) const

Public Attributes

- std::map< int, BCP_tm_node ∗ > active_nodes

  A map from the process ids to the nodes (what they work on)
- BCP_vec< BCP_tm_node ∗ > next_phase_nodes

  a vector of nodes to be processed in the next phase

User provided members

- BCP_tm_user ∗ **user**
- BCP_user_pack ∗ packer

  A class that holds the methods about how to pack things.
- BCP_message_environment ∗ **msg_env**

Statistics in the other processes

- BCP_lp_statistics ∗ **lp_stat**
- BCP_solution ∗ **feas_sol**

Parameters

- BCP_parameter_set< BCP_tm_par > **par**
- BCP_slave_params **slave_pars**

Flags

- BCP_tm_flags **flags**

Message passing related fields

- BCP_buffer **msg_buf**
- std::vector< int > **ts_procs**
- std::vector< int > **lp_procs**
- BCP_scheduler **lp_scheduler**
- double root_node_sent_

  members to measure how long it took to process the root node.
- double **root_node_received_**

The description of the core of the problem

- BCP_problem_core ∗ **core**
- BCP_problem_core_change ∗ **core_as_change**

Vectors indicating the number of leaf nodes assigned to each CP/VP

- BCP_vec< std::pair< int, int > > **leaves_per_cp**
- BCP_vec< std::pair< int, int > > **leaves_per_vp**

Static Public Attributes

- static double lb_multiplier

  The lower bounds of the unexplored search tree nodes.

4.51.1 Detailed Description

NO OLD DOC.

Definition at line 136 of file BCP_tm.hpp.

4.51.2 Member Data Documentation

4.51.2.1 BCP_user_pack∗ BCP_tm_prob::packer

A class that holds the methods about how to pack things.

Definition at line 153 of file BCP_tm.hpp.

4.51.2.2 double BCP_tm_prob::root_node_sent_

members to measure how long it took to process the root node.

Needed for the scheduler (both are in wallclock)

Definition at line 191 of file BCP_tm.hpp.

4.51.2.3 double BCP_tm_prob::lb_multiplier `[static]`

The lower bounds of the unexplored search tree nodes.

To avoid rounding errors the lower bounds are stored as floor(lb∗lb_multiplier)

Definition at line 198 of file BCP_tm.hpp.

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

- BCP_tm.hpp

4.52 BCP_tm_stat Class Reference

4.52.1 Detailed Description

Definition at line 85 of file BCP_tm.hpp.

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

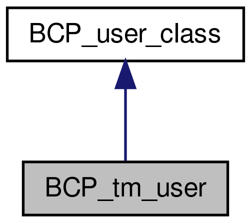
- BCP_tm.hpp

4.53 BCP_tm_user Class Reference

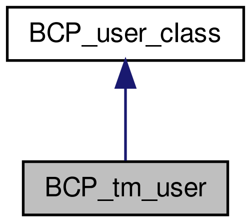
The BCP_tm_user class is the base class from which the user can derive a problem specific class to be used in the TM process.

`#include <BCP_tm_user.hpp>`

Inheritance diagram for BCP_tm_user:



Collaboration diagram for BCP_tm_user:



Public Member Functions

- int process_id () const

  What is the process id of the current process.
- void send_message (const int target, const BCP_buffer &buf)

  Send a message to a particular process.
- void broadcast_message (const BCP_process_t proc_type, const BCP_buffer &buf)

  Broadcast the message to all processes of the given type.
- virtual void process_message (BCP_buffer &buf)

  Process a message that has been sent by another process' user part to this process' user part.
- virtual void display_feasible_solution (const BCP_solution ∗sol)

  Display a feasible solution.
- virtual void display_node_information (BCP_tree &search_tree, const BCP_tm_node &node)

  Display user information just before a new node is sent to the LP or diving into a node is acknowledged.
- virtual void display_node_information (BCP_tree &search_tree, const BCP_tm_node &node, bool after_-processing_node)

Display user information.
- virtual void display_final_information (const BCP_lp_statistics &lp_stat)

  Display information after BCP finished processing the search tree.

Methods to set and get the pointer to the BCP_tm_prob

object.

It is unlikely that the users would want to muck around with these (especially with the set method!) but they are here to provide total control.

- void setTmProblemPointer (BCP_tm_prob ∗ptr)

  Set the pointer.
- BCP_tm_prob ∗ getTmProblemPointer () const

  Get the pointer.

Informational methods for the user.

- double upper_bound () const

  Return what is the best known upper bound (might be BCP_DBL_MAX)
- double lower_bound () const

  Return a global lower bound.

Methods to get/set BCP parameters on the fly

- char **get_param** (const BCP_tm_par::chr_params key) const
- int **get_param** (const BCP_tm_par::int_params key) const
- double **get_param** (const BCP_tm_par::dbl_params key) const
- const BCP_string & **get_param** (const BCP_tm_par::str_params key) const
- void **set_param** (const BCP_tm_par::chr_params key, const bool val)
- void **set_param** (const BCP_tm_par::chr_params key, const char val)
- void **set_param** (const BCP_tm_par::int_params key, const int val)
- void **set_param** (const BCP_tm_par::dbl_params key, const double val)
- void **set_param** (const BCP_tm_par::str_params key, const char ∗val)

Constructor, Destructor

- **BCP_tm_user** ()
- virtual ∼BCP_tm_user ()

  Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Packing and unpacking methods

- virtual void pack_module_data (BCP_buffer &buf, BCP_process_t ptype)

  Pack the initial information (info that the user wants to send over) for the process specified by the last argument.
- virtual BCP_solution ∗ unpack_feasible_solution (BCP_buffer &buf)

  Unpack a MIP feasible solution that was packed by the BCP_lp_user::pack_feasible_solution() method.
- virtual bool replace_solution (const BCP_solution ∗old_sol, const BCP_solution ∗new_sol)

  Decide whether to replace old_sol with new_sol.

Initial setup (creating core and root)

- virtual void initialize_core (BCP_vec< BCP_var_core ∗ > &vars, BCP_vec< BCP_cut_core ∗ > &cuts, BCP-_lp_relax ∗&matrix)

  Create the core of the problem by filling out the last three arguments.

- virtual void create_root (BCP_vec< BCP_var ∗ > &added_vars, BCP_vec< BCP_cut ∗ > &added_cuts, BC-P_user_data ∗&user_data)

  Create the set of extra variables and cuts that should be added to the formulation in the root node.

Initialize new phase

- virtual void init_new_phase (int phase, BCP_column_generation &colgen, **CoinSearchTreeBase** ∗&candidates)

  Do whatever initialization is necessary before the `phase`-th phase.

If desired, change the tree (the candidate list) in the search

tree manager using the setTree() method.

This method is invoked after every insertion into the candidate list and also whenever a new solution is found. In the latter case `new_solution` is `true`.

The default invokes the newSolution() and the reevaluateSearchStrategy() methods from **CoinSearchTreeManager**.

- virtual void **change_candidate_heap** (**CoinSearchTreeManager** &candidates, const bool new_solution)

4.53.1 Detailed Description

The BCP_tm_user class is the base class from which the user can derive a problem specific class to be used in the TM process.

In that derived class the user can store data to be used in the methods she overrides. Also that is the object the user must return in the USER_initialize::tm_init() method.

There are two kind of methods in the class. The non-virtual methods are helper functions for the built-in defaults, but the user can use them as well. The virtual methods execute steps in the BCP algorithm where the user might want to override the default behavior.

The default implementations fall into three major categories.

- Empty; doesn't do anything and immediately returns. (E.g., pack_module_data().
- There is no reasonable default, so throw an exception. This happens if the parameter settings drive the flow of in a way that BCP can't perform the necessary function. This behavior is correct since such methods are invoked only if the parameter settings drive the flow of the algorithm that way, in which case the user better implement those methods. (At the momemnt there is no such method in TM.)
- A default is given. Frequently there are multiple defaults and parameters govern which one is selected (e.g., compare_tree_nodes()).

Definition at line 58 of file BCP_tm_user.hpp.

4.53.2 Constructor & Destructor Documentation

4.53.2.1 virtual BCP_tm_user::∼BCP_tm_user ( ) `[inline]`, `[virtual]`

Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Definition at line 113 of file BCP_tm_user.hpp.

4.53.3 Member Function Documentation

4.53.3.1 double BCP_tm_user::lower_bound ( ) const

Return a global lower bound.

This value is the minimum of the true lower bounds in the candidate list and the true lower bounds of the nodes currently processed

4.53.3.2 virtual void BCP_tm_user::pack_module_data (**BCP_buffer** & *buf*, BCP_process_t *ptype*) [virtual]

Pack the initial information (info that the user wants to send over) for the process specified by the last argument.

The information packed here will be unpacked in the `unpack_module_data()` method of the user defined class in the appropriate process.

Default: empty method.

4.53.3.3 virtual **BCP_solution**∗ BCP_tm_user::unpack_feasible_solution (**BCP_buffer** & *buf*) [virtual]

Unpack a MIP feasible solution that was packed by the BCP_lp_user::pack_feasible_solution() method.

Default: Unpacks a BCP_solution_generic object. The built-in default should be used if and only if the built-in default was used in BCP_lp_user::pack_feasible_solution().

4.53.3.4 virtual bool BCP_tm_user::replace_solution (const **BCP_solution** ∗ *old_sol*, const **BCP_solution** ∗ *new_sol*) [virtual]

Decide whether to replace old_sol with new_sol.

When this method is invoked it has already been tested that they have the same objective function value. The purpose of the method is that the user can have a secondary objective function.

4.53.3.5 virtual void BCP_tm_user::process_message (**BCP_buffer** & *buf*) [virtual]

Process a message that has been sent by another process' user part to this process' user part.

4.53.3.6 virtual void BCP_tm_user::initialize_core (**BCP_vec**< **BCP_var_core** ∗ > & *vars*, **BCP_vec**< **BCP_cut_core** ∗ > & *cuts*, **BCP_lp_relax** ∗& *matrix*) [virtual]

Create the core of the problem by filling out the last three arguments.

These variables/cuts will always stay in the LP relaxation and the corresponding matrix is described by the specified matrix. If there is no core variable or cut then the returned pointer for to the matrix should be a null pointer.

Default: empty method, meaning that there are no variables/cuts in the core and this the core matrix is empty (0 pointer) as well.

4.53.3.7 virtual void BCP_tm_user::create_root (**BCP_vec**< **BCP_var** ∗ > & *added_vars*, **BCP_vec**< **BCP_cut** ∗ > & *added_cuts*, **BCP_user_data** ∗& *user_data*) [virtual]

Create the set of extra variables and cuts that should be added to the formulation in the root node.

Also decide how variable pricing shuld be done, that is, if column generation is requested in the init_new_phase() method of this class then column generation should be performed according to `pricing_status`.

Default: empty method, meaning that no variables/cuts are added, there is no user data and no pricing should be done.

4.53.3.8 virtual void BCP_tm_user::display_node_information (**BCP_tree** & *search_tree,* const **BCP_tm_node** & *node*) [virtual]

Display user information just before a new node is sent to the LP or diving into a node is acknowledged.

This method is deprecated in favor of the one with 3 args.

4.53.3.9 virtual void BCP_tm_user::display_node_information (**BCP_tree** & *search_tree,* const **BCP_tm_node** & *node,* bool *after_processing_node*) [virtual]

Display user information.

This method is called just before a node is sent ot for processing (or diving into the node is acknowledged) and just after a node description has been received.

4.53.3.10 virtual void BCP_tm_user::display_final_information (const **BCP_lp_statistics** & *lp_stat*) [virtual]

Display information after BCP finished processing the search tree.

4.53.3.11 virtual void BCP_tm_user::init_new_phase (int *phase,* BCP_column_generation & *colgen,* **CoinSearchTreeBase** ∗& *candidates*) [virtual]

Do whatever initialization is necessary before the `phase`-th phase.

(E.g., setting the pricing strategy.)

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

- BCP_tm_user.hpp

4.54 BCP_tree Class Reference

NO OLD DOC.

`#include <BCP_tm_node.hpp>`

Public Member Functions

Constructor and destructor

- **BCP_tree** ()
- ∼**BCP_tree** ()

Query methods

- BCP_vec< BCP_tm_node ∗ >::iterator **begin** ()
- BCP_vec< BCP_tm_node ∗ >::iterator **end** ()
- BCP_tm_node ∗ **root** ()
- BCP_tm_node ∗ **operator[]** (int index)
- size_t **size** () const
- int **maxdepth** () const
- int **processed** () const
- void **increase_processed** ()

Modifying methods

- double true_lower_bound (const BCP_tm_node ∗node) const

Return the worst true lower bound in the search tree.
- void **enumerate_leaves** (BCP_tm_node ∗node, const double obj_limit)
- void **insert** (BCP_tm_node ∗node)
- void **remove** (int index)

4.54.1 Detailed Description

NO OLD DOC.

Definition at line 178 of file BCP_tm_node.hpp.

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

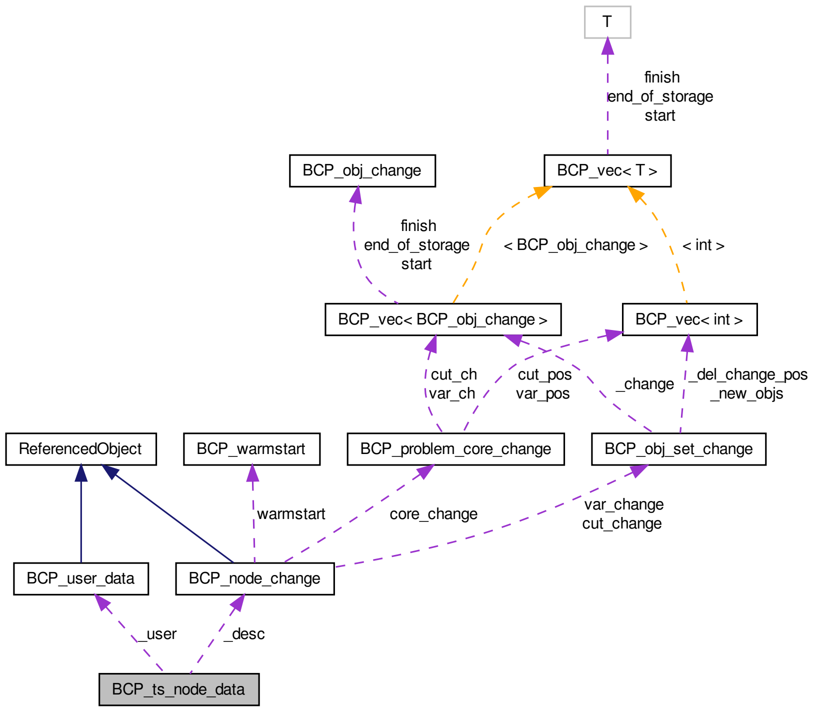
- BCP_tm_node.hpp

4.55 BCP_ts_node_data Struct Reference

Same as BCP_tm_node_data, just there's no need to use smart pointers in this process.

`#include <BCP_tmstorage.hpp>`

Collaboration diagram for BCP_ts_node_data:



4.55.1 Detailed Description

Same as BCP_tm_node_data, just there's no need to use smart pointers in this process.

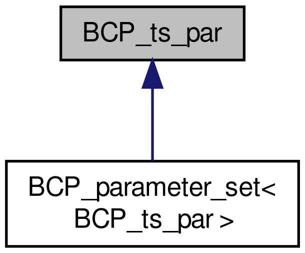
Definition at line 67 of file BCP_tmstorage.hpp.

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

- BCP_tmstorage.hpp

4.56 BCP_ts_par Struct Reference

Inheritance diagram for BCP_ts_par:



Public Types

- enum int_params { MaxHeapSize }

4.56.1 Detailed Description

Definition at line 30 of file BCP_tmstorage.hpp.

4.56.2 Member Enumeration Documentation

4.56.2.1 enum BCP_ts_par::int_params

Enumerator:

MaxHeapSize The maximum size of the memory heap the TS can use. The amount of free memory the TS indicates towards the TM uses this value. If positive, it's the number of megabytes. 0 indicates the TS should attempt to find this out. -1 indicates that the TS can use as much memory as it wants. Default: 0.

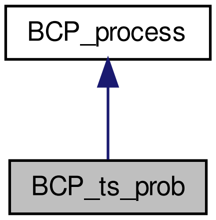
Definition at line 37 of file BCP_tmstorage.hpp.

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

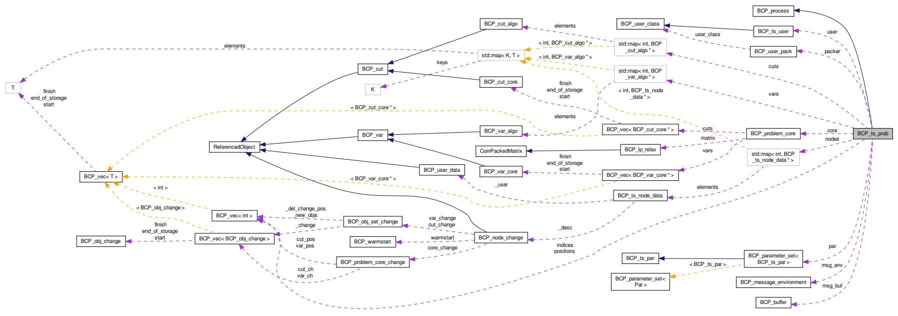
- BCP_tmstorage.hpp

4.57 BCP_ts_prob Class Reference

Inheritance diagram for BCP_ts_prob:



Collaboration diagram for BCP_ts_prob:



Public Attributes

- BCP_vec< int > indices

  a list of indices of nodes/vars/cuts that are requested/tobe deleted
- BCP_vec< int > positions

  positions in the TM of requested nodes/vars/cuts

4.57.1 Detailed Description

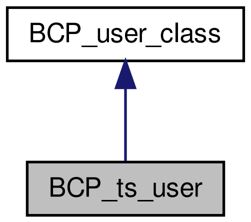
Definition at line 74 of file BCP_tmstorage.hpp.

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

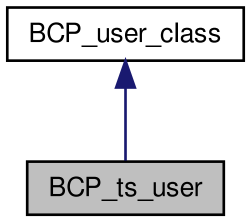
- BCP_tmstorage.hpp

4.58 BCP_ts_user Class Reference

Inheritance diagram for BCP_ts_user:



Collaboration diagram for BCP_ts_user:



Public Member Functions

- virtual void unpack_module_data (BCP_buffer &buf)

  Unpack the initial information sent to the LP process by the Tree Manager.

Methods to set and get the pointer to the BCP_ts_prob

object.

It is unlikely that the users would want to muck around with these (especially with the set method!) but they are here to provide total control.

- void setTsProblemPointer (BCP_ts_prob ∗ptr)

  Set the pointer.
- BCP_ts_prob ∗ getTsProblemPointer ()

  Get the pointer.

Methods to get/set BCP parameters on the fly

- char **get_param** (const BCP_ts_par::chr_params key) const
- int **get_param** (const BCP_ts_par::int_params key) const
- double **get_param** (const BCP_ts_par::dbl_params key) const
- const BCP_string & **get_param** (const BCP_ts_par::str_params key) const
- void **set_param** (const BCP_ts_par::chr_params key, const bool val)
- void **set_param** (const BCP_ts_par::chr_params key, const char val)
- void **set_param** (const BCP_ts_par::int_params key, const int val)
- void **set_param** (const BCP_ts_par::dbl_params key, const double val)
- void **set_param** (const BCP_ts_par::str_params key, const char ∗val)

Constructor, Destructor

- **BCP_ts_user** ()
- virtual ∼BCP_ts_user ()

    Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

4.58.1 Detailed Description

Definition at line 125 of file BCP_tmstorage.hpp.

4.58.2 Constructor & Destructor Documentation

4.58.2.1 virtual BCP_ts_user::∼BCP_ts_user ( ) [inline], [virtual]

Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Definition at line 169 of file BCP_tmstorage.hpp.

4.58.3 Member Function Documentation

4.58.3.1 virtual void BCP_ts_user::unpack_module_data (BCP_buffer & *buf*) [virtual]

Unpack the initial information sent to the LP process by the Tree Manager.

This information was packed by the method BCP_ts_user::pack_module_data() invoked with `BCP_ProcessType_TS` as the third (target process type) argument.

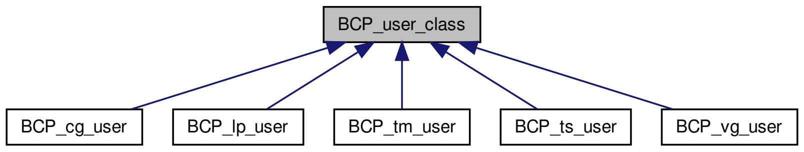
Default: empty method.

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

- BCP_tmstorage.hpp

4.59 BCP_user_class Class Reference

Inheritance diagram for BCP_user_class:



4.59.1 Detailed Description

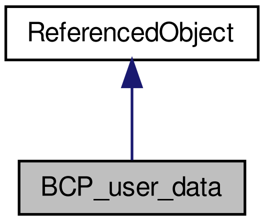
Definition at line 26 of file BCP_USER.hpp.

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

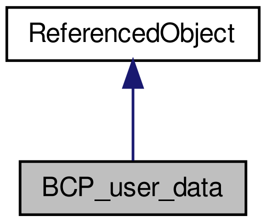
- BCP_USER.hpp

4.60 BCP_user_data Class Reference

Inheritance diagram for BCP_user_data:



Collaboration diagram for BCP_user_data:



4.60.1 Detailed Description

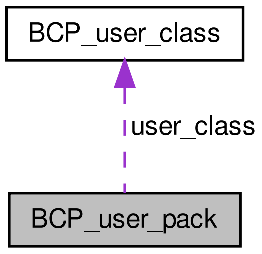
Definition at line 19 of file BCP_USER.hpp.

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

- BCP_USER.hpp

4.61 BCP_user_pack Class Reference

Collaboration diagram for BCP_user_pack:



Public Member Functions

- virtual void pack_warmstart (const BCP_warmstart ∗ws, BCP_buffer &buf, bool report_if_default=false)

  Pack warmstarting information.
- virtual BCP_warmstart ∗ unpack_warmstart (BCP_buffer &buf, bool report_if_default=false)

  Unpack warmstarting information.

- virtual void pack_var_algo (const BCP_var_algo ∗var, BCP_buffer &buf)

    Pack an algorithmic variable.
- virtual BCP_var_algo ∗ unpack_var_algo (BCP_buffer &buf)

    Unpack an algorithmic variable.
- virtual void pack_cut_algo (const BCP_cut_algo ∗cut, BCP_buffer &buf)

    Pack an algorithmic cut.
- virtual BCP_cut_algo ∗ unpack_cut_algo (BCP_buffer &buf)

    Unpack an algorithmic cut.
- virtual void pack_user_data (const BCP_user_data ∗ud, BCP_buffer &buf)

    Pack an user data.
- virtual BCP_user_data ∗ unpack_user_data (BCP_buffer &buf)

    Unpack an user data.

Public Attributes

- BCP_user_class ∗ user_class

    A pointer ot the usr class of the process from which the methods of this class are invoked from.

4.61.1 Detailed Description

Definition at line 48 of file BCP_USER.hpp.

4.61.2 Member Data Documentation

4.61.2.1 BCP_user_class∗ BCP_user_pack::user_class

A pointer ot the usr class of the process from which the methods of this class are invoked from.

The user can try to dynamic cast `user_class` into BCP_tm_user∗, BCP_lp_user∗, etc., and when the cast succeeds the user knows in which process the methods was called, and also, the user will have access to the other methods of that class.

Definition at line 55 of file BCP_USER.hpp.

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

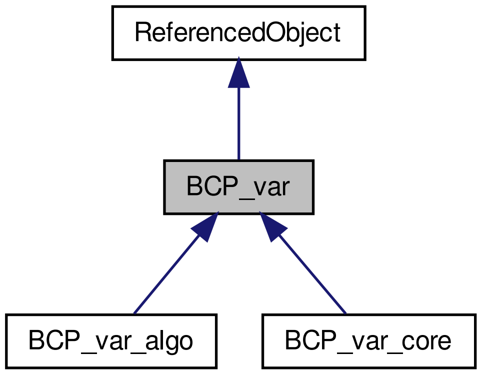
- BCP_USER.hpp

4.62 BCP_var Class Reference

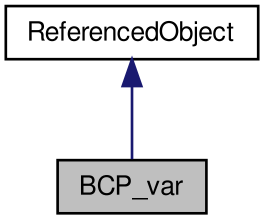
Abstract base class that defines members common to all types of variables.

```
#include <BCP_var.hpp>
```

Inheritance diagram for BCP_var:



Collaboration diagram for BCP_var:



Public Member Functions

Constructor and destructor

Note that there is no default constructor.

There is no such thing as "default variable".

- BCP_var (const BCP_var_t var_type, const double obj, const double lb, const double ub)
  The constructor sets the internal index of the variable to zero and the other data members to the given arguments.
- virtual ∼BCP_var ()
  The destructor is virtual so that the appropriate destructor is invoked for every variable.

Query methods

- virtual BCP_object_t obj_type () const =0

Return the type of the variable.
- BCP_var_t var_type () const

  Return the integrality type of the variable.
- double obj () const

  Return the objective coefficient.
- double lb () const

  Return the lower bound.
- double ub () const

  Return the upper bound.
- int bcpind () const

  Return the internal index of the variable.

Query methods about the status of the variable

- BCP_obj_status status () const

  Return the status of the variable.
- bool dont_send_to_pool () const

  Return whether the variable should be sent to the Variable Pool process.
- bool is_fixed () const

  Return whether the variable is fixed or not.
- bool is_fixed_to_zero () const

  Return whether the variable is fixed to zero or not.
- bool is_non_removable () const

  Return whether the variable is marked NotRemovable.
- bool is_removable () const

  Return whether the variable is removable from the formulation at the time of the query.
- bool is_to_be_removed () const

  Return whether the variable must be removed from the formulation.

Modifying methods

- void test_inactive ()

  Test (and set) whether the var is fixed (inactive)
- void set_var_type (const BCP_var_t type)

  Set the integrality type of the variable.
- void set_obj (const double obj)

  Set the objective coefficient.
- void set_lb (const double lb)

  Set the lower bound.
- void set_ub (const double ub)

  Set the upper bound.
- void set_lb_ub (const double lb, const double ub)

  Set both lower and upper bounds.
- void change_lb_ub_st (const BCP_obj_change &change)

  Set the lower/upper bounds and the status of the variable simultaneously to the values given in the data members of the argument.
- void change_bounds (const double lb, const double ub)

  Change the lower and upper bounds to the given values.
- void set_bcpind (const int bcpind)

  Set the internal index of the variable.
- void set_bcpind_flip ()

  Flip the internal index of the variable to its negative.

Status modifying methods

- void set_status (const BCP_obj_status status)
  Set the status of the variable.
- void dont_send_to_pool (bool flag)
  Set/unset the flag controlling whether the variable could be sent to the Variable Pool process.
- void make_active ()
  Mark the variable as active.
- void make_non_removable ()
  Mark the variable as NotRemovable.
- void make_to_be_removed ()
  Mark the variable as ToBeRemoved.

Display

- void display (const double val) const
  Display the object type, internal index, and the value given in the argument.

Protected Attributes

Protected data members

- BCP_var_t _var_type
  The integrality type of the variable.
- double _obj
  The objective coefficient.
- double _lb
  Lower bound on the value the variable can take.
- double _ub
  Upper bound on the value the variable can take.

4.62.1 Detailed Description

Abstract base class that defines members common to all types of variables.

Classes describing the three types of variables (core and algorithmic) are derived from this class. No object of type BCP_var can exist (having purely virtual members in the class enforces this restriction).

Definition at line 28 of file BCP_var.hpp.

4.62.2 Constructor & Destructor Documentation

4.62.2.1 BCP_var::BCP_var (const BCP_var_t *var_type,* const double *obj,* const double *lb,* const double *ub*) `[inline]`

The constructor sets the internal index of the variable to zero and the other data members to the given arguments.

Definition at line 71 of file BCP_var.hpp.

4.62.2.2 virtual BCP_var::∼BCP_var ( ) `[inline]`, `[virtual]`

The destructor is virtual so that the appropriate destructor is invoked for every variable.

Definition at line 77 of file BCP_var.hpp.

4.62.3 Member Function Documentation

4.62.3.1 virtual BCP_object_t BCP_var::obj_type ( ) const [pure virtual]

Return the type of the variable.

Implemented in BCP_var_algo, and BCP_var_core.

4.62.3.2 BCP_var_t BCP_var::var_type ( ) const [inline]

Return the integrality type of the variable.

Definition at line 85 of file BCP_var.hpp.

4.62.3.3 double BCP_var::obj ( ) const [inline]

Return the objective coefficient.

Definition at line 87 of file BCP_var.hpp.

4.62.3.4 double BCP_var::lb ( ) const [inline]

Return the lower bound.

Definition at line 89 of file BCP_var.hpp.

4.62.3.5 double BCP_var::ub ( ) const [inline]

Return the upper bound.

Definition at line 91 of file BCP_var.hpp.

4.62.3.6 int BCP_var::bcpind ( ) const [inline]

Return the internal index of the variable.

Definition at line 93 of file BCP_var.hpp.

4.62.3.7 BCP_obj_status BCP_var::status ( ) const [inline]

Return the status of the variable.

Definition at line 98 of file BCP_var.hpp.

4.62.3.8 bool BCP_var::dont_send_to_pool ( ) const [inline]

Return whether the variable should be sent to the Variable Pool process.

(Assuming that it stays in the formulation long enough to qualify to be sent to the Variable Pool at all.

Definition at line 102 of file BCP_var.hpp.

4.62.3.9 bool BCP_var::is_fixed ( ) const [inline]

Return whether the variable is fixed or not.

Definition at line 106 of file BCP_var.hpp.

4.62.3.10 bool BCP_var::is_fixed_to_zero ( ) const [inline]

Return whether the variable is fixed to zero or not.

Definition at line 110 of file BCP_var.hpp.

4.62.3.11 bool BCP_var::is_non_removable ( ) const [inline]

Return whether the variable is marked NotRemovable.

Examples of such variables include ???

Definition at line 115 of file BCP_var.hpp.

4.62.3.12 bool BCP_var::is_removable ( ) const [inline]

Return whether the variable is removable from the formulation at the time of the query.

(It is if it is not non_removable and it is fixed to zero.)

Definition at line 121 of file BCP_var.hpp.

4.62.3.13 bool BCP_var::is_to_be_removed ( ) const [inline]

Return whether the variable must be removed from the formulation.

There are very few circumstances when this flag is set; all of them are completely internal to BCP.

Definition at line 127 of file BCP_var.hpp.

4.62.3.14 void BCP_var::set_var_type (const BCP_var_t *type*) [inline]

Set the integrality type of the variable.

Definition at line 141 of file BCP_var.hpp.

4.62.3.15 void BCP_var::set_obj (const double *obj*) [inline]

Set the objective coefficient.

Definition at line 143 of file BCP_var.hpp.

4.62.3.16 void BCP_var::set_lb (const double *lb*) [inline]

Set the lower bound.

Definition at line 145 of file BCP_var.hpp.

4.62.3.17 void BCP_var::set_ub (const double *ub*) [inline]

Set the upper bound.

Definition at line 150 of file BCP_var.hpp.

4.62.3.18 void BCP_var::set_lb_ub (const double *lb,* const double *ub*) [inline]

Set both lower and upper bounds.

Definition at line 155 of file BCP_var.hpp.

4.62.3.19 void BCP_var::change_bounds (const double *lb,* const double *ub*) [inline]

Change the lower and upper bounds to the given values.

Definition at line 170 of file BCP_var.hpp.

4.62.3.20 void BCP_var::set_bcpind (const int *bcpind*) [inline]

Set the internal index of the variable.

Definition at line 176 of file BCP_var.hpp.

4.62.3.21 void BCP_var::set_status (const BCP_obj_status *status*) [inline]

Set the status of the variable.

Definition at line 183 of file BCP_var.hpp.

4.62.3.22 void BCP_var::dont_send_to_pool (bool *flag*) [inline]

Set/unset the flag controlling whether the variable could be sent to the Variable Pool process.

Definition at line 186 of file BCP_var.hpp.

4.62.3.23 void BCP_var::make_active ( ) [inline]

Mark the variable as active.

Note that when this method is invoked the lp formulation must be modified as well: the original bounds of the variable must be reset.

Definition at line 195 of file BCP_var.hpp.

4.62.3.24 void BCP_var::make_non_removable ( ) [inline]

Mark the variable as NotRemovable.

Definition at line 199 of file BCP_var.hpp.

4.62.3.25 void BCP_var::make_to_be_removed ( ) [inline]

Mark the variable as ToBeRemoved.

It will actually be removed immediately after all variables that have to be marked this way are marked.

Definition at line 207 of file BCP_var.hpp.

4.62.3.26 void BCP_var::display (const double *val*) const

Display the object type, internal index, and the value given in the argument.

(This value is usually the variable's value in an LP solution.)

4.62.4 Member Data Documentation

4.62.4.1 BCP_var_t BCP_var::_var_type [protected]

The integrality type of the variable.

Definition at line 55 of file BCP_var.hpp.

4.62.4.2 double BCP_var::_obj [protected]

The objective coefficient.

Definition at line 57 of file BCP_var.hpp.

4.62.4.3 double BCP_var::_lb `[protected]`

Lower bound on the value the variable can take.

Definition at line 59 of file BCP_var.hpp.

4.62.4.4 double BCP_var::_ub `[protected]`

Upper bound on the value the variable can take.

Definition at line 61 of file BCP_var.hpp.

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

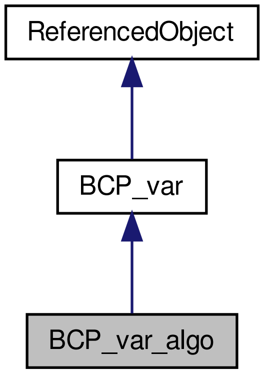
- BCP_var.hpp

4.63 BCP_var_algo Class Reference

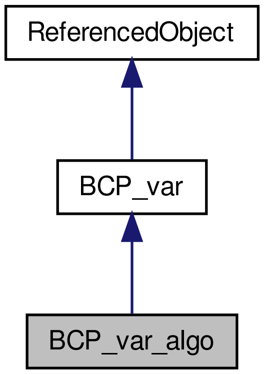
This is the class from which the user should derive her own algorithmic variables.

`#include <BCP_var.hpp>`

Inheritance diagram for BCP_var_algo:



Collaboration diagram for BCP_var_algo:



Public Member Functions

Constructor and destructor

- BCP_var_algo (const BCP_var_t var_type, const double obj, const double lb, const double ub)
  This constructor just sets the data members to the given values.
- virtual ∼BCP_var_algo ()=0
  The destructor deletes the object.

Query methods

- BCP_object_t obj_type () const
  Return `BCP_AlgoObj` indicating that the object is an algorithmic variable.

Additional Inherited Members

4.63.1 Detailed Description

This is the class from which the user should derive her own algorithmic variables.

Note that such an object cannot be constructed (it has pure virtual methods), only objects with types derived from BCP-_var_algo can be created. Such objects are constructed either directly by the user or by the unpacking functions of the `BCP_xx_user` classes.

Definition at line 277 of file BCP_var.hpp.

4.63.2 Constructor & Destructor Documentation

4.63.2.1 BCP_var_algo::BCP_var_algo (const BCP_var_t *var_type,* const double *obj,* const double *lb,* const double *ub*) `[inline]`

This constructor just sets the data members to the given values.

See also the constructor of BCP_var.

Definition at line 294 of file BCP_var.hpp.

4.63.2.2 virtual BCP_var_algo::∼BCP_var_algo ( ) [pure virtual]

The destructor deletes the object.

4.63.3 Member Function Documentation

4.63.3.1 BCP_object_t BCP_var_algo::obj_type ( ) const [inline], [virtual]

Return `BCP_AlgoObj` indicating that the object is an algorithmic variable.

Implements BCP_var.

Definition at line 305 of file BCP_var.hpp.

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

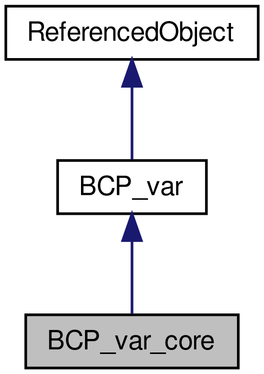
- BCP_var.hpp

4.64 BCP_var_core Class Reference

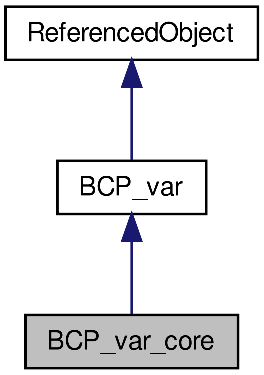
Core variables are the variables that always stay in the LP formulation.

`#include <BCP_var.hpp>`

Inheritance diagram for BCP_var_core:



Collaboration diagram for BCP_var_core:



Public Member Functions

Constructors and destructor

- BCP_var_core (const BCP_var_core &x)
  The copy constructor makes a replica of the argument.
- BCP_var_core (const BCP_var_t var_type, const double obj, const double lb, const double ub)
  This constructor just sets the data members to the given values.
- ∼BCP_var_core ()
  The destructor deletes the object.

Query methods

- BCP_object_t obj_type () const
  Return BCP_CoreObj indicating that the object is a core variable.

Additional Inherited Members

4.64.1 Detailed Description

Core variables are the variables that always stay in the LP formulation.

Therefore the data members in the base class are quite sufficient to describe the variable. The only thing that has to be done here is overriding the pure virtual method obj_type().

Definition at line 230 of file BCP_var.hpp.

4.64.2 Constructor & Destructor Documentation

4.64.2.1 BCP_var_core::BCP_var_core (const BCP_var_core & *x*) [inline]

The copy constructor makes a replica of the argument.

Definition at line 244 of file BCP_var.hpp.

4.64.2.2 BCP_var_core::BCP_var_core (const BCP_var_t *var_type,* const double *obj,* const double *lb,* const double *ub*) `[inline]`

This constructor just sets the data members to the given values.

See also the constructor BCP_var.

Definition at line 251 of file BCP_var.hpp.

4.64.2.3 BCP_var_core::∼BCP_var_core ( ) `[inline]`

The destructor deletes the object.

Definition at line 255 of file BCP_var.hpp.

4.64.3 Member Function Documentation

4.64.3.1 BCP_object_t BCP_var_core::obj_type ( ) const `[inline], [virtual]`

Return BCP_CoreObj indicating that the object is a core variable.

Implements BCP_var.

Definition at line 261 of file BCP_var.hpp.

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

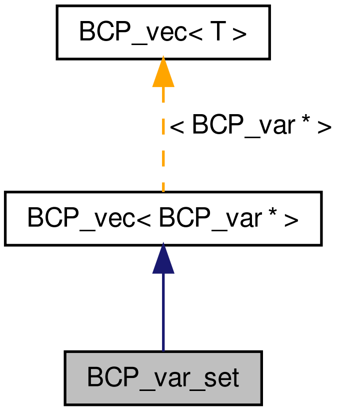
- BCP_var.hpp

4.65 BCP_var_set Class Reference

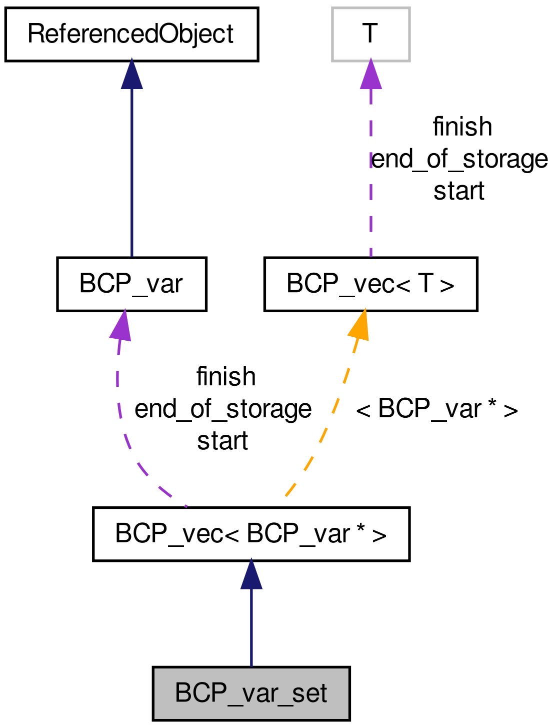
This class is just a collection of pointers to variables with a number of methods to manipulate these variables and/or select certain entries.

```
#include <BCP_var.hpp>
```

Inheritance diagram for BCP_var_set:



Collaboration diagram for BCP_var_set:



Public Member Functions

Constructor and destructor

- BCP_var_set ()

  The default constructor creates a variable set with no variables in it.
- ∼BCP_var_set ()

  The destructor empties the variable set.

Modifying methods

- void append (const BCP_vec< BCP_var ∗ > &x)

  Append the variables in the vector `x` *to the end of the variable set.*
- void append (BCP_var_set::const_iterator first, BCP_var_set::const_iterator last)

  Append the variables in `[first, last)` *to the end of the variable set.*
- void set_lb_ub (const BCP_vec< int > &pos, BCP_vec< double >::const_iterator bounds)

  Set the lower/upper bound pairs of the entries given by the contents of `pos` *to the values in* `[bounds, bounds + pos.`
- void set_lb_ub_st (const BCP_vec< BCP_obj_change > &vc)

  Set the lower/upper bound pairs and the stati of the first `cc`.

- void set_lb_ub_st (BCP_vec< int >::const_iterator pos, const BCP_vec< BCP_obj_change > &vc)
  Set the lower/upper bound pairs and the stati of the entries given by the content of `[pos, pos + cc`.

Methods related to deleting variables from the variable set

- void deletable (const int bvarnum, BCP_vec< int > &collection)
  Collect the indices of the variables marked to be removed.

Additional Inherited Members

4.65.1 Detailed Description

This class is just a collection of pointers to variables with a number of methods to manipulate these variables and/or select certain entries.

Definition at line 316 of file BCP_var.hpp.

4.65.2 Constructor & Destructor Documentation

4.65.2.1 BCP_var_set::∼BCP_var_set ( ) `[inline]`

The destructor empties the variable set.

NOTE: the destructor does NOT delete the variables the members of the variable set point to.

Definition at line 335 of file BCP_var.hpp.

4.65.3 Member Function Documentation

4.65.3.1 void BCP_var_set::append (const BCP_vec< BCP_var ∗ > & *x*) `[inline]`

Append the variables in the vector `x` to the end of the variable set.

Reimplemented from BCP_vec< BCP_var ∗ >.

Definition at line 342 of file BCP_var.hpp.

4.65.3.2 void BCP_var_set::append (BCP_var_set::const_iterator *first*, BCP_var_set::const_iterator *last*) `[inline]`

Append the variables in `[first, last)` to the end of the variable set.

Reimplemented from BCP_vec< BCP_var ∗ >.

Definition at line 347 of file BCP_var.hpp.

4.65.3.3 void BCP_var_set::set_lb_ub (const BCP_vec< int > & *pos*, BCP_vec< double >::const_iterator *bounds*)

Set the lower/upper bound pairs of the entries given by the contents of `pos` to the values in `[bounds, bounds + pos.`

`size()).`

4.65.3.4 void BCP_var_set::set_lb_ub_st (const BCP_vec< BCP_obj_change > & *vc*)

Set the lower/upper bound pairs and the stati of the first `cc`.

`size()` entries to the triplets given in the vector. This method is invoked when the variable set is all the variables in the current formulation and we want to change the triplets for the core variables, which are at the beginning of that variable set.

4.65.3.5 void BCP_var_set::set_lb_ub_st (BCP_vec< int >::const_iterator *pos*, const BCP_vec< BCP_obj_change > & *vc*)

Set the lower/upper bound pairs and the stati of the entries given by the content of `[pos, pos + cc.size())` to the triplets contained in `cc`.

4.65.3.6 void BCP_var_set::deletable (const int *bvarnum*, BCP_vec< int > & *collection*)

Collect the indices of the variables marked to be removed.

Since core variables are never removed, we pass the number of core variables in the first argument to speed up things a little.

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

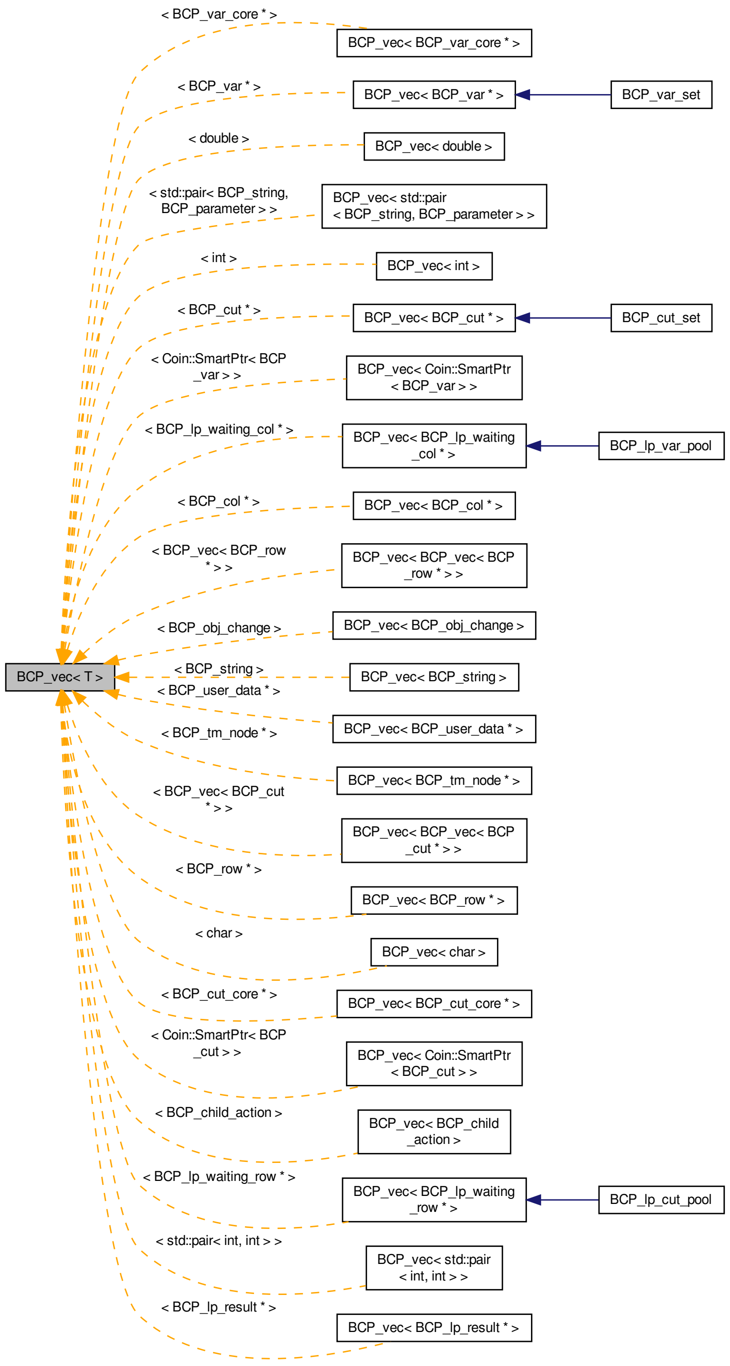
- BCP_var.hpp

4.66 BCP_vec< T > Class Template Reference

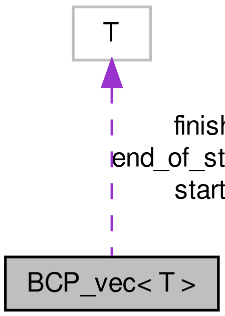
The class BCP_vec serves the same purpose as the vector class in the standard template library.

`#include <BCP_vector.hpp>`

Inheritance diagram for BCP_vec< T >:


BCP_vec< T >
< BCP_var_core * >
BCP_vec< BCP_var_core * >
< BCP_var * >
BCP_vec< BCP_var * >
BCP_var_set
< double >
BCP_vec< double >
< std::pair< BCP_string, BCP_parameter > >
BCP_vec< std::pair < BCP_string, BCP_parameter > >
< int >
BCP_vec< int >
< BCP_cut * >
BCP_vec< BCP_cut * >
BCP_cut_set
< Coin::SmartPtr< BCP _var > >
BCP_vec< Coin::SmartPtr < BCP_var > >
< BCP_lp_waiting_col * >
BCP_vec< BCP_lp_waiting _col * >
BCP_lp_var_pool
< BCP_col * >
BCP_vec< BCP_col * >
< BCP_vec< BCP_row * > >
BCP_vec< BCP_vec< BCP _row * > >
< BCP_obj_change >
BCP_vec< BCP_obj_change >
< BCP_string >
BCP_vec< BCP_string >
< BCP_user_data * >
BCP_vec< BCP_user_data * >
< BCP_tm_node * >
BCP_vec< BCP_tm_node * >
< BCP_vec< BCP_cut * > >
BCP_vec< BCP_vec< BCP _cut * > >
< BCP_row * >
BCP_vec< BCP_row * >
< char >
BCP_vec< char >
< BCP_cut_core * >
BCP_vec< BCP_cut_core * >
< Coin::SmartPtr< BCP _cut > >
BCP_vec< Coin::SmartPtr < BCP_cut > >
< BCP_child_action >
BCP_vec< BCP_child _action >
< BCP_lp_waiting_row * >
BCP_vec< BCP_lp_waiting _row * >
BCP_lp_cut_pool
< std::pair< int, int > >
BCP_vec< std::pair < int, int > >
< BCP_lp_result * >
BCP_vec< BCP_lp_result * >

Collaboration diagram for BCP_vec< T >:



Public Types

Type definitions (needed for using the STL)

- typedef size_t **size_type**
- typedef T **value_type**
- typedef T ∗ **iterator**
- typedef const T ∗ **const_iterator**
- typedef T & **reference**
- typedef const T & **const_reference**

Public Member Functions

Constructors / Destructor

- BCP_vec ()
  The default constructor initializes the data members as 0 pointers.
- BCP_vec (const BCP_vec< T > &x)
  The copy constructor copies over the content of `x`.
- BCP_vec (const size_t n, const_reference value=T())
  Construct a `BCP_vec` with `n` elements, all initialized with the second argument (or initialized with the default constructor of `T` if the second argument is missing).
- BCP_vec (const_iterator first, const_iterator last)
  Construct a `BCP_vec` by copying the elements from `first` to `last-1`.
- BCP_vec (const T ∗x, const size_t num)
  Construct a `BCP_vec` by copying `num` objects of type `T` from the memory starting at `x`.
- virtual ∼BCP_vec ()
  The destructor deallocates the memory allocated for the `BCP_vec`.

Query methods

- iterator begin ()
  Return an iterator to the beginning of the object.

- const_iterator begin () const

  Return a const iterator to the beginning of the object.
- iterator end ()

  Return an iterator to the end of the object.
- const_iterator end () const

  Return a const iterator to the end of the object.
- iterator entry (const int i)

  Return an iterator to the `i`-th entry.
- const_iterator entry (const int i) const

  Return a const iterator to the `i`-th entry.
- size_t index (const_iterator pos) const

  Return the index of the entry pointed to by `pos`.
- size_t size () const

  Return the current number of entries.
- size_t capacity () const

  Return the capacity of the object (space allocated for this many entries).
- bool empty () const

  Test if there are any entries in the object.
- reference operator[] (const size_t i)

  Return a reference to the `i`-th entry.
- const_reference operator[] (const size_t i) const

  Return a const reference to the `i`-th entry.
- reference front ()

  Return a reference to the first entry.
- const_reference front () const

  Return a const reference to the first entry.
- reference back ()

  Return a reference to the last entry.
- const_reference back () const

  Return a const reference to the last entry.

General modifying methods

- void reserve (const size_t n)

  Reallocate the object to make space for `n` entries.
- void swap (BCP_vec< T > &x)

  Exchange the contents of the object with that of `x`.
- BCP_vec< T > & operator= (const BCP_vec< T > &x)

  Copy the contents of `x` into the object and return a reference the the object itself.
- void assign (const void ∗x, const size_t num)

  Copy `num` entries of type `T` starting at the memory location `x` into the object.
- void insert (iterator position, const void ∗first, const size_t num)

  Insert `num` entries starting from memory location `first` into the vector from position `pos`.
- void insert (iterator position, const_iterator first, const_iterator last)

  Insert the entries `[first,last)` into the vector from position `pos`.
- void insert (iterator position, const size_t n, const_reference x)

  Insert `n` copies of `x` into the vector from position `pos`.
- iterator insert (iterator position, const_reference x)

  Insert `x` (a single entry) into the vector at position `pos`.
- void append (const BCP_vec< T > &x)

  Append the entries in `x` to the end of the vector.
- void append (const_iterator first, const_iterator last)

  Append the entries `[first,last)` to the end of the vector.
- void push_back (const_reference x)

Append x to the end of the vector.
- void unchecked_push_back (const_reference x)

  Append x to the end of the vector.
- void pop_back ()

  Delete the last entry.
- void clear ()

  Delete every entry.
- void update (const BCP_vec< int > &positions, const BCP_vec< T > &values)

  Update those entries listed in `positions` to the given `values`.
- void unchecked_update (const BCP_vec< int > &positions, const BCP_vec< T > &values)

  Same as the previous method but without sanity checks.

Methods for selectively keeping entries

- void keep (iterator pos)

  Keep only the entry pointed to by `pos`.
- void keep (iterator first, iterator last)

  Keep the entries `[first,last)`.
- void keep_by_index (const BCP_vec< int > &positions)

  Keep the entries indexed by `indices`.
- void unchecked_keep_by_index (const BCP_vec< int > &positions)

  Same as the previous method but without the sanity checks.
- void keep_by_index (const int ∗firstpos, const int ∗lastpos)

  Keep the entries indexed by the values in `[firstpos,lastpos)`.
- void unchecked_keep_by_index (const int ∗firstpos, const int ∗lastpos)

  Same as the previous method but without the sanity checks.

Methods for selectively erasing entries

- void erase (iterator pos)

  Erase the entry pointed to by `pos`.
- void erase (iterator first, iterator last)

  Erase the entries `[first,last)`.
- void erase_by_index (const BCP_vec< int > &positions)

  Erase the entries indexed by `indices`.
- void unchecked_erase_by_index (const BCP_vec< int > &positions)

  Same as the previous method but without the sanity check.
- void erase_by_index (const int ∗firstpos, const int ∗lastpos)

  Like the other `erase_by_index` method (including sanity checks), just the indices of the entries to be erased are given in `[firstpos,lastpos)`.
- void unchecked_erase_by_index (const int ∗firstpos, const int ∗lastpos)

  Same as the previous method but without the sanity checks.

Protected Member Functions

Internal methods

- iterator allocate (size_t len)

  allocate raw, uninitialized memory for `len` entries.
- void deallocate ()

  Destroy the entries in the vector and free the memory allocated for the vector.
- void insert_aux (iterator position, const_reference x)

  insert x into the given `position` in the vector.

Protected Attributes

Data members

- iterator start
  Iterator pointing to the beginning of the memory array where the vector is stored.
- iterator finish
  Iterator pointing to right after the last entry in the vector.
- iterator end_of_storage
  Iterator pointing to right after the last memory location usable by the vector without reallocation.

4.66.1 Detailed Description

template<class T>class BCP_vec< T >

The class BCP_vec serves the same purpose as the vector class in the standard template library.

The main difference is that while the vector class is *likely* to be implemented as a memory array, BCP_vec *is* implemented that way. Also, BCP_vec has a number of extra member methods, most of them exist to speed up operations (e.g., there are *unchecked* versions of the insert member methods, i.e., the method does not check whether there is enough space allocated to fit the new elements).

Definition at line 24 of file BCP_vector.hpp.

4.66.2 Constructor & Destructor Documentation

4.66.2.1 template<class T > BCP_vec< T >::BCP_vec ( )

The default constructor initializes the data members as 0 pointers.

Definition at line 65 of file BCP_vector_general.hpp.

4.66.2.2 template<class T> BCP_vec< T >::BCP_vec (const BCP_vec< T > & *x*)

The copy constructor copies over the content of `x`.

Definition at line 68 of file BCP_vector_general.hpp.

4.66.2.3 template<class T> BCP_vec< T >::BCP_vec (const size_t *n*, const_reference *value* = `T()`)

Construct a `BCP_vec` with `n` elements, all initialized with the second argument (or initialized with the default constructor of `T` if the second argument is missing).

Definition at line 75 of file BCP_vector_general.hpp.

4.66.2.4 template<class T> BCP_vec< T >::BCP_vec (const_iterator *first*, const_iterator *last*)

Construct a `BCP_vec` by copying the elements from `first` to `last-1`.

Definition at line 86 of file BCP_vector_general.hpp.

4.66.2.5 template<class T> BCP_vec< T >::BCP_vec (const T * *x*, const size_t *num*)

Construct a `BCP_vec` by copying `num` objects of type `T` from the memory starting at `x`.

Definition at line 97 of file BCP_vector_general.hpp.

4.66.2.6 template<class T> virtual BCP_vec< T >::~BCP_vec ( ) [inline], [virtual]

The destructor deallocates the memory allocated for the BCP_vec.

Definition at line 93 of file BCP_vector.hpp.

4.66.3 Member Function Documentation

4.66.3.1 template<class T > BCP_vec< T >::iterator BCP_vec< T >::allocate (size_t *len*) [inline], [protected]

allocate raw, uninitialized memory for `len` entries.

Definition at line 30 of file BCP_vector_general.hpp.

4.66.3.2 template<class T > void BCP_vec< T >::deallocate ( ) [inline], [protected]

Destroy the entries in the vector and free the memory allocated for the vector.

Definition at line 35 of file BCP_vector_general.hpp.

4.66.3.3 template<class T > void BCP_vec< T >::insert_aux (iterator *position,* const_reference *x*) [protected]

insert `x` into the given `position` in the vector.

Reallocate the vector if necessary.

Definition at line 43 of file BCP_vector_general.hpp.

4.66.3.4 template<class T> iterator BCP_vec< T >::begin ( ) [inline]

Return an iterator to the beginning of the object.

Definition at line 99 of file BCP_vector.hpp.

4.66.3.5 template<class T> const_iterator BCP_vec< T >::begin ( ) const [inline]

Return a const iterator to the beginning of the object.

Definition at line 101 of file BCP_vector.hpp.

4.66.3.6 template<class T> iterator BCP_vec< T >::end ( ) [inline]

Return an iterator to the end of the object.

Definition at line 104 of file BCP_vector.hpp.

4.66.3.7 template<class T> const_iterator BCP_vec< T >::end ( ) const [inline]

Return a const iterator to the end of the object.

Definition at line 106 of file BCP_vector.hpp.

4.66.3.8 template<class T> iterator BCP_vec< T >::entry (const int *i*) [inline]

Return an iterator to the `i`-th entry.

Definition at line 109 of file BCP_vector.hpp.

4.66.3.9 template<class T> const_iterator BCP_vec< T >::entry (const int *i*) const `[inline]`

Return a const iterator to the `i`-th entry.

Definition at line 111 of file BCP_vector.hpp.

4.66.3.10 template<class T> size_t BCP_vec< T >::index (const_iterator *pos*) const `[inline]`

Return the index of the entry pointed to by `pos`.

Definition at line 114 of file BCP_vector.hpp.

4.66.3.11 template<class T> size_t BCP_vec< T >::size ( ) const `[inline]`

Return the current number of entries.

Definition at line 116 of file BCP_vector.hpp.

4.66.3.12 template<class T> size_t BCP_vec< T >::capacity ( ) const `[inline]`

Return the capacity of the object (space allocated for this many entries).

Definition at line 119 of file BCP_vector.hpp.

4.66.3.13 template<class T> bool BCP_vec< T >::empty ( ) const `[inline]`

Test if there are any entries in the object.

Definition at line 121 of file BCP_vector.hpp.

4.66.3.14 template<class T> reference BCP_vec< T >::operator[] (const size_t *i*) `[inline]`

Return a reference to the `i`-th entry.

Definition at line 124 of file BCP_vector.hpp.

4.66.3.15 template<class T> const_reference BCP_vec< T >::operator[] (const size_t *i*) const `[inline]`

Return a const reference to the `i`-th entry.

Definition at line 126 of file BCP_vector.hpp.

4.66.3.16 template<class T> reference BCP_vec< T >::front ( ) `[inline]`

Return a reference to the first entry.

Definition at line 129 of file BCP_vector.hpp.

4.66.3.17 template<class T> const_reference BCP_vec< T >::front ( ) const `[inline]`

Return a const reference to the first entry.

Definition at line 131 of file BCP_vector.hpp.

4.66.3.18 template<class T> reference BCP_vec< T >::back ( ) `[inline]`

Return a reference to the last entry.

Definition at line 133 of file BCP_vector.hpp.

4.66.3.19 template<class T> const_reference BCP_vec< T >::back ( ) const [inline]

Return a const reference to the last entry.

Definition at line 135 of file BCP_vector.hpp.

4.66.3.20 template<class T > void BCP_vec< T >::reserve (const size_t *n*)

Reallocate the object to make space for `n` entries.

Definition at line 112 of file BCP_vector_general.hpp.

4.66.3.21 template<class T> void BCP_vec< T >::swap (BCP_vec< T > & *x*) [inline]

Exchange the contents of the object with that of `x`.

Definition at line 124 of file BCP_vector_general.hpp.

4.66.3.22 template<class T> BCP_vec< T > & BCP_vec< T >::operator= (const BCP_vec< T > & *x*)

Copy the contents of `x` into the object and return a reference the the object itself.

Definition at line 131 of file BCP_vector_general.hpp.

4.66.3.23 template<class T > void BCP_vec< T >::assign (const void ∗ *x*, const size_t *num*)

Copy `num` entries of type `T` starting at the memory location `x` into the object.

(`x` is a void pointer since it might be located somewhere in a buffer and therefore might not be aligned for type `T` entries.)

Definition at line 157 of file BCP_vector_general.hpp.

4.66.3.24 template<class T > void BCP_vec< T >::insert (iterator *position*, const void ∗ *first*, const size_t *num*)

Insert `num` entries starting from memory location `first` into the vector from position `pos`.

Definition at line 176 of file BCP_vector_general.hpp.

4.66.3.25 template<class T > void BCP_vec< T >::insert (iterator *position*, const_iterator *first*, const_iterator *last*)

Insert the entries `[first,last)` into the vector from position `pos`.

Definition at line 225 of file BCP_vector_general.hpp.

4.66.3.26 template<class T > void BCP_vec< T >::insert (iterator *position*, const size_t *n*, const_reference *x*)

Insert `n` copies of `x` into the vector from position `pos`.

Definition at line 256 of file BCP_vector_general.hpp.

4.66.3.27 template<class T > BCP_vec< T >::iterator BCP_vec< T >::insert (iterator *position*, const_reference *x*) [inline]

Insert `x` (a single entry) into the vector at position `pos`.

Return an iterator to the newly inserted entry.

Definition at line 287 of file BCP_vector_general.hpp.

4.66.3.28 template<class T> void BCP_vec< T >::append (const BCP_vec< T > & *x*) [inline]

Append the entries in `x` to the end of the vector.

Reimplemented in BCP_var_set, and BCP_cut_set.

Definition at line 169 of file BCP_vector.hpp.

4.66.3.29 template<class T > void BCP_vec< T >::push_back (const_reference *x*) [inline]

Append `x` to the end of the vector.

Check if enough space is allocated (reallocate if necessary).

Definition at line 300 of file BCP_vector_general.hpp.

4.66.3.30 template<class T > void BCP_vec< T >::unchecked_push_back (const_reference *x*) [inline]

Append `x` to the end of the vector.

Does not check if enough space is allcoated.

Definition at line 308 of file BCP_vector_general.hpp.

4.66.3.31 template<class T > void BCP_vec< T >::pop_back ( ) [inline]

Delete the last entry.

Definition at line 313 of file BCP_vector_general.hpp.

4.66.3.32 template<class T > void BCP_vec< T >::clear ( ) [inline]

Delete every entry.

Definition at line 318 of file BCP_vector_general.hpp.

4.66.3.33 template<class T> void BCP_vec< T >::update (const BCP_vec< int > & *positions,* const BCP_vec< T > & *values*) [inline]

Update those entries listed in `positions` to the given `values`.

The two argument vector must be of equal length. Sanity checks are done on the given positions.

Definition at line 336 of file BCP_vector_general.hpp.

4.66.3.34 template<class T> void BCP_vec< T >::unchecked_update (const BCP_vec< int > & *positions,* const BCP_vec< T > & *values*) [inline]

Same as the previous method but without sanity checks.

Definition at line 324 of file BCP_vector_general.hpp.

4.66.3.35 template<class T > void BCP_vec< T >::keep (iterator *pos*) [inline]

Keep only the entry pointed to by `pos`.

Definition at line 347 of file BCP_vector_general.hpp.

4.66.3.36 template<class T > void BCP_vec< T >::keep (iterator *first,* iterator *last*) [inline]

Keep the entries `[first,last)`.

Definition at line 354 of file BCP_vector_general.hpp.

4.66.3.37 template<class T > void BCP_vec< T >::keep_by_index (const BCP_vec< int > & *positions*) [inline]

Keep the entries indexed by `indices`.

Abort if the indices are not in increasing order, if there are duplicate indices or if any of the indices is outside of the range `[0,size())`.

Definition at line 362 of file BCP_vector_general.hpp.

4.66.3.38 template<class T > void **BCP_vec**< T >::unchecked_keep_by_index (const **BCP_vec**< int > & *positions*) [inline]

Same as the previous method but without the sanity checks.

Definition at line 369 of file BCP_vector_general.hpp.

4.66.3.39 template<class T> void **BCP_vec**< T >::keep_by_index (const int ∗ *firstpos,* const int ∗ *lastpos*) [inline]

Keep the entries indexed by the values in `[firstpos,lastpos)`.

Abort if the indices are not in increasing order, if there are duplicate indices or if any of the indices is outside of the range `[0,size())`.

4.66.3.40 template<class T> void **BCP_vec**< T >::unchecked_keep_by_index (const int ∗ *firstpos,* const int ∗ *lastpos*)

Same as the previous method but without the sanity checks.

4.66.3.41 template<class T > void **BCP_vec**< T >::erase (iterator *pos*) [inline]

Erase the entry pointed to by `pos`.

Definition at line 399 of file BCP_vector_general.hpp.

4.66.3.42 template<class T > void **BCP_vec**< T >::erase (iterator *first,* iterator *last*) [inline]

Erase the entries `[first,last)`.

Definition at line 406 of file BCP_vector_general.hpp.

4.66.3.43 template<class T > void **BCP_vec**< T >::erase_by_index (const **BCP_vec**< int > & *positions*) [inline]

Erase the entries indexed by `indices`.

Abort if the indices are not in increasing order, if there are duplicate indices or if any of the indices is outside of the range `[0,size())`.

Definition at line 414 of file BCP_vector_general.hpp.

4.66.3.44 template<class T > void **BCP_vec**< T >::unchecked_erase_by_index (const **BCP_vec**< int > & *positions*) [inline]

Same as the previous method but without the sanity check.

Definition at line 421 of file BCP_vector_general.hpp.

4.66.3.45 template<class T> void **BCP_vec**< T >::erase_by_index (const int ∗ *firstpos,* const int ∗ *lastpos*) [inline]

Like the other `erase_by_index` method (including sanity checks), just the indices of the entries to be erased are given in `[firstpos,lastpos)`.

4.66.3.46 template<class T> void **BCP_vec**< T >::unchecked_erase_by_index (const int ∗ *firstpos,* const int ∗ *lastpos*)

Same as the previous method but without the sanity checks.

4.66.4 Member Data Documentation

4.66.4.1 template<class T> iterator BCP_vec< T >::start [protected]

Iterator pointing to the beginning of the memory array where the vector is stored.

Definition at line 66 of file BCP_vector.hpp.

4.66.4.2 template<class T> iterator BCP_vec< T >::finish [protected]

Iterator pointing to right after the last entry in the vector.

Definition at line 68 of file BCP_vector.hpp.

4.66.4.3 template<class T> iterator BCP_vec< T >::end_of_storage [protected]

Iterator pointing to right after the last memory location usable by the vector without reallocation.

Definition at line 71 of file BCP_vector.hpp.

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

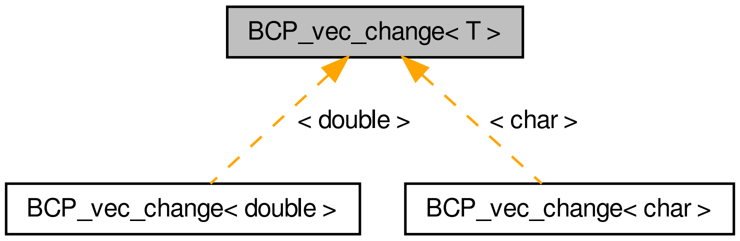
- BCP_vector.hpp
- BCP_vector_general.hpp

4.67 BCP_vec_change< T > Class Template Reference

This class stores a vector explicitly or relatively to another vector.

`#include <BCP_vector_change.hpp>`

Inheritance diagram for BCP_vec_change< T >:



Public Member Functions

Constructors and destructor

- BCP_vec_change (const BCP_storage_t st)
  Construct a vector change that's empty and is of the given storage.
- BCP_vec_change (const T ∗first, const T ∗last)
  Construct an explicit description describing the vector bounded by the two iterators.

- BCP_vec_change (const BCP_vec_change< T > &old_vec, const BCP_vec_change< T > &new_vec, const BCP_vec< int > &del_pos)

  Construct a relative description.
- BCP_vec_change (const BCP_vec_change< T > &old_vec, const BCP_vec_change< T > &new_vec, const BCP_vec< int > &del_pos, const double etol)

  Construct a relative description.
- BCP_vec_change (BCP_buffer &buf)

  Construct the object by unpacking it from a buffer.
- ∼BCP_vec_change ()

  The destructor need not do anything.

Query methods

- BCP_storage_t storage () const

  Return the storage type of the vector.
- const BCP_vec< T > & explicit_vector () const

  Return a const reference to the vector if it is explicitly stored, otherwise throw an exception.
- int storage_size () const

  Return how much memory it'll take to pack this info.

Modifying methods

- void update (const BCP_vec_change< T > &change)

  Update the current vector with the argument vector.

Packing and unpacking

- void pack (BCP_buffer &buf) const

  Pack the data into a buffer.
- void unpack (BCP_buffer &buf)

  Unpack the data from a buffer.

4.67.1 Detailed Description

template<class T>class BCP_vec_change< T >

This class stores a vector explicitly or relatively to another vector.

The class can be used when we want to store a sequence of vectors in as small a space as possible. The first vector must be stored explicitly, the others can be either explicit ones or stored with respect to the previous vector.

Definition at line 20 of file BCP_vector_change.hpp.

4.67.2 Constructor & Destructor Documentation

4.67.2.1 template<class T> BCP_vec_change< T >::BCP_vec_change (const BCP_storage_t *st*) [inline]

Construct a vector change that's empty and is of the given storage.

If the storage is explicit then this creates an explicit vector with no entry. If it is relative, then the data describes that there is no change with respect to the parent vector.

Definition at line 43 of file BCP_vector_change.hpp.

4.67.2.2 template<class T> BCP_vec_change< T >::BCP_vec_change (const T ∗ *first*, const T ∗ *last*) [inline]

Construct an explicit description describing the vector bounded by the two iterators.

Definition at line 47 of file BCP_vector_change.hpp.

4.67.2.3 template<class T> BCP_vec_change< T >::BCP_vec_change (const BCP_vec_change< T > & *old_vec*, const BCP_vec_change< T > & *new_vec*, const BCP_vec< int > & *del_pos*) [inline]

Construct a relative description.

Parameters

new_vec	the vector that should result after the change is applied to
old_vec	the original vector
del_pos	specifies which entries are to be deleted from old_vec before the change in this object can be applied.

Definition at line 59 of file BCP_vector_change.hpp.

4.67.2.4 template<class T> BCP_vec_change< T >::BCP_vec_change (const BCP_vec_change< T > & *old_vec*, const BCP_vec_change< T > & *new_vec*, const BCP_vec< int > & *del_pos*, const double *etol*) [inline]

Construct a relative description.

Same as the previous constructor, except that there is an extra argument, the epsilon tolerance for equality testing. This makes sense only if `is double`, so this constructor should be used only in that case.

Definition at line 100 of file BCP_vector_change.hpp.

4.67.2.5 template<class T> BCP_vec_change< T >::BCP_vec_change (BCP_buffer & *buf*) [inline]

Construct the object by unpacking it from a buffer.

Definition at line 137 of file BCP_vector_change.hpp.

4.67.2.6 template<class T> BCP_vec_change< T >::∼BCP_vec_change ( ) [inline]

The destructor need not do anything.

Definition at line 142 of file BCP_vector_change.hpp.

4.67.3 Member Function Documentation

4.67.3.1 template<class T> int BCP_vec_change< T >::storage_size ( ) const [inline]

Return how much memory it'll take to pack this info.

It is used when comparing which sort of storage is smaller

Definition at line 161 of file BCP_vector_change.hpp.

4.67.3.2 template<class T> void BCP_vec_change< T >::update (const BCP_vec_change< T > & *change*) [inline]

Update the current vector with the argument vector.

If the argument vector is explicit then just replace the current vector. If it is relative and the current vector is explicit, then perform the update. Otherwise throw an exception (the operation is impossible.)

Definition at line 175 of file BCP_vector_change.hpp.

4.67.3.3 template<class T> void BCP_vec_change< T >::pack (BCP_buffer & *buf*) const [inline]

Pack the data into a buffer.

Definition at line 208 of file BCP_vector_change.hpp.

4.67.3.4 template<class T> void BCP_vec_change< T >::unpack (BCP_buffer & *buf*) [inline]

Unpack the data from a buffer.

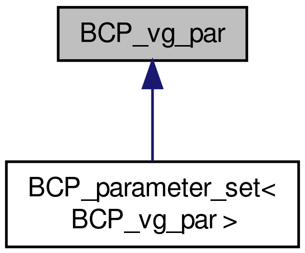
Definition at line 213 of file BCP_vector_change.hpp.

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

- BCP_vector_change.hpp

4.68 BCP_vg_par Struct Reference

Inheritance diagram for BCP_vg_par:



Public Types

- enum chr_params { MessagePassingIsSerial, ReportWhenDefaultIsExecuted }
- enum int_params { NiceLevel }

4.68.1 Detailed Description

Definition at line 6 of file BCP_vg_param.hpp.

4.68.2 Member Enumeration Documentation

4.68.2.1 enum BCP_vg_par::chr_params

Enumerator:

MessagePassingIsSerial Indicates whether message passing is serial (all processes are on the same processor) or not.

Values: true (1), false (0). Default: 1.

ReportWhenDefaultIsExecuted Print out a message when the default version of an overridable method is executed. Default: 1.

Definition at line 7 of file BCP_vg_param.hpp.

4.68.2.2 enum BCP_vg_par::int_params

Enumerator:

NiceLevel The "nice" level the process should run at. On Linux and on AIX this value should be between -20 and 20. The higher this value the less resource the process will receive from the system. Note that

1. The process starts with 0 priority and it can only be increased.
2. If the load is low on the machine (e.g., when all other processes are interactive like netscape or text editors) then even with 20 priority the process will use close to 100% of the cpu, and the interactive processes will be noticably more responsive than they would be if this process ran with 0 priority.

See Also

the man page of the `setpriority` system function.

Definition at line 18 of file BCP_vg_param.hpp.

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

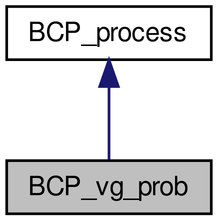
- BCP_vg_param.hpp

4.69 BCP_vg_prob Class Reference

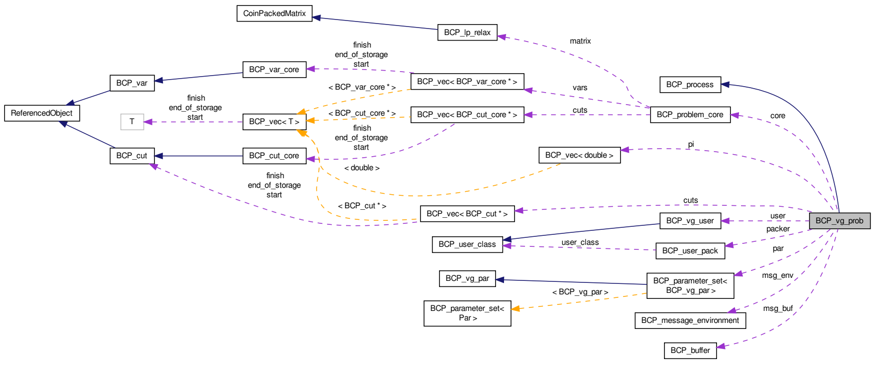
This class is the central class of the Variable Generator process.

`#include <BCP_vg.hpp>`

Inheritance diagram for BCP_vg_prob:



Collaboration diagram for BCP_vg_prob:



Public Member Functions

- BCP_cut ∗ unpack_cut ()

  Unpack a cut.

Constructor and destructor

- BCP_vg_prob (int my_id, int parent)

  The default constructor.
- virtual ∼BCP_vg_prob ()

  The destructor deletes everything.

Query methods

- bool has_ub () const

  Return true/false indicating whether any upper bound has been found.
- double ub () const

  Return the current upper bound (`BCP_DBL_MAX/10` if there's no upper bound found yet.)

Modifying methods

- void ub (const double bd)

  Set the upper bound equal to the argument.
- bool probe_messages ()

  Test if there is a message in the message queue waiting to be processed.

Public Attributes

Data members

- BCP_vg_user ∗ user

  The user object holding the user's data.
- BCP_user_pack ∗ packer

  A class that holds the methods about how to pack things.
- BCP_message_environment ∗ msg_env

The message passing environment.

- BCP_buffer msg_buf

  The message buffer of the Variable Generator process.
- BCP_parameter_set< BCP_vg_par > par

  The parameters controlling the Variable Generator process.
- BCP_problem_core ∗ core

  The description of the core of the problem.
- double upper_bound

  The proc id of the Tree Manager.
- BCP_vec< BCP_cut ∗ > cuts

  Variables are to be generated for the LP solution given by these cuts and their values (next member).
- BCP_vec< double > pi

  The dual values corresponding to the cuts above.
- int sender

  The process id of the LP process that sent the solution.
- int phase

  The phase the algorithm is in.
- int node_level

  The level of search tree node where the solution was generated.
- int node_index

  The index of search tree node where the solution was generated.
- int node_iteration

  The iteration within the search tree node where the solution was generated.

4.69.1 Detailed Description

This class is the central class of the Variable Generator process.

Only one object of this type is created and that holds all the data in the VG process. A reference to that object is passed to (almost) every function (or member method) that's invoked within the VG process.

Definition at line 32 of file BCP_vg.hpp.

4.69.2 Constructor & Destructor Documentation

4.69.2.1 BCP_vg_prob::BCP_vg_prob (int *my_id,* int *parent*)

The default constructor.

Initializes every data member to a natural state.

4.69.2.2 virtual BCP_vg_prob::∼BCP_vg_prob ( ) `[virtual]`

The destructor deletes everything.

4.69.3 Member Function Documentation

4.69.3.1 bool BCP_vg_prob::has_ub ( ) const `[inline]`

Return true/false indicating whether any upper bound has been found.

Definition at line 113 of file BCP_vg.hpp.

4.69.3.2 void BCP_vg_prob::ub (const double *bd*) [inline]

Set the upper bound equal to the argument.

Definition at line 122 of file BCP_vg.hpp.

4.69.3.3 bool BCP_vg_prob::probe_messages ( )

Test if there is a message in the message queue waiting to be processed.

4.69.3.4 BCP_cut* BCP_vg_prob::unpack_cut ( )

Unpack a cut.

Invoked from the built-in BCP_vg_user::unpack_dual_solution().

4.69.4 Member Data Documentation

4.69.4.1 BCP_vg_user* BCP_vg_prob::user

The user object holding the user's data.

This object is created by a call to the appropriate member of [`USER_initialize`]{USER_initialize.html}.

Definition at line 49 of file BCP_vg.hpp.

4.69.4.2 BCP_user_pack* BCP_vg_prob::packer

A class that holds the methods about how to pack things.

Definition at line 52 of file BCP_vg.hpp.

4.69.4.3 BCP_message_environment* BCP_vg_prob::msg_env

The message passing environment.

This object is created by a call to the appropriate member of [`USER_initialize`]{USER_initialize.html}.

Definition at line 57 of file BCP_vg.hpp.

4.69.4.4 BCP_buffer BCP_vg_prob::msg_buf

The message buffer of the Variable Generator process.

Definition at line 60 of file BCP_vg.hpp.

4.69.4.5 BCP_parameter_set<BCP_vg_par> BCP_vg_prob::par

The parameters controlling the Variable Generator process.

Definition at line 63 of file BCP_vg.hpp.

4.69.4.6 BCP_problem_core* BCP_vg_prob::core

The description of the core of the problem.

Definition at line 66 of file BCP_vg.hpp.

4.69.4.7 double BCP_vg_prob::upper_bound

The proc id of the Tree Manager.

The best currently known upper bound.

Definition at line 72 of file BCP_vg.hpp.

4.69.4.8 BCP_vec<BCP_cut*> BCP_vg_prob::cuts

Variables are to be generated for the LP solution given by these cuts and their values (next member).

Not all cuts need to be listed (e.g., list only those that have nonzero dual values in the current solution).

See Also

BCP_lp_user::pack_dual_solution()

Definition at line 82 of file BCP_vg.hpp.

4.69.4.9 BCP_vec<double> BCP_vg_prob::pi

The dual values corresponding to the cuts above.

Definition at line 84 of file BCP_vg.hpp.

4.69.4.10 int BCP_vg_prob::sender

The process id of the LP process that sent the solution.

Definition at line 86 of file BCP_vg.hpp.

4.69.4.11 int BCP_vg_prob::phase

The phase the algorithm is in.

Definition at line 89 of file BCP_vg.hpp.

4.69.4.12 int BCP_vg_prob::node_level

The level of search tree node where the solution was generated.

Definition at line 91 of file BCP_vg.hpp.

4.69.4.13 int BCP_vg_prob::node_index

The index of search tree node where the solution was generated.

Definition at line 93 of file BCP_vg.hpp.

4.69.4.14 int BCP_vg_prob::node_iteration

The iteration within the search tree node where the solution was generated.

Definition at line 96 of file BCP_vg.hpp.

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

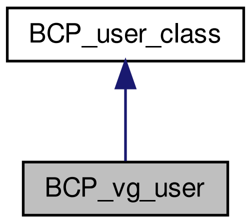
- BCP_vg.hpp

4.70 BCP_vg_user Class Reference

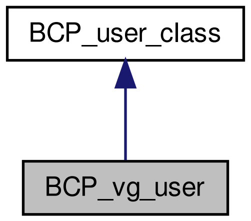
The BCP_vg_user class is the base class from which the user can derive a problem specific class to be used in the Cut Generator process.

```
#include <BCP_vg_user.hpp>
```

Inheritance diagram for BCP_vg_user:



Collaboration diagram for BCP_vg_user:



Public Member Functions

- void send_var (const BCP_var &var)

  Pack the argument into the message buffer and send it to the sender of the LP solution.
- virtual void unpack_module_data (BCP_buffer &buf)

  Unpack the initial information sent to the Variable Generator process by the Tree Manager.
- virtual void unpack_dual_solution (BCP_buffer &buf)

  Unpack the LP solution arriving from the LP process.
- virtual void generate_vars (BCP_vec< BCP_cut $*$ > &cuts, BCP_vec< double > &pi)

  Perform the actual variable generation.

Methods to set and get the pointer to the BCP_vg_prob

object.

It is unlikely that the users would want to muck around with these (especially with the set method!) but they are here to provide total control.

- void setVgProblemPointer (BCP_vg_prob ∗ptr)
  Set the pointer.
- BCP_vg_prob ∗ getVgProblemPointer ()
  Get the pointer.

Informational methods for the user.

- double upper_bound () const
  Return what is the best known upper bound (might be BCP_DBL_MAX)
- int current_phase () const
  Return the phase the algorithm is in.
- int current_level () const
  Return the level of the search tree node for which cuts are being generated.
- int current_index () const
  Return the internal index of the search tree node for which cuts are being generated.
- int current_iteration () const
  Return the iteration count within the search tree node for which cuts are being generated.

Methods to get/set BCP parameters on the fly

- char **get_param** (const BCP_vg_par::chr_params key) const
- int **get_param** (const BCP_vg_par::int_params key) const
- double **get_param** (const BCP_vg_par::dbl_params key) const
- const BCP_string & **get_param** (const BCP_vg_par::str_params key) const
- void **set_param** (const BCP_vg_par::chr_params key, const bool val)
- void **set_param** (const BCP_vg_par::chr_params key, const char val)
- void **set_param** (const BCP_vg_par::int_params key, const int val)
- void **set_param** (const BCP_vg_par::dbl_params key, const double val)
- void **set_param** (const BCP_vg_par::str_params key, const char ∗val)

Constructor, Destructor

- **BCP_vg_user** ()
- virtual ∼BCP_vg_user ()
  Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

4.70.1 Detailed Description

The BCP_vg_user class is the base class from which the user can derive a problem specific class to be used in the Cut Generator process.

In that derived class the user can store data to be used in the methods she overrides. Also that is the object the user must return in the USER_initialize::vg_init() method.

There are two kind of methods in the class. The non-virtual methods are helper functions for the built-in defaults, but the user can use them as well. The virtual methods execute steps in the BCP algorithm where the user might want to override the default behavior.

The default implementations fall into three major categories.

- Empty; doesn't do anything and immediately returns (e.g., unpack_module_data()). `unpack_module_data.`)
- There is no reasonable default, so throw an exception. This happens if the parameter settings drive the flow of in a way that BCP can't perform the necessary function. This behavior is correct since such methods are invoked only if the parameter settings drive the flow of the algorithm that way, in which case the user better implement those methods. (At the momemnt there is no such method in VG.)
- A default is given. Frequently there are multiple defaults and parameters govern which one is selected (e.g., unpack_dual_solution()).

Definition at line 51 of file BCP_vg_user.hpp.

4.70.2 Constructor & Destructor Documentation

4.70.2.1 virtual BCP_vg_user::∼BCP_vg_user ( ) `[inline]`, `[virtual]`

Being virtual, the destructor invokes the destructor for the real type of the object being deleted.

Definition at line 120 of file BCP_vg_user.hpp.

4.70.3 Member Function Documentation

4.70.3.1 void BCP_vg_user::send_var (const **BCP_var** & *var*)

Pack the argument into the message buffer and send it to the sender of the LP solution.

Whenever the user generates a variable in the generate_vars() method she should invoke this method to immediately send off the variable to the LP process.

4.70.3.2 virtual void BCP_vg_user::unpack_module_data (**BCP_buffer** & *buf*) `[virtual]`

Unpack the initial information sent to the Variable Generator process by the Tree Manager.

This information was packed by the method BCP_tm_user::pack_module_data() invoked with `BCP_ProcessType-_VG` as the third (target process type) argument.

Default: empty method.

4.70.3.3 virtual void BCP_vg_user::unpack_dual_solution (**BCP_buffer** & *buf*) `[virtual]`

Unpack the LP solution arriving from the LP process.

This method is invoked only if the user packs the info necessary for variable generation by herself, i.e., she overrides the BCP_lp_user::pack_dual_solution() method. If that's the case the user has to unpack the same info she has packed in the LP process.

4.70.3.4 virtual void BCP_vg_user::generate_vars (**BCP_vec**< **BCP_cut** ∗ > & *cuts*, **BCP_vec**< double > & *pi*) `[virtual]`

Perform the actual variable generation.

Whenever a variable is generated, the user should invoke the send_var() method to send the generated variable back to the LP process.

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

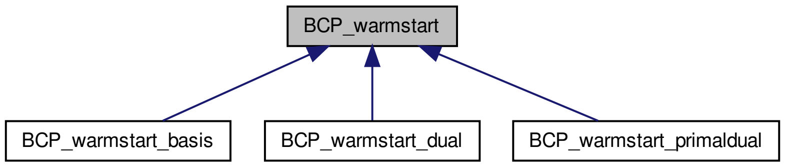
- BCP_vg_user.hpp

4.71 BCP_warmstart Class Reference

Warmstarting information for the LP solver.

```
#include <BCP_warmstart.hpp>
```

Inheritance diagram for BCP_warmstart:



Public Member Functions

- virtual ∼BCP_warmstart ()

  The destructor is pure virtual.
- virtual **CoinWarmStart** ∗ convert_to_CoinWarmStart () const =0

  Return an OsiWarmStart object that can be fed to the LP engine.
- virtual BCP_storage_t storage () const =0

  Return how the warmstarting info is stored.
- virtual void update (const BCP_warmstart ∗const change)=0

  Update the current data with the one in the argument.
- virtual BCP_warmstart ∗ as_change (const BCP_warmstart ∗const old_ws, const BCP_vec< int > &del_vars, const BCP_vec< int > &del_cuts, const double petol, const double detol) const =0

  Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.
- virtual BCP_warmstart ∗ clone () const =0

  Make a replica of the current warmstart information.
- virtual BCP_warmstart ∗ empty_wrt_this () const =0

  Create a warmstart info describing that no change should be done.
- virtual int storage_size () const =0

  Return how much memory it'll take to pack this warmstart info.

4.71.1 Detailed Description

Warmstarting information for the LP solver.

A realization of the warmstarting information must done in a way that allows to keep the information either in an explicit way or as a change relative to another warmstarting information.

Definition at line 24 of file BCP_warmstart.hpp.

4.71.2 Constructor & Destructor Documentation

4.71.2.1 virtual BCP_warmstart::∼BCP_warmstart ( ) `[inline], [virtual]`

The destructor is pure virtual.

(There are no data members here.)

Definition at line 27 of file BCP_warmstart.hpp.

4.71.3 Member Function Documentation

4.71.3.1 virtual BCP_storage_t BCP_warmstart::storage ( ) const [pure virtual]

Return how the warmstarting info is stored.

Implemented in BCP_warmstart_basis, BCP_warmstart_primaldual, and BCP_warmstart_dual.

4.71.3.2 virtual void BCP_warmstart::update (const **BCP_warmstart** *const *change*) [pure virtual]

Update the current data with the one in the argument.

If the argument is of explicit storage then just replace the current data. If it is relative and the current data is explicit then perform the update. Otherwise throw an exception.

Implemented in BCP_warmstart_primaldual, BCP_warmstart_basis, and BCP_warmstart_dual.

4.71.3.3 virtual **BCP_warmstart*** BCP_warmstart::as_change (const **BCP_warmstart** *const *old_ws,* const **BCP_vec**< int > & *del_vars,* const **BCP_vec**< int > & *del_cuts,* const double *petol,* const double *detol*) const [pure virtual]

Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.

However, if the currently stored data is shorter to store than the change, then this method can return a copy of the current data! The current data must be explicitly stored and `old_ws` must be either explicit or can contain no data. Otherwise an exception must be thrown.

Parameters

old_ws	the old warmstart info
del_vars	the indices of the variables that are deleted from the formulation `old_ws` was created for
del_cuts	same for the cuts
petol	primal zero tolerance
detol	dual zero tolerance

Implemented in BCP_warmstart_primaldual, BCP_warmstart_basis, and BCP_warmstart_dual.

4.71.3.4 virtual **BCP_warmstart*** BCP_warmstart::clone ( ) const [pure virtual]

Make a replica of the current warmstart information.

Implemented in BCP_warmstart_primaldual, BCP_warmstart_basis, and BCP_warmstart_dual.

4.71.3.5 virtual **BCP_warmstart*** BCP_warmstart::empty_wrt_this ( ) const [pure virtual]

Create a warmstart info describing that no change should be done.

This is really the task of a constructor, but BCP does not know the type of warmstart the user will use, so it will invoke this method for a warmstart that was created by the user. Tricky, isn't it?

Implemented in BCP_warmstart_primaldual, BCP_warmstart_basis, and BCP_warmstart_dual.

4.71.3.6 virtual int BCP_warmstart::storage_size ( ) const [pure virtual]

Return how much memory it'll take to pack this warmstart info.

It is used when comparing which sort of storage is smaller.

Implemented in BCP_warmstart_primaldual, BCP_warmstart_basis, and BCP_warmstart_dual.

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

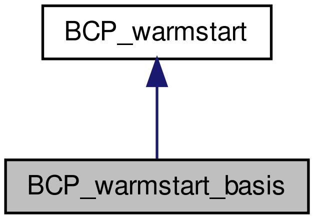
- BCP_warmstart.hpp

4.72 BCP_warmstart_basis Class Reference

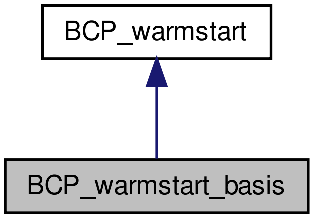
This class describes a warmstart information that consists of basis information for structural and artificial variables.

`#include <BCP_warmstart_basis.hpp>`

Inheritance diagram for BCP_warmstart_basis:



Collaboration diagram for BCP_warmstart_basis:



Public Member Functions

- virtual **CoinWarmStart** ∗ convert_to_CoinWarmStart () const

  Return an CoinwarmStart object that can be fed to the LP engine.
- virtual BCP_storage_t storage () const

  Return how the warmstarting info is stored.
- virtual BCP_warmstart ∗ clone () const

  Make a replica of the current warmstart information.

- virtual BCP_warmstart $*$ empty_wrt_this () const

  Create a warmstart info describing that no change should be done.
- virtual int storage_size () const

  Return how much memory it'll take to pack this warmstart info.
- virtual void update (const BCP_warmstart $*$const change)

  Update the current data with the one in the argument.
- virtual BCP_warmstart $*$ as_change (const BCP_warmstart $*$const old_ws, const BCP_vec$<$ int $>$ &del_vars, const BCP_vec$<$ int $>$ &del_cuts, const double petol, const double detol) const

  Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.
- void pack (BCP_buffer &buf) const

  Pack the warmstart info into a buffer.

Constructors and destructor

- BCP_warmstart_basis (BCP_buffer &buf)

  Create the object by unpacking it from a buffer.
- BCP_warmstart_basis (const char $*$vfirst, const char $*$vlast, const char $*$cfirst, const char $*$clast)

  Create an explicitly stored warmstart info by considering the two character arrays (`[vfirst,vlast)` and `[cfirst,clast)`) as the status arrays for the variables/cuts.
- BCP_warmstart_basis (const BCP_warmstart_basis &ws)

  Copy constructor.
- ~**BCP_warmstart_basis** ()

4.72.1 Detailed Description

This class describes a warmstart information that consists of basis information for structural and artificial variables.

"basic / on_upper / on_lower" info is stored on 2 bits for every variable. Only the added methods are documented in details on this page, for the inherited methods see the description of the base class.

Definition at line 24 of file BCP_warmstart_basis.hpp.

4.72.2 Member Function Documentation

4.72.2.1 virtual CoinWarmStart$*$ BCP_warmstart_basis::convert_to_CoinWarmStart ( ) const [virtual]

Return an CoinwarmStart object that can be fed to the LP engine.

The implementation for this class will return an CoinwarmStartBasis object.

Implements BCP_warmstart.

4.72.2.2 virtual BCP_storage_t BCP_warmstart_basis::storage ( ) const [virtual]

Return how the warmstarting info is stored.

Implements BCP_warmstart.

4.72.2.3 virtual BCP_warmstart$*$ BCP_warmstart_basis::clone ( ) const [inline], [virtual]

Make a replica of the current warmstart information.

Implements BCP_warmstart.

Definition at line 69 of file BCP_warmstart_basis.hpp.

4.72.2.4 virtual **BCP_warmstart**∗ BCP_warmstart_basis::empty_wrt_this ( ) const [inline], [virtual]

Create a warmstart info describing that no change should be done.

This is really the task of a constructor, but BCP does not know the type of warmstart the user will use, so it will invoke this method for a warmstart that was created by the user. Tricky, isn't it?

Implements BCP_warmstart.

Definition at line 73 of file BCP_warmstart_basis.hpp.

4.72.2.5 virtual int BCP_warmstart_basis::storage_size ( ) const [inline], [virtual]

Return how much memory it'll take to pack this warmstart info.

It is used when comparing which sort of storage is smaller.

Implements BCP_warmstart.

Definition at line 79 of file BCP_warmstart_basis.hpp.

4.72.2.6 virtual void BCP_warmstart_basis::update (const **BCP_warmstart** ∗const *change*) [virtual]

Update the current data with the one in the argument.

If the argument is of explicit storage then just replace the current data. If it is relative and the current data is explicit then perform the update. Otherwise throw an exception.

Implements BCP_warmstart.

4.72.2.7 virtual **BCP_warmstart**∗ BCP_warmstart_basis::as_change (const **BCP_warmstart** ∗const *old_ws,* const **BCP_vec**< int > & *del_vars,* const **BCP_vec**< int > & *del_cuts,* const double *petol,* const double *detol*) const [virtual]

Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.

However, if the currently stored data is shorter to store than the change, then this method can return a copy of the current data! The current data must be explicitly stored and `old_ws` must be either explicit or can contain no data. Otherwise an exception must be thrown.

Parameters

old_ws	the old warmstart info
del_vars	the indices of the variables that are deleted from the formulation `old_ws` was created for
del_cuts	same for the cuts
petol	primal zero tolerance
detol	dual zero tolerance

Implements BCP_warmstart.

4.72.2.8 void BCP_warmstart_basis::pack (**BCP_buffer** & *buf*) const [inline]

Pack the warmstart info into a buffer.

Definition at line 94 of file BCP_warmstart_basis.hpp.

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

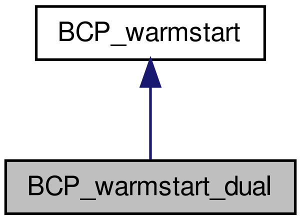
- BCP_warmstart_basis.hpp

4.73 BCP_warmstart_dual Class Reference

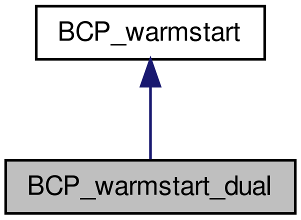
This class describes a warmstart information that consists solely of the dual vector.

`#include <BCP_warmstart_dual.hpp>`

Inheritance diagram for BCP_warmstart_dual:



Collaboration diagram for BCP_warmstart_dual:



Public Member Functions

- virtual **CoinWarmStart** ∗ convert_to_CoinWarmStart () const

  Return an **CoinWarmStart** *object that can be fed to the LP engine.*
- virtual BCP_storage_t storage () const

  Return how the warmstarting info is stored.
- virtual BCP_warmstart ∗ clone () const

  Make a replica of the current warmstart information.
- virtual BCP_warmstart ∗ empty_wrt_this () const

  Create a warmstart info describing that no change should be done.
- virtual int storage_size () const

  Return how much memory it'll take to pack this warmstart info.

- virtual void update (const BCP_warmstart ∗const change)

    Update the current data with the one in the argument.
- virtual BCP_warmstart ∗ as_change (const BCP_warmstart ∗const old_ws, const BCP_vec< int > &del_vars, const BCP_vec< int > &del_cuts, const double petol, const double detol) const

    Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.
- void pack (BCP_buffer &buf) const

    Pack the warmstart info into a buffer.

Constructors and destructor

- BCP_warmstart_dual (BCP_buffer &buf)

    Create the object by unpacking it from a buffer.
- BCP_warmstart_dual (const double ∗first, const double ∗last)

    Create an explicitly stored warmstart info by considering the double array `[first,last)` as the dual vector.
- BCP_warmstart_dual (const BCP_warmstart_dual &ws)

    Copy constructor.
- virtual ∼**BCP_warmstart_dual** ()

4.73.1 Detailed Description

This class describes a warmstart information that consists solely of the dual vector.

Definition at line 20 of file BCP_warmstart_dual.hpp.

4.73.2 Constructor & Destructor Documentation

4.73.2.1 BCP_warmstart_dual::BCP_warmstart_dual (const double ∗ *first,* const double ∗ *last*) [inline]

Create an explicitly stored warmstart info by considering the double array `[first,last)` as the dual vector.

Definition at line 42 of file BCP_warmstart_dual.hpp.

4.73.3 Member Function Documentation

4.73.3.1 virtual CoinWarmStart∗ BCP_warmstart_dual::convert_to_CoinWarmStart ( ) const [virtual]

Return an **CoinWarmStart** object that can be fed to the LP engine.

The implementation for this class will return an **CoinWarmStartDual** object.

Implements BCP_warmstart.

4.73.3.2 virtual BCP_storage_t BCP_warmstart_dual::storage ( ) const [inline], [virtual]

Return how the warmstarting info is stored.

Implements BCP_warmstart.

Definition at line 56 of file BCP_warmstart_dual.hpp.

4.73.3.3 virtual BCP_warmstart∗ BCP_warmstart_dual::clone ( ) const [inline], [virtual]

Make a replica of the current warmstart information.

Implements BCP_warmstart.

Definition at line 58 of file BCP_warmstart_dual.hpp.

4.73.3.4 virtual BCP_warmstart* BCP_warmstart_dual::empty_wrt_this ( ) const [inline], [virtual]

Create a warmstart info describing that no change should be done.

This is really the task of a constructor, but BCP does not know the type of warmstart the user will use, so it will invoke this method for a warmstart that was created by the user. Tricky, isn't it?

Implements BCP_warmstart.

Definition at line 62 of file BCP_warmstart_dual.hpp.

4.73.3.5 virtual int BCP_warmstart_dual::storage_size ( ) const [inline], [virtual]

Return how much memory it'll take to pack this warmstart info.

It is used when comparing which sort of storage is smaller.

Implements BCP_warmstart.

Definition at line 68 of file BCP_warmstart_dual.hpp.

4.73.3.6 virtual void BCP_warmstart_dual::update (const BCP_warmstart *const *change*) [virtual]

Update the current data with the one in the argument.

If the argument is of explicit storage then just replace the current data. If it is relative and the current data is explicit then perform the update. Otherwise throw an exception.

Implements BCP_warmstart.

4.73.3.7 virtual BCP_warmstart* BCP_warmstart_dual::as_change (const BCP_warmstart *const *old_ws*, const BCP_vec< int > & *del_vars*, const BCP_vec< int > & *del_cuts*, const double *petol*, const double *detol*) const [virtual]

Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.

However, if the currently stored data is shorter to store than the change, then this method can return a copy of the current data! The current data must be explicitly stored and `old_ws` must be either explicit or can contain no data. Otherwise an exception must be thrown.

Parameters

old_ws	the old warmstart info
del_vars	the indices of the variables that are deleted from the formulation `old_ws` was created for
del_cuts	same for the cuts
petol	primal zero tolerance
detol	dual zero tolerance

Implements BCP_warmstart.

4.73.3.8 void BCP_warmstart_dual::pack (BCP_buffer & *buf*) const [inline]

Pack the warmstart info into a buffer.

Definition at line 83 of file BCP_warmstart_dual.hpp.

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

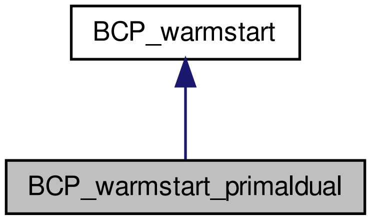
- BCP_warmstart_dual.hpp

4.74 BCP_warmstart_primaldual Class Reference

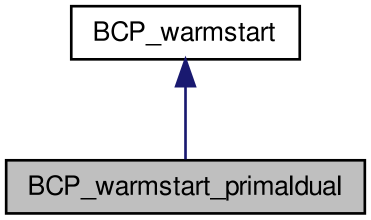
This class describes a warmstart information that consists solely of the dual vector.

`#include <BCP_warmstart_primaldual.hpp>`

Inheritance diagram for BCP_warmstart_primaldual:



Collaboration diagram for BCP_warmstart_primaldual:



Public Member Functions

- virtual **CoinWarmStart** ∗ convert_to_CoinWarmStart () const

    Return an **CoinWarmStart** *object that can be fed to the LP engine.*
- virtual BCP_storage_t storage () const

    Return how the warmstarting info is stored.
- virtual BCP_warmstart ∗ clone () const

    Make a replica of the current warmstart information.
- virtual BCP_warmstart ∗ empty_wrt_this () const

    Create a warmstart info describing that no change should be done.
- virtual int storage_size () const

    Return how much memory it'll take to pack this warmstart info.

- virtual void update (const BCP_warmstart ∗const change)

  Update the current data with the one in the argument.
- virtual BCP_warmstart ∗ as_change (const BCP_warmstart ∗const old_ws, const BCP_vec< int > &del_vars, const BCP_vec< int > &del_cuts, const double petol, const double detol) const

  Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.
- void pack (BCP_buffer &buf) const

  Pack the warmstart info into a buffer.

Constructors and destructor

- BCP_warmstart_primaldual (BCP_buffer &buf)

  Create the object by unpacking it from a buffer.
- BCP_warmstart_primaldual (const double ∗pfirst, const double ∗plast, const double ∗dfirst, const double ∗dlast)

  Create an explicitly stored warmstart info by considering the double arrays `[fpirst,plast)` and `[dpirst,dlast)` as the primal and dual vectors.
- BCP_warmstart_primaldual (const BCP_warmstart_primaldual &ws)

  Copy constructor.
- virtual ∼**BCP_warmstart_primaldual** ()

4.74.1 Detailed Description

This class describes a warmstart information that consists solely of the dual vector.

Definition at line 21 of file BCP_warmstart_primaldual.hpp.

4.74.2 Constructor & Destructor Documentation

4.74.2.1 BCP_warmstart_primaldual::BCP_warmstart_primaldual (const double ∗ *pfirst*, const double ∗ *plast*, const double ∗ *dfirst*, const double ∗ *dlast*) `[inline]`

Create an explicitly stored warmstart info by considering the double arrays `[fpirst,plast)` and `[dpirst,dlast)` as the primal and dual vectors.

Definition at line 47 of file BCP_warmstart_primaldual.hpp.

4.74.3 Member Function Documentation

4.74.3.1 virtual CoinWarmStart∗ BCP_warmstart_primaldual::convert_to_CoinWarmStart ( ) const `[virtual]`

Return an **CoinWarmStart** object that can be fed to the LP engine.

The implementation for this class will return an **CoinWarmStartDual** object.

Implements BCP_warmstart.

4.74.3.2 virtual BCP_storage_t BCP_warmstart_primaldual::storage ( ) const `[inline]`, `[virtual]`

Return how the warmstarting info is stored.

Implements BCP_warmstart.

Definition at line 63 of file BCP_warmstart_primaldual.hpp.

4.74.3.3 virtual BCP_warmstart∗ BCP_warmstart_primaldual::clone ( ) const [inline], [virtual]

Make a replica of the current warmstart information.

Implements BCP_warmstart.

Definition at line 82 of file BCP_warmstart_primaldual.hpp.

4.74.3.4 virtual BCP_warmstart∗ BCP_warmstart_primaldual::empty_wrt_this ( ) const [inline], [virtual]

Create a warmstart info describing that no change should be done.

This is really the task of a constructor, but BCP does not know the type of warmstart the user will use, so it will invoke this method for a warmstart that was created by the user. Tricky, isn't it?

Implements BCP_warmstart.

Definition at line 86 of file BCP_warmstart_primaldual.hpp.

4.74.3.5 virtual int BCP_warmstart_primaldual::storage_size ( ) const [inline], [virtual]

Return how much memory it'll take to pack this warmstart info.

It is used when comparing which sort of storage is smaller.

Implements BCP_warmstart.

Definition at line 92 of file BCP_warmstart_primaldual.hpp.

4.74.3.6 virtual void BCP_warmstart_primaldual::update (const BCP_warmstart ∗const *change*) [virtual]

Update the current data with the one in the argument.

If the argument is of explicit storage then just replace the current data. If it is relative and the current data is explicit then perform the update. Otherwise throw an exception.

Implements BCP_warmstart.

4.74.3.7 virtual BCP_warmstart∗ BCP_warmstart_primaldual::as_change (const BCP_warmstart ∗const *old_ws,* const BCP_vec< int > & *del_vars,* const BCP_vec< int > & *del_cuts,* const double *petol,* const double *detol*) const [virtual]

Return a pointer to a warmstart info describing the currently stored data as a change with respect to that stored in `old_ws`.

However, if the currently stored data is shorter to store than the change, then this method can return a copy of the current data! The current data must be explicitly stored and `old_ws` must be either explicit or can contain no data. Otherwise an exception must be thrown.

Parameters

old_ws	the old warmstart info
del_vars	the indices of the variables that are deleted from the formulation `old_ws` was created for
del_cuts	same for the cuts
petol	primal zero tolerance
detol	dual zero tolerance

Implements BCP_warmstart.

4.74.3.8 void BCP_warmstart_primaldual::pack (BCP_buffer & *buf*) const [inline]

Pack the warmstart info into a buffer.

Definition at line 107 of file BCP_warmstart_primaldual.hpp.

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

- BCP_warmstart_primaldual.hpp

4.75 USER_initialize Class Reference

This class is an abstract base class for the initializer class the user has to provide.

```
#include <BCP_USER.hpp>
```

Public Member Functions

Destructor

- virtual ∼USER_initialize ()
  virtual destructor

Message passing environment

- virtual BCP_message_environment ∗ msgenv_init (int argc, char ∗argv[ ])
  Create a message passing environment.

User object initialization in the processes

These methods are invoked when the appropriate process starts.

They have to return pointers to user objects, i.e., objects derived from the return value of each init method. Those objects can be used by the user to store information about the problem; information that will be used in the member methods of the user objects.

- virtual BCP_tm_user ∗ **tm_init** (BCP_tm_prob &p, const int argnum, const char ∗const ∗arglist)
- virtual BCP_ts_user ∗ **ts_init** (BCP_ts_prob &p)
- virtual BCP_lp_user ∗ **lp_init** (BCP_lp_prob &p)
- virtual BCP_vg_user ∗ **vg_init** (BCP_vg_prob &p)
- virtual BCP_cg_user ∗ **cg_init** (BCP_cg_prob &p)
- virtual BCP_vp_user ∗ **vp_init** (BCP_vp_prob &p)
- virtual BCP_cp_user ∗ **cp_init** (BCP_cp_prob &p)
- virtual BCP_user_pack ∗ **packer_init** (BCP_user_class ∗p)

4.75.1 Detailed Description

This class is an abstract base class for the initializer class the user has to provide.

The user will have to return an instance of the initializer class when the BCP_user_init() function is invoked. The member methods of that instance will be invoked to create the various objects (well, pointers to them) that are used/controlled by the user during the course of a run.

Definition at line 160 of file BCP_USER.hpp.

4.75.2 Member Function Documentation

4.75.2.1 virtual BCP_message_environment∗ USER_initialize::msgenv_init (int *argc*, char ∗ *argv[]*) [virtual]

Create a message passing environment.

Currently implemented environments are single and PVM, the default is single. To use PVM, the user has to override this method and return a pointer to a new `BCP_pvm_environment` object.

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

- BCP_USER.hpp

File Documentation

Index