# Cbc

2.9

Generated by Doxygen 1.8.9.1

Thu Oct 8 2015 22:42:51

# Contents

# Chapter 1

# Hierarchical Index

## 1.1 Class Hierarchy

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

- std::deque< T >::const_reverse_iterator
- std::map< K, T >::const_reverse_iterator
- std::unordered_map< K, T >::const_reverse_iterator
- std::set< K >::const_reverse_iterator
- std::unordered_set< K >::const_reverse_iterator
- consys_struct [external]
- cut [external]
- cut_list [external]
- cutParams [external]
- LAP::Cuts [external]
- cycle [external]
- cycle_list [external]
- CbcGenCtlBlk::debugSolInfo_struct . . . . . . . . . . 237
- std::deque< T >
- std::deque< StdVectorDouble >
- DGG_constraint_t [external]
- DGG_data_t [external]
- DGG_list_t [external]
- disaggregationAction [external]
- CbcGenCtlBlk::djFixCtl_struct . . . . . . . . . . 237
- dropped_zero [external]
- dualColumnResult [external]
- edge [external]
- EKKHlink [external]
- ENV [external]
- std::error_category
- std::error_code
- std::error_condition
- std::exception
  - std::bad_alloc
  - std::bad_cast
  - std::bad_exception
  - std::bad_typeid
  - std::ios_base::failure
  - std::logic_error
    - std::domain_error
    - std::invalid_argument
    - std::length_error
    - std::out_of_range
  - std::runtime_error
    - std::overflow_error
    - std::range_error
    - std::underflow_error
- FactorPointers [external]
- std::forward_list< T >
- CbcGenCtlBlk::genParamsInfo_struct . . . . . . . . . . 238
- glp_prob [external]
- hel_tag [external]
- Idiot [external]
- IdiotResult [external]
- ilp [external]
- Info [external]
- info_weak [external]
- INV [external]

- std::ios_base
  - basic_ios< char >
  - basic_ios< wchar_t >
  - std::basic_ios
    - basic_istream< char >
    - basic_istream< wchar_t >
    - basic_ostream< char >
    - basic_ostream< wchar_t >
    - std::basic_istream
      - basic_ifstream< char >
      - basic_ifstream< wchar_t >
      - basic_iostream< char >
      - basic_iostream< wchar_t >
      - basic_istringstream< char >
      - basic_istringstream< wchar_t >
      - std::basic_ifstream
        - std::ifstream
        - std::wifstream
      - std::basic_iostream
        - basic_fstream< char >
        - basic_fstream< wchar_t >
        - basic_stringstream< char >
        - basic_stringstream< wchar_t >
        - std::basic_fstream
          - std::fstream
          - std::wfstream
        - std::basic_stringstream
          - std::stringstream
          - std::wstringstream
      - std::basic_istringstream
        - std::istringstream
        - std::wistringstream
      - std::istream
      - std::wistream
    - std::basic_ostream
      - basic_iostream< char >
      - basic_iostream< wchar_t >
      - basic_ofstream< char >
      - basic_ofstream< wchar_t >
      - basic_ostringstream< char >
      - basic_ostringstream< wchar_t >
      - std::basic_iostream
      - std::basic_ofstream
        - std::ofstream
        - std::wofstream
      - std::basic_ostringstream
        - std::ostringstream
        - std::wostringstream
      - std::ostream
      - std::wostream
    - std::ios
    - std::wios
- OsiCuts::iterator `[external]`
- std::unordered_multimap< K, T >::iterator

std::valarray< T >
LAP::Validator [external]
std::vector< T >
std::vector< bool >
std::vector< CbcHeuristicNode * >
std::vector< CbcNode * >
std::vector< ColumnSelectionStrategy >
std::vector< double >
std::vector< int >
std::vector< Node >
std::vector< RowSelectionStrategy >
std::vector< std::pair< std::string, double > >
std::vector< std::string >
VOL_alpha_factor [external]
VOL_dual [external]
VOL_dvector [external]
VOL_indc [external]
VOL_ivector [external]
VOL_parms [external]
VOL_primal [external]
VOL_problem [external]
VOL_swing [external]
VOL_user_hooks [external]
    OsiVolSolverInterface [external]
VOL_vh [external]
std::weak_ptr< T >
K
S
T
U

# Chapter 2

# Class Index

## 2.1 Class List

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

# Chapter 3

# File Index

## 3.1 File List

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

# Chapter 4

# Class Documentation

## 4.1 CbcGenCtlBlk::babState_struct Struct Reference

State of branch-and-cut.

`#include <CbcGenCtlBlk.hpp>`

Collaboration diagram for CbcGenCtlBlk::babState_struct:

### 4.1.1 Detailed Description

State of branch-and-cut.

Major and minor status codes, and a solver holding the answer, assuming we have a valid answer. See the documentation with the BACMajor, BACMinor, and BACWhere enums for the meaning of the codes.

Definition at line 718 of file CbcGenCtlBlk.hpp.

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

- CbcGenCtlBlk.hpp

## 4.2 CbcBaseModel Class Reference

Base model.

`#include <CbcThread.hpp>`

### 4.2.1 Detailed Description

Base model.

Definition at line 437 of file CbcThread.hpp.

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

- CbcThread.hpp

## 4.3 CbcBranchAllDifferent Class Reference

Define a branch class that branches so that it is only satsified if all members have different values So cut is x <= y-1 or x >= y+1.

`#include <CbcBranchAllDifferent.hpp>`

Inheritance diagram for CbcBranchAllDifferent:

Collaboration diagram for CbcBranchAllDifferent:

### Public Member Functions

- CbcBranchAllDifferent (CbcModel ∗model, int number, const int ∗which)

  *Useful constructor - passed set of integer variables which must all be different.*
- virtual CbcObject ∗ clone () const

  *Clone.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗info, int &preferredWay) const

  *Infeasibility - large is 0.5.*
- virtual CbcBranchingObject ∗ createCbcBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way)

  *Creates a branching object.*

### Protected Attributes

- int numberInSet_

  *data*
- int ∗ which_

  *Which variables.*

### 4.3.1 Detailed Description

Define a branch class that branches so that it is only satsified if all members have different values So cut is x <= y-1 or x >= y+1.

Definition at line 22 of file CbcBranchAllDifferent.hpp.

### 4.3.2 Member Data Documentation

#### 4.3.2.1 int CbcBranchAllDifferent::numberInSet_ `[protected]`

data

Number of entries

Definition at line 57 of file CbcBranchAllDifferent.hpp.

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

- CbcBranchAllDifferent.hpp

## 4.4 CbcBranchCut Class Reference

Define a cut branching class.

`#include <CbcBranchCut.hpp>`

Inheritance diagram for CbcBranchCut:

Collaboration diagram for CbcBranchCut:

### Public Member Functions

- CbcBranchCut (CbcModel ∗model)

  *In to maintain normal methods.*
- virtual CbcObject ∗ clone () const

  *Clone.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗info, int &preferredWay) const

  *Infeasibility.*
- virtual void feasibleRegion ()

  *Set bounds to contain the current solution.*
- virtual bool boundBranch () const

  *Return true if branch created by object should fix variables.*
- virtual CbcBranchingObject ∗ createCbcBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way)

  *Creates a branching object.*
- virtual CbcBranchingObject ∗ preferredNewFeasible () const

  *Given a valid solution (with reduced costs, etc.), return a branching object which would give a new feasible point in the good direction.*
- virtual CbcBranchingObject ∗ notPreferredNewFeasible () const

  *Given a valid solution (with reduced costs, etc.), return a branching object which would give a new feasible point in a bad direction.*
- virtual void resetBounds ()

  *Reset original upper and lower bound values from the solver.*

### Additional Inherited Members

### 4.4.1 Detailed Description

Define a cut branching class.

At present empty - all stuff in descendants

Definition at line 17 of file CbcBranchCut.hpp.

### 4.4.2 Member Function Documentation

#### 4.4.2.1 virtual void CbcBranchCut::feasibleRegion ( ) `[virtual]`

Set bounds to contain the current solution.

More precisely, for the variable associated with this object, take the value given in the current solution, force it within the current bounds if required, then set the bounds to fix the variable at the integer nearest the solution value.

At present this will do nothing

Implements CbcObject.

#### 4.4.2.2 virtual CbcBranchingObject* CbcBranchCut::preferredNewFeasible ( ) const [virtual]

Given a valid solution (with reduced costs, etc.), return a branching object which would give a new feasible point in the good direction.

The preferred branching object will force the variable to be +/-1 from its current value, depending on the reduced cost and objective sense. If movement in the direction which improves the objective is impossible due to bounds on the variable, the branching object will move in the other direction. If no movement is possible, the method returns NULL.

Only the bounds on this variable are considered when determining if the new point is feasible.

At present this does nothing

Reimplemented from CbcObject.

#### 4.4.2.3 virtual CbcBranchingObject* CbcBranchCut::notPreferredNewFeasible ( ) const [virtual]

Given a valid solution (with reduced costs, etc.), return a branching object which would give a new feasible point in a bad direction.

As for preferredNewFeasible(), but the preferred branching object will force movement in a direction that degrades the objective.

At present this does nothing

Reimplemented from CbcObject.

#### 4.4.2.4 virtual void CbcBranchCut::resetBounds ( ) [virtual]

Reset original upper and lower bound values from the solver.

Handy for updating bounds held in this object after bounds held in the solver have been tightened.

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

- CbcBranchCut.hpp

## 4.5 CbcBranchDecision Class Reference

Inheritance diagram for CbcBranchDecision:

Collaboration diagram for CbcBranchDecision:

### Public Member Functions

- CbcBranchDecision ()

  *Default Constructor.*
- virtual ~CbcBranchDecision ()

  *Destructor.*
- virtual CbcBranchDecision * clone () const =0

*Clone.*

- virtual void initialize (CbcModel ∗model)=0

  *Initialize e.g. before starting to choose a branch at a node.*

- virtual int betterBranch (CbcBranchingObject ∗thisOne, CbcBranchingObject ∗bestSoFar, double changeUp, int numberInfeasibilitiesUp, double changeDown, int numberInfeasibilitiesDown)=0

  *Compare two branching objects.*

- virtual int bestBranch (CbcBranchingObject ∗∗objects, int numberObjects, int numberUnsatisfied, double ∗changeUp, int ∗numberInfeasibilitiesUp, double ∗changeDown, int ∗numberInfeasibilitiesDown, double objectiveValue)

  *Compare N branching objects.*

- virtual int whichMethod ()

  *Says whether this method can handle both methods - 1 better, 2 best, 3 both.*

- virtual void saveBranchingObject (**OsiBranchingObject** ∗)

  *Saves a clone of current branching object.*

- virtual void updateInformation (**OsiSolverInterface** ∗, const CbcNode ∗)

  *Pass in information on branch just done.*

- virtual void setBestCriterion (double)

  *Sets or gets best criterion so far.*

- virtual void generateCpp (FILE ∗)

  *Create C++ lines to get to current state.*

- CbcModel ∗ cbcModel () const

  *Model.*

- void setChooseMethod (const **OsiChooseVariable** &method)

  *Set (clone) chooseMethod.*

## Protected Attributes

- CbcModel ∗ model_

  *Pointer to model.*

### 4.5.1 Detailed Description

Definition at line 28 of file CbcBranchDecision.hpp.

### 4.5.2 Member Function Documentation

#### 4.5.2.1 virtual int CbcBranchDecision::betterBranch ( CbcBranchingObject ∗ *thisOne*, CbcBranchingObject ∗ *bestSoFar*, double *changeUp*, int *numberInfeasibilitiesUp*, double *changeDown*, int *numberInfeasibilitiesDown* ) `[pure virtual]`

Compare two branching objects.

Return nonzero if branching using `thisOne` is better than branching using `bestSoFar`.

If `bestSoFar` is NULL, the routine should return a nonzero value. This routine is used only after strong branching. Either this or bestBranch is used depending which user wants.

Implemented in CbcBranchDynamicDecision, and CbcBranchDefaultDecision.

#### 4.5.2.2 virtual int CbcBranchDecision::bestBranch ( CbcBranchingObject ** *objects*, int *numberObjects*, int *numberUnsatisfied*, double * *changeUp*, int * *numberInfeasibilitiesUp*, double * *changeDown*, int * *numberInfeasibilitiesDown*, double *objectiveValue* ) [virtual]

Compare N branching objects.

Return index of best and sets way of branching in chosen object.

Either this or betterBranch is used depending which user wants.

Reimplemented in CbcBranchDefaultDecision.

#### 4.5.2.3 virtual void CbcBranchDecision::saveBranchingObject ( OsiBranchingObject * ) [inline], [virtual]

Saves a clone of current branching object.

Can be used to update information on object causing branch - after branch

Reimplemented in CbcBranchDynamicDecision.

Definition at line 80 of file CbcBranchDecision.hpp.

#### 4.5.2.4 virtual void CbcBranchDecision::updateInformation ( OsiSolverInterface *, const CbcNode * ) [inline], [virtual]

Pass in information on branch just done.

assumes object can get information from solver

Reimplemented in CbcBranchDynamicDecision.

Definition at line 83 of file CbcBranchDecision.hpp.

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

- CbcBranchDecision.hpp

## 4.6 CbcBranchDefaultDecision Class Reference

Branching decision default class.

`#include <CbcBranchDefaultDecision.hpp>`

Inheritance diagram for CbcBranchDefaultDecision:

Collaboration diagram for CbcBranchDefaultDecision:

### Public Member Functions

- virtual CbcBranchDecision * clone () const

  *Clone.*
- virtual void initialize (CbcModel *model)

  *Initialize, e.g. before the start of branch selection at a node.*
- virtual int betterBranch (CbcBranchingObject *thisOne, CbcBranchingObject *bestSoFar, double changeUp, int numInfUp, double changeDn, int numInfDn)

  *Compare two branching objects.*

- virtual void setBestCriterion (double value)

  *Sets or gets best criterion so far.*

- virtual int bestBranch (CbcBranchingObject ∗∗objects, int numberObjects, int numberUnsatisfied, double ∗changeUp, int ∗numberInfeasibilitiesUp, double ∗changeDown, int ∗numberInfeasibilitiesDown, double objectiveValue)

  *Compare N branching objects.*

## Additional Inherited Members

### 4.6.1 Detailed Description

Branching decision default class.

This class implements a simple default algorithm (betterBranch()) for choosing a branching variable.

Definition at line 18 of file CbcBranchDefaultDecision.hpp.

### 4.6.2 Member Function Documentation

#### 4.6.2.1 virtual int CbcBranchDefaultDecision::betterBranch ( **CbcBranchingObject** ∗ ***thisOne,*** **CbcBranchingObject** ∗ ***bestSoFar,*** double ***changeUp,*** int ***numInfUp,*** double ***changeDn,*** int ***numInfDn*** ) `[virtual]`

Compare two branching objects.

Return nonzero if `thisOne` is better than `bestSoFar`.

The routine compares branches using the values supplied in `numInfUp` and `numInfDn` until a solution is found by search, after which it uses the values supplied in `changeUp` and `changeDn`. The best branching object seen so far and the associated parameter values are remembered in the `CbcBranchDefaultDecision` object. The nonzero return value is +1 if the up branch is preferred, -1 if the down branch is preferred.

As the names imply, the assumption is that the values supplied for `numInfUp` and `numInfDn` will be the number of infeasibilities reported by the branching object, and `changeUp` and `changeDn` will be the estimated change in objective. Other measures can be used if desired.

Because an `CbcBranchDefaultDecision` object remembers the current best branching candidate (#best←
Object_) as well as the values used in the comparison, the parameter `bestSoFar` is redundant, hence unused.

Implements CbcBranchDecision.

#### 4.6.2.2 virtual int CbcBranchDefaultDecision::bestBranch ( **CbcBranchingObject** ∗∗ ***objects,*** int ***numberObjects,*** int ***numberUnsatisfied,*** double ∗ ***changeUp,*** int ∗ ***numberInfeasibilitiesUp,*** double ∗ ***changeDown,*** int ∗ ***numberInfeasibilitiesDown,*** double ***objectiveValue*** ) `[virtual]`

Compare N branching objects.

Return index of best and sets way of branching in chosen object.

This routine is used only after strong branching.

Reimplemented from CbcBranchDecision.

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

- CbcBranchDefaultDecision.hpp

## 4.7 CbcBranchDynamicDecision Class Reference

Branching decision dynamic class.

`#include <CbcBranchDynamic.hpp>`

Inheritance diagram for CbcBranchDynamicDecision:

Collaboration diagram for CbcBranchDynamicDecision:

### Public Member Functions

- virtual CbcBranchDecision ∗ clone () const

  *Clone.*
- virtual void initialize (CbcModel ∗model)

  *Initialize, e.g. before the start of branch selection at a node.*
- virtual int betterBranch (CbcBranchingObject ∗thisOne, CbcBranchingObject ∗bestSoFar, double changeUp, int numInfUp, double changeDn, int numInfDn)

  *Compare two branching objects.*
- virtual void setBestCriterion (double value)

  *Sets or gets best criterion so far.*
- virtual int whichMethod ()

  *Says whether this method can handle both methods - 1 better, 2 best, 3 both.*
- virtual void saveBranchingObject (**OsiBranchingObject** ∗object)

  *Saves a clone of current branching object.*
- virtual void updateInformation (**OsiSolverInterface** ∗solver, const CbcNode ∗node)

  *Pass in information on branch just done.*

### Additional Inherited Members

### 4.7.1 Detailed Description

Branching decision dynamic class.

This class implements a simple algorithm (betterBranch()) for choosing a branching variable when dynamic pseudo costs.

Definition at line 19 of file CbcBranchDynamic.hpp.

### 4.7.2 Member Function Documentation

#### 4.7.2.1 virtual int CbcBranchDynamicDecision::betterBranch ( **CbcBranchingObject** ∗ *thisOne,* **CbcBranchingObject** ∗ *bestSoFar,* double *changeUp,* int *numInfUp,* double *changeDn,* int *numInfDn* ) `[virtual]`

Compare two branching objects.

Return nonzero if `thisOne` is better than `bestSoFar`.

The routine compares branches using the values supplied in `numInfUp` and `numInfDn` until a solution is found by search, after which it uses the values supplied in `changeUp` and `changeDn`. The best branching object seen so far and the associated parameter values are remembered in the `CbcBranchDynamicDecision` object. The nonzero return value is +1 if the up branch is preferred, -1 if the down branch is preferred.

As the names imply, the assumption is that the values supplied for `numInfUp` and `numInfDn` will be the number of infeasibilities reported by the branching object, and `changeUp` and `changeDn` will be the estimated change in objective. Other measures can be used if desired.

Because an `CbcBranchDynamicDecision` object remembers the current best branching candidate (#best←Object_) as well as the values used in the comparison, the parameter `bestSoFar` is redundant, hence unused.

Implements CbcBranchDecision.

#### 4.7.2.2 virtual void CbcBranchDynamicDecision::saveBranchingObject ( OsiBranchingObject ∗ *object* ) [virtual]

Saves a clone of current branching object.

Can be used to update information on object causing branch - after branch

Reimplemented from CbcBranchDecision.

#### 4.7.2.3 virtual void CbcBranchDynamicDecision::updateInformation ( OsiSolverInterface ∗ *solver,* const CbcNode ∗ *node* ) [virtual]

Pass in information on branch just done.

assumes object can get information from solver

Reimplemented from CbcBranchDecision.

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

- CbcBranchDynamic.hpp

## 4.8 CbcBranchingObject Class Reference

Abstract branching object base class Now just difference with **OsiBranchingObject**.

`#include <CbcBranchingObject.hpp>`

Inheritance diagram for CbcBranchingObject:

Collaboration diagram for CbcBranchingObject:

### Public Member Functions

- CbcBranchingObject ()

  *Default Constructor.*
- CbcBranchingObject (CbcModel ∗model, int variable, int way, double **value**)

  *Constructor.*
- CbcBranchingObject (const CbcBranchingObject &)

  *Copy constructor.*
- CbcBranchingObject & operator= (const CbcBranchingObject &rhs)

  *Assignment operator.*
- virtual CbcBranchingObject ∗ clone () const =0

  *Clone.*
- virtual ∼CbcBranchingObject ()

*Destructor.*

- virtual int fillStrongInfo (CbcStrongInfo &)

  *Some branchingObjects may claim to be able to skip strong branching.*
- void resetNumberBranchesLeft ()

  *Reset number of branches left to original.*
- void setNumberBranches (int **value**)

  *Set number of branches to do.*
- virtual double branch ()=0

  *Execute the actions required to branch, as specified by the current state of the branching object, and advance the object's state.*
- virtual double branch (**OsiSolverInterface** ∗)

  *Execute the actions required to branch, as specified by the current state of the branching object, and advance the object's state.*
- virtual void fix (**OsiSolverInterface** ∗, double ∗, double ∗, int) const

  *Update bounds in solver as in 'branch' and update given bounds.*
- virtual bool tighten (**OsiSolverInterface** ∗)

  *Change (tighten) bounds in object to reflect bounds in solver.*
- virtual void previousBranch ()

  *Reset every information so that the branching object appears to point to the previous child.*
- virtual void print () const

  *Print something about branch - only if log level high.*
- int variable () const

  *Index identifying the associated CbcObject within its class.*
- int way () const

  *Get the state of the branching object.*
- void way (int way)

  *Set the state of the branching object.*
- void setModel (CbcModel ∗model)

  *update model*
- CbcModel ∗ model () const

  *Return model.*
- CbcObject ∗ object () const

  *Return pointer back to object which created.*
- void setOriginalObject (CbcObject ∗object)

  *Set pointer back to object which created.*
- virtual CbcBranchObjType type () const =0

  *Return the type (an integer identifier) of `this`.*
- virtual int compareOriginalObject (const CbcBranchingObject ∗brObj) const

  *Compare the original object of `this` with the original object of `brObj`.*
- virtual CbcRangeCompare compareBranchingObject (const CbcBranchingObject ∗brObj, const bool replaceIf↩ Overlap=false)=0

  *Compare the `this` with `brObj`.*

### Protected Attributes

- CbcModel ∗ model_

  *The model that owns this branching object.*
- CbcObject ∗ originalCbcObject_

  *Pointer back to object which created.*
- int variable_

  *Branching variable (0 is first integer)*
- int way_

  *The state of the branching object.*

### 4.8.1 Detailed Description

Abstract branching object base class Now just difference with **OsiBranchingObject**.

In the abstract, an CbcBranchingObject contains instructions for how to branch. We want an abstract class so that we can describe how to branch on simple objects (*e.g.*, integers) and more exotic objects (*e.g.*, cliques or hyperplanes).

The branch() method is the crucial routine: it is expected to be able to step through a set of branch arms, executing the actions required to create each subproblem in turn. The base class is primarily virtual to allow for a wide range of problem modifications.

See CbcObject for an overview of the three classes (CbcObject, CbcBranchingObject, and CbcBranchDecision) which make up cbc's branching model.

Definition at line 53 of file CbcBranchingObject.hpp.

### 4.8.2 Member Function Documentation

#### 4.8.2.1 virtual int CbcBranchingObject::fillStrongInfo ( **CbcStrongInfo &** ) [inline], [virtual]

Some branchingObjects may claim to be able to skip strong branching.

If so they have to fill in CbcStrongInfo. The object mention in incoming CbcStrongInfo must match. Returns nonzero if skip is wanted

Reimplemented in CbcDynamicPseudoCostBranchingObject.

Definition at line 79 of file CbcBranchingObject.hpp.

#### 4.8.2.2 virtual double CbcBranchingObject::branch ( ) [pure virtual]

Execute the actions required to branch, as specified by the current state of the branching object, and advance the object's state.

Mainly for diagnostics, whether it is true branch or strong branching is also passed. Returns change in guessed objective on next branch

Reimplemented from **OsiBranchingObject**.

Implemented in CbcIntegerPseudoCostBranchingObject, CbcOrbitalBranchingObject, CbcLongCliqueBranching↩Object, CbcSOSBranchingObject, CbcLotsizeBranchingObject, CbcCliqueBranchingObject, CbcDynamicPseudo↩CostBranchingObject, CbcCutBranchingObject, CbcNWayBranchingObject, CbcFixingBranchingObject, CbcInteger↩BranchingObject, and CbcDummyBranchingObject.

#### 4.8.2.3 virtual double CbcBranchingObject::branch ( OsiSolverInterface * ) [inline], [virtual]

Execute the actions required to branch, as specified by the current state of the branching object, and advance the object's state.

Mainly for diagnostics, whether it is true branch or strong branching is also passed. Returns change in guessed objective on next branch

Implements **OsiBranchingObject**.

Definition at line 105 of file CbcBranchingObject.hpp.

#### 4.8.2.4 virtual void CbcBranchingObject::fix ( OsiSolverInterface *, double *, double *, int ) const [inline], [virtual]

Update bounds in solver as in 'branch' and update given bounds.

branchState is -1 for 'down' +1 for 'up'

Reimplemented in CbcOrbitalBranchingObject, CbcSOSBranchingObject, and CbcIntegerBranchingObject.

Definition at line 110 of file CbcBranchingObject.hpp.

#### 4.8.2.5 virtual bool CbcBranchingObject::tighten ( OsiSolverInterface * ) [inline], [virtual]

Change (tighten) bounds in object to reflect bounds in solver.

Return true if now fixed

Reimplemented in CbcIntegerBranchingObject.

Definition at line 116 of file CbcBranchingObject.hpp.

#### 4.8.2.6 virtual void CbcBranchingObject::previousBranch ( ) [inline], [virtual]

Reset every information so that the branching object appears to point to the previous child.

This method does not need to modify anything in any solver.

Reimplemented in CbcOrbitalBranchingObject, and CbcSOSBranchingObject.

Definition at line 121 of file CbcBranchingObject.hpp.

#### 4.8.2.7 int CbcBranchingObject::variable ( ) const [inline]

Index identifying the associated CbcObject within its class.

The name is misleading, and typically the index will *not* refer directly to a variable. Rather, it identifies an CbcObject within the class of similar CbcObjects

*E.g.*, for an CbcSimpleInteger, variable() is the index of the integer variable in the set of integer variables (*not* the index of the variable in the set of all variables).

Definition at line 143 of file CbcBranchingObject.hpp.

#### 4.8.2.8 int CbcBranchingObject::way ( ) const [inline]

Get the state of the branching object.

Returns a code indicating the active arm of the branching object. The precise meaning is defined in the derived class.

**See also**

way_

Definition at line 154 of file CbcBranchingObject.hpp.

#### 4.8.2.9 void CbcBranchingObject::way ( int *way* ) [inline]

Set the state of the branching object.

See way()

Definition at line 162 of file CbcBranchingObject.hpp.

#### 4.8.2.10 virtual CbcBranchObjType CbcBranchingObject::type ( ) const [pure virtual]

Return the type (an integer identifier) of `this`.

See definition of CbcBranchObjType above for possibilities

Implemented in CbcIntegerPseudoCostBranchingObject, CbcOrbitalBranchingObject, CbcLongCliqueBranchingObject, CbcSOSBranchingObject, CbcLotsizeBranchingObject, CbcCliqueBranchingObject, CbcDynamicPseudoCostBranchingObject, CbcCutBranchingObject, CbcNWayBranchingObject, CbcIntegerBranchingObject, CbcFixingBranchingObject, and CbcDummyBranchingObject.

#### 4.8.2.11 virtual int CbcBranchingObject::compareOriginalObject ( const **CbcBranchingObject** ∗ *brObj* ) const [inline], [virtual]

Compare the original object of `this` with the original object of `brObj`.

Assumes that there is an ordering of the original objects. This method should be invoked only if `this` and brObj are of the same type. Return negative/0/positive depending on whether `this` is smaller/same/larger than the argument.

Reimplemented in CbcOrbitalBranchingObject, CbcLongCliqueBranchingObject, CbcSOSBranchingObject, CbcCliqueBranchingObject, CbcCutBranchingObject, CbcNWayBranchingObject, CbcFixingBranchingObject, and CbcDummyBranchingObject.

Definition at line 199 of file CbcBranchingObject.hpp.

#### 4.8.2.12 virtual CbcRangeCompare CbcBranchingObject::compareBranchingObject ( const **CbcBranchingObject** ∗ *brObj*, const bool *replaceIfOverlap* = `false` ) [pure virtual]

Compare the `this` with `brObj`.

`this` and `brObj` must be of the same type and must have the same original object, but they may have different feasible regions. Return the appropriate CbcRangeCompare value (first argument being the sub/superset if that's the case). In case of overlap (and if `replaceIfOverlap` is true) replace the current branching object with one whose feasible region is the overlap.

Implemented in CbcIntegerPseudoCostBranchingObject, CbcOrbitalBranchingObject, CbcLongCliqueBranchingObject, CbcSOSBranchingObject, CbcLotsizeBranchingObject, CbcCliqueBranchingObject, CbcCutBranchingObject, CbcNWayBranchingObject, CbcIntegerBranchingObject, CbcFixingBranchingObject, and CbcDummyBranchingObject.

### 4.8.3 Member Data Documentation

#### 4.8.3.1 int CbcBranchingObject::way_ [protected]

The state of the branching object.

Specifies the active arm of the branching object. Coded as -1 to take the 'down' arm, +1 for the 'up' arm. 'Down' and 'up' are defined based on the natural meaning (floor and ceiling, respectively) for a simple integer. The precise meaning is defined in the derived class.

Definition at line 232 of file CbcBranchingObject.hpp.

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

- CbcBranchingObject.hpp

## 4.9 CbcBranchToFixLots Class Reference

Define a branch class that branches so that one way variables are fixed while the other way cuts off that solution.

`#include <CbcBranchToFixLots.hpp>`

Inheritance diagram for CbcBranchToFixLots:

Collaboration diagram for CbcBranchToFixLots:

### Public Member Functions

- CbcBranchToFixLots (CbcModel ∗model, double djTolerance, double fractionFixed, int depth, int numberClean=0, const char ∗mark=NULL, bool alwaysCreate=false)

  *Useful constructor - passed reduced cost tolerance and fraction we would like fixed.*
- virtual CbcObject ∗ clone () const

  *Clone.*
- int shallWe () const

  *Does a lot of the work, Returns 0 if no good, 1 if dj, 2 if clean, 3 if both FIXME: should use enum or equivalent to make these numbers clearer.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗info, int &preferredWay) const

  *Infeasibility for an integer variable - large is 0.5, but also can be infinity when known infeasible.*
- virtual bool canDoHeuristics () const

  *Return true if object can take part in normal heuristics.*
- virtual CbcBranchingObject ∗ createCbcBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way)

  *Creates a branching object.*
- virtual void redoSequenceEtc (CbcModel ∗model, int numberColumns, const int ∗originalColumns)

  *Redoes data when sequence numbers change.*

### Protected Attributes

- double djTolerance_

  *data*
- double fractionFixed_

*We only need to make sure this fraction fixed.*
- char * mark_

  *Never fix ones marked here.*
- **CoinPackedMatrix** matrixByRow_

  *Matrix by row.*
- int depth_

  *Do if depth multiple of this.*
- int numberClean_

  *number of ==1 rows which need to be clean*
- bool alwaysCreate_

  *If true then always create branch.*

### 4.9.1 Detailed Description

Define a branch class that branches so that one way variables are fixed while the other way cuts off that solution.

a) On reduced cost b) When enough ==1 or <=1 rows have been satisfied (not fixed - satisfied)

Definition at line 23 of file CbcBranchToFixLots.hpp.

### 4.9.2 Constructor & Destructor Documentation

#### 4.9.2.1 CbcBranchToFixLots::CbcBranchToFixLots ( **CbcModel** * *model*, double *djTolerance*, double *fractionFixed*, int *depth*, int *numberClean* = `0`, const char * *mark* = `NULL`, bool *alwaysCreate* = `false` )

Useful constructor - passed reduced cost tolerance and fraction we would like fixed.

Also depth level to do at. Also passed number of 1 rows which when clean triggers fix Always does if all 1 rows cleaned up and number>0 or if fraction columns reached Also whether to create branch if can't reach fraction.

### 4.9.3 Member Data Documentation

#### 4.9.3.1 double CbcBranchToFixLots::djTolerance_ `[protected]`

data

Reduced cost tolerance i.e. dj has to be >= this before fixed

Definition at line 79 of file CbcBranchToFixLots.hpp.

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

- CbcBranchToFixLots.hpp

## 4.10 CbcCbcParam Class Reference

Class for control parameters that act on a CbcModel object.

`#include <CbcGenCbcParam.hpp>`

Inheritance diagram for CbcCbcParam:

Collaboration diagram for CbcCbcParam:

## Public Types

### Subtypes

- enum CbcCbcParamCode
  *Enumeration for parameters that control a CbcModel object.*

## Public Member Functions

### Constructors and Destructors

*Be careful how you specify parameters for the constructors! There's great potential for confusion.*

- CbcCbcParam ()
  *Default constructor.*
- CbcCbcParam (CbcCbcParamCode code, std::string **name**, std::string help, double lower, double upper, double dflt=0.0, bool **display**=true)
  *Constructor for a parameter with a double value.*
- CbcCbcParam (CbcCbcParamCode code, std::string **name**, std::string help, int lower, int upper, int dflt=0, bool **display**=true)
  *Constructor for a parameter with an integer value.*
- CbcCbcParam (CbcCbcParamCode code, std::string **name**, std::string help, std::string firstValue, int dflt, bool **display**=true)
  *Constructor for a parameter with keyword values.*
- CbcCbcParam (CbcCbcParamCode code, std::string **name**, std::string help, std::string dflt, bool **display**=true)
  *Constructor for a string parameter.*
- CbcCbcParam (CbcCbcParamCode code, std::string **name**, std::string help, bool **display**=true)
  *Constructor for an action parameter.*
- CbcCbcParam (const CbcCbcParam &orig)
  *Copy constructor.*
- CbcCbcParam ∗ clone ()
  *Clone.*
- CbcCbcParam & operator= (const CbcCbcParam &rhs)
  *Assignment.*
- ∼CbcCbcParam ()
  *Destructor.*

### Methods to query and manipulate a parameter object

- CbcCbcParamCode paramCode () const
  *Get the parameter code.*
- void setParamCode (CbcCbcParamCode code)
  *Set the parameter code.*
- CbcModel ∗ obj () const
  *Get the underlying CbcModel object.*
- void setObj (CbcModel ∗obj)
  *Set the underlying CbcModel object.*

## 4.10.1 Detailed Description

Class for control parameters that act on a CbcModel object.

Adds parameter type codes and push/pull functions to the generic parameter object.

Definition at line 31 of file CbcGenCbcParam.hpp.

### 4.10.2 Member Enumeration Documentation

#### 4.10.2.1 enum CbcCbcParam::CbcCbcParamCode

Enumeration for parameters that control a CbcModel object.

These are parameters that control the operation of a CbcModel object. CBCCBC_FIRSTPARAM and CBCCBC_LAS←TPARAM are markers to allow convenient separation of parameter groups.

Definition at line 45 of file CbcGenCbcParam.hpp.

### 4.10.3 Constructor & Destructor Documentation

#### 4.10.3.1 CbcCbcParam::CbcCbcParam ( CbcCbcParamCode *code*, std::string *name*, std::string *help*, double *lower*, double *upper*, double *dflt* = `0.0`, bool *display* = `true` )

Constructor for a parameter with a double value.

The default value is 0.0. Be careful to clearly indicate that `lower` and `upper` are real (double) values to distinguish this constructor from the constructor for an integer parameter.

#### 4.10.3.2 CbcCbcParam::CbcCbcParam ( CbcCbcParamCode *code*, std::string *name*, std::string *help*, int *lower*, int *upper*, int *dflt* = `0`, bool *display* = `true` )

Constructor for a parameter with an integer value.

The default value is 0.

#### 4.10.3.3 CbcCbcParam::CbcCbcParam ( CbcCbcParamCode *code*, std::string *name*, std::string *help*, std::string *firstValue*, int *dflt*, bool *display* = `true` )

Constructor for a parameter with keyword values.

The string supplied as `firstValue` becomes the first keyword. Additional keywords can be added using **append←Kwd()**. Keywords are numbered from zero. It's necessary to specify both the first keyword (`firstValue`) and the default keyword index (`dflt`) in order to distinguish this constructor from the string and action parameter constructors.

#### 4.10.3.4 CbcCbcParam::CbcCbcParam ( CbcCbcParamCode *code*, std::string *name*, std::string *help*, std::string *dflt*, bool *display* = `true` )

Constructor for a string parameter.

The default string value must be specified explicitly to distinguish a string constructor from an action parameter constructor.

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

- CbcGenCbcParam.hpp

## 4.11 CbcClique Class Reference

Branching object for cliques.

`#include <CbcClique.hpp>`

Inheritance diagram for CbcClique:

Collaboration diagram for CbcClique:

## Public Member Functions

- CbcClique ()

  *Default Constructor.*
- CbcClique (CbcModel *model, int cliqueType, int numberMembers, const int *which, const char *type, int identifier, int slack=-1)

  *Useful constructor (which are integer indices) slack can denote a slack in set.*
- CbcClique (const CbcClique &)

  *Copy constructor.*
- virtual CbcObject * clone () const

  *Clone.*
- CbcClique & operator= (const CbcClique &rhs)

  *Assignment operator.*
- virtual ~CbcClique ()

  *Destructor.*
- virtual double infeasibility (const **OsiBranchingInformation** *info, int &preferredWay) const

  *Infeasibility - large is 0.5.*
- virtual void feasibleRegion ()

  *This looks at solution and sets bounds to contain solution.*
- virtual CbcBranchingObject * createCbcBranch (**OsiSolverInterface** *solver, const **OsiBranchingInformation** *info, int way)

  *Creates a branching object.*
- int numberMembers () const

  *Number of members.*
- int numberNonSOSMembers () const

  *Number of variables with -1 coefficient.*
- const int * members () const

  *Members (indices in range 0 ... numberIntegers_-1)*
- char type (int index) const

  *Type of each member, i.e., which way is strong.*
- int cliqueType () const

  *Clique type: 0 is <=, 1 is ==.*
- virtual void redoSequenceEtc (CbcModel *model, int numberColumns, const int *originalColumns)

  *Redoes data when sequence numbers change.*

## Protected Attributes

- int numberMembers_

  *data Number of members*
- int numberNonSOSMembers_

  *Number of Non SOS members i.e. fixing to zero is strong.*
- int * members_

*Members (indices in range 0 ... numberIntegers_-1)*

- char * type_

  *Strong value for each member.*

- int cliqueType_

  *Clique type.*

- int slack_

  *Slack variable for the clique.*

### 4.11.1 Detailed Description

Branching object for cliques.

A clique is defined to be a set of binary variables where fixing any one variable to its 'strong' value fixes all other variables. An example is the most common SOS1 construction: a set of binary variables x_j s.t. SUM{j} x_j = 1. Setting any one variable to 1 forces all other variables to 0. (See comments for CbcSOS below.)

Other configurations are possible, however: Consider x1-x2+x3 <= 0. Setting x1 (x3) to 1 forces x2 to 1 and x3 (x1) to 0. Setting x2 to 0 forces x1 and x3 to 0.

The proper point of view to take when interpreting CbcClique is 'generalisation of SOS1 on binary variables.' To get into the proper frame of mind, here's an example.

Consider the following sequence, where x_j = (1-y_j):

```
 x1 + x2 + x3 <=  1           all strong at 1
 x1 - y2 + x3 <=  0           y2 strong at 0; x1, x3 strong at 1
-y1 - y2 + x3 <= -1           y1, y2 strong at 0, x3 strong at 1
-y1 - y2 - y3 <= -2           all strong at 0
```

The first line is a standard SOS1 on binary variables.

Variables with +1 coefficients are 'SOS-style' and variables with -1 coefficients are 'non-SOS-style'. So numberNonSOSMembers_ simply tells you how many variables have -1 coefficients. The implicit rhs for a clique is 1-numberNonSOSMembers_.

Definition at line 41 of file CbcClique.hpp.

### 4.11.2 Constructor & Destructor Documentation

#### 4.11.2.1 CbcClique::CbcClique ( **CbcModel** * *model,* int *cliqueType,* int *numberMembers,* const int * *which,* const char * *type,* int *identifier,* int *slack* = -1 )

Useful constructor (which are integer indices) slack can denote a slack in set.

If type == NULL then as if 1

### 4.11.3 Member Function Documentation

#### 4.11.3.1 int CbcClique::numberNonSOSMembers ( ) const [inline]

Number of variables with -1 coefficient.

Number of non-SOS members, i.e., fixing to zero is strong. See comments at head of class, and comments for type_.

Definition at line 86 of file CbcClique.hpp.

#### 4.11.3.2 char CbcClique::type ( int *index* ) const [inline]

Type of each member, i.e., which way is strong.

This also specifies whether a variable has a +1 or -1 coefficient.

- 0 => -1 coefficient, 0 is strong value
- 1 => +1 coefficient, 1 is strong value If unspecified, all coefficients are assumed to be positive.

Indexed as 0 .. numberMembers_-1

Definition at line 104 of file CbcClique.hpp.

### 4.11.4 Member Data Documentation

#### 4.11.4.1 char* CbcClique::type_ [protected]

Strong value for each member.

This also specifies whether a variable has a +1 or -1 coefficient.

- 0 => -1 coefficient, 0 is strong value
- 1 => +1 coefficient, 1 is strong value If unspecified, all coefficients are assumed to be positive.

Indexed as 0 .. numberMembers_-1

Definition at line 136 of file CbcClique.hpp.

#### 4.11.4.2 int CbcClique::cliqueType_ [protected]

Clique type.

0 defines a <= relation, 1 an equality. The assumed value of the rhs is numberNonSOSMembers_+1. (See comments for the class.)

Definition at line 143 of file CbcClique.hpp.

#### 4.11.4.3 int CbcClique::slack_ [protected]

Slack variable for the clique.

Identifies the slack variable for the clique (typically added to convert a <= relation to an equality). Value is sequence number within clique menbers.

Definition at line 151 of file CbcClique.hpp.

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

- CbcClique.hpp

## 4.12 CbcCliqueBranchingObject Class Reference

Branching object for unordered cliques.

`#include <CbcClique.hpp>`

Inheritance diagram for CbcCliqueBranchingObject:

Collaboration diagram for CbcCliqueBranchingObject:

## Public Member Functions

- virtual CbcBranchingObject ∗ clone () const

  *Clone.*
- virtual double branch ()

  *Does next branch and updates state.*
- virtual void print ()

  *Print something about branch - only if log level high.*
- virtual CbcBranchObjType type () const

  *Return the type (an integer identifier) of `this`.*
- virtual int compareOriginalObject (const CbcBranchingObject ∗brObj) const

  *Compare the original object of `this` with the original object of `brObj`.*
- virtual CbcRangeCompare compareBranchingObject (const CbcBranchingObject ∗brObj, const bool replaceIf←Overlap=false)

  *Compare the `this` with `brObj`.*

## Additional Inherited Members

### 4.12.1 Detailed Description

Branching object for unordered cliques.

Intended for cliques which are long enough to make it worthwhile but <= 64 members. There will also be ones for long cliques.

Variable_ is the clique id number (redundant, as the object also holds a pointer to the clique.

Definition at line 162 of file CbcClique.hpp.

### 4.12.2 Member Function Documentation

#### 4.12.2.1 virtual int CbcCliqueBranchingObject::compareOriginalObject ( const CbcBranchingObject ∗ *brObj* ) const [virtual]

Compare the original object of `this` with the original object of `brObj`.

Assumes that there is an ordering of the original objects. This method should be invoked only if `this` and brObj are of the same type. Return negative/0/positive depending on whether `this` is smaller/same/larger than the argument.

Reimplemented from CbcBranchingObject.

#### 4.12.2.2 virtual CbcRangeCompare CbcCliqueBranchingObject::compareBranchingObject ( const CbcBranchingObject ∗ *brObj*, const bool *replaceIfOverlap* = false ) [virtual]

Compare the `this` with `brObj`.

`this` and `brObj` must be of the same type and must have the same original object, but they may have different feasible regions. Return the appropriate CbcRangeCompare value (first argument being the sub/superset if that's the case). In case of overlap (and if `replaceIfOverlap` is true) replace the current branching object with one whose feasible region is the overlap.

Implements CbcBranchingObject.

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

- CbcClique.hpp

## 4.13 CbcCompare Class Reference

Collaboration diagram for CbcCompare:

### Public Member Functions

- bool alternateTest (CbcNode *x, CbcNode *y)

  *This is alternate test function.*
- CbcCompareBase * comparisonObject () const

  *return comparison object*

### 4.13.1 Detailed Description

Definition at line 11 of file CbcCompare.hpp.

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

- CbcCompare.hpp

## 4.14 CbcCompareBase Class Reference

Inheritance diagram for CbcCompareBase:

Collaboration diagram for CbcCompareBase:

### Public Member Functions

- virtual bool newSolution (CbcModel *)

  *Reconsider behaviour after discovering a new solution.*
- virtual bool newSolution (CbcModel *, double, int)

  *Reconsider behaviour after discovering a new solution.*
- virtual bool fullScan () const

  *Returns true if wants code to do scan with alternate criterion NOTE - this is temporarily disabled.*
- virtual void generateCpp (FILE *)

  *Create C++ lines to get to current state.*
- virtual CbcCompareBase * clone () const

  *Clone.*

- virtual bool test (CbcNode ∗, CbcNode ∗)

  *This is test function.*
- virtual bool alternateTest (CbcNode ∗x, CbcNode ∗y)

  *This is alternate test function.*
- bool equalityTest (CbcNode ∗x, CbcNode ∗y) const

  *Further test if everything else equal.*
- void sayThreaded ()

  *Say threaded.*

### 4.14.1 Detailed Description

Definition at line 27 of file CbcCompareBase.hpp.

### 4.14.2 Member Function Documentation

#### 4.14.2.1 virtual bool CbcCompareBase::newSolution ( CbcModel ∗ ) [inline], [virtual]

Reconsider behaviour after discovering a new solution.

This allows any method to change its behaviour. It is called after each solution.

The method should return true if changes are made which will alter the evaluation criteria applied to a node. (So that in cases where the search tree is sorted, it can be properly rebuilt.)

Definition at line 45 of file CbcCompareBase.hpp.

#### 4.14.2.2 virtual bool CbcCompareBase::newSolution ( CbcModel ∗ , double , int ) [inline], [virtual]

Reconsider behaviour after discovering a new solution.

This allows any method to change its behaviour. It is called after each solution.

The method should return true if changes are made which will alter the evaluation criteria applied to a node. (So that in cases where the search tree is sorted, it can be properly rebuilt.)

Reimplemented in CbcCompareDefault.

Definition at line 57 of file CbcCompareBase.hpp.

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

- CbcCompareBase.hpp

## 4.15 CbcCompareDefault Class Reference

Inheritance diagram for CbcCompareDefault:

Collaboration diagram for CbcCompareDefault:

### Public Member Functions

- CbcCompareDefault ()

*Default Constructor.*
- CbcCompareDefault (double weight)

  *Constructor with weight.*
- CbcCompareDefault (const CbcCompareDefault &rhs)

  *Copy constructor.*
- CbcCompareDefault & operator= (const CbcCompareDefault &rhs)

  *Assignment operator.*
- virtual CbcCompareBase ∗ clone () const

  *Clone.*
- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*
- virtual bool test (CbcNode ∗x, CbcNode ∗y)

  *This is test function.*
- virtual bool newSolution (CbcModel ∗model, double objectiveAtContinuous, int numberInfeasibilitiesAt←Continuous)

  *This allows method to change behavior as it is called after each solution.*
- virtual bool every1000Nodes (CbcModel ∗model, int numberNodes)

  *This allows method to change behavior Return true if want tree re-sorted.*
- double getCutoff () const

  *Cutoff.*
- double getBestPossible () const

  *Best possible solution.*
- void setBreadthDepth (int value)

  *Depth above which want to explore first.*
- void startDive (CbcModel ∗model)

  *Start dive.*
- void cleanDive ()

  *Clean up diving (i.e. switch off or prepare)*

## Protected Attributes

- double weight_

  *Weight for each infeasibility.*
- double saveWeight_

  *Weight for each infeasibility - computed from solution.*
- double cutoff_

  *Cutoff.*
- double bestPossible_

  *Best possible solution.*
- int numberSolutions_

  *Number of solutions.*
- int treeSize_

  *Tree size (at last check)*
- int breadthDepth_

  *Depth above which want to explore first.*
- int startNodeNumber_

  *Chosen node from estimated (-1 is off)*

- int afterNodeNumber_

    *Node number when dive started.*
- bool setupForDiving_

    *Indicates doing setup for diving.*

### 4.15.1 Detailed Description

Definition at line 31 of file CbcCompareDefault.hpp.

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

- CbcCompareDefault.hpp

## 4.16 CbcCompareDepth Class Reference

Inheritance diagram for CbcCompareDepth:

Collaboration diagram for CbcCompareDepth:

### Public Member Functions

- virtual CbcCompareBase ∗ clone () const

    *Clone.*
- virtual void generateCpp (FILE ∗fp)

    *Create C++ lines to get to current state.*
- virtual bool test (CbcNode ∗x, CbcNode ∗y)

    *This is test function.*

### 4.16.1 Detailed Description

Definition at line 25 of file CbcCompareDepth.hpp.

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

- CbcCompareDepth.hpp

## 4.17 CbcCompareEstimate Class Reference

Inheritance diagram for CbcCompareEstimate:

Collaboration diagram for CbcCompareEstimate:

### Public Member Functions

- virtual CbcCompareBase ∗ clone () const

    *Clone.*
- virtual void generateCpp (FILE ∗fp)

*Create C++ lines to get to current state.*
- virtual bool test (CbcNode *x, CbcNode *y)

  *This is test function.*

### 4.17.1 Detailed Description

Definition at line 27 of file CbcCompareEstimate.hpp.

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

- CbcCompareEstimate.hpp

## 4.18 CbcCompareObjective Class Reference

Inheritance diagram for CbcCompareObjective:

Collaboration diagram for CbcCompareObjective:

### Public Member Functions

- virtual CbcCompareBase * clone () const

  *Clone.*
- virtual void generateCpp (FILE *fp)

  *Create C++ lines to get to current state.*
- virtual bool test (CbcNode *x, CbcNode *y)

  *This is test function.*

### 4.18.1 Detailed Description

Definition at line 26 of file CbcCompareObjective.hpp.

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

- CbcCompareObjective.hpp

## 4.19 CbcConsequence Class Reference

Abstract base class for consequent bounds.

`#include <CbcConsequence.hpp>`

Inheritance diagram for CbcConsequence:

### Public Member Functions

- virtual CbcConsequence * clone () const =0

  *Clone.*
- virtual ~CbcConsequence ()

*Destructor.*
- virtual void applyToSolver (**OsiSolverInterface** ∗solver, int state) const =0

  *Apply to an LP solver.*

### 4.19.1 Detailed Description

Abstract base class for consequent bounds.

When a variable is branched on it normally interacts with other variables by means of equations. There are cases where we want to step outside LP and do something more directly e.g. fix bounds. This class is for that.

At present it need not be virtual as only instance is CbcFixVariable, but ...

Definition at line 22 of file CbcConsequence.hpp.

### 4.19.2 Member Function Documentation

#### 4.19.2.1 virtual void CbcConsequence::applyToSolver ( **OsiSolverInterface** ∗ *solver,* int *state* ) const [pure virtual]

Apply to an LP solver.

Action depends on state

Implemented in CbcFixVariable.

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

- CbcConsequence.hpp

## 4.20 CbcCountRowCut Class Reference

**OsiRowCut** augmented with bookkeeping.

`#include <CbcCountRowCut.hpp>`

Inheritance diagram for CbcCountRowCut:

Collaboration diagram for CbcCountRowCut:

### Public Member Functions

- void increment (int change=1)

  *Increment the number of references.*
- int decrement (int change=1)

  *Decrement the number of references and return the number left.*
- void setInfo (CbcNodeInfo ∗, int whichOne)

  *Set the information associating this cut with a node.*
- int numberPointingToThis ()

  *Number of other CbcNodeInfo objects pointing to this row cut.*
- int whichCutGenerator () const

  *Which generator for cuts - as user order.*
- bool canDropCut (const **OsiSolverInterface** ∗solver, int **row**) const

*Returns true if can drop cut if slack basic.*

**Constructors & destructors**

- CbcCountRowCut ()
  *Default Constructor.*
- CbcCountRowCut (const **OsiRowCut** &)
  *'Copy' constructor using an* **OsiRowCut**
- CbcCountRowCut (const **OsiRowCut** &, CbcNodeInfo ∗, int whichOne, int whichGenerator=-1, int numberPointingToThis=0)
  *'Copy' constructor using an* **OsiRowCut** *and an CbcNodeInfo*
- virtual ∼CbcCountRowCut ()
  *Destructor.*

### 4.20.1 Detailed Description

**OsiRowCut** augmented with bookkeeping.

CbcCountRowCut is an **OsiRowCut** object augmented with bookkeeping information: a reference count and information that specifies the the generator that created the cut and the node to which it's associated.

The general principles for handling the reference count are as follows:

- Once it's determined how the node will branch, increment the reference count under the assumption that all children will use all cuts currently tight at the node and will survive to be placed in the search tree.
- As this assumption is proven incorrect (a cut becomes loose, or a child is fathomed), decrement the reference count accordingly.

When all possible uses of a cut have been demonstrated to be unnecessary, the reference count (#numberPointingToThis_) will fall to zero. The CbcCountRowCut object (and its included **OsiRowCut** object) are then deleted.

Definition at line 35 of file CbcCountRowCut.hpp.

### 4.20.2 Constructor & Destructor Documentation

#### 4.20.2.1 virtual CbcCountRowCut::∼CbcCountRowCut ( ) [virtual]

Destructor.

**Note**

The destructor will reach out (via #owner_) and NULL the reference to the cut in the owner's cuts_ list.

### 4.20.3 Member Function Documentation

#### 4.20.3.1 void CbcCountRowCut::setInfo ( CbcNodeInfo ∗ , int *whichOne* )

Set the information associating this cut with a node.

An CbcNodeInfo object and an index in the cut set of the node. For locally valid cuts, the node will be the search tree node where the cut was generated. For globally valid cuts, it's the node where the cut was activated.

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

- CbcCountRowCut.hpp

## 4.21 CbcCutBranchingObject Class Reference

Cut branching object.

`#include <CbcBranchCut.hpp>`

Inheritance diagram for CbcCutBranchingObject:

Collaboration diagram for CbcCutBranchingObject:

### Public Member Functions

- CbcCutBranchingObject ()

  *Default constructor.*
- CbcCutBranchingObject (CbcModel ∗model, **OsiRowCut** &down, **OsiRowCut** &up, bool canFix)

  *Create a cut branching object.*
- CbcCutBranchingObject (const CbcCutBranchingObject &)

  *Copy constructor.*
- CbcCutBranchingObject & operator= (const CbcCutBranchingObject &rhs)

  *Assignment operator.*
- virtual CbcBranchingObject ∗ clone () const

  *Clone.*
- virtual ∼CbcCutBranchingObject ()

  *Destructor.*
- virtual double branch ()

  *Sets the bounds for variables or adds a cut depending on the current arm of the branch and advances the object state to the next arm.*
- virtual void print ()

  *Print something about branch - only if log level high.*
- virtual bool boundBranch () const

  *Return true if branch should fix variables.*
- virtual CbcBranchObjType type () const

  *Return the type (an integer identifier) of `this`.*
- virtual int compareOriginalObject (const CbcBranchingObject ∗brObj) const

  *Compare the original object of `this` with the original object of `brObj`.*
- virtual CbcRangeCompare compareBranchingObject (const CbcBranchingObject ∗brObj, const bool replaceIf↩ Overlap=false)

  *Compare the `this` with `brObj`.*

### Protected Attributes

- **OsiRowCut** down_

  *Cut for the down arm (way_ = -1)*
- **OsiRowCut** up_

  *Cut for the up arm (way_ = 1)*
- bool canFix_

  *True if one way can fix variables.*

### 4.21.1 Detailed Description

Cut branching object.

This object can specify a two-way branch in terms of two cuts

Definition at line 108 of file CbcBranchCut.hpp.

### 4.21.2 Constructor & Destructor Documentation

#### 4.21.2.1 CbcCutBranchingObject::CbcCutBranchingObject ( **CbcModel** ∗ *model,* **OsiRowCut** & *down,* **OsiRowCut** & *up,* bool *canFix* )

Create a cut branching object.

Cut down will applied on way=-1, up on way==1 Assumed down will be first so way_ set to -1

### 4.21.3 Member Function Documentation

#### 4.21.3.1 virtual double CbcCutBranchingObject::branch ( ) `[virtual]`

Sets the bounds for variables or adds a cut depending on the current arm of the branch and advances the object state to the next arm.

Returns change in guessed objective on next branch

Implements CbcBranchingObject.

#### 4.21.3.2 virtual int CbcCutBranchingObject::compareOriginalObject ( const **CbcBranchingObject** ∗ *brObj* ) const `[virtual]`

Compare the original object of `this` with the original object of `brObj`.

Assumes that there is an ordering of the original objects. This method should be invoked only if `this` and brObj are of the same type. Return negative/0/positive depending on whether `this` is smaller/same/larger than the argument.

Reimplemented from CbcBranchingObject.

#### 4.21.3.3 virtual CbcRangeCompare CbcCutBranchingObject::compareBranchingObject ( const **CbcBranchingObject** ∗ *brObj,* const bool *replaceIfOverlap* = `false` ) `[virtual]`

Compare the `this` with `brObj`.

`this` and `brObj` must be os the same type and must have the same original object, but they may have different feasible regions. Return the appropriate CbcRangeCompare value (first argument being the sub/superset if that's the case). In case of overlap (and if `replaceIfOverlap` is true) replace the current branching object with one whose feasible region is the overlap.

Implements CbcBranchingObject.

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

- CbcBranchCut.hpp

## 4.22 CbcCutGenerator Class Reference

Interface between Cbc and Cut Generation Library.

`#include <CbcCutGenerator.hpp>`

### Public Member Functions

#### Generate Cuts

- bool generateCuts (**OsiCuts** &cs, int fullScan, **OsiSolverInterface** ∗solver, CbcNode ∗node)
  *Generate cuts for the client model.*

#### Constructors and destructors

- CbcCutGenerator ()
  *Default constructor.*
- CbcCutGenerator (CbcModel ∗model, **CglCutGenerator** ∗generator, int howOften=1, const char ∗name=N←ULL, bool normal=true, bool atSolution=false, bool infeasible=false, int howOftenInsub=-100, int whatDepth=-1, int whatDepthInSub=-1, int switchOffIfLessThan=0)
  *Normal constructor.*
- CbcCutGenerator (const CbcCutGenerator &)
  *Copy constructor.*
- CbcCutGenerator & operator= (const CbcCutGenerator &rhs)
  *Assignment operator.*
- ∼CbcCutGenerator ()
  *Destructor.*

#### Gets and sets

- void refreshModel (CbcModel ∗model)
  *Set the client model.*
- const char ∗ cutGeneratorName () const
  *return name of generator*
- void generateTuning (FILE ∗fp)
  *Create C++ lines to show how to tune.*
- void setHowOften (int value)
  *Set the cut generation interval.*
- int howOften () const
  *Get the cut generation interval.*
- int howOftenInSub () const
  *Get the cut generation interval.in sub tree.*
- int inaccuracy () const
  *Get level of cut inaccuracy (0 means exact e.g. cliques)*
- void setInaccuracy (int level)
  *Set level of cut inaccuracy (0 means exact e.g. cliques)*
- void setWhatDepth (int value)
  *Set the cut generation depth.*
- void setWhatDepthInSub (int value)
  *Set the cut generation depth in sub tree.*
- int whatDepth () const
  *Get the cut generation depth criterion.*

- int whatDepthInSub () const

  *Get the cut generation depth criterion.in sub tree.*
- void setMaximumTries (int value)

  *Set maximum number of times to enter.*
- int maximumTries () const

  *Get maximum number of times to enter.*
- int switches () const

  *Get switches.*
- void setSwitches (int value)

  *Set switches (for copying from virgin state)*
- bool normal () const

  *Get whether the cut generator should be called in the normal place.*
- void setNormal (bool value)

  *Set whether the cut generator should be called in the normal place.*
- bool atSolution () const

  *Get whether the cut generator should be called when a solution is found.*
- void setAtSolution (bool value)

  *Set whether the cut generator should be called when a solution is found.*
- bool whenInfeasible () const

  *Get whether the cut generator should be called when the subproblem is found to be infeasible.*
- void setWhenInfeasible (bool value)

  *Set whether the cut generator should be called when the subproblem is found to be infeasible.*
- bool timing () const

  *Get whether the cut generator is being timed.*
- void setTiming (bool value)

  *Set whether the cut generator is being timed.*
- double timeInCutGenerator () const

  *Return time taken in cut generator.*
- void **incrementTimeInCutGenerator** (double value)
- **CglCutGenerator** ∗ generator () const

  *Get the* **CglCutGenerator** *corresponding to this* `CbcCutGenerator`*.*
- int numberTimesEntered () const

  *Number times cut generator entered.*
- void **setNumberTimesEntered** (int value)
- void **incrementNumberTimesEntered** (int value=1)
- int numberCutsInTotal () const

  *Total number of cuts added.*
- void **setNumberCutsInTotal** (int value)
- void **incrementNumberCutsInTotal** (int value=1)
- int numberElementsInTotal () const

  *Total number of elements added.*
- void **setNumberElementsInTotal** (int value)
- void **incrementNumberElementsInTotal** (int value=1)
- int numberColumnCuts () const

  *Total number of column cuts.*
- void **setNumberColumnCuts** (int value)
- void **incrementNumberColumnCuts** (int value=1)
- int numberCutsActive () const

  *Total number of cuts active after (at end of n cut passes at each node)*
- void **setNumberCutsActive** (int value)
- void **incrementNumberCutsActive** (int value=1)
- void **setSwitchOffIfLessThan** (int value)
- int **switchOffIfLessThan** () const

- bool needsOptimalBasis () const
  *Say if optimal basis needed.*
- void setNeedsOptimalBasis (bool yesNo)
  *Set if optimal basis needed.*
- bool mustCallAgain () const
  *Whether generator MUST be called again if any cuts (i.e. ignore break from loop)*
- void setMustCallAgain (bool yesNo)
  *Set whether generator MUST be called again if any cuts (i.e. ignore break from loop)*
- bool switchedOff () const
  *Whether generator switched off for moment.*
- void setSwitchedOff (bool yesNo)
  *Set whether generator switched off for moment.*
- bool ineffectualCuts () const
  *Whether last round of cuts did little.*
- void setIneffectualCuts (bool yesNo)
  *Set whether last round of cuts did little.*
- bool whetherToUse () const
  *Whether to use if any cuts generated.*
- void setWhetherToUse (bool yesNo)
  *Set whether to use if any cuts generated.*
- bool whetherInMustCallAgainMode () const
  *Whether in must call again mode (or after others)*
- void setWhetherInMustCallAgainMode (bool yesNo)
  *Set whether in must call again mode (or after others)*
- bool whetherCallAtEnd () const
  *Whether to call at end.*
- void setWhetherCallAtEnd (bool yesNo)
  *Set whether to call at end.*
- bool needsRefresh () const
  *Whether needs refresh on copy.*
- void setNeedsRefresh (bool yesNo)
  *Set whether needs refresh on copy.*
- int numberCutsAtRoot () const
  *Number of cuts generated at root.*
- void **setNumberCutsAtRoot** (int value)
- int numberActiveCutsAtRoot () const
  *Number of cuts active at root.*
- void **setNumberActiveCutsAtRoot** (int value)
- int numberShortCutsAtRoot () const
  *Number of short cuts at root.*
- void **setNumberShortCutsAtRoot** (int value)
- void setModel (CbcModel ∗model)
  *Set model.*
- bool globalCutsAtRoot () const
  *Whether global cuts at root.*
- void setGlobalCutsAtRoot (bool yesNo)
  *Set whether global cuts at root.*
- bool globalCuts () const
  *Whether global cuts.*
- void setGlobalCuts (bool yesNo)
  *Set whether global cuts.*
- void addStatistics (const CbcCutGenerator ∗other)
  *Add in statistics from other.*
- void scaleBackStatistics (int factor)
  *Scale back statistics by factor.*

### 4.22.1 Detailed Description

Interface between Cbc and Cut Generation Library.

`CbcCutGenerator` is intended to provide an intelligent interface between Cbc and the cutting plane algorithms in the CGL. A `CbcCutGenerator` is bound to a **CglCutGenerator** and to an `CbcModel`. It contains parameters which control when and how the `generateCuts` method of the **CglCutGenerator** will be called.

The builtin decision criteria available to use when deciding whether to generate cuts are limited: every *X* nodes, when a solution is found, and when a subproblem is found to be infeasible. The idea is that the class will grow more intelligent with time.

Definition at line 49 of file CbcCutGenerator.hpp.

### 4.22.2 Member Function Documentation

#### 4.22.2.1 bool CbcCutGenerator::generateCuts ( OsiCuts & *cs,* int *fullScan,* OsiSolverInterface ∗ *solver,* CbcNode ∗ *node* )

Generate cuts for the client model.

Evaluate the state of the client model and decide whether to generate cuts. The generated cuts are inserted into and returned in the collection of cuts `cs`.

If `fullScan` is !=0, the generator is obliged to call the CGL `generateCuts` routine. Otherwise, it is free to make a local decision. Negative fullScan says things like at integer solution The current implementation uses `whenCutGenerator_` to decide.

The routine returns true if reoptimisation is needed (because the state of the solver interface has been modified).

If node then can find out depth

#### 4.22.2.2 void CbcCutGenerator::refreshModel ( CbcModel ∗ *model* )

Set the client model.

In addition to setting the client model, refreshModel also calls the `refreshSolver` method of the **CglCutGenerator** object.

#### 4.22.2.3 void CbcCutGenerator::setHowOften ( int *value* )

Set the cut generation interval.

Set the number of nodes evaluated between calls to the Cgl object's `generateCuts` routine.

If `value` is positive, cuts will always be generated at the specified interval. If `value` is negative, cuts will initially be generated at the specified interval, but Cbc may adjust the value depending on the success of cuts produced by this generator.

A value of -100 disables the generator, while a value of -99 means just at root.

#### 4.22.2.4 void CbcCutGenerator::setWhatDepth ( int *value* )

Set the cut generation depth.

Set the depth criterion for calls to the Cgl object's `generateCuts` routine. Only active if $>$ 0.

If whenCutGenerator is positive and this is positive then this overrides. If whenCutGenerator is -1 then this is used as criterion if any cuts were generated at root node. If whenCutGenerator is anything else this is ignored.

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

- CbcCutGenerator.hpp

## 4.23 CbcCutModifier Class Reference

Abstract cut modifier base class.

`#include <CbcCutModifier.hpp>`

Inheritance diagram for CbcCutModifier:

### Public Member Functions

- CbcCutModifier ()

  *Default Constructor.*
- virtual ~CbcCutModifier ()

  *Destructor.*
- CbcCutModifier & operator= (const CbcCutModifier &rhs)

  *Assignment.*
- virtual CbcCutModifier * clone () const =0

  *Clone.*
- virtual int modify (const **OsiSolverInterface** *solver, **OsiRowCut** &**cut**)=0

  *Returns 0 unchanged 1 strengthened 2 weakened 3 deleted.*
- virtual void generateCpp (FILE *)

  *Create C++ lines to get to current state.*

### 4.23.1 Detailed Description

Abstract cut modifier base class.

In exotic circumstances - cuts may need to be modified a) strengthened - changed b) weakened - changed c) deleted - set to NULL d) unchanged

Definition at line 27 of file CbcCutModifier.hpp.

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

- CbcCutModifier.hpp

## 4.24 CbcCutSubsetModifier Class Reference

Simple cut modifier base class.

`#include <CbcCutSubsetModifier.hpp>`

Inheritance diagram for CbcCutSubsetModifier:

Collaboration diagram for CbcCutSubsetModifier:

### Public Member Functions

- CbcCutSubsetModifier ()

    *Default Constructor.*
- CbcCutSubsetModifier (int firstOdd)

    *Useful Constructor.*
- virtual ~CbcCutSubsetModifier ()

    *Destructor.*
- CbcCutSubsetModifier & operator= (const CbcCutSubsetModifier &rhs)

    *Assignment.*
- virtual CbcCutModifier * clone () const

    *Clone.*
- virtual int modify (const **OsiSolverInterface** *solver, **OsiRowCut** &**cut**)

    *Returns 0 unchanged 1 strengthened 2 weakened 3 deleted.*
- virtual void generateCpp (FILE *)

    *Create C++ lines to get to current state.*

### Protected Attributes

- int firstOdd_

    *data First odd variable*

### 4.24.1 Detailed Description

Simple cut modifier base class.

In exotic circumstances - cuts may need to be modified a) strengthened - changed b) weakened - changed c) deleted - set to NULL d) unchanged

initially get rid of cuts with variables >= k could weaken

Definition at line 31 of file CbcCutSubsetModifier.hpp.

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

- CbcCutSubsetModifier.hpp

## 4.25 CbcDummyBranchingObject Class Reference

Dummy branching object.

`#include <CbcDummyBranchingObject.hpp>`

Inheritance diagram for CbcDummyBranchingObject:

Collaboration diagram for CbcDummyBranchingObject:

### Public Member Functions

- CbcDummyBranchingObject (CbcModel *model=NULL)

    *Default constructor.*

- CbcDummyBranchingObject (const CbcDummyBranchingObject &)

  *Copy constructor.*
- CbcDummyBranchingObject & operator= (const CbcDummyBranchingObject &rhs)

  *Assignment operator.*
- virtual CbcBranchingObject ∗ clone () const

  *Clone.*
- virtual ∼CbcDummyBranchingObject ()

  *Destructor.*
- virtual double branch ()

  *Dummy branch.*
- virtual void print ()

  *Print something about branch - only if log level high.*
- virtual CbcBranchObjType type () const

  *Return the type (an integer identifier) of `this`.*
- virtual int compareOriginalObject (const CbcBranchingObject ∗brObj) const

  *Compare the original object of `this` with the original object of `brObj`.*
- virtual CbcRangeCompare compareBranchingObject (const CbcBranchingObject ∗brObj, const bool replaceIf←Overlap=false)

  *Compare the `this` with `brObj`.*

## Additional Inherited Members

### 4.25.1 Detailed Description

Dummy branching object.

This object specifies a one-way dummy branch. This is so one can carry on branching even when it looks feasible

Definition at line 18 of file CbcDummyBranchingObject.hpp.

### 4.25.2 Member Function Documentation

#### 4.25.2.1 virtual int CbcDummyBranchingObject::compareOriginalObject ( const CbcBranchingObject ∗ *brObj* ) const [virtual]

Compare the original object of `this` with the original object of `brObj`.

Assumes that there is an ordering of the original objects. This method should be invoked only if `this` and brObj are of the same type. Return negative/0/positive depending on whether `this` is smaller/same/larger than the argument.

Reimplemented from CbcBranchingObject.

#### 4.25.2.2 virtual CbcRangeCompare CbcDummyBranchingObject::compareBranchingObject ( const CbcBranchingObject ∗ *brObj*, const bool *replaceIfOverlap* = `false` ) [virtual]

Compare the `this` with `brObj`.

`this` and `brObj` must be os the same type and must have the same original object, but they may have different feasible regions. Return the appropriate CbcRangeCompare value (first argument being the sub/superset if that's the case). In case of overlap (and if `replaceIfOverlap` is true) replace the current branching object with one whose feasible region is the overlap.

Implements CbcBranchingObject.

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

- CbcDummyBranchingObject.hpp

## 4.26 CbcDynamicPseudoCostBranchingObject Class Reference

Simple branching object for an integer variable with pseudo costs.

`#include <CbcBranchDynamic.hpp>`

Inheritance diagram for CbcDynamicPseudoCostBranchingObject:

Collaboration diagram for CbcDynamicPseudoCostBranchingObject:

### Public Member Functions

- CbcDynamicPseudoCostBranchingObject ()

  *Default constructor.*
- CbcDynamicPseudoCostBranchingObject (CbcModel ∗model, int variable, int way, double **value**, CbcSimple←IntegerDynamicPseudoCost ∗object)

  *Create a standard floor/ceiling branch object.*
- CbcDynamicPseudoCostBranchingObject (CbcModel ∗model, int variable, int way, double lowerValue, double upperValue)

  *Create a degenerate branch object.*
- CbcDynamicPseudoCostBranchingObject (const CbcDynamicPseudoCostBranchingObject &)

  *Copy constructor.*
- CbcDynamicPseudoCostBranchingObject & operator= (const CbcDynamicPseudoCostBranchingObject &rhs)

  *Assignment operator.*
- virtual CbcBranchingObject ∗ clone () const

  *Clone.*
- virtual ∼CbcDynamicPseudoCostBranchingObject ()

  *Destructor.*
- void fillPart (int variable, int way, double **value**, CbcSimpleIntegerDynamicPseudoCost ∗object)

  *Does part of constructor.*
- virtual double branch ()

  *Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.*
- virtual int fillStrongInfo (CbcStrongInfo &info)

  *Some branchingObjects may claim to be able to skip strong branching.*
- double changeInGuessed () const

  *Change in guessed.*
- void setChangeInGuessed (double **value**)

  *Set change in guessed.*
- CbcSimpleIntegerDynamicPseudoCost ∗ object () const

  *Return object.*
- void setObject (CbcSimpleIntegerDynamicPseudoCost ∗object)

  *Set object.*
- virtual CbcBranchObjType type () const

  *Return the type (an integer identifier) of `this`.*

## Protected Attributes

- double changeInGuessed_

    *Change in guessed objective value for next branch.*
- CbcSimpleIntegerDynamicPseudoCost ∗ object_

    *Pointer back to object.*

### 4.26.1 Detailed Description

Simple branching object for an integer variable with pseudo costs.

This object can specify a two-way branch on an integer variable. For each arm of the branch, the upper and lower bounds on the variable can be independently specified.

Variable_ holds the index of the integer variable in the integerVariable_ array of the model.

Definition at line 111 of file CbcBranchDynamic.hpp.

### 4.26.2 Constructor & Destructor Documentation

#### 4.26.2.1 CbcDynamicPseudoCostBranchingObject::CbcDynamicPseudoCostBranchingObject ( **CbcModel** ∗ *model,* int *variable,* int *way,* double *value,* **CbcSimpleIntegerDynamicPseudoCost** ∗ *object* )

Create a standard floor/ceiling branch object.

Specifies a simple two-way branch. Let `value` = x∗. One arm of the branch will be is lb <= x <= floor(x∗), the other ceil(x∗) <= x <= ub. Specify way = -1 to set the object state to perform the down arm first, way = 1 for the up arm.

#### 4.26.2.2 CbcDynamicPseudoCostBranchingObject::CbcDynamicPseudoCostBranchingObject ( **CbcModel** ∗ *model,* int *variable,* int *way,* double *lowerValue,* double *upperValue* )

Create a degenerate branch object.

Specifies a ‘one-way branch'. Calling branch() for this object will always result in lowerValue <= x <= upperValue. Used to fix a variable when lowerValue = upperValue.

### 4.26.3 Member Function Documentation

#### 4.26.3.1 virtual double CbcDynamicPseudoCostBranchingObject::branch ( ) `[virtual]`

Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.

This version also changes guessed objective value

Reimplemented from CbcIntegerBranchingObject.

#### 4.26.3.2 virtual int CbcDynamicPseudoCostBranchingObject::fillStrongInfo ( **CbcStrongInfo** & *info* ) `[virtual]`

Some branchingObjects may claim to be able to skip strong branching.

If so they have to fill in CbcStrongInfo. The object mention in incoming CbcStrongInfo must match. Returns nonzero if skip is wanted

Reimplemented from CbcBranchingObject.

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

- CbcBranchDynamic.hpp

## 4.27 CbcEventHandler Class Reference

Base class for Cbc event handling.

`#include <CbcEventHandler.hpp>`

Collaboration diagram for CbcEventHandler:

### Public Types

- enum CbcEvent {
  node = 200, treeStatus, solution, heuristicSolution,
  beforeSolution1, beforeSolution2, afterHeuristic, smallBranchAndBound,
  heuristicPass, convertToCuts, endSearch }

  *Events known to cbc.*
- enum CbcAction {
  noAction = -1, stop = 0, restart, restartRoot,
  addCuts, killSolution, takeAction }

  *Action codes returned by the event handler.*
- typedef std::map< CbcEvent, CbcAction > eaMapPair

  *Data type for event/action pairs.*

### Public Member Functions

#### Event Processing

- virtual CbcAction event (CbcEvent whichEvent)

  *Return the action to be taken for an event.*
- virtual CbcAction event (CbcEvent whichEvent, void ∗data)

  *Return the action to be taken for an event - and modify data.*

#### Constructors and destructors

- CbcEventHandler (CbcModel ∗model=0)

  *Default constructor.*
- CbcEventHandler (const CbcEventHandler &orig)

  *Copy constructor.*
- CbcEventHandler & operator= (const CbcEventHandler &rhs)

  *Assignment.*
- virtual CbcEventHandler ∗ clone () const

  *Clone (virtual) constructor.*
- virtual ∼CbcEventHandler ()

  *Destructor.*

#### Set/Get methods

- void setModel (CbcModel *model)

  *Set model.*
- const CbcModel * getModel () const

  *Get model.*
- void setDfltAction (CbcAction action)

  *Set the default action.*
- void setAction (CbcEvent event, CbcAction action)

  *Set the action code associated with an event.*

## Protected Attributes

### Data members

*Protected (as opposed to private) to allow access by derived classes.*

- CbcModel * model_

  *Pointer to associated CbcModel.*
- CbcAction dfltAction_

  *Default action.*
- eaMapPair * eaMap_

  *Pointer to a map that holds non-default event/action pairs.*

## 4.27.1 Detailed Description

Base class for Cbc event handling.

Up front: We're not talking about unanticipated events here. We're talking about anticipated events, in the sense that the code is going to make a call to event() and is prepared to obey the return value that it receives.

The general pattern for usage is as follows:

1. Create a CbcEventHandler object. This will be initialised with a set of default actions for every recognised event.
2. Attach the event handler to the CbcModel object.
3. When execution reaches the point where an event occurs, call the event handler as CbcEventHandler::event(the event). The return value will specify what the code should do in response to the event.

The return value associated with an event can be changed at any time.

Definition at line 82 of file CbcEventHandler.hpp.

## 4.27.2 Member Enumeration Documentation

### 4.27.2.1 enum CbcEventHandler::CbcEvent

Events known to cbc.

**Enumerator**

***node*** Processing of the current node is complete.

***treeStatus*** A tree status interval has arrived.

***solution*** A solution has been found.

***heuristicSolution*** A heuristic solution has been found.

***beforeSolution1*** A solution will be found unless user takes action (first check).
***beforeSolution2*** A solution will be found unless user takes action (thorough check).
***afterHeuristic*** After failed heuristic.
***smallBranchAndBound*** On entry to small branch and bound.
***heuristicPass*** After a pass of heuristic.
***convertToCuts*** When converting constraints to cuts.
***endSearch*** End of search.

Definition at line 88 of file CbcEventHandler.hpp.

#### 4.27.2.2 enum CbcEventHandler::CbcAction

Action codes returned by the event handler.

Specific values are chosen to match **ClpEventHandler** return codes.

**Enumerator**

***noAction*** Continue — no action required.
***stop*** Stop — abort the current run at the next opportunity.
***restart*** Restart — restart branch-and-cut search; do not undo root node processing.
***restartRoot*** RestartRoot — undo root node and start branch-and-cut afresh.
***addCuts*** Add special cuts.
***killSolution*** Pretend solution never happened.
***takeAction*** Take action on modified data.

Definition at line 117 of file CbcEventHandler.hpp.

### 4.27.3 Constructor & Destructor Documentation

#### 4.27.3.1 CbcEventHandler::CbcEventHandler ( CbcModel * *model* = 0 )

Default constructor.

#### 4.27.3.2 CbcEventHandler::CbcEventHandler ( const CbcEventHandler & *orig* )

Copy constructor.

#### 4.27.3.3 virtual CbcEventHandler::~CbcEventHandler ( ) [virtual]

Destructor.

### 4.27.4 Member Function Documentation

#### 4.27.4.1 virtual CbcAction CbcEventHandler::event ( CbcEvent *whichEvent* ) [virtual]

Return the action to be taken for an event.

Return the action that should be taken in response to the event passed as the parameter. The default implementation simply reads a return code from a map.

#### 4.27.4.2 virtual CbcAction CbcEventHandler::event ( CbcEvent *whichEvent*, void * *data* ) [virtual]

Return the action to be taken for an event - and modify data.

Return the action that should be taken in response to the event passed as the parameter. The default implementation simply reads a return code from a map.

#### 4.27.4.3 CbcEventHandler& CbcEventHandler::operator= ( const CbcEventHandler & *rhs* )

Assignment.

#### 4.27.4.4 virtual CbcEventHandler* CbcEventHandler::clone ( ) const [virtual]

Clone (virtual) constructor.

#### 4.27.4.5 void CbcEventHandler::setModel ( CbcModel * *model* ) [inline]

Set model.

Definition at line 193 of file CbcEventHandler.hpp.

#### 4.27.4.6 const CbcModel* CbcEventHandler::getModel ( ) const [inline]

Get model.

Definition at line 199 of file CbcEventHandler.hpp.

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

- CbcEventHandler.hpp

## 4.28 CbcFathom Class Reference

Fathom base class.

`#include <CbcFathom.hpp>`

Inheritance diagram for CbcFathom:

Collaboration diagram for CbcFathom:

### Public Member Functions

- virtual void setModel (CbcModel *model)

  *update model (This is needed if cliques update matrix etc)*
- virtual CbcFathom * clone () const =0

  *Clone.*
- virtual void resetModel (CbcModel *model)=0

  *Resets stuff if model changes.*
- virtual int fathom (double *&newSolution)=0

  *returns 0 if no fathoming attempted, 1 fully fathomed, 2 incomplete search, 3 incomplete search but treat as complete.*

### Protected Attributes

- CbcModel ∗ model_

  *Model.*
- bool possible_

  *Possible - if this method of fathoming can be used.*

### 4.28.1 Detailed Description

Fathom base class.

The idea is that after some branching the problem will be effectively smaller than the original problem and maybe there will be a more specialized technique which can completely fathom this branch quickly.

One method is to presolve the problem to give a much smaller new problem and then do branch and cut on that. Another might be dynamic programming.

Definition at line 32 of file CbcFathom.hpp.

### 4.28.2 Member Function Documentation

#### 4.28.2.1 virtual int CbcFathom::fathom ( double ∗& *newSolution* ) [pure virtual]

returns 0 if no fathoming attempted, 1 fully fathomed, 2 incomplete search, 3 incomplete search but treat as complete.

If solution then newSolution will not be NULL and will be freed by CbcModel. It is expected that the solution is better than best so far but CbcModel will double check.

If returns 3 then of course there is no guarantee of global optimum

Implemented in CbcFathomDynamicProgramming.

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

- CbcFathom.hpp

## 4.29 CbcFathomDynamicProgramming Class Reference

FathomDynamicProgramming class.

`#include <CbcFathomDynamicProgramming.hpp>`

Inheritance diagram for CbcFathomDynamicProgramming:

Collaboration diagram for CbcFathomDynamicProgramming:

### Public Member Functions

- virtual void setModel (CbcModel ∗model)

  *update model (This is needed if cliques update matrix etc)*
- virtual CbcFathom ∗ clone () const

  *Clone.*
- virtual void resetModel (CbcModel ∗model)

  *Resets stuff if model changes.*

- virtual int fathom (double *&newSolution)

  *returns 0 if no fathoming attempted, 1 fully fathomed , 2 incomplete search, 3 incomplete search but treat as complete.*
- int maximumSize () const

  *Maximum size allowed.*
- int checkPossible (int allowableSize=0)

  *Returns type of algorithm and sets up arrays.*
- bool tryColumn (int numberElements, const int *rows, const double *coefficients, double cost, int upper=COIN↩ _INT_MAX)

  *Tries a column returns true if was used in making any changes.*
- const double * cost () const

  *Returns cost array.*
- const int * back () const

  *Returns back array.*
- int target () const

  *Gets bit pattern for target result.*
- void setTarget (int value)

  *Sets bit pattern for target result.*

## Protected Attributes

- int size_

  *Size of states (power of 2 unless just one constraint)*
- int type_

  *Type - 0 coefficients and rhs all 1, 1 - coefficients > 1 or rhs > 1.*
- double * cost_

  *Space for states.*
- int * back_

  *Which state produced this cheapest one.*
- int * lookup_

  *Some rows may be satisified so we need a lookup.*
- int * indices_

  *Space for sorted indices.*
- int numberActive_

  *Number of active rows.*
- int maximumSizeAllowed_

  *Maximum size allowed.*
- int * startBit_

  *Start bit for each active row.*
- int * numberBits_

  *Number bits for each active row.*
- int * rhs_

  *Effective rhs.*
- int * coefficients_

  *Space for sorted coefficients.*
- int target_

  *Target pattern.*
- int numberNonOne_

*Number of Non 1 rhs.*
- int bitPattern_

  *Current bit pattern.*
- int algorithm_

  *Current algorithm.*

### 4.29.1 Detailed Description

FathomDynamicProgramming class.

The idea is that after some branching the problem will be effectively smaller than the original problem and maybe there will be a more specialized technique which can completely fathom this branch quickly.

This is a dynamic programming implementation which is very fast for some specialized problems. It expects small integral rhs, an all integer problem and positive integral coefficients. At present it can not do general set covering problems just set partitioning. It can find multiple optima for various rhs combinations.

The main limiting factor is size of state space. Each 1 rhs doubles the size of the problem. 2 or 3 rhs quadruples, 4,5,6,7 by 8 etc.

Definition at line 28 of file CbcFathomDynamicProgramming.hpp.

### 4.29.2 Member Function Documentation

#### 4.29.2.1 virtual int CbcFathomDynamicProgramming::fathom ( double ∗& *newSolution* ) [virtual]

returns 0 if no fathoming attempted, 1 fully fathomed , 2 incomplete search, 3 incomplete search but treat as complete.

If solution then newSolution will not be NULL and will be freed by CbcModel. It is expected that the solution is better than best so far but CbcModel will double check.

If returns 3 then of course there is no guarantee of global optimum

Implements CbcFathom.

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

- CbcFathomDynamicProgramming.hpp

## 4.30 CbcFeasibilityBase Class Reference

### Public Member Functions

- virtual int feasible (CbcModel ∗, int)

  *On input mode: 0 - called after a solve but before any cuts -1 - called after strong branching Returns : 0 - no opinion -1 pretend infeasible 1 pretend integer solution.*
- virtual CbcFeasibilityBase ∗ clone () const

  *Clone.*

### 4.30.1 Detailed Description

Definition at line 22 of file CbcFeasibilityBase.hpp.

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

- CbcFeasibilityBase.hpp

## 4.31 CbcFixingBranchingObject Class Reference

General Branching Object class.

`#include <CbcFollowOn.hpp>`

Inheritance diagram for CbcFixingBranchingObject:

Collaboration diagram for CbcFixingBranchingObject:

### Public Member Functions

- virtual CbcBranchingObject * clone () const

  *Clone.*
- virtual double branch ()

  *Does next branch and updates state.*
- virtual void print ()

  *Print something about branch - only if log level high.*
- virtual CbcBranchObjType type () const

  *Return the type (an integer identifier) of `this`.*
- virtual int compareOriginalObject (const CbcBranchingObject *brObj) const

  *Compare the original object of `this` with the original object of `brObj`.*
- virtual CbcRangeCompare compareBranchingObject (const CbcBranchingObject *brObj, const bool replaceIf←Overlap=false)

  *Compare the `this` with `brObj`.*

### Additional Inherited Members

### 4.31.1 Detailed Description

General Branching Object class.

Each way fixes some variables to lower bound

Definition at line 74 of file CbcFollowOn.hpp.

### 4.31.2 Member Function Documentation

#### 4.31.2.1 virtual int CbcFixingBranchingObject::compareOriginalObject ( const CbcBranchingObject * *brObj* ) const [virtual]

Compare the original object of `this` with the original object of `brObj`.

Assumes that there is an ordering of the original objects. This method should be invoked only if `this` and brObj are of the same type. Return negative/0/positive depending on whether `this` is smaller/same/larger than the argument.

Reimplemented from CbcBranchingObject.

#### 4.31.2.2 virtual CbcRangeCompare CbcFixingBranchingObject::compareBranchingObject ( const CbcBranchingObject * *brObj*, const bool *replaceIfOverlap* = false ) [virtual]

Compare the `this` with `brObj`.

`this` and `brObj` must be os the same type and must have the same original object, but they may have different feasible regions. Return the appropriate CbcRangeCompare value (first argument being the sub/superset if that's the case). In case of overlap (and if `replaceIfOverlap` is true) replace the current branching object with one whose feasible region is the overlap.

Implements CbcBranchingObject.

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

- CbcFollowOn.hpp

## 4.32 CbcFixVariable Class Reference

Class for consequent bounds.

`#include <CbcFixVariable.hpp>`

Inheritance diagram for CbcFixVariable:

Collaboration diagram for CbcFixVariable:

### Public Member Functions

- virtual CbcConsequence * clone () const

  *Clone.*
- virtual ~CbcFixVariable ()

  *Destructor.*
- virtual void applyToSolver (**OsiSolverInterface** *solver, int state) const

  *Apply to an LP solver.*

### Protected Attributes

- int numberStates_

  *Number of states.*
- int * states_

  *Values of integers for various states.*
- int * startLower_

  *Start of information for each state (setting new lower)*
- int * startUpper_

  *Start of information for each state (setting new upper)*
- double * newBound_

  *For each variable new bounds.*
- int * variable_

  *Variable.*

### 4.32.1 Detailed Description

Class for consequent bounds.

When a variable is branched on it normally interacts with other variables by means of equations. There are cases where we want to step outside LP and do something more directly e.g. fix bounds. This class is for that.

A state of -9999 means at LB, +9999 means at UB, others mean if fixed to that value.

Definition at line 22 of file CbcFixVariable.hpp.

### 4.32.2 Member Function Documentation

#### 4.32.2.1 virtual void CbcFixVariable::applyToSolver ( **OsiSolverInterface** ∗ *solver,* int *state* ) const [virtual]

Apply to an LP solver.

Action depends on state

Implements CbcConsequence.

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

- CbcFixVariable.hpp

## 4.33 CbcFollowOn Class Reference

Define a follow on class.

`#include <CbcFollowOn.hpp>`

Inheritance diagram for CbcFollowOn:

Collaboration diagram for CbcFollowOn:

### Public Member Functions

- CbcFollowOn (CbcModel ∗model)

  *Useful constructor.*
- virtual CbcObject ∗ clone () const

  *Clone.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗info, int &preferredWay) const

  *Infeasibility - large is 0.5.*
- virtual void feasibleRegion ()

  *This looks at solution and sets bounds to contain solution.*
- virtual CbcBranchingObject ∗ createCbcBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way)

  *Creates a branching object.*
- virtual int gutsOfFollowOn (int &otherRow, int &preferredWay) const

  *As some computation is needed in more than one place - returns row.*

### Protected Attributes

- **CoinPackedMatrix** matrix_

  *data Matrix*
- **CoinPackedMatrix** matrixByRow_

  *Matrix by row.*
- int ∗ rhs_

  *Possible rhs (if 0 then not possible)*

### 4.33.1 Detailed Description

Define a follow on class.

The idea of this is that in air-crew scheduling problems crew may fly in on flight A and out on flight B or on some other flight. A useful branch is one which on one side fixes all which go out on flight B to 0, while the other branch fixes all those that do NOT go out on flight B to 0.

This branching rule should be in addition to normal rules and have a high priority.

Definition at line 25 of file CbcFollowOn.hpp.

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

- CbcFollowOn.hpp

## 4.34 CbcFullNodeInfo Class Reference

Information required to recreate the subproblem at this node.

`#include <CbcFullNodeInfo.hpp>`

Inheritance diagram for CbcFullNodeInfo:

Collaboration diagram for CbcFullNodeInfo:

### Public Member Functions

- virtual void applyToModel (CbcModel ∗model, **CoinWarmStartBasis** ∗&basis, CbcCountRowCut ∗∗addCuts, int &currentNumberCuts) const

  *Modify model according to information at node.*
- virtual int applyBounds (int iColumn, double &lower, double &upper, int force)

  *Just apply bounds to one variable - force means overwrite by lower,upper (1=>infeasible)*
- virtual CbcNodeInfo ∗ buildRowBasis (**CoinWarmStartBasis** &basis) const

  *Builds up row basis backwards (until original model).*
- CbcFullNodeInfo (CbcModel ∗model, int numberRowsAtContinuous)

  *Constructor from continuous or satisfied.*
- virtual CbcNodeInfo ∗ clone () const

  *Clone.*
- const double ∗ lower () const

  *Lower bounds.*
- void setColLower (int sequence, double value)

*Set a bound.*

- double * mutableLower () const

  *Mutable lower bounds.*

- const double * upper () const

  *Upper bounds.*

- void setColUpper (int sequence, double value)

  *Set a bound.*

- double * mutableUpper () const

  *Mutable upper bounds.*

## Protected Attributes

- **CoinWarmStartBasis** * basis_

  *Full basis.*

### 4.34.1 Detailed Description

Information required to recreate the subproblem at this node.

When a subproblem is initially created, it is represented by a CbcNode object and an attached CbcNodeInfo object.

The CbcNode contains information needed while the subproblem remains live. The CbcNode is deleted when the last branch arm has been evaluated.

The CbcNodeInfo contains information required to maintain the branch-and-cut search tree structure (links and reference counts) and to recreate the subproblem for this node (basis, variable bounds, cutting planes). A CbcNodeInfo object remains in existence until all nodes have been pruned from the subtree rooted at this node.

The principle used to maintain the reference count is that the reference count is always the sum of all potential and actual children of the node. Specifically,

- Once it's determined how the node will branch, the reference count is set to the number of potential children (*i.e.*, the number of arms of the branch).
- As each child is created by CbcNode::branch() (converting a potential child to the active subproblem), the reference count is decremented.
- If the child survives and will become a node in the search tree (converting the active subproblem into an actual child), increment the reference count.

Notice that the active subproblem lives in a sort of limbo, neither a potential or an actual node in the branch-and-cut tree.

CbcNodeInfo objects come in two flavours. A CbcFullNodeInfo object contains a full record of the information required to recreate a subproblem. A CbcPartialNodeInfo object expresses this information in terms of differences from the parent.Holds complete information for recreating a subproblem.

A CbcFullNodeInfo object contains all necessary information (bounds, basis, and cuts) required to recreate a subproblem.

Definition at line 81 of file CbcFullNodeInfo.hpp.

### 4.34.2 Member Function Documentation

#### 4.34.2.1 virtual void CbcFullNodeInfo::applyToModel ( CbcModel * *model*, CoinWarmStartBasis *& *basis*, CbcCountRowCut ** *addCuts*, int & *currentNumberCuts* ) const [virtual]

Modify model according to information at node.

The routine modifies the model according to bound information at node, creates a new basis according to information at node, but with the size passed in through basis, and adds any cuts to the addCuts array.

**Note**

The basis passed in via basis is solely a vehicle for passing in the desired basis size. It will be deleted and a new basis returned.

Implements CbcNodeInfo.

#### 4.34.2.2 virtual CbcNodeInfo* CbcFullNodeInfo::buildRowBasis ( CoinWarmStartBasis & *basis* ) const [virtual]

Builds up row basis backwards (until original model).

Returns NULL or previous one to apply . Depends on Free being 0 and impossible for cuts

Implements CbcNodeInfo.

### 4.34.3 Member Data Documentation

#### 4.34.3.1 CoinWarmStartBasis* CbcFullNodeInfo::basis_ [protected]

Full basis.

This MUST BE A POINTER to avoid cutting extra information in derived warm start classes.

Definition at line 151 of file CbcFullNodeInfo.hpp.

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

- CbcFullNodeInfo.hpp

## 4.35 CbcGenCtlBlk Class Reference

Collaboration diagram for CbcGenCtlBlk:

### Classes

- struct babState_struct

  *State of branch-and-cut.*
- struct cbcParamsInfo_struct

  *Start and end of CbcModel parameters in parameter vector.*
- struct chooseStrongCtl_struct

  *Control variables for a strong branching method.*
- struct debugSolInfo_struct

*Array of primal variable values for debugging.*
- struct djFixCtl_struct

  *Control use of reduced cost fixing prior to B&C.*
- struct genParamsInfo_struct

  *Start and end of cbc-generic parameters in parameter vector.*
- struct osiParamsInfo_struct

  *Start and end of* **OsiSolverInterface** *parameters in parameter vector.*

## Public Types

### Enumeration types used for cbc-generic control variables

- enum IPPControl

  *Codes to control integer preprocessing.*
- enum CGControl

  *Codes to control the use of cut generators and heuristics.*
- enum BPControl

  *Codes to specify the assignment of branching priorities.*
- enum BACMajor

  *Major status codes for branch-and-cut.*
- enum BACMinor

  *Minor status codes.*
- enum BACWhere

  *Codes to specify where branch-and-cut stopped.*

## Public Member Functions

### Constructors and destructors

- CbcGenCtlBlk ()

  *Default constructor.*
- ∼CbcGenCtlBlk ()

  *Destructor.*

### Access and Control Functions for Cut Generators and Heuristics

*Control functions, plus lazy creation functions for cut generators and heuristics*

*cbc-generic avoids creating objects for cut generators and heuristics unless they're actually used. For cut generators, a prototype is created and reused. For heuristics, the default is to create a new object with each call, because the model may have changed. The object is returned through the reference parameter. The return value of the function is the current action state.*

*Cut generator and heuristic objects created by these calls will be deleted with the destruction of the CbcGenCtlBlk object.*

- int getCutDepth ()

  *Get cut depth setting.*
- void setCutDepth (int cutDepth)

  *Set cut depth setting.*
- IPPControl **getIPPAction** ()
- void setIPPAction (IPPControl action)

  *Set action state for use of integer preprocessing.*
- CGControl getProbing (**CglCutGenerator** ∗&gen)

*Obtain a prototype for a probing cut generator.*
- void setProbingAction (CGControl action)

  *Set action state for use of probing cut generator.*
- CGControl getClique (**CglCutGenerator** ∗&gen)

  *Obtain a prototype for a clique cut generator.*
- void setCliqueAction (CGControl action)

  *Set action state for use of clique cut generator.*
- CGControl getFlow (**CglCutGenerator** ∗&gen)

  *Obtain a prototype for a flow cover cut generator.*
- void setFlowAction (CGControl action)

  *Set action state for use of flow cover cut generator.*
- CGControl getGomory (**CglCutGenerator** ∗&gen)

  *Obtain a prototype for a Gomory cut generator.*
- void setGomoryAction (CGControl action)

  *Set action state for use of Gomory cut generator.*
- CGControl getKnapsack (**CglCutGenerator** ∗&gen)

  *Obtain a prototype for a knapsack cover cut generator.*
- void setKnapsackAction (CGControl action)

  *Set action state for use of knapsack cut generator.*
- CGControl getMir (**CglCutGenerator** ∗&gen)

  *Obtain a prototype for a mixed integer rounding (MIR) cut generator.*
- void setMirAction (CGControl action)

  *Set action state for use of MIR cut generator.*
- CGControl getRedSplit (**CglCutGenerator** ∗&gen)

  *Obtain a prototype for a reduce and split cut generator.*
- void setRedSplitAction (CGControl action)

  *Set action state for use of reduce and split cut generator.*
- CGControl getTwomir (**CglCutGenerator** ∗&gen)

  *Obtain a prototype for a 2-MIR cut generator.*
- void setTwomirAction (CGControl action)

  *Set action state for use of 2-MIR cut generator.*
- CGControl getFPump (CbcHeuristic ∗&gen, CbcModel ∗model, bool alwaysCreate=true)

  *Obtain a feasibility pump heuristic.*
- void setFPumpAction (CGControl action)

  *Set action state for use of feasibility pump heuristic.*
- CGControl getCombine (CbcHeuristic ∗&gen, CbcModel ∗model, bool alwaysCreate=true)

  *Obtain a local search/combine heuristic.*
- void setCombineAction (CGControl action)

  *Set action state for use of local search/combine heuristic.*
- CGControl getGreedyCover (CbcHeuristic ∗&gen, CbcModel ∗model, bool alwaysCreate=true)

  *Obtain a greedy cover heuristic.*
- void setGreedyCoverAction (CGControl action)

  *Set action state for use of greedy cover heuristic.*
- CGControl getGreedyEquality (CbcHeuristic ∗&gen, CbcModel ∗model, bool alwaysCreate=true)

  *Obtain a greedy equality heuristic.*
- void setGreedyEqualityAction (CGControl action)

  *Set action state for use of greedy equality heuristic.*
- CGControl getRounding (CbcHeuristic ∗&gen, CbcModel ∗model, bool alwaysCreate=true)

  *Obtain a simple rounding heuristic.*
- void setRoundingAction (CGControl action)

  *Set action state for use of simple rounding heuristic.*
- CGControl getTreeLocal (CbcTreeLocal ∗&localTree, CbcModel ∗model, bool alwaysCreate=true)

  *Obtain a local search tree object.*
- void setTreeLocalAction (CGControl action)

  *Set action state for use of local tree.*

#### Status Functions

*Convenience routines for status codes.*

- void setBaBStatus (BACMajor majorStatus, BACMinor minorStatus, BACWhere where, bool haveAnswer, **OsiSolverInterface** ∗answerSolver)

  *Set the result of branch-and-cut search.*
- void setBaBStatus (const CbcModel ∗model, BACWhere where, bool haveAnswer=false, **OsiSolverInterface** ∗answerSolver=0)

  *Set the result of branch-and-cut search.*
- BACMajor translateMajor (int status)

  *Translate CbcModel major status to BACMajor.*
- BACMinor translateMinor (int status)

  *Translate CbcModel minor status to BACMinor.*
- BACMinor translateMinor (const **OsiSolverInterface** ∗osi)

  *Translate* **OsiSolverInterface** *status to BACMinor.*
- void printBaBStatus ()

  *Print the status block.*

### Public Attributes

#### Parameter parsing and input/output.

- std::string version_

  *cbc-generic version*
- std::string dfltDirectory_

  *Default directory prefix.*
- std::string lastMpsIn_

  *Last MPS input file.*
- bool allowImportErrors_

  *Allow/disallow errors when importing a model.*
- std::string lastSolnOut_

  *Last solution output file.*
- int printMode_

  *Solution printing mode.*
- std::string printMask_

  *Print mask.*
- CoinParamVec ∗ paramVec_

  *The parameter vector.*
- struct CbcGenCtlBlk::genParamsInfo_struct **genParams_**
- struct CbcGenCtlBlk::cbcParamsInfo_struct **cbcParams_**
- struct CbcGenCtlBlk::osiParamsInfo_struct **osiParams_**
- int verbose_

  *Verbosity level for help messages.*
- int paramsProcessed_

  *Number of parameters processed.*
- std::vector< bool > setByUser_

  *Record of parameters changed by user command.*
- bool defaultSettings_

  *False if the user has made nontrivial modifications to the default control settings.*
- std::string debugCreate_

  *Control debug file creation.*
- std::string debugFile_

  *Last debug input file.*

- struct CbcGenCtlBlk::debugSolInfo_struct **debugSol_**

- double totalTime_

  *Total elapsed time for this run.*

#### Models of various flavours

- CbcModel ∗ model_

  *The reference CbcModel object.*
- **OsiSolverInterface** ∗ dfltSolver_

  *The current default LP solver.*
- bool goodModel_

  *True if we have a valid model loaded, false otherwise.*
- struct CbcGenCtlBlk::babState_struct **bab_**

#### Various algorithm control variables and settings

- struct CbcGenCtlBlk::djFixCtl_struct **djFix_**
- BPControl priorityAction_

  *Control the assignment of branching priorities to integer variables.*

#### Branching Method Control

*Usage control and prototypes for branching methods.*

*Looking to the future, this covers only OsiChoose methods.*

- struct CbcGenCtlBlk::chooseStrongCtl_struct **chooseStrong_**

### Messages and statistics

- int printOpt_

  *When greater than 0, integer presolve gives more information and branch-and-cut provides statistics.*
- **CoinMessageHandler** & message (CbcGenMsgCode inID)

  *Print a message.*
- void passInMessageHandler (**CoinMessageHandler** ∗handler)

  *Supply a new message handler.*
- **CoinMessageHandler** ∗ messageHandler () const

  *Return a pointer to the message handler.*
- void setMessages (**CoinMessages::Language** lang=CoinMessages::us_en)

  *Set up messages in the specified language.*
- void setLogLevel (int lvl)

  *Set log level.*
- int logLevel () const

  *Get log level.*

### 4.35.1 Detailed Description

Definition at line 67 of file CbcGenCtlBlk.hpp.

### 4.35.2 Member Enumeration Documentation

#### 4.35.2.1 enum CbcGenCtlBlk::IPPControl

Codes to control integer preprocessing.

- IPPOff: Integer preprocessing is off.
- IPPOn: Integer preprocessing is on.
- IPPSave: IPPOn, plus preprocessed system will be saved to presolved.mps.
- IPPEqual: IPPOn, plus '<=' cliques are converted to '=' cliques.
- IPPSOS: IPPOn, plus will create SOS sets (see below).
- IPPTrySOS: IPPOn, plus will create SOS sets (see below).
- IPPEqualAll: IPPOn, plus turns all valid inequalities into equalities with integer slacks.
- IPPStrategy: look to CbcStrategy object for instructions.

IPPSOS will create SOS sets if all binary variables (except perhaps one) can be covered by SOS sets with no overlap between sets. IPPTrySOS will allow any number of binary variables to be uncovered.

Definition at line 99 of file CbcGenCtlBlk.hpp.

#### 4.35.2.2 enum CbcGenCtlBlk::CGControl

Codes to control the use of cut generators and heuristics.

- CGOff: the cut generator will not be installed
- CGOn: the cut generator will be installed; exactly how often it's activated depends on the settings at installation
- CGRoot: the cut generator will be installed with settings that restrict it to activation at the root node only.
- CGIfMove: the cut generator will be installed with settings that allow it to remain active only so long as it's generating cuts that tighten the relaxation.
- CGForceOn: the cut generator will be installed with settings that force it to be called at every node
- CGForceBut: the cut generator will be installed with settings that force it to be called at every node, but more active at root (probing only)
- CGMarker: a convenience to mark the end of the codes.

The same codes are used for heuristics.

Definition at line 129 of file CbcGenCtlBlk.hpp.

#### 4.35.2.3 enum CbcGenCtlBlk::BPControl

Codes to specify the assignment of branching priorities.

- BPOff: no priorities are passed to cbc
- BPCost: a priority vector is constructed based on objective coefficients
- BPOrder: a priority vector is constructed based on column order
- BPExt: the user has provided a priority vector

Definition at line 141 of file CbcGenCtlBlk.hpp.

#### 4.35.2.4 enum CbcGenCtlBlk::BACMajor

Major status codes for branch-and-cut.

- BACInvalid: status not yet set
- BACNotRun: branch-and-cut has not yet run for the current problem
- BACFinish: branch-and-cut has finished normally
- BACStop: branch-and-cut has stopped on a limit
- BACAbandon: branch-and-cut abandoned the problem
- BACUser: branch-and-cut stopped on user signal

Consult minorStatus_ for details.

These codes are (mostly) set to match the codes used by CbcModel. Additions to CbcModel codes should be reflected here and in translateMajor.

Definition at line 158 of file CbcGenCtlBlk.hpp.

#### 4.35.2.5 enum CbcGenCtlBlk::BACMinor

Minor status codes.

- BACmInvalid status not yet set
- BACmFinish search exhausted the tree; optimal solution found
- BACmInfeas problem is infeasible
- BACmUbnd problem is unbounded
- BACmGap stopped on integrality gap
- BACmNodeLimit stopped on node limit
- BACmTimeLimit stopped on time limit
- BACmSolnLimit stopped on number of solutions limit
- BACmUser stopped due to user event

- BACmOther nothing else is appropriate

It's not possible to make these codes agree with CbcModel. The meaning varies according to context: if the BACWhere code specifies a relaxation, then the minor status reflects the underlying OSI solver. Otherwise, it reflects the integer problem.

Definition at line 181 of file CbcGenCtlBlk.hpp.

#### 4.35.2.6 enum **CbcGenCtlBlk::BACWhere**

Codes to specify where branch-and-cut stopped.

- BACwNotStarted stopped before we ever got going
- BACwBareRoot stopped after initial solve of root relaxation
- BACwIPP stopped after integer preprocessing
- BACwIPPRelax stopped after initial solve of preprocessed problem
- BACwBAC stopped at some point in branch-and-cut

Definition at line 195 of file CbcGenCtlBlk.hpp.

### 4.35.3 Member Function Documentation

#### 4.35.3.1 int CbcGenCtlBlk::getCutDepth ( ) [inline]

Get cut depth setting.

The name is a bit of a misnomer. Essentially, this overrides the ‘every so many nodes' control with ‘execute when (depth in tree) mod (cut depth) == 0'.

Definition at line 236 of file CbcGenCtlBlk.hpp.

#### 4.35.3.2 void CbcGenCtlBlk::setCutDepth ( int *cutDepth* ) [inline]

Set cut depth setting.

See comments for getCutDepth().

Definition at line 245 of file CbcGenCtlBlk.hpp.

#### 4.35.3.3 **CGControl** CbcGenCtlBlk::getProbing ( **CglCutGenerator** ∗& *gen* )

Obtain a prototype for a probing cut generator.

#### 4.35.3.4 void CbcGenCtlBlk::setProbingAction ( **CGControl** *action* ) [inline]

Set action state for use of probing cut generator.

Definition at line 267 of file CbcGenCtlBlk.hpp.

#### 4.35.3.5 CGControl CbcGenCtlBlk::getClique ( CglCutGenerator *& gen )

Obtain a prototype for a clique cut generator.

#### 4.35.3.6 void CbcGenCtlBlk::setCliqueAction ( CGControl action ) [inline]

Set action state for use of clique cut generator.

Definition at line 277 of file CbcGenCtlBlk.hpp.

#### 4.35.3.7 CGControl CbcGenCtlBlk::getFlow ( CglCutGenerator *& gen )

Obtain a prototype for a flow cover cut generator.

#### 4.35.3.8 void CbcGenCtlBlk::setFlowAction ( CGControl action ) [inline]

Set action state for use of flow cover cut generator.

Definition at line 287 of file CbcGenCtlBlk.hpp.

#### 4.35.3.9 CGControl CbcGenCtlBlk::getGomory ( CglCutGenerator *& gen )

Obtain a prototype for a Gomory cut generator.

#### 4.35.3.10 void CbcGenCtlBlk::setGomoryAction ( CGControl action ) [inline]

Set action state for use of Gomory cut generator.

Definition at line 297 of file CbcGenCtlBlk.hpp.

#### 4.35.3.11 CGControl CbcGenCtlBlk::getKnapsack ( CglCutGenerator *& gen )

Obtain a prototype for a knapsack cover cut generator.

#### 4.35.3.12 void CbcGenCtlBlk::setKnapsackAction ( CGControl action ) [inline]

Set action state for use of knapsack cut generator.

Definition at line 307 of file CbcGenCtlBlk.hpp.

#### 4.35.3.13 void CbcGenCtlBlk::setMirAction ( CGControl action ) [inline]

Set action state for use of MIR cut generator.

Definition at line 329 of file CbcGenCtlBlk.hpp.

#### 4.35.3.14 CGControl CbcGenCtlBlk::getRedSplit ( CglCutGenerator *& gen )

Obtain a prototype for a reduce and split cut generator.

#### 4.35.3.15 void CbcGenCtlBlk::setRedSplitAction ( CGControl *action* ) [inline]

Set action state for use of reduce and split cut generator.

Definition at line 339 of file CbcGenCtlBlk.hpp.

#### 4.35.3.16 CGControl CbcGenCtlBlk::getTwomir ( CglCutGenerator *& *gen* )

Obtain a prototype for a 2-MIR cut generator.

#### 4.35.3.17 void CbcGenCtlBlk::setTwomirAction ( CGControl *action* ) [inline]

Set action state for use of 2-MIR cut generator.

Definition at line 349 of file CbcGenCtlBlk.hpp.

#### 4.35.3.18 CGControl CbcGenCtlBlk::getFPump ( CbcHeuristic *& *gen*, CbcModel * *model*, bool *alwaysCreate* = true )

Obtain a feasibility pump heuristic.

By default, any existing object is deleted and a new object is created and loaded with `model`. Set alwaysCreate = false to return an existing object if one exists.

#### 4.35.3.19 void CbcGenCtlBlk::setFPumpAction ( CGControl *action* ) [inline]

Set action state for use of feasibility pump heuristic.

Definition at line 366 of file CbcGenCtlBlk.hpp.

#### 4.35.3.20 CGControl CbcGenCtlBlk::getCombine ( CbcHeuristic *& *gen*, CbcModel * *model*, bool *alwaysCreate* = true )

Obtain a local search/combine heuristic.

By default, any existing object is deleted and a new object is created and loaded with `model`. Set alwaysCreate = false to return an existing object if one exists.

#### 4.35.3.21 void CbcGenCtlBlk::setCombineAction ( CGControl *action* ) [inline]

Set action state for use of local search/combine heuristic.

Definition at line 382 of file CbcGenCtlBlk.hpp.

#### 4.35.3.22 CGControl CbcGenCtlBlk::getGreedyCover ( CbcHeuristic *& *gen*, CbcModel * *model*, bool *alwaysCreate* = true )

Obtain a greedy cover heuristic.

By default, any existing object is deleted and a new object is created and loaded with `model`. Set alwaysCreate = false to return an existing object if one exists.

#### 4.35.3.23 void CbcGenCtlBlk::setGreedyCoverAction ( CGControl *action* ) [inline]

Set action state for use of greedy cover heuristic.

Definition at line 398 of file CbcGenCtlBlk.hpp.

#### 4.35.3.24 CGControl CbcGenCtlBlk::getGreedyEquality ( CbcHeuristic *& *gen*, CbcModel * *model*, bool *alwaysCreate* = true )

Obtain a greedy equality heuristic.

By default, any existing object is deleted and a new object is created and loaded with `model`. Set alwaysCreate = false to return an existing object if one exists.

#### 4.35.3.25 void CbcGenCtlBlk::setGreedyEqualityAction ( CGControl *action* ) [inline]

Set action state for use of greedy equality heuristic.

Definition at line 414 of file CbcGenCtlBlk.hpp.

#### 4.35.3.26 CGControl CbcGenCtlBlk::getRounding ( CbcHeuristic *& *gen*, CbcModel * *model*, bool *alwaysCreate* = true )

Obtain a simple rounding heuristic.

By default, any existing object is deleted and a new object is created and loaded with `model`. Set alwaysCreate = false to return an existing object if one exists.

#### 4.35.3.27 void CbcGenCtlBlk::setRoundingAction ( CGControl *action* ) [inline]

Set action state for use of simple rounding heuristic.

Definition at line 430 of file CbcGenCtlBlk.hpp.

#### 4.35.3.28 CGControl CbcGenCtlBlk::getTreeLocal ( CbcTreeLocal *& *localTree*, CbcModel * *model*, bool *alwaysCreate* = true )

Obtain a local search tree object.

By default, any existing object is deleted and a new object is created and loaded with `model`. Set alwaysCreate = false to return an existing object if one exists.

#### 4.35.3.29 void CbcGenCtlBlk::setTreeLocalAction ( CGControl *action* ) [inline]

Set action state for use of local tree.

Definition at line 446 of file CbcGenCtlBlk.hpp.

#### 4.35.3.30 void CbcGenCtlBlk::setBaBStatus ( const CbcModel * *model*, BACWhere *where*, bool *haveAnswer* = false, OsiSolverInterface * *answerSolver* = 0 )

Set the result of branch-and-cut search.

This version will extract the necessary information from the CbcModel object and set appropriate status based on the value passed for where.

#### 4.35.3.31 BACMajor CbcGenCtlBlk::translateMajor ( int *status* )

Translate CbcModel major status to BACMajor.

See the BACMajor enum for details.

#### 4.35.3.32 BACMinor CbcGenCtlBlk::translateMinor ( int *status* )

Translate CbcModel minor status to BACMinor.

See the BACMinor enum for details.

#### 4.35.3.33 BACMinor CbcGenCtlBlk::translateMinor ( const OsiSolverInterface * *osi* )

Translate **OsiSolverInterface** status to BACMinor.

See the BACMinor enum for details. Optimal, infeasible, and unbounded get their own codes; everything else maps to BACmOther.

#### 4.35.3.34 CoinMessageHandler& CbcGenCtlBlk::message ( CbcGenMsgCode *inID* )

Print a message.

Uses the current message handler and messages.

#### 4.35.3.35 void CbcGenCtlBlk::passInMessageHandler ( CoinMessageHandler * *handler* )

Supply a new message handler.

Replaces the current message handler. The current handler is destroyed if ourMsgHandler_ is true, and the call will set ourMsgHandler_ = true.

#### 4.35.3.36 void CbcGenCtlBlk::setMessages ( CoinMessages::Language *lang* = `CoinMessages::us_en` )

Set up messages in the specified language.

Building a set of messages in a given language implies rebuilding the whole set of messages, for reasons explained in the body of the code. Hence there's no separate setLanguage routine. Use this routine for the initial setup of messages and any subsequent change in language. Note that the constructor gives you a message handler by default, but *not* messages. You need to call setMessages explicitly.

The default value specified here for lang effectively sets the default language.

### 4.35.4 Member Data Documentation

#### 4.35.4.1 int CbcGenCtlBlk::printMode_

Solution printing mode.

Controls the amount of information printed when printing a solution. Coding is set by the keyword declarations for the printingOptions command.

Definition at line 583 of file CbcGenCtlBlk.hpp.

#### 4.35.4.2 std::string CbcGenCtlBlk::printMask_

Print mask.

Used to specify row/column names to be printed. Not implemented as of 060920.

Definition at line 590 of file CbcGenCtlBlk.hpp.

#### 4.35.4.3 int CbcGenCtlBlk::verbose_

Verbosity level for help messages.

Interpretation is bitwise:

- (0): short help
- (1): long help
- (2): unused (for compatibility with cbc; indicates AMPL)
- (3): show parameters with display = false.

Definition at line 628 of file CbcGenCtlBlk.hpp.

#### 4.35.4.4 bool CbcGenCtlBlk::defaultSettings_

False if the user has made nontrivial modifications to the default control settings.

Initially true. Specifying DJFIX, TIGHTENFACTOR, or any cut or heuristic parameter will set this to false.

Definition at line 644 of file CbcGenCtlBlk.hpp.

#### 4.35.4.5 std::string CbcGenCtlBlk::debugCreate_

Control debug file creation.

At the conclusion of branch-and-cut, dump the full solution in a binary format to debug.file in the current directory. When set to "createAfterPre", the solution is dumped before integer presolve transforms are removed. When set to "create", the solution is dumped after integer presolve transforms are backed out.

Definition at line 654 of file CbcGenCtlBlk.hpp.

#### 4.35.4.6 std::string CbcGenCtlBlk::debugFile_

Last debug input file.

The file is expected to be in a binary format understood by activateRowCutDebugger.

Definition at line 662 of file CbcGenCtlBlk.hpp.

#### 4.35.4.7 double CbcGenCtlBlk::totalTime_

Total elapsed time for this run.

Definition at line 680 of file CbcGenCtlBlk.hpp.

#### 4.35.4.8 CbcModel* CbcGenCtlBlk::model_

The reference CbcModel object.

This is the CbcModel created when cbc-generic boots up. It holds the default solver with the current constraint system. CbcCbcParam parameters are applied here, and CbcOsiParam parameters are applied to the solver. Major modifications for branch-and-cut (integer preprocessing, installation of heuristics and cut generators) are performed on a clone. The solution is transferred back into this object.

Definition at line 697 of file CbcGenCtlBlk.hpp.

#### 4.35.4.9 OsiSolverInterface* CbcGenCtlBlk::dfltSolver_

The current default LP solver.

This is a pointer to a reference copy. If you want the solver associated with model_, ask for it directly.

Definition at line 705 of file CbcGenCtlBlk.hpp.

#### 4.35.4.10 bool CbcGenCtlBlk::goodModel_

True if we have a valid model loaded, false otherwise.

Definition at line 709 of file CbcGenCtlBlk.hpp.

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

- CbcGenCtlBlk.hpp

## 4.36 CbcGeneral Class Reference

Define a catch all class.

`#include <CbcGeneral.hpp>`

Inheritance diagram for CbcGeneral:

Collaboration diagram for CbcGeneral:

### Public Member Functions

- CbcGeneral (CbcModel *model)

  *Useful constructor Just needs to point to model.*
- virtual CbcObject * clone () const =0

  *Clone.*
- virtual double infeasibility (const **OsiBranchingInformation** *info, int &preferredWay) const

  *Infeasibility - large is 0.5.*

- virtual void feasibleRegion ()=0

  *This looks at solution and sets bounds to contain solution.*
- virtual CbcBranchingObject * createCbcBranch (**OsiSolverInterface** *solver, const **OsiBranchingInformation** *info, int way)

  *Creates a branching object.*
- virtual void redoSequenceEtc (CbcModel *model, int numberColumns, const int *originalColumns)=0

  *Redoes data when sequence numbers change.*

### Additional Inherited Members

### 4.36.1 Detailed Description

Define a catch all class.

This will create a list of subproblems

Definition at line 17 of file CbcGeneral.hpp.

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

- CbcGeneral.hpp

## 4.37 CbcGenParam Class Reference

Class for cbc-generic control parameters.

`#include <CbcGenParam.hpp>`

Inheritance diagram for CbcGenParam:

Collaboration diagram for CbcGenParam:

### Public Types

**Subtypes**

- enum CbcGenParamCode

  *Enumeration for cbc-generic parameters.*

### Public Member Functions

**Constructors and Destructors**

*Be careful how you specify parameters for the constructors! There's great potential for confusion.*

- CbcGenParam ()

  *Default constructor.*
- CbcGenParam (CbcGenParamCode code, std::string **name**, std::string help, double lower, double upper, double dflt=0.0, bool **display**=true)

  *Constructor for a parameter with a double value.*
- CbcGenParam (CbcGenParamCode code, std::string **name**, std::string help, int lower, int upper, int dflt=0, bool **display**=true)

  *Constructor for a parameter with an integer value.*

- CbcGenParam (CbcGenParamCode code, std::string **name**, std::string help, std::string firstValue, int dflt, bool **display**=true)
  *Constructor for a parameter with keyword values.*
- CbcGenParam (CbcGenParamCode code, std::string **name**, std::string help, std::string dflt, bool **display**=true)
  *Constructor for a string parameter.*
- CbcGenParam (CbcGenParamCode code, std::string **name**, std::string help, bool **display**=true)
  *Constructor for an action parameter.*
- CbcGenParam (const CbcGenParam &orig)
  *Copy constructor.*
- CbcGenParam ∗ clone ()
  *Clone.*
- CbcGenParam & operator= (const CbcGenParam &rhs)
  *Assignment.*
- ∼CbcGenParam ()
  *Destructor.*

**Methods to query and manipulate a parameter object**

- CbcGenParamCode paramCode () const
  *Get the parameter code.*
- void setParamCode (CbcGenParamCode code)
  *Set the parameter code.*
- CbcGenCtlBlk ∗ obj () const
  *Get the underlying cbc-generic control object.*
- void setObj (CbcGenCtlBlk ∗obj)
  *Set the underlying cbc-generic control object.*

## 4.37.1 Detailed Description

Class for cbc-generic control parameters.

Adds parameter type codes and push/pull functions to the generic parameter object.

Definition at line 34 of file CbcGenParam.hpp.

## 4.37.2 Member Enumeration Documentation

### 4.37.2.1 enum CbcGenParam::CbcGenParamCode

Enumeration for cbc-generic parameters.

These are parameters that control the operation of the cbc-generic main program by operating on a CbcGenCtlBlk object. CBCGEN_FIRSTPARAM and CBCGEN_LASTPARM are markers to allow convenient separation of parameter groups.

Definition at line 49 of file CbcGenParam.hpp.

## 4.37.3 Constructor & Destructor Documentation

### 4.37.3.1 CbcGenParam::CbcGenParam ( CbcGenParamCode *code,* std::string *name,* std::string *help,* double *lower,* double *upper,* double *dflt* = `0.0`, bool *display* = `true` )

Constructor for a parameter with a double value.

The default value is 0.0. Be careful to clearly indicate that `lower` and `upper` are real (double) values to distinguish this constructor from the constructor for an integer parameter.

#### 4.37.3.2 CbcGenParam::CbcGenParam ( CbcGenParamCode *code*, std::string *name*, std::string *help*, int *lower*, int *upper*, int *dflt* = 0, bool *display* = true )

Constructor for a parameter with an integer value.

The default value is 0.

#### 4.37.3.3 CbcGenParam::CbcGenParam ( CbcGenParamCode *code*, std::string *name*, std::string *help*, std::string *firstValue*, int *dflt*, bool *display* = true )

Constructor for a parameter with keyword values.

The string supplied as `firstValue` becomes the first keyword. Additional keywords can be added using **append←Kwd()**. Keywords are numbered from zero. It's necessary to specify both the first keyword (`firstValue`) and the default keyword index (`dflt`) in order to distinguish this constructor from the string and action parameter constructors.

#### 4.37.3.4 CbcGenParam::CbcGenParam ( CbcGenParamCode *code*, std::string *name*, std::string *help*, std::string *dflt*, bool *display* = true )

Constructor for a string parameter.

The default string value must be specified explicitly to distinguish a string constructor from an action parameter constructor.

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

- CbcGenParam.hpp

## 4.38 CbcHeuristic Class Reference

Heuristic base class.

`#include <CbcHeuristic.hpp>`

Inheritance diagram for CbcHeuristic:

Collaboration diagram for CbcHeuristic:

### Public Member Functions

- virtual CbcHeuristic ∗ clone () const =0

    *Clone.*
- CbcHeuristic & operator= (const CbcHeuristic &rhs)

    *Assignment operator.*
- virtual void setModel (CbcModel ∗model)

    *update model (This is needed if cliques update matrix etc)*
- virtual void resetModel (CbcModel ∗model)=0

    *Resets stuff if model changes.*
- virtual int solution (double &objectiveValue, double ∗newSolution)=0

    *returns 0 if no solution, 1 if valid solution with better objective value than one passed in Sets solution values if good, sets objective value This is called after cuts have been added - so can not add cuts*
- virtual int solution2 (double &, double ∗, **OsiCuts** &)

*returns 0 if no solution, 1 if valid solution, -1 if just returning an estimate of best possible solution with better objective value than one passed in Sets solution values if good, sets objective value (only if nonzero code) This is called at same time as cut generators - so can add cuts Default is do nothing*

- virtual void validate ()

  *Validate model i.e. sets when_ to 0 if necessary (may be NULL)*
- void setWhen (int value)

  *Sets "when" flag - 0 off, 1 at root, 2 other than root, 3 always.*
- int when () const

  *Gets "when" flag - 0 off, 1 at root, 2 other than root, 3 always.*
- void setNumberNodes (int value)

  *Sets number of nodes in subtree (default 200)*
- int numberNodes () const

  *Gets number of nodes in a subtree (default 200)*
- void setSwitches (int value)

  *Switches (does not apply equally to all heuristics) 1 bit - stop once allowable gap on objective reached 2 bit - always do given number of passes 4 bit - weaken cutoff by 5% every 50 passes? 8 bit - if has cutoff and suminf bobbling for 20 passes then first try halving distance to best possible then try keep halving distance to known cutoff 16 bit - needs new solution to run 1024 bit - stop all heuristics on max time.*
- int switches () const

  *Switches (does not apply equally to all heuristics) 1 bit - stop once allowable gap on objective reached 2 bit - always do given number of passes 4 bit - weaken cutoff by 5% every 50 passes? 8 bit - if has cutoff and suminf bobbling for 20 passes then first try halving distance to best possible then try keep halving distance to known cutoff 16 bit - needs new solution to run 1024 bit - stop all heuristics on max time 65536 bit and above used for temporary communication.*
- bool exitNow (double bestObjective) const

  *Whether to exit at once on gap.*
- void setFeasibilityPumpOptions (int value)

  *Sets feasibility pump options (-1 is off)*
- int feasibilityPumpOptions () const

  *Gets feasibility pump options (-1 is off)*
- void setModelOnly (CbcModel ∗model)

  *Just set model - do not do anything else.*
- void setFractionSmall (double value)

  *Sets fraction of new(rows+columns)/old(rows+columns) before doing small branch and bound (default 1.0)*
- double fractionSmall () const

  *Gets fraction of new(rows+columns)/old(rows+columns) before doing small branch and bound (default 1.0)*
- int numberSolutionsFound () const

  *Get how many solutions the heuristic thought it got.*
- void incrementNumberSolutionsFound ()

  *Increment how many solutions the heuristic thought it got.*
- int smallBranchAndBound (**OsiSolverInterface** ∗solver, int numberNodes, double ∗newSolution, double &new←SolutionValue, double cutoff, std::string name) const

  *Do mini branch and bound - return 0 not finished - no solution 1 not finished - solution 2 finished - no solution 3 finished - solution (could add global cut if finished) -1 returned on size -2 time or user event.*
- virtual void generateCpp (FILE ∗)

  *Create C++ lines to get to current state.*
- void generateCpp (FILE ∗fp, const char ∗heuristic)

  *Create C++ lines to get to current state - does work for base class.*
- virtual bool canDealWithOdd () const

  *Returns true if can deal with "odd" problems e.g. sos type 2.*

- const char ∗ heuristicName () const

  *return name of heuristic*
- void setHeuristicName (const char ∗name)

  *set name of heuristic*
- void setSeed (int value)

  *Set random number generator seed.*
- int getSeed () const

  *Get random number generator seed.*
- void setDecayFactor (double value)

  *Sets decay factor (for howOften) on failure.*
- void setInputSolution (const double ∗solution, double objValue)

  *Set input solution.*
- void setShallowDepth (int value)

  *Upto this depth we call the tree shallow and the heuristic can be called multiple times.*
- void setHowOftenShallow (int value)

  *How often to invoke the heuristics in the shallow part of the tree.*
- void setMinDistanceToRun (int value)

  *How "far" should this node be from every other where the heuristic was run in order to allow the heuristic to run in this node, too.*
- virtual bool shouldHeurRun (int whereFrom)

  *Check whether the heuristic should run at all 0 - before cuts at root node (or from doHeuristics) 1 - during cuts at root 2 - after root node cuts 3 - after cuts at other nodes 4 - during cuts at other nodes 8 added if previous heuristic in loop found solution.*
- bool shouldHeurRun_randomChoice ()

  *Check whether the heuristic should run this time.*
- int numRuns () const

  *how many times the heuristic has actually run*
- int numCouldRun () const

  *How many times the heuristic could run.*
- **OsiSolverInterface** ∗ cloneBut (int type)

  *Clone, but ...*

## Protected Attributes

- CbcModel ∗ model_

  *Model.*
- int when_

  *When flag - 0 off, 1 at root, 2 other than root, 3 always.*
- int numberNodes_

  *Number of nodes in any sub tree.*
- int feasibilityPumpOptions_

  *Feasibility pump options , -1 is off >=0 for feasibility pump itself -2 quick proximity search -3 longer proximity search.*
- double fractionSmall_

  *Fraction of new(rows+columns)/old(rows+columns) before doing small branch and bound.*
- **CoinThreadRandom** randomNumberGenerator_

  *Thread specific random number generator.*
- std::string heuristicName_

*Name for printing.*

- int howOften_

  *How often to do (code can change)*

- double decayFactor_

  *How much to increase how often.*

- int switches_

  *Switches (does not apply equally to all heuristics) 1 bit - stop once allowable gap on objective reached 2 bit - always do given number of passes 4 bit - weaken cutoff by 5% every 50 passes? 8 bit - if has cutoff and suminf bobbling for 20 passes then first try halving distance to best possible then try keep halving distance to known cutoff 16 bit - needs new solution to run 1024 bit - stop all heuristics on max time.*

- int shallowDepth_

  *Upto this depth we call the tree shallow and the heuristic can be called multiple times.*

- int howOftenShallow_

  *How often to invoke the heuristics in the shallow part of the tree.*

- int numInvocationsInShallow_

  *How many invocations happened within the same node when in a shallow part of the tree.*

- int numInvocationsInDeep_

  *How many invocations happened when in the deep part of the tree.*

- int lastRunDeep_

  *After how many deep invocations was the heuristic run last time.*

- int numRuns_

  *how many times the heuristic has actually run*

- int minDistanceToRun_

  *How "far" should this node be from every other where the heuristic was run in order to allow the heuristic to run in this node, too.*

- CbcHeuristicNodeList runNodes_

  *The description of the nodes where this heuristic has been applied.*

- int numCouldRun_

  *How many times the heuristic could run.*

- int numberSolutionsFound_

  *How many solutions the heuristic thought it got.*

- int numberNodesDone_

  *How many nodes the heuristic did this go.*

### 4.38.1 Detailed Description

Heuristic base class.

Definition at line 77 of file CbcHeuristic.hpp.

### 4.38.2 Member Function Documentation

#### 4.38.2.1 void CbcHeuristic::setWhen ( int *value* ) [inline]

Sets "when" flag - 0 off, 1 at root, 2 other than root, 3 always.

If 10 added then don't worry if validate says there are funny objects as user knows it will be fine

Definition at line 134 of file CbcHeuristic.hpp.

#### 4.38.2.2 void CbcHeuristic::setShallowDepth ( int *value* ) [inline]

Upto this depth we call the tree shallow and the heuristic can be called multiple times.

That is, the test whether the current node is far from the others where the jeuristic was invoked will not be done, only the frequency will be tested. After that depth the heuristic will can be invoked only once per node, right before branching. That's when it'll be tested whether the heur should run at all.

Definition at line 268 of file CbcHeuristic.hpp.

#### 4.38.2.3 void CbcHeuristic::setMinDistanceToRun ( int *value* ) [inline]

How "far" should this node be from every other where the heuristic was run in order to allow the heuristic to run in this node, too.

Currently this is tested, but we may switch to avgDistanceToRun_ in the future.

Definition at line 278 of file CbcHeuristic.hpp.

#### 4.38.2.4 OsiSolverInterface* CbcHeuristic::cloneBut ( int *type* )

Clone, but ...

If type is

- 0 clone the solver for the model,
- 1 clone the continuous solver for the model
- Add 2 to say without integer variables which are at low priority
- Add 4 to say quite likely infeasible so give up easily (clp only).

### 4.38.3 Member Data Documentation

#### 4.38.3.1 int CbcHeuristic::shallowDepth_ [protected]

Upto this depth we call the tree shallow and the heuristic can be called multiple times.

That is, the test whether the current node is far from the others where the jeuristic was invoked will not be done, only the frequency will be tested. After that depth the heuristic will can be invoked only once per node, right before branching. That's when it'll be tested whether the heur should run at all.

Definition at line 364 of file CbcHeuristic.hpp.

#### 4.38.3.2 int CbcHeuristic::numInvocationsInShallow_ [protected]

How many invocations happened within the same node when in a shallow part of the tree.

Definition at line 369 of file CbcHeuristic.hpp.

#### 4.38.3.3 int CbcHeuristic::numInvocationsInDeep_ [protected]

How many invocations happened when in the deep part of the tree.

For every node we count only one invocation.

Definition at line 372 of file CbcHeuristic.hpp.

#### 4.38.3.4 int CbcHeuristic::minDistanceToRun_ [protected]

How "far" should this node be from every other where the heuristic was run in order to allow the heuristic to run in this node, too.

Currently this is tested, but we may switch to avgDistanceToRun_ in the future.

Definition at line 380 of file CbcHeuristic.hpp.

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

- CbcHeuristic.hpp

## 4.39 CbcHeuristicCrossover Class Reference

Crossover Search class.

`#include <CbcHeuristicLocal.hpp>`

Inheritance diagram for CbcHeuristicCrossover:

Collaboration diagram for CbcHeuristicCrossover:

### Public Member Functions

- virtual CbcHeuristic ∗ clone () const

  *Clone.*
- CbcHeuristicCrossover & operator= (const CbcHeuristicCrossover &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel ∗model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel ∗model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double ∗newSolution)

  *returns 0 if no solution, 1 if valid solution.*
- void setNumberSolutions (int value)

  *Sets number of solutions to use.*

### Protected Attributes

- std::vector< double > attempts_

  *Attempts.*
- double random_ [10]

  *Random numbers to stop same search happening.*
- int numberSolutions_

  *Number of solutions so we only do after new solution.*
- int useNumber_

  *Number of solutions to use.*

### 4.39.1 Detailed Description

Crossover Search class.

Definition at line 211 of file CbcHeuristicLocal.hpp.

### 4.39.2 Member Function Documentation

#### 4.39.2.1 virtual int CbcHeuristicCrossover::solution ( double & *objectiveValue,* double ∗ *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Fix variables if agree in useNumber_ solutions when_ 0 off, 1 only at new solutions, 2 also every now and then add 10 to make only if agree at lower bound

Implements CbcHeuristic.

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

- CbcHeuristicLocal.hpp

## 4.40 CbcHeuristicDINS Class Reference

Inheritance diagram for CbcHeuristicDINS:

Collaboration diagram for CbcHeuristicDINS:

### Public Member Functions

- virtual CbcHeuristic ∗ clone () const

  *Clone.*
- CbcHeuristicDINS & operator= (const CbcHeuristicDINS &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel ∗model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel ∗model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double ∗newSolution)

  *returns 0 if no solution, 1 if valid solution.*
- int solutionFix (double &objectiveValue, double ∗newSolution, const int ∗keep)

  *This version fixes stuff and does IP.*
- void setHowOften (int value)

  *Sets how often to do it.*
- void setMaximumKeep (int value)

  *Sets maximum number of solutions kept.*
- void setConstraint (int value)

  *Sets tightness of extra constraint.*

### Protected Attributes

- int numberSolutions_

    *Number of solutions so we can do something at solution.*
- int howOften_

    *How often to do (code can change)*
- int numberSuccesses_

    *Number of successes.*
- int numberTries_

    *Number of tries.*
- int maximumKeepSolutions_

    *Maximum number of solutions to keep.*
- int numberKeptSolutions_

    *Number of solutions kept.*
- int numberIntegers_

    *Number of integer variables.*
- int localSpace_

    *Local parameter.*
- int ** values_

    *Values of integer variables.*

### 4.40.1 Detailed Description

Definition at line 14 of file CbcHeuristicDINS.hpp.

### 4.40.2 Member Function Documentation

#### 4.40.2.1 virtual int CbcHeuristicDINS::solution ( double & *objectiveValue*, double * *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) This does Relaxation Induced Neighborhood Search

Implements CbcHeuristic.

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

- CbcHeuristicDINS.hpp

## 4.41 CbcHeuristicDive Class Reference

Dive class.

`#include <CbcHeuristicDive.hpp>`

Inheritance diagram for CbcHeuristicDive:

Collaboration diagram for CbcHeuristicDive:

## Classes

- struct PriorityType

  *Array of priorities.*

## Public Member Functions

- virtual CbcHeuristicDive ∗ clone () const =0

  *Clone.*
- CbcHeuristicDive & operator= (const CbcHeuristicDive &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE ∗)

  *Create C++ lines to get to current state.*
- void generateCpp (FILE ∗fp, const char ∗heuristic)

  *Create C++ lines to get to current state - does work for base class.*
- virtual void resetModel (CbcModel ∗model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel ∗model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double ∗newSolution)

  *returns 0 if no solution, 1 if valid solution with better objective value than one passed in Sets solution values if good, sets objective value (only if good) This is called after cuts have been added - so can not add cuts This does Fractional Diving*
- int solution (double &objectiveValue, int &numberNodes, int &numberCuts, **OsiRowCut** ∗∗cuts, CbcSubProblem ∗∗&nodes, double ∗newSolution)

  *inner part of dive*
- int fathom (CbcModel ∗model, int &numberNodes, CbcSubProblem ∗∗&nodes)

  *returns 0 if no solution, 1 if valid solution with better objective value than one passed in also returns list of nodes This does Fractional Diving*
- virtual void validate ()

  *Validate model i.e. sets when_ to 0 if necessary (may be NULL)*
- void setPriorities ()

  *Sets priorities if any.*
- void selectBinaryVariables ()

  *Select candidate binary variables for fixing.*
- void setPercentageToFix (double value)

  *Set percentage of integer variables to fix at bounds.*
- void setMaxIterations (int value)

  *Set maximum number of iterations.*
- void setMaxSimplexIterations (int value)

  *Set maximum number of simplex iterations.*
- int maxSimplexIterations () const

  *Get maximum number of simplex iterations.*
- void setMaxSimplexIterationsAtRoot (int value)

  *Set maximum number of simplex iterations at root node.*
- void setMaxTime (double value)

  *Set maximum time allowed.*
- virtual bool canHeuristicRun ()

  *Tests if the heuristic can run.*

- virtual bool selectVariableToBranch (**OsiSolverInterface** ∗solver, const double ∗newSolution, int &bestColumn, int &bestRound)=0

  *Selects the next variable to branch on Returns true if all the fractional variables can be trivially rounded.*
- virtual void initializeData ()

  *Initializes any data which is going to be used repeatedly in selectVariableToBranch.*
- int reducedCostFix (**OsiSolverInterface** ∗solver)

  *Perform reduced cost fixing on integer variables.*
- virtual int fixOtherVariables (**OsiSolverInterface** ∗solver, const double ∗solution, PseudoReducedCost ∗candidate, const double ∗random)

  *Fix other variables at bounds.*

### Protected Attributes

- double ∗ downArray_

  *Extra down array (number Integers long)*
- double ∗ upArray_

  *Extra up array (number Integers long)*

### 4.41.1 Detailed Description

Dive class.

Definition at line 21 of file CbcHeuristicDive.hpp.

### 4.41.2 Member Function Documentation

#### 4.41.2.1 virtual bool CbcHeuristicDive::selectVariableToBranch ( OsiSolverInterface ∗ *solver*, const double ∗ *newSolution*, int & *bestColumn*, int & *bestRound* ) [pure virtual]

Selects the next variable to branch on Returns true if all the fractional variables can be trivially rounded.

Returns false, if there is at least one fractional variable that is not trivially roundable. In this case, the bestColumn returned will not be trivially roundable.

Implemented in CbcHeuristicDiveGuided, CbcHeuristicDiveCoefficient, CbcHeuristicDiveFractional, CbcHeuristicDiveLineSearch, CbcHeuristicDivePseudoCost, and CbcHeuristicDiveVectorLength.

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

- CbcHeuristicDive.hpp

## 4.42 CbcHeuristicDiveCoefficient Class Reference

DiveCoefficient class.

`#include <CbcHeuristicDiveCoefficient.hpp>`

Inheritance diagram for CbcHeuristicDiveCoefficient:

Collaboration diagram for CbcHeuristicDiveCoefficient:

### Public Member Functions

- virtual CbcHeuristicDiveCoefficient ∗ clone () const

  *Clone.*
- CbcHeuristicDiveCoefficient & operator= (const CbcHeuristicDiveCoefficient &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*
- virtual bool selectVariableToBranch (**OsiSolverInterface** ∗solver, const double ∗newSolution, int &bestColumn, int &bestRound)

  *Selects the next variable to branch on.*

### Additional Inherited Members

### 4.42.1 Detailed Description

DiveCoefficient class.

Definition at line 14 of file CbcHeuristicDiveCoefficient.hpp.

### 4.42.2 Member Function Documentation

#### 4.42.2.1 virtual bool CbcHeuristicDiveCoefficient::selectVariableToBranch ( OsiSolverInterface ∗ *solver,* const double ∗ *newSolution,* int & *bestColumn,* int & *bestRound* ) [virtual]

Selects the next variable to branch on.

Returns true if all the fractional variables can be trivially rounded. Returns false, if there is at least one fractional variable that is not trivially roundable. In this case, the bestColumn returned will not be trivially roundable.

Implements CbcHeuristicDive.

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

- CbcHeuristicDiveCoefficient.hpp

## 4.43 CbcHeuristicDiveFractional Class Reference

DiveFractional class.

`#include <CbcHeuristicDiveFractional.hpp>`

Inheritance diagram for CbcHeuristicDiveFractional:

Collaboration diagram for CbcHeuristicDiveFractional:

### Public Member Functions

- virtual CbcHeuristicDiveFractional ∗ clone () const

  *Clone.*
- CbcHeuristicDiveFractional & operator= (const CbcHeuristicDiveFractional &rhs)

*Assignment operator.*

- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*

- virtual bool selectVariableToBranch (**OsiSolverInterface** ∗solver, const double ∗newSolution, int &bestColumn, int &bestRound)

  *Selects the next variable to branch on.*

### Additional Inherited Members

### 4.43.1 Detailed Description

DiveFractional class.

Definition at line 14 of file CbcHeuristicDiveFractional.hpp.

### 4.43.2 Member Function Documentation

#### 4.43.2.1 virtual bool CbcHeuristicDiveFractional::selectVariableToBranch ( OsiSolverInterface ∗ *solver,* const double ∗ *newSolution,* int & *bestColumn,* int & *bestRound* ) [virtual]

Selects the next variable to branch on.

Returns true if all the fractional variables can be trivially rounded. Returns false, if there is at least one fractional variable that is not trivially roundable. In this case, the bestColumn returned will not be trivially roundable.

Implements CbcHeuristicDive.

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

- CbcHeuristicDiveFractional.hpp

## 4.44 CbcHeuristicDiveGuided Class Reference

DiveGuided class.

`#include <CbcHeuristicDiveGuided.hpp>`

Inheritance diagram for CbcHeuristicDiveGuided:

Collaboration diagram for CbcHeuristicDiveGuided:

### Public Member Functions

- virtual CbcHeuristicDiveGuided ∗ clone () const

  *Clone.*

- CbcHeuristicDiveGuided & operator= (const CbcHeuristicDiveGuided &rhs)

  *Assignment operator.*

- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*

- virtual bool canHeuristicRun ()

  *Tests if the heuristic can run.*

- virtual bool selectVariableToBranch (**OsiSolverInterface** ∗solver, const double ∗newSolution, int &bestColumn, int &bestRound)

    *Selects the next variable to branch on.*

### Additional Inherited Members

### 4.44.1 Detailed Description

DiveGuided class.

Definition at line 14 of file CbcHeuristicDiveGuided.hpp.

### 4.44.2 Member Function Documentation

#### 4.44.2.1 virtual bool CbcHeuristicDiveGuided::selectVariableToBranch ( OsiSolverInterface ∗ *solver,* const double ∗ *newSolution,* int & *bestColumn,* int & *bestRound* ) [virtual]

Selects the next variable to branch on.

Returns true if all the fractional variables can be trivially rounded. Returns false, if there is at least one fractional variable that is not trivially roundable. In this case, the bestColumn returned will not be trivially roundable.

Implements CbcHeuristicDive.

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

- CbcHeuristicDiveGuided.hpp

## 4.45 CbcHeuristicDiveLineSearch Class Reference

DiveLineSearch class.

`#include <CbcHeuristicDiveLineSearch.hpp>`

Inheritance diagram for CbcHeuristicDiveLineSearch:

Collaboration diagram for CbcHeuristicDiveLineSearch:

### Public Member Functions

- virtual CbcHeuristicDiveLineSearch ∗ clone () const

    *Clone.*

- CbcHeuristicDiveLineSearch & operator= (const CbcHeuristicDiveLineSearch &rhs)

    *Assignment operator.*

- virtual void generateCpp (FILE ∗fp)

    *Create C++ lines to get to current state.*

- virtual bool selectVariableToBranch (**OsiSolverInterface** ∗solver, const double ∗newSolution, int &bestColumn, int &bestRound)

    *Selects the next variable to branch on.*

### Additional Inherited Members

### 4.45.1 Detailed Description

DiveLineSearch class.

Definition at line 14 of file CbcHeuristicDiveLineSearch.hpp.

### 4.45.2 Member Function Documentation

#### 4.45.2.1 virtual bool CbcHeuristicDiveLineSearch::selectVariableToBranch ( **OsiSolverInterface** ∗ *solver,* const double ∗ *newSolution,* int & *bestColumn,* int & *bestRound* ) [virtual]

Selects the next variable to branch on.

Returns true if all the fractional variables can be trivially rounded. Returns false, if there is at least one fractional variable that is not trivially roundable. In this case, the bestColumn returned will not be trivially roundable.

Implements CbcHeuristicDive.

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

- CbcHeuristicDiveLineSearch.hpp

## 4.46 CbcHeuristicDivePseudoCost Class Reference

DivePseudoCost class.

`#include <CbcHeuristicDivePseudoCost.hpp>`

Inheritance diagram for CbcHeuristicDivePseudoCost:

Collaboration diagram for CbcHeuristicDivePseudoCost:

### Public Member Functions

- virtual CbcHeuristicDivePseudoCost ∗ clone () const

  *Clone.*
- CbcHeuristicDivePseudoCost & operator= (const CbcHeuristicDivePseudoCost &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*
- virtual bool selectVariableToBranch (**OsiSolverInterface** ∗solver, const double ∗newSolution, int &bestColumn, int &bestRound)

  *Selects the next variable to branch on.*
- virtual void initializeData ()

  *Initializes any data which is going to be used repeatedly in selectVariableToBranch.*
- virtual int fixOtherVariables (**OsiSolverInterface** ∗solver, const double ∗solution, PseudoReducedCost ∗candidate, const double ∗random)

  *Fix other variables at bounds.*

### Additional Inherited Members

### 4.46.1 Detailed Description

DivePseudoCost class.

Definition at line 14 of file CbcHeuristicDivePseudoCost.hpp.

### 4.46.2 Member Function Documentation

#### 4.46.2.1 virtual bool CbcHeuristicDivePseudoCost::selectVariableToBranch ( **OsiSolverInterface** ∗ *solver,* const double ∗ *newSolution,* int & *bestColumn,* int & *bestRound* ) [virtual]

Selects the next variable to branch on.

Returns true if all the fractional variables can be trivially rounded. Returns false, if there is at least one fractional variable that is not trivially roundable. In this case, the bestColumn returned will not be trivially roundable.

Implements CbcHeuristicDive.

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

- CbcHeuristicDivePseudoCost.hpp

## 4.47 CbcHeuristicDiveVectorLength Class Reference

DiveVectorLength class.

`#include <CbcHeuristicDiveVectorLength.hpp>`

Inheritance diagram for CbcHeuristicDiveVectorLength:

Collaboration diagram for CbcHeuristicDiveVectorLength:

### Public Member Functions

- virtual CbcHeuristicDiveVectorLength ∗ clone () const
  *Clone.*
- CbcHeuristicDiveVectorLength & operator= (const CbcHeuristicDiveVectorLength &rhs)
  *Assignment operator.*
- virtual void generateCpp (FILE ∗fp)
  *Create C++ lines to get to current state.*
- virtual bool selectVariableToBranch (**OsiSolverInterface** ∗solver, const double ∗newSolution, int &bestColumn, int &bestRound)
  *Selects the next variable to branch on.*

### Additional Inherited Members

### 4.47.1 Detailed Description

DiveVectorLength class.

Definition at line 14 of file CbcHeuristicDiveVectorLength.hpp.

### 4.47.2 Member Function Documentation

#### 4.47.2.1 virtual bool CbcHeuristicDiveVectorLength::selectVariableToBranch ( OsiSolverInterface * *solver*, const double * *newSolution*, int & *bestColumn*, int & *bestRound* ) [virtual]

Selects the next variable to branch on.

Returns true if all the fractional variables can be trivially rounded. Returns false, if there is at least one fractional variable that is not trivially roundable. In this case, the bestColumn returned will not be trivially roundable.

Implements CbcHeuristicDive.

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

- CbcHeuristicDiveVectorLength.hpp

## 4.48 CbcHeuristicDW Class Reference

This is unlike the other heuristics in that it is very very compute intensive.

`#include <CbcHeuristicDW.hpp>`

Inheritance diagram for CbcHeuristicDW:

Collaboration diagram for CbcHeuristicDW:

### Public Member Functions

- virtual CbcHeuristic * clone () const

  *Clone.*
- CbcHeuristicDW & operator= (const CbcHeuristicDW &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE *fp)

  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel *model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel *model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double *newSolution)

  *returns 0 if no solution, 1 if valid solution.*
- int numberBlocks () const

  *Return number of blocks <=0 - no usable structure.*
- void passInSolution (const double *solution)

  *Pass in a solution.*
- void passInContinuousSolution (const double *solution)

  *Pass in continuous solution.*
- void setProposalActions (int fullDWEverySoOften)

  *DW Proposal actions fullDWEverySoOften - 0 - off k - every k times solution gets better.*
- double objectiveValueWhen (int whichDW) const

  *Objective value when whichDw created.*
- int numberColumnsDW (int whichDW) const

*Number of columns in DW.*

- **OsiSolverInterface** ∗ solver () const

    *Solver.*

- **OsiSolverInterface** ∗ DWModel (int whichDW) const

    *DW model (user must delete)*

- double bestObjective () const

    *Best objective value.*

- const double ∗ bestSolution () const

    *Best solution found so far.*

- const double ∗ continuousSolution () const

    *Continuous solution.*

- const double ∗ fixedDj () const

    *Reduced costs of fixed solution.*

- const double ∗ objectiveDW () const

    *Objective at which DW updated.*

- int numberDWTimes () const

    *Number of times we have added to DW model.*

- const int ∗ numberColumnsDW () const

    *Number of columns in DW.*

- void setNumberPasses (int value)

    *Set number of passes.*

- void setNumberBadPasses (int value)

    *Set number of passes without better solution.*

- void setNumberNeeded (int value)

    *Set number free integers needed (Base value)*

- int getNumberNeeded () const

    *Get number free integers needed (Base value)*

- void setCurrentNumberNeeded (int value)

    *Set number free integers needed (Current value)*

- int getCurrentNumberNeeded () const

    *Get number free integers needed (Current value)*

- void setNumberNodes (int value)

    *Set number nodes (could be done in callback) (Base value)*

- int getNumberNodes () const

    *Get number nodes (could be done in callback) (Base value)*

- void setCurrentNumberNodes (int value)

    *Set number nodes (could be done in callback) (Current value)*

- int getCurrentNumberNodes () const

    *Get number nodes (could be done in callback) (Current value)*

- void setTargetObjective (double value)

    *Set target objective.*

- void setHowOften (int value)

    *Sets how often to do it.*

- const int ∗ whichRowBlock () const

    *Block for every row.*

- const int ∗ whichColumnBlock () const

    *Block for every column.*

- double ∗ initialLower () const

  *Initial Lower bounds.*
- double ∗ initialUpper () const

  *Initial Upper bounds.*
- int ∗ intArrays () const

  *Local integer arrays (each numberBlocks_ long)*
- double ∗ doubleArrays () const

  *Local double arrays (each numberBlocks_ long)*
- int phase () const

  *Phase of solution.*
- int pass () const

  *Pass number.*
- const int ∗ columnsInBlock () const

  *Which columns are in block.*
- const int ∗ startColumnBlock () const

  *Starts for columnsInBlock.*
- const int ∗ intsInBlock () const

  *Number of integer variables in each block.*
- double objectiveValue (const double ∗solution)

  *Objective value (could also check validity)*

## Protected Attributes

- double targetObjective_

  *Target objective.*
- double bestObjective_

  *Best objective value.*
- double lastObjective_

  *Objective value last time.*
- heuristicCallBack functionPointer_

  *Call back whereFrom - 0 - after blocks found but before data setup 1 - after blocks sorted but before used 2 - just before normal branch and bound 3 - after DW has been updated 4 - if better solution found 5 - every time a block might be used next few for adjustment of nNeeded etc 6 - complete search done - no solution 7 - stopped on nodes - no improvement 8 - improving (same as 4 but after nNeeded changed Pointers to local data given by following pointers.*
- int ∗ intArray_

  *Local integer arrays (each numberBlocks_ long)*
- double ∗ doubleArray_

  *Local double arrays (each numberBlocks_ long)*
- **OsiSolverInterface** ∗ solver_

  *Base solver.*
- **OsiSolverInterface** ∗ dwSolver_

  *DW solver.*
- double ∗ bestSolution_

  *Best solution found so far.*
- double ∗ continuousSolution_

  *Continuous solution.*
- double ∗ fixedDj_

*Reduced costs of fixed solution.*

- double ∗ saveLower_

  *Original lower bounds.*
- double ∗ saveUpper_

  *Original Upper bounds.*
- double ∗ random_

  *random numbers for master rows*
- double ∗ weights_

  *Weights for each proposal.*
- double ∗ objectiveDW_

  *Objective at which DW updated.*
- int ∗ numberColumnsDW_

  *Number of columns in each DW.*
- int ∗ whichRowBlock_

  *Block for every row.*
- int ∗ whichColumnBlock_

  *Block for every column.*
- int ∗ dwBlock_

  *Block number for each proposal.*
- int ∗ backwardRow_

  *Points back to master rows.*
- int ∗ rowsInBlock_

  *Which rows are in blocke.*
- int ∗ columnsInBlock_

  *Which columns are in block.*
- int ∗ startRowBlock_

  *Starts for rowsInBlock.*
- int ∗ startColumnBlock_

  *Starts for columnsInBlock.*
- int ∗ intsInBlock_

  *Number of integer variables in each block.*
- unsigned int ∗ fingerPrint_

  *Bits set for 1 integers in each block.*
- unsigned short ∗ affinity_

  *Affinity each block has for other (will be triangular?)*
- int fullDWEverySoOften_

  *DW Proposal actions fullDWEverySoOften - 0 - off k - every k times solution gets better.*
- int numberPasses_

  *Number of passes.*
- int howOften_

  *How often to do (code can change)*
- int maximumDW_

  *Current maximum number of DW proposals.*
- int numberDW_

  *Number of DW proposals.*
- int numberDWTimes_

  *Number of times we have added to DW model.*

- int sizeFingerPrint_

  *Number of unsigned ints needed for each block of fingerPrint.*
- int numberMasterColumns_

  *Number of columns in master.*
- int numberMasterRows_

  *Number of rows in master.*
- int numberBlocks_

  *Number of blocks.*
- int keepContinuous_

  *Action on decomposition - 1 keep continuous, 0 don't.*
- int phase_

  *Phase of solution.*
- int pass_

  *Pass number.*
- int nNeededBase_

  *Base number of integers needed.*
- int nNodesBase_

  *Base number of nodes needed.*
- int nNeeded_

  *Base number of integers needed.*
- int nNodes_

  *Base number of nodes needed.*
- int numberBadPasses_

  *Number of passes without better solution.*

### 4.48.1 Detailed Description

This is unlike the other heuristics in that it is very very compute intensive.

It tries to find a DW structure and use that

Definition at line 17 of file CbcHeuristicDW.hpp.

### 4.48.2 Member Function Documentation

#### 4.48.2.1 virtual int CbcHeuristicDW::solution ( double & *objectiveValue*, double * *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) This does Relaxation Induced Neighborhood Search

Implements CbcHeuristic.

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

- CbcHeuristicDW.hpp

## 4.49 CbcHeuristicDynamic3 Class Reference

heuristic - just picks up any good solution

`#include <CbcLinked.hpp>`

Inheritance diagram for CbcHeuristicDynamic3:

Collaboration diagram for CbcHeuristicDynamic3:

### Public Member Functions

- virtual CbcHeuristic ∗ clone () const

  *Clone.*
- virtual void setModel (CbcModel ∗model)

  *update model*
- virtual int solution (double &objectiveValue, double ∗newSolution)

  *returns 0 if no solution, 1 if valid solution.*
- virtual void resetModel (CbcModel ∗model)

  *Resets stuff if model changes.*
- virtual bool canDealWithOdd () const

  *Returns true if can deal with "odd" problems e.g. sos type 2.*

### Additional Inherited Members

### 4.49.1 Detailed Description

heuristic - just picks up any good solution

Definition at line 379 of file CbcLinked.hpp.

### 4.49.2 Member Function Documentation

#### 4.49.2.1 virtual int CbcHeuristicDynamic3::solution ( double & *objectiveValue,* double ∗ *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) We leave all variables which are at one at this node of the tree to that value and will initially set all others to zero. We then sort all variables in order of their cost divided by the number of entries in rows which are not yet covered. We randomize that value a bit so that ties will be broken in different ways on different runs of the heuristic. We then choose the best one and set it to one and repeat the exercise.

Implements CbcHeuristic.

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

- CbcLinked.hpp

## 4.50 CbcHeuristicFPump Class Reference

Feasibility Pump class.

`#include <CbcHeuristicFPump.hpp>`

Inheritance diagram for CbcHeuristicFPump:

Collaboration diagram for CbcHeuristicFPump:

## Public Member Functions

- CbcHeuristicFPump & operator= (const CbcHeuristicFPump &rhs)

  *Assignment operator.*
- virtual CbcHeuristic ∗ clone () const

  *Clone.*
- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel ∗model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel ∗model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double ∗newSolution)

  *returns 0 if no solution, 1 if valid solution with better objective value than one passed in Sets solution values if good, sets objective value (only if good) This is called after cuts have been added - so can not add cuts.*
- void setMaximumTime (double value)

  *Set maximum Time (default off) - also sets starttime to current.*
- double maximumTime () const

  *Get maximum Time (default 0.0 == time limit off)*
- void setFakeCutoff (double value)

  *Set fake cutoff (default COIN_DBL_MAX == off)*
- double fakeCutoff () const

  *Get fake cutoff (default 0.0 == off)*
- void setAbsoluteIncrement (double value)

  *Set absolute increment (default 0.0 == off)*
- double absoluteIncrement () const

  *Get absolute increment (default 0.0 == off)*
- void setRelativeIncrement (double value)

  *Set relative increment (default 0.0 == off)*
- double relativeIncrement () const

  *Get relative increment (default 0.0 == off)*
- void setDefaultRounding (double value)

  *Set default rounding (default 0.5)*
- double defaultRounding () const

  *Get default rounding (default 0.5)*
- void setInitialWeight (double value)

  *Set initial weight (default 0.0 == off)*
- double initialWeight () const

  *Get initial weight (default 0.0 == off)*
- void setWeightFactor (double value)

  *Set weight factor (default 0.1)*
- double weightFactor () const

*Get weight factor (default 0.1)*
- void setArtificialCost (double value)

  *Set threshold cost for using original cost - even on continuous (default infinity)*
- double artificialCost () const

  *Get threshold cost for using original cost - even on continuous (default infinity)*
- double iterationRatio () const

  *Get iteration to size ratio.*
- void setIterationRatio (double value)

  *Set iteration to size ratio.*
- void setMaximumPasses (int value)

  *Set maximum passes (default 100)*
- int maximumPasses () const

  *Get maximum passes (default 100)*
- void setMaximumRetries (int value)

  *Set maximum retries (default 1)*
- int maximumRetries () const

  *Get maximum retries (default 1)*
- void setAccumulate (int value)

  *Set use of multiple solutions and solves 0 - do not reuse solves, do not accumulate integer solutions for local search 1 - do not reuse solves, accumulate integer solutions for local search 2 - reuse solves, do not accumulate integer solutions for local search 3 - reuse solves, accumulate integer solutions for local search If we add 4 then use second form of problem (with extra rows and variables for general integers) At some point (date?), I added.*
- int accumulate () const

  *Get accumulation option.*
- void setFixOnReducedCosts (int value)

  *Set whether to fix variables on known solution 0 - do not fix 1 - fix integers on reduced costs 2 - fix integers on reduced costs but only on entry.*
- int fixOnReducedCosts () const

  *Get reduced cost option.*
- void setReducedCostMultiplier (double value)

  *Set reduced cost multiplier 1.0 as normal $<$1.0 (x) - pretend gap is x$*$ actual gap - just for fixing.*
- double reducedCostMultiplier () const

  *Get reduced cost multiplier.*

## Protected Attributes

- double startTime_

  *Start time.*
- double maximumTime_

  *Maximum Cpu seconds.*
- double fakeCutoff_

  *Fake cutoff value.*
- double absoluteIncrement_

  *If positive carry on after solution expecting gain of at least this.*
- double relativeIncrement_

  *If positive carry on after solution expecting gain of at least this times objective.*
- double defaultRounding_

  *Default is round up if $>$ this.*

- double initialWeight_

  *Initial weight for true objective.*
- double weightFactor_

  *Factor for decreasing weight.*
- double artificialCost_

  *Threshold cost for using original cost - even on continuous.*
- double iterationRatio_

  *If iterationRatio >0 use instead of maximumPasses_ test is iterations > ratio*(2*nrow+ncol)*
- double reducedCostMultiplier_

  *Reduced cost multiplier 1.0 as normal <1.0 (x) - pretend gap is x* actual gap - just for fixing.*
- int maximumPasses_

  *Maximum number of passes.*
- int maximumRetries_

  *Maximum number of retries if we find a solution.*
- int accumulate_

  *Set use of multiple solutions and solves 0 - do not reuse solves, do not accumulate integer solutions for local search 1 - do not reuse solves, accumulate integer solutions for local search 2 - reuse solves, do not accumulate integer solutions for local search 3 - reuse solves, accumulate integer solutions for local search If we add 4 then use second form of problem (with extra rows and variables for general integers) If we do not accumulate solutions then no mini branch and bounds will be done reuse - refers to initial solve after adding in new "cut" If we add 8 then can run after initial cuts (if no solution)*
- int fixOnReducedCosts_

  *Set whether to fix variables on known solution 0 - do not fix 1 - fix integers on reduced costs 2 - fix integers on reduced costs but only on entry.*
- bool roundExpensive_

  *If true round to expensive.*

### 4.50.1 Detailed Description

Feasibility Pump class.

Definition at line 15 of file CbcHeuristicFPump.hpp.

### 4.50.2 Member Function Documentation

#### 4.50.2.1 virtual int CbcHeuristicFPump::solution ( double & *objectiveValue*, double * *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution with better objective value than one passed in Sets solution values if good, sets objective value (only if good) This is called after cuts have been added - so can not add cuts.

It may make sense for user to call this outside Branch and Cut to get solution. Or normally is just at root node.

new meanings for when_ - on first try then set back to 1 11 - at end fix all integers at same bound throughout 12 - also fix all integers staying at same internal integral value throughout 13 - also fix all continuous variables staying at same bound throughout 14 - also fix all continuous variables staying at same internal value throughout 15 - as 13 but no internal integers And beyond that, it's apparently possible for the range to be between 21 and 25, in which case it's reduced on entry to solution() to be between 11 and 15 and allSlack is set to true. Then, if we're not processing general integers, we'll use an all-slack basis to solve ... what? Don't see that yet.

Implements CbcHeuristic.

#### 4.50.2.2 void CbcHeuristicFPump::setAccumulate ( int *value* ) [inline]

Set use of multiple solutions and solves 0 - do not reuse solves, do not accumulate integer solutions for local search 1 - do not reuse solves, accumulate integer solutions for local search 2 - reuse solves, do not accumulate integer solutions for local search 3 - reuse solves, accumulate integer solutions for local search If we add 4 then use second form of problem (with extra rows and variables for general integers) At some point (date?), I added.

And then there are a few bit fields: 4 - something about general integers So my (lh) guess for 4 was at least in the ballpark, but I'll have to rethink 8 entirely (and it may well not mean the same thing as it did when I added that comment. 8 - determines whether we process general integers

And on 090831, John added

If we add 4 then use second form of problem (with extra rows and variables for general integers) If we add 8 then can run after initial cuts (if no solution)

Definition at line 175 of file CbcHeuristicFPump.hpp.

### 4.50.3 Member Data Documentation

#### 4.50.3.1 double CbcHeuristicFPump::fakeCutoff_ [protected]

Fake cutoff value.

If set then better of real cutoff and this used to add a constraint

Definition at line 215 of file CbcHeuristicFPump.hpp.

#### 4.50.3.2 int CbcHeuristicFPump::maximumRetries_ [protected]

Maximum number of retries if we find a solution.

If negative we clean out used array

Definition at line 241 of file CbcHeuristicFPump.hpp.

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

- CbcHeuristicFPump.hpp

## 4.51 CbcHeuristicGreedyCover Class Reference

Greedy heuristic classes.

`#include <CbcHeuristicGreedy.hpp>`

Inheritance diagram for CbcHeuristicGreedyCover:

Collaboration diagram for CbcHeuristicGreedyCover:

### Public Member Functions

- virtual CbcHeuristic * clone () const

  *Clone.*
- CbcHeuristicGreedyCover & operator= (const CbcHeuristicGreedyCover &rhs)

  *Assignment operator.*

- virtual void generateCpp (FILE ∗fp)

    *Create C++ lines to get to current state.*
- virtual void setModel (CbcModel ∗model)

    *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double ∗newSolution)

    *returns 0 if no solution, 1 if valid solution.*
- virtual void validate ()

    *Validate model i.e. sets when_ to 0 if necessary (may be NULL)*
- virtual void resetModel (CbcModel ∗model)

    *Resets stuff if model changes.*

## Protected Member Functions

- void gutsOfConstructor (CbcModel ∗model)

    *Guts of constructor from a CbcModel.*

## Protected Attributes

- int numberTimes_

    *Do this many times.*

### 4.51.1 Detailed Description

Greedy heuristic classes.

Definition at line 13 of file CbcHeuristicGreedy.hpp.

### 4.51.2 Member Function Documentation

#### 4.51.2.1 virtual int CbcHeuristicGreedyCover::solution ( double & *objectiveValue,* double ∗ *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) We leave all variables which are at one at this node of the tree to that value and will initially set all others to zero. We then sort all variables in order of their cost divided by the number of entries in rows which are not yet covered. We randomize that value a bit so that ties will be broken in different ways on different runs of the heuristic. We then choose the best one and set it to one and repeat the exercise.

Implements CbcHeuristic.

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

- CbcHeuristicGreedy.hpp

## 4.52 CbcHeuristicGreedyEquality Class Reference

Inheritance diagram for CbcHeuristicGreedyEquality:

Collaboration diagram for CbcHeuristicGreedyEquality:

## Public Member Functions

- virtual CbcHeuristic * clone () const

  *Clone.*
- CbcHeuristicGreedyEquality & operator= (const CbcHeuristicGreedyEquality &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE *fp)

  *Create C++ lines to get to current state.*
- virtual void setModel (CbcModel *model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double *newSolution)

  *returns 0 if no solution, 1 if valid solution.*
- virtual void validate ()

  *Validate model i.e. sets when_ to 0 if necessary (may be NULL)*
- virtual void resetModel (CbcModel *model)

  *Resets stuff if model changes.*

## Protected Member Functions

- void gutsOfConstructor (CbcModel *model)

  *Guts of constructor from a CbcModel.*

## Protected Attributes

- int numberTimes_

  *Do this many times.*

### 4.52.1 Detailed Description

Definition at line 98 of file CbcHeuristicGreedy.hpp.

### 4.52.2 Member Function Documentation

#### 4.52.2.1 virtual int CbcHeuristicGreedyEquality::solution ( double & *objectiveValue*, double * *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) We leave all variables which are at one at this node of the tree to that value and will initially set all others to zero. We then sort all variables in order of their cost divided by the number of entries in rows which are not yet covered. We randomize that value a bit so that ties will be broken in different ways on different runs of the heuristic. We then choose the best one and set it to one and repeat the exercise.

Implements CbcHeuristic.

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

- CbcHeuristicGreedy.hpp

## 4.53 CbcHeuristicGreedySOS Class Reference

Greedy heuristic for SOS and L rows (and positive elements)

`#include <CbcHeuristicGreedy.hpp>`

Inheritance diagram for CbcHeuristicGreedySOS:

Collaboration diagram for CbcHeuristicGreedySOS:

### Public Member Functions

- virtual CbcHeuristic ∗ clone () const

    *Clone.*
- CbcHeuristicGreedySOS & operator= (const CbcHeuristicGreedySOS &rhs)

    *Assignment operator.*
- virtual void generateCpp (FILE ∗fp)

    *Create C++ lines to get to current state.*
- virtual void setModel (CbcModel ∗model)

    *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double ∗newSolution)

    *returns 0 if no solution, 1 if valid solution.*
- virtual void validate ()

    *Validate model i.e. sets when_ to 0 if necessary (may be NULL)*
- virtual void resetModel (CbcModel ∗model)

    *Resets stuff if model changes.*

### Protected Member Functions

- void gutsOfConstructor (CbcModel ∗model)

    *Guts of constructor from a CbcModel.*

### Protected Attributes

- int numberTimes_

    *Do this many times.*

### 4.53.1 Detailed Description

Greedy heuristic for SOS and L rows (and positive elements)

Definition at line 193 of file CbcHeuristicGreedy.hpp.

### 4.53.2 Member Function Documentation

#### 4.53.2.1 virtual int CbcHeuristicGreedySOS::solution ( double & *objectiveValue,* double ∗ *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) We leave all variables which are at one at this node of the tree to that value and will initially set all others to zero. We then sort all variables in order of their cost divided by the number of entries in rows which are not yet covered. We randomize that value a bit so that ties will be broken in different ways on different runs of the heuristic. We then choose the best one and set it to one and repeat the exercise.

Implements CbcHeuristic.

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

- CbcHeuristicGreedy.hpp

## 4.54 CbcHeuristicJustOne Class Reference

Just One class - this chooses one at random.

`#include <CbcHeuristic.hpp>`

Inheritance diagram for CbcHeuristicJustOne:

Collaboration diagram for CbcHeuristicJustOne:

### Public Member Functions

- virtual CbcHeuristicJustOne ∗ clone () const

  *Clone.*
- CbcHeuristicJustOne & operator= (const CbcHeuristicJustOne &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*
- virtual int solution (double &objectiveValue, double ∗newSolution)

  *returns 0 if no solution, 1 if valid solution with better objective value than one passed in Sets solution values if good, sets objective value (only if good) This is called after cuts have been added - so can not add cuts This does Fractional Diving*
- virtual void resetModel (CbcModel ∗model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel ∗model)

  *update model (This is needed if cliques update matrix etc)*
- virtual bool selectVariableToBranch (**OsiSolverInterface** ∗, const double ∗, int &, int &)

  *Selects the next variable to branch on.*
- virtual void validate ()

  *Validate model i.e. sets when_ to 0 if necessary (may be NULL)*
- void addHeuristic (const CbcHeuristic ∗heuristic, double probability)

  *Adds an heuristic with probability.*
- void normalizeProbabilities ()

  *Normalize probabilities.*

### Additional Inherited Members

### 4.54.1 Detailed Description

Just One class - this chooses one at random.

Definition at line 611 of file CbcHeuristic.hpp.

### 4.54.2 Member Function Documentation

#### 4.54.2.1 virtual bool CbcHeuristicJustOne::selectVariableToBranch ( OsiSolverInterface ∗ , const double ∗ , int & , int & ) [inline], [virtual]

Selects the next variable to branch on.

Returns true if all the fractional variables can be trivially rounded. Returns false, if there is at least one fractional variable that is not trivially roundable. In this case, the bestColumn returned will not be trivially roundable. This is dummy as never called

Definition at line 655 of file CbcHeuristic.hpp.

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

- CbcHeuristic.hpp

## 4.55 CbcHeuristicLocal Class Reference

LocalSearch class.

`#include <CbcHeuristicLocal.hpp>`

Inheritance diagram for CbcHeuristicLocal:

Collaboration diagram for CbcHeuristicLocal:

### Public Member Functions

- virtual CbcHeuristic ∗ clone () const
  *Clone.*
- CbcHeuristicLocal & operator= (const CbcHeuristicLocal &rhs)
  *Assignment operator.*
- virtual void generateCpp (FILE ∗fp)
  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel ∗model)
  *Resets stuff if model changes.*
- virtual void setModel (CbcModel ∗model)
  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double ∗newSolution)
  *returns 0 if no solution, 1 if valid solution.*
- int solutionFix (double &objectiveValue, double ∗newSolution, const int ∗keep)
  *This version fixes stuff and does IP.*
- void setSearchType (int value)
  *Sets type of search.*
- int ∗ used () const
  *Used array so we can set.*

### Protected Attributes

- int ∗ used_
  *Whether a variable has been in a solution (also when)*

### 4.55.1 Detailed Description

LocalSearch class.

Definition at line 13 of file CbcHeuristicLocal.hpp.

### 4.55.2 Member Function Documentation

#### 4.55.2.1 virtual int CbcHeuristicLocal::solution ( double & *objectiveValue,* double ∗ *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) This is called after cuts have been added - so can not add cuts First tries setting a variable to better value. If feasible then tries setting others. If not feasible then tries swaps

This first version does not do LP's and does swaps of two integer variables. Later versions could do Lps.

Implements CbcHeuristic.

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

- CbcHeuristicLocal.hpp

## 4.56 CbcHeuristicNaive Class Reference

Naive class a) Fix all ints as close to zero as possible b) Fix all ints with nonzero costs and < large to zero c) Put bounds round continuous and UIs and maximize.

`#include <CbcHeuristicLocal.hpp>`

Inheritance diagram for CbcHeuristicNaive:

Collaboration diagram for CbcHeuristicNaive:

### Public Member Functions

- virtual CbcHeuristic ∗ clone () const

  *Clone.*
- CbcHeuristicNaive & operator= (const CbcHeuristicNaive &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel ∗model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel ∗model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double ∗newSolution)

  *returns 0 if no solution, 1 if valid solution.*
- void setLargeValue (double value)

  *Sets large cost value.*
- double largeValue () const

  *Gets large cost value.*

### Protected Attributes

- double large_

  *Data Large value.*

### 4.56.1 Detailed Description

Naive class a) Fix all ints as close to zero as possible b) Fix all ints with nonzero costs and < large to zero c) Put bounds round continuous and UIs and maximize.

Definition at line 154 of file CbcHeuristicLocal.hpp.

### 4.56.2 Member Function Documentation

#### 4.56.2.1 virtual int CbcHeuristicNaive::solution ( double & *objectiveValue*, double ∗ *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good)

Implements CbcHeuristic.

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

- CbcHeuristicLocal.hpp

## 4.57 CbcHeuristicNode Class Reference

A class describing the branching decisions that were made to get to the node where a heuristic was invoked from.

`#include <CbcHeuristic.hpp>`

### 4.57.1 Detailed Description

A class describing the branching decisions that were made to get to the node where a heuristic was invoked from.

Definition at line 28 of file CbcHeuristic.hpp.

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

- CbcHeuristic.hpp

## 4.58 CbcHeuristicNodeList Class Reference

### 4.58.1 Detailed Description

Definition at line 52 of file CbcHeuristic.hpp.

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

- CbcHeuristic.hpp

## 4.59 CbcHeuristicPartial Class Reference

Partial solution class If user knows a partial solution this tries to get an integer solution it uses hotstart information.

`#include <CbcHeuristic.hpp>`

Inheritance diagram for CbcHeuristicPartial:

Collaboration diagram for CbcHeuristicPartial:

### Public Member Functions

- CbcHeuristicPartial (CbcModel &model, int fixPriority=10000, int numberNodes=200)

  *Constructor with model - assumed before cuts Fixes all variables with priority <= given and does given number of nodes.*
- CbcHeuristicPartial & operator= (const CbcHeuristicPartial &rhs)

  *Assignment operator.*
- virtual CbcHeuristic ∗ clone () const

  *Clone.*
- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel ∗model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel ∗model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double ∗newSolution)

  *returns 0 if no solution, 1 if valid solution with better objective value than one passed in Sets solution values if good, sets objective value (only if good) This is called after cuts have been added - so can not add cuts*
- virtual void validate ()

  *Validate model i.e. sets when_ to 0 if necessary (may be NULL)*
- void setFixPriority (int value)

  *Set priority level.*
- virtual bool shouldHeurRun (int whereFrom)

  *Check whether the heuristic should run at all.*

### Additional Inherited Members

### 4.59.1 Detailed Description

Partial solution class If user knows a partial solution this tries to get an integer solution it uses hotstart information.

Definition at line 499 of file CbcHeuristic.hpp.

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

- CbcHeuristic.hpp

## 4.60 CbcHeuristicPivotAndFix Class Reference

LocalSearch class.

`#include <CbcHeuristicPivotAndFix.hpp>`

Inheritance diagram for CbcHeuristicPivotAndFix:

Collaboration diagram for CbcHeuristicPivotAndFix:

### Public Member Functions

- virtual CbcHeuristic * clone () const

  *Clone.*
- CbcHeuristicPivotAndFix & operator= (const CbcHeuristicPivotAndFix &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE *fp)

  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel *model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel *model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double *newSolution)

  *returns 0 if no solution, 1 if valid solution.*

### Additional Inherited Members

### 4.60.1 Detailed Description

LocalSearch class.

Definition at line 13 of file CbcHeuristicPivotAndFix.hpp.

### 4.60.2 Member Function Documentation

#### 4.60.2.1 virtual int CbcHeuristicPivotAndFix::solution ( double & *objectiveValue,* double * *newSolution* ) `[virtual]`

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) needs comments

Implements CbcHeuristic.

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

- CbcHeuristicPivotAndFix.hpp

## 4.61 CbcHeuristicProximity Class Reference

Inheritance diagram for CbcHeuristicProximity:

Collaboration diagram for CbcHeuristicProximity:

### Public Member Functions

- virtual CbcHeuristic * clone () const

  *Clone.*
- CbcHeuristicProximity & operator= (const CbcHeuristicProximity &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE *fp)

  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel *model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel *model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double *newSolution)

  *returns 0 if no solution, 1 if valid solution.*
- void setIncrement (double value)

  *Set extra increment.*
- int * used () const

  *Used array so we can set.*

### Protected Attributes

- double increment_

  *Increment to use if no change.*
- CbcHeuristicFPump * feasibilityPump_

  *Copy of Feasibility pump.*
- int numberSolutions_

  *Number of solutions so we only do after new solution.*
- int * used_

  *Whether a variable has been in a solution (also when)*

## 4.61.1 Detailed Description

Definition at line 90 of file CbcHeuristicLocal.hpp.

## 4.61.2 Member Function Documentation

#### 4.61.2.1 virtual int CbcHeuristicProximity::solution ( double & *objectiveValue,* double * *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good)

Implements CbcHeuristic.

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

- CbcHeuristicLocal.hpp

## 4.62 CbcHeuristicRandRound Class Reference

LocalSearch class.

`#include <CbcHeuristicRandRound.hpp>`

Inheritance diagram for CbcHeuristicRandRound:

Collaboration diagram for CbcHeuristicRandRound:

### Public Member Functions

- virtual CbcHeuristic * clone () const

  *Clone.*
- CbcHeuristicRandRound & operator= (const CbcHeuristicRandRound &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE *fp)

  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel *model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel *model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double *newSolution)

  *returns 0 if no solution, 1 if valid solution.*

### Additional Inherited Members

### 4.62.1 Detailed Description

LocalSearch class.

Definition at line 13 of file CbcHeuristicRandRound.hpp.

### 4.62.2 Member Function Documentation

#### 4.62.2.1 virtual int CbcHeuristicRandRound::solution ( double & *objectiveValue*, double * *newSolution* ) `[virtual]`

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) needs comments

Implements CbcHeuristic.

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

- CbcHeuristicRandRound.hpp

## 4.63 CbcHeuristicRENS Class Reference

LocalSearch class.

`#include <CbcHeuristicRENS.hpp>`

Inheritance diagram for CbcHeuristicRENS:

Collaboration diagram for CbcHeuristicRENS:

## Public Member Functions

- virtual CbcHeuristic * clone () const

    *Clone.*
- CbcHeuristicRENS & operator= (const CbcHeuristicRENS &rhs)

    *Assignment operator.*
- virtual void resetModel (CbcModel *model)

    *Resets stuff if model changes.*
- virtual void setModel (CbcModel *model)

    *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double *newSolution)

    *returns 0 if no solution, 1 if valid solution.*
- void setRensType (int value)

    *Set type.*

## Protected Attributes

- int numberTries_

    *Number of tries.*
- int rensType_

    *Type 0 - fix at LB 1 - fix on dj 2 - fix at UB as well 3 - fix on 0.01*average dj add 16 to allow two tries 32 - if solution exists use to keep more variables 64 - if priorities keep high priority 128 - if priorities keep low priority.*

### 4.63.1 Detailed Description

LocalSearch class.

Definition at line 16 of file CbcHeuristicRENS.hpp.

### 4.63.2 Member Function Documentation

#### 4.63.2.1 virtual int CbcHeuristicRENS::solution ( double & *objectiveValue*, double * *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) This does Relaxation Extension Neighborhood Search Does not run if when_<2 and a solution exists

Implements CbcHeuristic.

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

- CbcHeuristicRENS.hpp

## 4.64 CbcHeuristicRINS Class Reference

LocalSearch class.

`#include <CbcHeuristicRINS.hpp>`

Inheritance diagram for CbcHeuristicRINS:

Collaboration diagram for CbcHeuristicRINS:

### Public Member Functions

- virtual CbcHeuristic ∗ clone () const
  *Clone.*
- CbcHeuristicRINS & operator= (const CbcHeuristicRINS &rhs)
  *Assignment operator.*
- virtual void generateCpp (FILE ∗fp)
  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel ∗model)
  *Resets stuff if model changes.*
- virtual void setModel (CbcModel ∗model)
  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double ∗newSolution)
  *returns 0 if no solution, 1 if valid solution.*
- int solutionFix (double &objectiveValue, double ∗newSolution, const int ∗keep)
  *This version fixes stuff and does IP.*
- void setHowOften (int value)
  *Sets how often to do it.*
- char ∗ used () const
  *Used array so we can set.*
- void setLastNode (int value)
  *Resets lastNode.*
- void setSolutionCount (int value)
  *Resets number of solutions.*

### Protected Attributes

- int numberSolutions_
  *Number of solutions so we can do something at solution.*
- int howOften_
  *How often to do (code can change)*
- int numberSuccesses_
  *Number of successes.*
- int numberTries_
  *Number of tries.*
- int stateOfFixing_
  *State of fixing continuous variables - 0 - not tried +n - this divisor makes small enough -n - this divisor still not small enough.*
- int lastNode_
  *Node when last done.*
- char ∗ used_
  *Whether a variable has been in a solution.*

### 4.64.1 Detailed Description

LocalSearch class.

Definition at line 17 of file CbcHeuristicRINS.hpp.

### 4.64.2 Member Function Documentation

#### 4.64.2.1 virtual int CbcHeuristicRINS::solution ( double & *objectiveValue*, double ∗ *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) This does Relaxation Induced Neighborhood Search

Implements CbcHeuristic.

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

- CbcHeuristicRINS.hpp

## 4.65 CbcHeuristicVND Class Reference

LocalSearch class.

`#include <CbcHeuristicVND.hpp>`

Inheritance diagram for CbcHeuristicVND:

Collaboration diagram for CbcHeuristicVND:

### Public Member Functions

- virtual CbcHeuristic ∗ clone () const

  *Clone.*
- CbcHeuristicVND & operator= (const CbcHeuristicVND &rhs)

  *Assignment operator.*
- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel ∗model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel ∗model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double ∗newSolution)

  *returns 0 if no solution, 1 if valid solution.*
- int solutionFix (double &objectiveValue, double ∗newSolution, const int ∗keep)

  *This version fixes stuff and does IP.*
- void setHowOften (int value)

  *Sets how often to do it.*
- double ∗ baseSolution () const

  *base solution array so we can set*

### Protected Attributes

- int numberSolutions_

  *Number of solutions so we can do something at solution.*
- int howOften_

  *How often to do (code can change)*
- int numberSuccesses_

  *Number of successes.*
- int numberTries_

  *Number of tries.*
- int lastNode_

  *Node when last done.*
- int stepSize_

  *Step size for decomposition.*
- double ∗ baseSolution_

  *Base solution.*

### 4.65.1 Detailed Description

LocalSearch class.

Definition at line 17 of file CbcHeuristicVND.hpp.

### 4.65.2 Member Function Documentation

#### 4.65.2.1 virtual int CbcHeuristicVND::solution ( double & *objectiveValue,* double ∗ *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) This does Relaxation Induced Neighborhood Search

Implements CbcHeuristic.

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

- CbcHeuristicVND.hpp

## 4.66 CbcIdiotBranch Class Reference

Define an idiotic idea class.

`#include <CbcFollowOn.hpp>`

Inheritance diagram for CbcIdiotBranch:

Collaboration diagram for CbcIdiotBranch:

### Public Member Functions

- CbcIdiotBranch (CbcModel ∗model)

  *Useful constructor.*

- virtual CbcObject * clone () const

  *Clone.*
- virtual double infeasibility (const **OsiBranchingInformation** *info, int &preferredWay) const

  *Infeasibility - large is 0.5.*
- virtual void feasibleRegion ()

  *This looks at solution and sets bounds to contain solution.*
- virtual CbcBranchingObject * createCbcBranch (**OsiSolverInterface** *solver, const **OsiBranchingInformation** *info, int way)

  *Creates a branching object.*
- virtual void initializeForBranching (CbcModel *)

  *Initialize for branching.*

### Protected Member Functions

- **OsiRowCut** buildCut (const **OsiBranchingInformation** *info, int type, int &preferredWay) const

  *Build "cut".*

### Protected Attributes

- **CoinThreadRandom** randomNumberGenerator_

  *data Thread specific random number generator*
- **CoinThreadRandom** savedRandomNumberGenerator_

  *Saved version of thread specific random number generator.*

### 4.66.1 Detailed Description

Define an idiotic idea class.

The idea of this is that we take some integer variables away from integer and sum them with some randomness to get signed sum close to 0.5. We then can branch to exclude that gap.

This branching rule should be in addition to normal rules and have a high priority.

Definition at line 161 of file CbcFollowOn.hpp.

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

- CbcFollowOn.hpp

## 4.67 CbcIntegerBranchingObject Class Reference

Simple branching object for an integer variable.

`#include <CbcSimpleInteger.hpp>`

Inheritance diagram for CbcIntegerBranchingObject:

Collaboration diagram for CbcIntegerBranchingObject:

## Public Member Functions

- CbcIntegerBranchingObject ()

    *Default constructor.*
- CbcIntegerBranchingObject (CbcModel ∗model, int variable, int way, double **value**)

    *Create a standard floor/ceiling branch object.*
- CbcIntegerBranchingObject (CbcModel ∗model, int variable, int way, double lowerValue, double upperValue)

    *Create a degenerate branch object.*
- CbcIntegerBranchingObject (const CbcIntegerBranchingObject &)

    *Copy constructor.*
- CbcIntegerBranchingObject & operator= (const CbcIntegerBranchingObject &rhs)

    *Assignment operator.*
- virtual CbcBranchingObject ∗ clone () const

    *Clone.*
- virtual ∼CbcIntegerBranchingObject ()

    *Destructor.*
- void fillPart (int variable, int way, double **value**)

    *Does part of constructor.*
- virtual double branch ()

    *Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.*
- virtual void fix (**OsiSolverInterface** ∗solver, double ∗lower, double ∗upper, int branchState) const

    *Update bounds in solver as in 'branch' and update given bounds.*
- virtual bool tighten (**OsiSolverInterface** ∗)

    *Change (tighten) bounds in object to reflect bounds in solver.*
- virtual void print ()

    *Print something about branch - only if log level high.*
- const double ∗ downBounds () const

    *Lower and upper bounds for down branch.*
- const double ∗ upBounds () const

    *Lower and upper bounds for up branch.*
- void setDownBounds (const double bounds[2])

    *Set lower and upper bounds for down branch.*
- void setUpBounds (const double bounds[2])

    *Set lower and upper bounds for up branch.*
- virtual CbcBranchObjType type () const

    *Return the type (an integer identifier) of `this`.*
- virtual CbcRangeCompare compareBranchingObject (const CbcBranchingObject ∗brObj, const bool replaceIf←Overlap=false)

    *Compare the `this` with `brObj`.*

## Protected Attributes

- double down_ [2]

    *Lower [0] and upper [1] bounds for the down arm (way_ = -1)*
- double up_ [2]

    *Lower [0] and upper [1] bounds for the up arm (way_ = 1)*

### 4.67.1 Detailed Description

Simple branching object for an integer variable.

This object can specify a two-way branch on an integer variable. For each arm of the branch, the upper and lower bounds on the variable can be independently specified.

Variable_ holds the index of the integer variable in the integerVariable_ array of the model.

Definition at line 23 of file CbcSimpleInteger.hpp.

### 4.67.2 Constructor & Destructor Documentation

#### 4.67.2.1 CbcIntegerBranchingObject::CbcIntegerBranchingObject ( CbcModel * *model*, int *variable*, int *way*, double *value* )

Create a standard floor/ceiling branch object.

Specifies a simple two-way branch. Let `value` = x*. One arm of the branch will be lb <= x <= floor(x*), the other ceil(x*) <= x <= ub. Specify way = -1 to set the object state to perform the down arm first, way = 1 for the up arm.

#### 4.67.2.2 CbcIntegerBranchingObject::CbcIntegerBranchingObject ( CbcModel * *model*, int *variable*, int *way*, double *lowerValue*, double *upperValue* )

Create a degenerate branch object.

Specifies a 'one-way branch'. Calling branch() for this object will always result in lowerValue <= x <= upperValue. Used to fix a variable when lowerValue = upperValue.

### 4.67.3 Member Function Documentation

#### 4.67.3.1 virtual double CbcIntegerBranchingObject::branch ( ) [virtual]

Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.

Returns change in guessed objective on next branch

Implements CbcBranchingObject.

Reimplemented in CbcIntegerPseudoCostBranchingObject, and CbcDynamicPseudoCostBranchingObject.

#### 4.67.3.2 virtual void CbcIntegerBranchingObject::fix ( OsiSolverInterface * *solver*, double * *lower*, double * *upper*, int *branchState* ) const [virtual]

Update bounds in solver as in 'branch' and update given bounds.

branchState is -1 for 'down' +1 for 'up'

Reimplemented from CbcBranchingObject.

#### 4.67.3.3 virtual bool CbcIntegerBranchingObject::tighten ( OsiSolverInterface * ) [virtual]

Change (tighten) bounds in object to reflect bounds in solver.

Return true if now fixed

Reimplemented from CbcBranchingObject.

#### 4.67.3.4 virtual CbcRangeCompare CbcIntegerBranchingObject::compareBranchingObject ( const CbcBranchingObject * *brObj*, const bool *replaceIfOverlap* = false ) [virtual]

Compare the this with brObj.

this and brObj must be os the same type and must have the same original object, but they may have different feasible regions. Return the appropriate CbcRangeCompare value (first argument being the sub/superset if that's the case). In case of overlap (and if replaceIfOverlap is true) replace the current branching object with one whose feasible region is the overlap.

Implements CbcBranchingObject.

Reimplemented in CbcIntegerPseudoCostBranchingObject.

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

- CbcSimpleInteger.hpp

## 4.68 CbcIntegerPseudoCostBranchingObject Class Reference

Simple branching object for an integer variable with pseudo costs.

`#include <CbcSimpleIntegerDynamicPseudoCost.hpp>`

Inheritance diagram for CbcIntegerPseudoCostBranchingObject:

Collaboration diagram for CbcIntegerPseudoCostBranchingObject:

### Public Member Functions

- CbcIntegerPseudoCostBranchingObject ()

  *Default constructor.*
- CbcIntegerPseudoCostBranchingObject (CbcModel *model, int variable, int way, double **value**)

  *Create a standard floor/ceiling branch object.*
- CbcIntegerPseudoCostBranchingObject (CbcModel *model, int variable, int way, double lowerValue, double upperValue)

  *Create a degenerate branch object.*
- CbcIntegerPseudoCostBranchingObject (const CbcIntegerPseudoCostBranchingObject &)

  *Copy constructor.*
- CbcIntegerPseudoCostBranchingObject & operator= (const CbcIntegerPseudoCostBranchingObject &rhs)

  *Assignment operator.*
- virtual CbcBranchingObject * clone () const

  *Clone.*
- virtual ~CbcIntegerPseudoCostBranchingObject ()

  *Destructor.*
- virtual double branch ()

  *Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.*
- double changeInGuessed () const

  *Change in guessed.*
- void setChangeInGuessed (double **value**)

*Set change in guessed.*

- virtual CbcBranchObjType type () const

  *Return the type (an integer identifier) of* `this`.

- virtual CbcRangeCompare compareBranchingObject (const CbcBranchingObject *brObj, const bool replaceIfOverlap=false)

  *Compare the* `this` *with* `brObj`.

### Protected Attributes

- double changeInGuessed_

  *Change in guessed objective value for next branch.*

### 4.68.1 Detailed Description

Simple branching object for an integer variable with pseudo costs.

This object can specify a two-way branch on an integer variable. For each arm of the branch, the upper and lower bounds on the variable can be independently specified.

Variable_ holds the index of the integer variable in the integerVariable_ array of the model.

Definition at line 393 of file CbcSimpleIntegerDynamicPseudoCost.hpp.

### 4.68.2 Constructor & Destructor Documentation

#### 4.68.2.1 CbcIntegerPseudoCostBranchingObject::CbcIntegerPseudoCostBranchingObject ( CbcModel * *model*, int *variable*, int *way*, double *value* )

Create a standard floor/ceiling branch object.

Specifies a simple two-way branch. Let `value` = x*. One arm of the branch will be is lb <= x <= floor(x*), the other ceil(x*) <= x <= ub. Specify way = -1 to set the object state to perform the down arm first, way = 1 for the up arm.

#### 4.68.2.2 CbcIntegerPseudoCostBranchingObject::CbcIntegerPseudoCostBranchingObject ( CbcModel * *model*, int *variable*, int *way*, double *lowerValue*, double *upperValue* )

Create a degenerate branch object.

Specifies a 'one-way branch'. Calling branch() for this object will always result in lowerValue <= x <= upperValue. Used to fix a variable when lowerValue = upperValue.

### 4.68.3 Member Function Documentation

#### 4.68.3.1 virtual double CbcIntegerPseudoCostBranchingObject::branch ( ) [virtual]

Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.

This version also changes guessed objective value

Reimplemented from CbcIntegerBranchingObject.

#### 4.68.3.2 virtual CbcRangeCompare CbcIntegerPseudoCostBranchingObject::compareBranchingObject ( const CbcBranchingObject ∗ *brObj*, const bool *replaceIfOverlap* = false ) [virtual]

Compare the `this` with `brObj`.

`this` and `brObj` must be os the same type and must have the same original object, but they may have different feasible regions. Return the appropriate CbcRangeCompare value (first argument being the sub/superset if that's the case). In case of overlap (and if `replaceIfOverlap` is true) replace the current branching object with one whose feasible region is the overlap.

Reimplemented from CbcIntegerBranchingObject.

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

- CbcSimpleIntegerDynamicPseudoCost.hpp

## 4.69 CbcLongCliqueBranchingObject Class Reference

Unordered Clique Branching Object class.

`#include <CbcClique.hpp>`

Inheritance diagram for CbcLongCliqueBranchingObject:

Collaboration diagram for CbcLongCliqueBranchingObject:

### Public Member Functions

- virtual CbcBranchingObject ∗ clone () const

    *Clone.*
- virtual double branch ()

    *Does next branch and updates state.*
- virtual void print ()

    *Print something about branch - only if log level high.*
- virtual CbcBranchObjType type () const

    *Return the type (an integer identifier) of `this`.*
- virtual int compareOriginalObject (const CbcBranchingObject ∗brObj) const

    *Compare the original object of `this` with the original object of `brObj`.*
- virtual CbcRangeCompare compareBranchingObject (const CbcBranchingObject ∗brObj, const bool replaceIf↵ Overlap=false)

    *Compare the `this` with `brObj`.*

### Additional Inherited Members

### 4.69.1 Detailed Description

Unordered Clique Branching Object class.

These are for cliques which are $>64$ members Variable is number of clique.

Definition at line 234 of file CbcClique.hpp.

### 4.69.2 Member Function Documentation

#### 4.69.2.1 virtual int CbcLongCliqueBranchingObject::compareOriginalObject ( const CbcBranchingObject * brObj ) const [virtual]

Compare the original object of `this` with the original object of `brObj`.

Assumes that there is an ordering of the original objects. This method should be invoked only if `this` and brObj are of the same type. Return negative/0/positive depending on whether `this` is smaller/same/larger than the argument.

Reimplemented from CbcBranchingObject.

#### 4.69.2.2 virtual CbcRangeCompare CbcLongCliqueBranchingObject::compareBranchingObject ( const CbcBranchingObject * brObj, const bool replaceIfOverlap = false ) [virtual]

Compare the `this` with `brObj`.

`this` and `brObj` must be os the same type and must have the same original object, but they may have different feasible regions. Return the appropriate CbcRangeCompare value (first argument being the sub/superset if that's the case). In case of overlap (and if `replaceIfOverlap` is true) replace the current branching object with one whose feasible region is the overlap.

Implements CbcBranchingObject.

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

- CbcClique.hpp

## 4.70 CbcLotsize Class Reference

Lotsize class.

`#include <CbcBranchLotsize.hpp>`

Inheritance diagram for CbcLotsize:

Collaboration diagram for CbcLotsize:

### Public Member Functions

- virtual CbcObject * clone () const

  *Clone.*
- virtual double infeasibility (const **OsiBranchingInformation** *info, int &preferredWay) const

  *Infeasibility - large is 0.5.*
- virtual void feasibleRegion ()

  *Set bounds to contain the current solution.*
- virtual CbcBranchingObject * createCbcBranch (**OsiSolverInterface** *solver, const **OsiBranchingInformation** *info, int way)

  *Creates a branching object.*
- virtual CbcBranchingObject * preferredNewFeasible () const

  *Given a valid solution (with reduced costs, etc.), return a branching object which would give a new feasible point in the good direction.*
- virtual CbcBranchingObject * notPreferredNewFeasible () const

*Given a valid solution (with reduced costs, etc.), return a branching object which would give a new feasible point in a bad direction.*

- virtual void resetBounds (const **OsiSolverInterface** ∗solver)

  *Reset original upper and lower bound values from the solver.*
- bool findRange (double value) const

  *Finds range of interest so value is feasible in range range_ or infeasible between hi[range_] and lo[range_+1].*
- virtual void floorCeiling (double &floorLotsize, double &ceilingLotsize, double value, double tolerance) const

  *Returns floor and ceiling.*
- int modelSequence () const

  *Model column number.*
- void setModelSequence (int value)

  *Set model column number.*
- virtual int columnNumber () const

  *Column number if single column object -1 otherwise, so returns >= 0 Used by heuristics.*
- double originalLowerBound () const

  *Original variable bounds.*
- int rangeType () const

  *Type - 1 points, 2 ranges.*
- int numberRanges () const

  *Number of points.*
- double ∗ bound () const

  *Ranges.*
- virtual bool canDoHeuristics () const

  *Return true if object can take part in normal heuristics.*

## Additional Inherited Members

### 4.70.1 Detailed Description

Lotsize class.

Definition at line 13 of file CbcBranchLotsize.hpp.

### 4.70.2 Member Function Documentation

#### 4.70.2.1 virtual void CbcLotsize::feasibleRegion ( ) [virtual]

Set bounds to contain the current solution.

More precisely, for the variable associated with this object, take the value given in the current solution, force it within the current bounds if required, then set the bounds to fix the variable at the integer nearest the solution value.

Implements CbcObject.

#### 4.70.2.2 virtual CbcBranchingObject∗ CbcLotsize::preferredNewFeasible ( ) const [virtual]

Given a valid solution (with reduced costs, etc.), return a branching object which would give a new feasible point in the good direction.

The preferred branching object will force the variable to be +/-1 from its current value, depending on the reduced cost and objective sense. If movement in the direction which improves the objective is impossible due to bounds on the variable, the branching object will move in the other direction. If no movement is possible, the method returns NULL.

Only the bounds on this variable are considered when determining if the new point is feasible.

Reimplemented from CbcObject.

#### 4.70.2.3 virtual **CbcBranchingObject**∗ CbcLotsize::notPreferredNewFeasible ( ) const [virtual]

Given a valid solution (with reduced costs, etc.), return a branching object which would give a new feasible point in a bad direction.

As for preferredNewFeasible(), but the preferred branching object will force movement in a direction that degrades the objective.

Reimplemented from CbcObject.

#### 4.70.2.4 virtual void CbcLotsize::resetBounds ( const **OsiSolverInterface** ∗ *solver* ) [virtual]

Reset original upper and lower bound values from the solver.

Handy for updating bounds held in this object after bounds held in the solver have been tightened.

Reimplemented from CbcObject.

#### 4.70.2.5 bool CbcLotsize::findRange ( double *value* ) const

Finds range of interest so value is feasible in range range_ or infeasible between hi[range_] and lo[range_+1].

Returns true if feasible.

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

- CbcBranchLotsize.hpp

## 4.71 CbcLotsizeBranchingObject Class Reference

Lotsize branching object.

`#include <CbcBranchLotsize.hpp>`

Inheritance diagram for CbcLotsizeBranchingObject:

Collaboration diagram for CbcLotsizeBranchingObject:

### Public Member Functions

- CbcLotsizeBranchingObject ()

  *Default constructor.*
- CbcLotsizeBranchingObject (CbcModel ∗model, int variable, int way, double **value**, const CbcLotsize ∗lotsize)

  *Create a lotsize floor/ceiling branch object.*
- CbcLotsizeBranchingObject (CbcModel ∗model, int variable, int way, double lowerValue, double upperValue)

  *Create a degenerate branch object.*

- CbcLotsizeBranchingObject (const CbcLotsizeBranchingObject &)

  *Copy constructor.*
- CbcLotsizeBranchingObject & operator= (const CbcLotsizeBranchingObject &rhs)

  *Assignment operator.*
- virtual CbcBranchingObject ∗ clone () const

  *Clone.*
- virtual ∼CbcLotsizeBranchingObject ()

  *Destructor.*
- virtual double branch ()

  *Sets the bounds for the variable according to the current arm of the branch and advances the object state to the next arm.*
- virtual void print ()

  *Print something about branch - only if log level high.*
- virtual CbcBranchObjType type () const

  *Return the type (an integer identifier) of `this`.*
- virtual CbcRangeCompare compareBranchingObject (const CbcBranchingObject ∗brObj, const bool replaceIf↩ Overlap=false)

  *Compare the `this` with `brObj`.*

### Protected Attributes

- double down_ [2]

  *Lower [0] and upper [1] bounds for the down arm (way_ = -1)*
- double up_ [2]

  *Lower [0] and upper [1] bounds for the up arm (way_ = 1)*

### 4.71.1 Detailed Description

Lotsize branching object.

This object can specify a two-way branch on an integer variable. For each arm of the branch, the upper and lower bounds on the variable can be independently specified.

Variable_ holds the index of the integer variable in the integerVariable_ array of the model.

Definition at line 166 of file CbcBranchLotsize.hpp.

### 4.71.2 Constructor & Destructor Documentation

#### 4.71.2.1 CbcLotsizeBranchingObject::CbcLotsizeBranchingObject ( CbcModel ∗ *model,* int *variable,* int *way,* double *value,* const CbcLotsize ∗ *lotsize* )

Create a lotsize floor/ceiling branch object.

Specifies a simple two-way branch. Let `value` = x∗. One arm of the branch will be is lb <= x <= valid range below(x∗), the other valid range above(x∗) <= x <= ub. Specify way = -1 to set the object state to perform the down arm first, way = 1 for the up arm.

#### 4.71.2.2 CbcLotsizeBranchingObject::CbcLotsizeBranchingObject ( CbcModel * *model*, int *variable*, int *way*, double *lowerValue*, double *upperValue* )

Create a degenerate branch object.

Specifies a 'one-way branch'. Calling branch() for this object will always result in lowerValue <= x <= upperValue. Used to fix in valid range

### 4.71.3 Member Function Documentation

#### 4.71.3.1 virtual CbcRangeCompare CbcLotsizeBranchingObject::compareBranchingObject ( const CbcBranchingObject * *brObj*, const bool *replaceIfOverlap* = `false` ) `[virtual]`

Compare the `this` with `brObj`.

`this` and `brObj` must be os the same type and must have the same original object, but they may have different feasible regions. Return the appropriate CbcRangeCompare value (first argument being the sub/superset if that's the case). In case of overlap (and if `replaceIfOverlap` is true) replace the current branching object with one whose feasible region is the overlap.

Implements CbcBranchingObject.

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

- CbcBranchLotsize.hpp

## 4.72 CbcMessage Class Reference

Inheritance diagram for CbcMessage:

Collaboration diagram for CbcMessage:

### Public Member Functions

**Constructors etc**

- CbcMessage (**Language language**=us_en)
  *Constructor.*

### 4.72.1 Detailed Description

Definition at line 81 of file CbcMessage.hpp.

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

- CbcMessage.hpp

## 4.73 CbcModel Class Reference

Simple Branch and bound class.

`#include <CbcModel.hpp>`

## Public Types

- enum CbcIntParam {
CbcMaxNumNode = 0, CbcMaxNumSol, CbcFathomDiscipline, CbcPrinting,
CbcNumberBranches, CbcLastIntParam }
- enum CbcDblParam {
CbcIntegerTolerance = 0, CbcInfeasibilityWeight, CbcCutoffIncrement, CbcAllowableGap,
CbcAllowableFractionGap, CbcMaximumSeconds, CbcCurrentCutoff, CbcOptimizationDirection,
CbcCurrentObjectiveValue, CbcCurrentMinimizationObjectiveValue, CbcStartSeconds, CbcHeuristicGap,
CbcHeuristicFractionGap, CbcSmallestChange, CbcSumChange, CbcLargestChange,
CbcSmallChange, CbcLastDblParam }

## Public Member Functions

### Presolve methods

- CbcModel ∗ findCliques (bool makeEquality, int atLeastThisMany, int lessThanThis, int defaultValue=1000)
  *Identify cliques and construct corresponding objects.*
- CbcModel ∗ integerPresolve (bool weak=false)
  *Do integer presolve, creating a new (presolved) model.*
- bool integerPresolveThisModel (**OsiSolverInterface** ∗originalSolver, bool weak=false)
  *Do integer presolve, modifying the current model.*
- void originalModel (CbcModel ∗presolvedModel, bool weak)
  *Put back information into the original model after integer presolve.*
- bool tightenVubs (int type, bool allowMultipleBinary=false, double useCutoff=1.0e50)
  *For variables involved in VUB constraints, see if we can tighten bounds by solving lp's.*
- bool tightenVubs (int numberVubs, const int ∗which, double useCutoff=1.0e50)
  *For variables involved in VUB constraints, see if we can tighten bounds by solving lp's.*
- void analyzeObjective ()
  *Analyze problem to find a minimum change in the objective function.*
- void AddIntegers ()
  *Add additional integers.*
- void saveModel (**OsiSolverInterface** ∗saveSolver, double ∗checkCutoffForRestart, bool ∗feasible)
  *Save copy of the model.*
- void flipModel ()
  *Flip direction of optimization on all models.*

### Object manipulation routines

*See* **OsiObject** *for an explanation of 'object' in the context of CbcModel.*

- int numberObjects () const
  *Get the number of objects.*
- void setNumberObjects (int number)
  *Set the number of objects.*
- **OsiObject** ∗∗ objects () const
  *Get the array of objects.*
- const **OsiObject** ∗ object (int which) const
  *Get the specified object.*
- **OsiObject** ∗ modifiableObject (int which) const
  *Get the specified object.*
- void **setOptionalInteger** (int index)
- void deleteObjects (bool findIntegers=true)

*Delete all object information (and just back to integers if true)*

- void addObjects (int numberObjects, **OsiObject** ∗∗objects)

  *Add in object information.*
- void addObjects (int numberObjects, CbcObject ∗∗objects)

  *Add in object information.*
- void synchronizeModel ()

  *Ensure attached objects point to this model.*
- void findIntegers (bool startAgain, int type=0)

  *Identify integer variables and create corresponding objects.*

### Parameter set/get methods

*The set methods return true if the parameter was set to the given value, false if the value of the parameter is out of range.*

*The get methods return the value of the parameter.*

- bool setIntParam (CbcIntParam key, int value)

  *Set an integer parameter.*
- bool setDblParam (CbcDblParam key, double value)

  *Set a double parameter.*
- int getIntParam (CbcIntParam key) const

  *Get an integer parameter.*
- double getDblParam (CbcDblParam key) const

  *Get a double parameter.*
- void setCutoff (double value)

  *Set cutoff bound on the objective function.*
- double getCutoff () const

  *Get the cutoff bound on the objective function - always as minimize.*
- bool setMaximumNodes (int value)

  *Set the maximum node limit .*
- int getMaximumNodes () const

  *Get the maximum node limit .*
- bool setMaximumSolutions (int value)

  *Set the maximum number of solutions desired.*
- int getMaximumSolutions () const

  *Get the maximum number of solutions desired.*
- bool setPrintingMode (int value)

  *Set the printing mode.*
- int getPrintingMode () const

  *Get the printing mode.*
- bool setMaximumSeconds (double value)

  *Set the maximum number of seconds desired.*
- double getMaximumSeconds () const

  *Get the maximum number of seconds desired.*
- double getCurrentSeconds () const

  *Current time since start of branchAndbound.*
- bool maximumSecondsReached () const

  *Return true if maximum time reached.*
- bool setIntegerTolerance (double value)

  *Set the integrality tolerance .*
- double getIntegerTolerance () const

  *Get the integrality tolerance .*
- bool setInfeasibilityWeight (double value)

*Set the weight per integer infeasibility .*

- double getInfeasibilityWeight () const

  *Get the weight per integer infeasibility .*
- bool setAllowableGap (double value)

  *Set the allowable gap between the best known solution and the best possible solution.*
- double getAllowableGap () const

  *Get the allowable gap between the best known solution and the best possible solution.*
- bool setAllowableFractionGap (double value)

  *Set the fraction allowable gap between the best known solution and the best possible solution.*
- double getAllowableFractionGap () const

  *Get the fraction allowable gap between the best known solution and the best possible solution.*
- bool setAllowablePercentageGap (double value)

  *Set the percentage allowable gap between the best known solution and the best possible solution.*
- double getAllowablePercentageGap () const

  *Get the percentage allowable gap between the best known solution and the best possible solution.*
- bool setHeuristicGap (double value)

  *Set the heuristic gap between the best known solution and the best possible solution.*
- double getHeuristicGap () const

  *Get the heuristic gap between the best known solution and the best possible solution.*
- bool setHeuristicFractionGap (double value)

  *Set the fraction heuristic gap between the best known solution and the best possible solution.*
- double getHeuristicFractionGap () const

  *Get the fraction heuristic gap between the best known solution and the best possible solution.*
- bool setCutoffIncrement (double value)

  *Set the CbcModel::CbcCutoffIncrement desired.*
- double getCutoffIncrement () const

  *Get the CbcModel::CbcCutoffIncrement desired.*
- bool canStopOnGap () const

  *See if can stop on gap.*
- void setHotstartSolution (const double ∗solution, const int ∗priorities=NULL)

  *Pass in target solution and optional priorities.*
- void setMinimumDrop (double value)

  *Set the minimum drop to continue cuts.*
- double getMinimumDrop () const

  *Get the minimum drop to continue cuts.*
- void setMaximumCutPassesAtRoot (int value)

  *Set the maximum number of cut passes at root node (default 20) Minimum drop can also be used for fine tuning.*
- int getMaximumCutPassesAtRoot () const

  *Get the maximum number of cut passes at root node.*
- void setMaximumCutPasses (int value)

  *Set the maximum number of cut passes at other nodes (default 10) Minimum drop can also be used for fine tuning.*
- int getMaximumCutPasses () const

  *Get the maximum number of cut passes at other nodes (default 10)*
- int getCurrentPassNumber () const

  *Get current cut pass number in this round of cuts.*
- void setCurrentPassNumber (int value)

  *Set current cut pass number in this round of cuts.*
- void setNumberStrong (int number)

  *Set the maximum number of candidates to be evaluated for strong branching.*
- int numberStrong () const

  *Get the maximum number of candidates to be evaluated for strong branching.*
- void setPreferredWay (int value)

*Set global preferred way to branch -1 down, +1 up, 0 no preference.*
- int getPreferredWay () const

  *Get the preferred way to branch (default 0)*
- int whenCuts () const

  *Get at which depths to do cuts.*
- void setWhenCuts (int value)

  *Set at which depths to do cuts.*
- bool doCutsNow (int allowForTopOfTree) const

  *Return true if we want to do cuts If allowForTopOfTree zero then just does on multiples of depth if 1 then allows for doing at top of tree if 2 then says if cuts allowed anywhere apart from root.*
- void setNumberBeforeTrust (int number)

  *Set the number of branches before pseudo costs believed in dynamic strong branching.*
- int numberBeforeTrust () const

  *get the number of branches before pseudo costs believed in dynamic strong branching.*
- void setNumberPenalties (int number)

  *Set the number of variables for which to compute penalties in dynamic strong branching.*
- int numberPenalties () const

  *get the number of variables for which to compute penalties in dynamic strong branching.*
- const CbcFullNodeInfo ∗ topOfTree () const

  *Pointer to top of tree.*
- void setNumberAnalyzeIterations (int number)

  *Number of analyze iterations to do.*
- int **numberAnalyzeIterations** () const
- double penaltyScaleFactor () const

  *Get scale factor to make penalties match strong.*
- void setPenaltyScaleFactor (double value)

  *Set scale factor to make penalties match strong.*
- void setProblemType (int number)

  *Problem type as set by user or found by analysis.*
- int **problemType** () const
- int currentDepth () const

  *Current depth.*
- void setHowOftenGlobalScan (int number)

  *Set how often to scan global cuts.*
- int howOftenGlobalScan () const

  *Get how often to scan global cuts.*
- int ∗ originalColumns () const

  *Original columns as created by integerPresolve or preprocessing.*
- void setOriginalColumns (const int ∗originalColumns, int numberGood=COIN_INT_MAX)

  *Set original columns as created by preprocessing.*
- **OsiRowCut** ∗ conflictCut (const **OsiSolverInterface** ∗solver, bool &localCuts)

  *Create conflict cut (well - most of)*
- void setPrintFrequency (int number)

  *Set the print frequency.*
- int printFrequency () const

  *Get the print frequency.*

### Methods returning info on how the solution process terminated

- bool isAbandoned () const

  *Are there a numerical difficulties?*
- bool isProvenOptimal () const

  *Is optimality proven?*

- bool isProvenInfeasible () const

  *Is infeasiblity proven (or none better than cutoff)?*
- bool isContinuousUnbounded () const

  *Was continuous solution unbounded.*
- bool isProvenDualInfeasible () const

  *Was continuous solution unbounded.*
- bool isNodeLimitReached () const

  *Node limit reached?*
- bool isSecondsLimitReached () const

  *Time limit reached?*
- bool isSolutionLimitReached () const

  *Solution limit reached?*
- int getIterationCount () const

  *Get how many iterations it took to solve the problem.*
- void incrementIterationCount (int value)

  *Increment how many iterations it took to solve the problem.*
- int getNodeCount () const

  *Get how many Nodes it took to solve the problem (including those in complete fathoming B&B inside CLP).*
- void incrementNodeCount (int value)

  *Increment how many nodes it took to solve the problem.*
- int getExtraNodeCount () const

  *Get how many Nodes were enumerated in complete fathoming B&B inside CLP.*
- int getFathomCount () const

  *Get how many times complete fathoming B&B was done.*
- int status () const

  *Final status of problem Some of these can be found out by is......*
- void **setProblemStatus** (int value)
- int secondaryStatus () const

  *Secondary status of problem -1 unset (status_ will also be -1) 0 search completed with solution 1 linear relaxation not feasible (or worse than cutoff) 2 stopped on gap 3 stopped on nodes 4 stopped on time 5 stopped on user event 6 stopped on solutions 7 linear relaxation unbounded 8 stopped on iteration limit.*
- void **setSecondaryStatus** (int value)
- bool isInitialSolveAbandoned () const

  *Are there numerical difficulties (for initialSolve) ?*
- bool isInitialSolveProvenOptimal () const

  *Is optimality proven (for initialSolve) ?*
- bool isInitialSolveProvenPrimalInfeasible () const

  *Is primal infeasiblity proven (for initialSolve) ?*
- bool isInitialSolveProvenDualInfeasible () const

  *Is dual infeasiblity proven (for initialSolve) ?*

### Problem information methods

*These methods call the solver's query routines to return information about the problem referred to by the current object.*

*Querying a problem that has no data associated with it result in zeros for the number of rows and columns, and NULL pointers from the methods that return vectors.*

*Const pointers returned from any data-query method are valid as long as the data is unchanged and the solver is not called.*

- int numberRowsAtContinuous () const

  *Number of rows in continuous (root) problem.*
- int getNumCols () const

*Get number of columns.*
- int getNumRows () const

  *Get number of rows.*
- CoinBigIndex getNumElements () const

  *Get number of nonzero elements.*
- int numberIntegers () const

  *Number of integers in problem.*
- const int ∗ **integerVariable** () const
- char integerType (int i) const

  *Whether or not integer.*
- const char ∗ integerType () const

  *Whether or not integer.*
- const double ∗ getColLower () const

  *Get pointer to array[getNumCols()] of column lower bounds.*
- const double ∗ getColUpper () const

  *Get pointer to array[getNumCols()] of column upper bounds.*
- const char ∗ getRowSense () const

  *Get pointer to array[getNumRows()] of row constraint senses.*
- const double ∗ getRightHandSide () const

  *Get pointer to array[getNumRows()] of rows right-hand sides.*
- const double ∗ getRowRange () const

  *Get pointer to array[getNumRows()] of row ranges.*
- const double ∗ getRowLower () const

  *Get pointer to array[getNumRows()] of row lower bounds.*
- const double ∗ getRowUpper () const

  *Get pointer to array[getNumRows()] of row upper bounds.*
- const double ∗ getObjCoefficients () const

  *Get pointer to array[getNumCols()] of objective function coefficients.*
- double getObjSense () const

  *Get objective function sense (1 for min (default), -1 for max)*
- bool isContinuous (int colIndex) const

  *Return true if variable is continuous.*
- bool isBinary (int colIndex) const

  *Return true if variable is binary.*
- bool isInteger (int colIndex) const

  *Return true if column is integer.*
- bool isIntegerNonBinary (int colIndex) const

  *Return true if variable is general integer.*
- bool isFreeBinary (int colIndex) const

  *Return true if variable is binary and not fixed at either bound.*
- const **CoinPackedMatrix** ∗ getMatrixByRow () const

  *Get pointer to row-wise copy of matrix.*
- const **CoinPackedMatrix** ∗ getMatrixByCol () const

  *Get pointer to column-wise copy of matrix.*
- double getInfinity () const

  *Get solver's value for infinity.*
- const double ∗ getCbcColLower () const

  *Get pointer to array[getNumCols()] (for speed) of column lower bounds.*
- const double ∗ getCbcColUpper () const

  *Get pointer to array[getNumCols()] (for speed) of column upper bounds.*
- const double ∗ getCbcRowLower () const

  *Get pointer to array[getNumRows()] (for speed) of row lower bounds.*

- const double ∗ getCbcRowUpper () const

  *Get pointer to array[getNumRows()] (for speed) of row upper bounds.*
- const double ∗ getCbcColSolution () const

  *Get pointer to array[getNumCols()] (for speed) of primal solution vector.*
- const double ∗ getCbcRowPrice () const

  *Get pointer to array[getNumRows()] (for speed) of dual prices.*
- const double ∗ getCbcReducedCost () const

  *Get a pointer to array[getNumCols()] (for speed) of reduced costs.*
- const double ∗ getCbcRowActivity () const

  *Get pointer to array[getNumRows()] (for speed) of row activity levels.*

#### Methods related to querying the solution

- double ∗ continuousSolution () const

  *Holds solution at continuous (after cuts if branchAndBound called)*
- int ∗ usedInSolution () const

  *Array marked whenever a solution is found if non-zero.*
- void incrementUsed (const double ∗solution)

  *Increases usedInSolution for nonzeros.*
- void setBestSolution (CBC_Message how, double &objectiveValue, const double ∗solution, int fixVariables=0)

  *Record a new incumbent solution and update objectiveValue.*
- void setBestObjectiveValue (double objectiveValue)

  *Just update objectiveValue.*
- CbcEventHandler::CbcAction dealWithEventHandler (CbcEventHandler::CbcEvent event, double objValue, const double ∗solution)

  *Deals with event handler and solution.*
- virtual double checkSolution (double cutoff, double ∗solution, int fixVariables, double originalObjValue)

  *Call this to really test if a valid solution can be feasible Solution is number columns in size.*
- bool feasibleSolution (int &numberIntegerInfeasibilities, int &numberObjectInfeasibilities) const

  *Test the current solution for feasiblility.*
- double ∗ currentSolution () const

  *Solution to the most recent lp relaxation.*
- const double ∗ testSolution () const

  *For testing infeasibilities - will point to currentSolution_ or solver–>getColSolution()*
- void **setTestSolution** (const double ∗solution)
- void reserveCurrentSolution (const double ∗solution=NULL)

  *Make sure region there and optionally copy solution.*
- const double ∗ getColSolution () const

  *Get pointer to array[getNumCols()] of primal solution vector.*
- const double ∗ getRowPrice () const

  *Get pointer to array[getNumRows()] of dual prices.*
- const double ∗ getReducedCost () const

  *Get a pointer to array[getNumCols()] of reduced costs.*
- const double ∗ getRowActivity () const

  *Get pointer to array[getNumRows()] of row activity levels.*
- double getCurrentObjValue () const

  *Get current objective function value.*
- double getCurrentMinimizationObjValue () const

  *Get current minimization objective function value.*
- double getMinimizationObjValue () const

  *Get best objective function value as minimization.*
- void setMinimizationObjValue (double value)

*Set best objective function value as minimization.*
- double getObjValue () const
  *Get best objective function value.*
- double getBestPossibleObjValue () const
  *Get best possible objective function value.*
- void setObjValue (double value)
  *Set best objective function value.*
- double getSolverObjValue () const
  *Get solver objective function value (as minimization)*
- double ∗ bestSolution () const
  *The best solution to the integer programming problem.*
- void setBestSolution (const double ∗solution, int numberColumns, double objectiveValue, bool check=false)
  *User callable setBestSolution.*
- int getSolutionCount () const
  *Get number of solutions.*
- void setSolutionCount (int value)
  *Set number of solutions (so heuristics will be different)*
- int numberSavedSolutions () const
  *Number of saved solutions (including best)*
- int maximumSavedSolutions () const
  *Maximum number of extra saved solutions.*
- void setMaximumSavedSolutions (int value)
  *Set maximum number of extra saved solutions.*
- const double ∗ savedSolution (int which) const
  *Return a saved solution (0==best) - NULL if off end.*
- double savedSolutionObjective (int which) const
  *Return a saved solution objective (0==best) - COIN_DBL_MAX if off end.*
- void deleteSavedSolution (int which)
  *Delete a saved solution and move others up.*
- int phase () const
  *Current phase (so heuristics etc etc can find out).*
- int getNumberHeuristicSolutions () const
  *Get number of heuristic solutions.*
- void setNumberHeuristicSolutions (int value)
  *Set number of heuristic solutions.*
- void setObjSense (double s)
  *Set objective function sense (1 for min (default), -1 for max,)*
- double getContinuousObjective () const
  *Value of objective at continuous.*
- void **setContinuousObjective** (double value)
- int getContinuousInfeasibilities () const
  *Number of infeasibilities at continuous.*
- void **setContinuousInfeasibilities** (int value)
- double rootObjectiveAfterCuts () const
  *Value of objective after root node cuts added.*
- double sumChangeObjective () const
  *Sum of Changes to objective by first solve.*
- int numberGlobalViolations () const
  *Number of times global cuts violated.*
- void **clearNumberGlobalViolations** ()
- bool resolveAfterTakeOffCuts () const
  *Whether to force a resolve after takeOffCuts.*

- void **setResolveAfterTakeOffCuts** (bool yesNo)
- int maximumRows () const

  *Maximum number of rows.*
- **CoinWarmStartBasis** & workingBasis ()

  *Work basis for temporary use.*
- int getStopNumberIterations () const

  *Get number of "iterations" to stop after.*
- void setStopNumberIterations (int value)

  *Set number of "iterations" to stop after.*
- CbcModel ∗ heuristicModel () const

  *A pointer to model from CbcHeuristic.*
- void setHeuristicModel (CbcModel ∗model)

  *Set a pointer to model from CbcHeuristic.*

### Node selection

- CbcCompareBase ∗ **nodeComparison** () const
- void **setNodeComparison** (CbcCompareBase ∗compare)
- void **setNodeComparison** (CbcCompareBase &compare)

### Problem feasibility checking

- CbcFeasibilityBase ∗ **problemFeasibility** () const
- void **setProblemFeasibility** (CbcFeasibilityBase ∗feasibility)
- void **setProblemFeasibility** (CbcFeasibilityBase &feasibility)

### Tree methods and subtree methods

- CbcTree ∗ tree () const

  *Tree method e.g. heap (which may be overridden by inheritance)*
- void passInTreeHandler (CbcTree &tree)

  *For modifying tree handling (original is cloned)*
- void passInSubTreeModel (CbcModel &model)

  *For passing in an CbcModel to do a sub Tree (with derived tree handlers).*
- CbcModel ∗ subTreeModel (**OsiSolverInterface** ∗solver=NULL) const

  *For retrieving a copy of subtree model with given OsiSolver.*
- int numberStoppedSubTrees () const

  *Returns number of times any subtree stopped on nodes, time etc.*
- void incrementSubTreeStopped ()

  *Says a sub tree was stopped.*
- int typePresolve () const

  *Whether to automatically do presolve before branch and bound (subTrees).*
- void **setTypePresolve** (int value)

### Branching Decisions

*See the CbcBranchDecision class for additional information.*

- CbcBranchDecision ∗ branchingMethod () const

  *Get the current branching decision method.*
- void setBranchingMethod (CbcBranchDecision ∗method)

  *Set the branching decision method.*
- void setBranchingMethod (CbcBranchDecision &method)

  *Set the branching method.*

- CbcCutModifier ∗ cutModifier () const

  *Get the current cut modifier method.*
- void setCutModifier (CbcCutModifier ∗modifier)

  *Set the cut modifier method.*
- void setCutModifier (CbcCutModifier &modifier)

  *Set the cut modifier method.*

### Row (constraint) and Column (variable) cut generation

- int stateOfSearch () const

  *State of search 0 - no solution 1 - only heuristic solutions 2 - branched to a solution 3 - no solution but many nodes.*
- void **setStateOfSearch** (int state)
- int searchStrategy () const

  *Strategy worked out - mainly at root node for use by CbcNode.*
- void setSearchStrategy (int value)

  *Set strategy worked out - mainly at root node for use by CbcNode.*
- int strongStrategy () const

  *Stong branching strategy.*
- void setStrongStrategy (int value)

  *Set strong branching strategy.*
- int numberCutGenerators () const

  *Get the number of cut generators.*
- CbcCutGenerator ∗∗ cutGenerators () const

  *Get the list of cut generators.*
- CbcCutGenerator ∗ cutGenerator (int i) const

  *Get the specified cut generator.*
- CbcCutGenerator ∗ virginCutGenerator (int i) const

  *Get the specified cut generator before any changes.*
- void addCutGenerator (**CglCutGenerator** ∗generator, int howOften=1, const char ∗name=NULL, bool normal=true, bool atSolution=false, bool infeasible=false, int howOftenInSub=-100, int whatDepth=-1, int whatDepthInSub=-1)

  *Add one generator - up to user to delete generators.*

### Strategy and sub models

*See the CbcStrategy class for additional information.*

- CbcStrategy ∗ strategy () const

  *Get the current strategy.*
- void setStrategy (CbcStrategy &strategy)

  *Set the strategy. Clones.*
- void setStrategy (CbcStrategy ∗strategy)

  *Set the strategy. assigns.*
- CbcModel ∗ parentModel () const

  *Get the current parent model.*
- void setParentModel (CbcModel &parentModel)

  *Set the parent model.*

### Heuristics and priorities

- void addHeuristic (CbcHeuristic ∗generator, const char ∗name=NULL, int before=-1)

  *Add one heuristic - up to user to delete.*
- CbcHeuristic ∗ heuristic (int i) const

*Get the specified heuristic.*
- int numberHeuristics () const
  *Get the number of heuristics.*
- void setNumberHeuristics (int value)
  *Set the number of heuristics.*
- CbcHeuristic ∗ lastHeuristic () const
  *Pointer to heuristic solver which found last solution (or NULL)*
- void setLastHeuristic (CbcHeuristic ∗last)
  *set last heuristic which found a solution*
- void passInPriorities (const int ∗priorities, bool ifNotSimpleIntegers)
  *Pass in branching priorities.*
- int priority (int sequence) const
  *Returns priority level for an object (or 1000 if no priorities exist)*
- void passInEventHandler (const CbcEventHandler ∗eventHandler)
  *Set an event handler.*
- CbcEventHandler ∗ getEventHandler () const
  *Retrieve a pointer to the event handler.*

### Setting/Accessing application data

- void setApplicationData (void ∗appData)
  *Set application data.*
- void ∗ getApplicationData () const
  *Get application data.*
- void passInSolverCharacteristics (**OsiBabSolver** ∗solverCharacteristics)
  *For advanced applications you may wish to modify the behavior of Cbc e.g.*
- const **OsiBabSolver** ∗ solverCharacteristics () const
  *Get solver characteristics.*

### Message handling etc

- void passInMessageHandler (**CoinMessageHandler** ∗handler)
  *Pass in Message handler (not deleted at end)*
- void newLanguage (**CoinMessages::Language** language)
  *Set language.*
- void **setLanguage** (**CoinMessages::Language** language)
- **CoinMessageHandler** ∗ messageHandler () const
  *Return handler.*
- **CoinMessages** & messages ()
  *Return messages.*
- **CoinMessages** ∗ messagesPointer ()
  *Return pointer to messages.*
- void setLogLevel (int value)
  *Set log level.*
- int logLevel () const
  *Get log level.*
- void setDefaultHandler (bool yesNo)
  *Set flag to say if handler_ is the default handler.*
- bool defaultHandler () const
  *Check default handler.*

### Specialized

- void setSpecialOptions (int value)

  *Set special options 0 bit (1) - check if cuts valid (if on debugger list) 1 bit (2) - use current basis to check integer solution (rather than all slack) 2 bit (4) - don't check integer solution (by solving LP) 3 bit (8) - fast analyze 4 bit (16) - non-linear model - so no well defined* **CoinPackedMatrix** *5 bit (32) - keep names 6 bit (64) - try for dominated columns 7 bit (128) - SOS type 1 but all declared integer 8 bit (256) - Set to say solution just found, unset by doing cuts 9 bit (512) - Try reduced model after 100 nodes 10 bit (1024) - Switch on some heuristics even if seems unlikely 11 bit (2048) - Mark as in small branch and bound 12 bit (4096) - Funny cuts so do slow way (in some places) 13 bit (8192) - Funny cuts so do slow way (in other places) 14 bit (16384) - Use Cplex! for fathoming 15 bit (32768) - Try reduced model after 0 nodes 16 bit (65536) - Original model had integer bounds 17 bit (131072) - Perturbation switched off 18 bit (262144) - donor CbcModel 19 bit (524288) - recipient CbcModel 20 bit (1048576) - waiting for sub model to return 22 bit (4194304) - do not initialize random seed in solver (user has) 23 bit (8388608) - leave solver_ with cuts 24 bit (16777216) - just get feasible if no cutoff.*
- int specialOptions () const

  *Get special options.*
- void setRandomSeed (int value)

  *Set random seed.*
- int getRandomSeed () const

  *Get random seed.*
- void setMultipleRootTries (int value)

  *Set multiple root tries.*
- int getMultipleRootTries () const

  *Get multiple root tries.*
- void sayEventHappened ()

  *Tell model to stop on event.*
- bool normalSolver () const

  *Says if normal solver i.e. has well defined* **CoinPackedMatrix**.
- bool waitingForMiniBranchAndBound () const

  *Says if model is sitting there waiting for mini branch and bound to finish This is because an event handler may only have access to parent model in mini branch and bound.*
- void setMoreSpecialOptions (int value)

  *Set more special options at present bottom 6 bits used for shadow price mode 1024 for experimental hotstart 2048,4096 breaking out of cuts 8192 slowly increase minimum drop 16384 gomory 32768 more heuristics in sub trees 65536 no cuts in preprocessing 131072 Time limits elapsed 18 bit (262144) - Perturb fathom nodes 19 bit (524288) - No limit on fathom nodes 20 bit (1048576) - Reduce sum of infeasibilities before cuts 21 bit (2097152) - Reduce sum of infeasibilities after cuts 22 bit (4194304) - Conflict analysis 23 bit (8388608) - Conflict analysis - temporary bit 24 bit (16777216) - Add cutoff as LP constraint (out) 25 bit (33554432) - diving/reordering 26 bit (67108864) - load global cuts from file 27 bit (134217728) - append binding global cuts to file 28 bit (268435456) - idiot branching 29 bit (536870912) - don't make fake objective 30 bit (1073741824) - Funny SOS or similar - be careful.*
- int moreSpecialOptions () const

  *Get more special options.*
- void setMoreSpecialOptions2 (int value)

  *Set more more special options 0 bit (1) - find switching variables 1 bit (2) - using fake objective until solution 2 bit (4) - switching variables exist 3 bit (8) - skip most of setBestSolution checks 4 bit (16) - very lightweight preprocessing in smallB&B 5 bit (32) - event handler needs to be cloned when parallel 6 bit (64) - testing - use probing to make cliques 7/8 bit (128) - try orbital branching (if nauty) 9 bit (512) - branching on objective (later) 10 bit (1024) - branching on constraints (later) 11/12 bit 2048 - intermittent cuts 13/14 bit 8192 - go to bitter end in strong branching (first time)*
- int moreSpecialOptions2 () const

  *Get more special options2.*
- void setCutoffAsConstraint (bool yesNo)

  *Set cutoff as constraint.*
- void setUseElapsedTime (bool yesNo)

  *Set time method.*
- bool useElapsedTime () const

  *Get time method.*

- void ∗ temporaryPointer () const

  *Get useful temporary pointer.*
- void setTemporaryPointer (void ∗pointer)

  *Set useful temporary pointer.*
- void goToDantzig (int numberNodes, **ClpDualRowPivot** ∗&savePivotMethod)

  *Go to dantzig pivot selection if easy problem (clp only)*
- bool ownObjects () const

  *Now we may not own objects - just point to solver's objects.*
- void checkModel ()

  *Check original model before it gets messed up.*

### Constructors and destructors etc

- CbcModel ()

  *Default Constructor.*
- CbcModel (const **OsiSolverInterface** &)

  *Constructor from solver.*
- void assignSolver (**OsiSolverInterface** ∗&solver, bool deleteSolver=true)

  *Assign a solver to the model (model assumes ownership)*
- void setModelOwnsSolver (bool ourSolver)

  *Set ownership of solver.*
- bool modelOwnsSolver ()

  *Get ownership of solver.*
- CbcModel (const CbcModel &rhs, bool cloneHandler=false)

  *Copy constructor .*
- virtual CbcModel ∗ clone (bool cloneHandler)

  *Clone.*
- CbcModel & operator= (const CbcModel &rhs)

  *Assignment operator.*
- virtual ∼CbcModel ()

  *Destructor.*
- **OsiSolverInterface** ∗ solver () const

  *Returns solver - has current state.*
- **OsiSolverInterface** ∗ swapSolver (**OsiSolverInterface** ∗solver)

  *Returns current solver - sets new one.*
- **OsiSolverInterface** ∗ continuousSolver () const

  *Returns solver with continuous state.*
- void createContinuousSolver ()

  *Create solver with continuous state.*
- void clearContinuousSolver ()

  *Clear solver with continuous state.*
- **OsiSolverInterface** ∗ referenceSolver () const

  *A copy of the solver, taken at constructor or by saveReferenceSolver.*
- void saveReferenceSolver ()

  *Save a copy of the current solver so can be reset to.*
- void resetToReferenceSolver ()

  *Uses a copy of reference solver to be current solver.*
- void gutsOfDestructor ()

  *Clears out as much as possible (except solver)*
- void gutsOfDestructor2 ()

  *Clears out enough to reset CbcModel as if no branch and bound done.*
- void resetModel ()

*Clears out enough to reset CbcModel cutoff etc.*
- void gutsOfCopy (const CbcModel &rhs, int mode=0)

  *Most of copy constructor mode - 0 copy but don't delete before 1 copy and delete before 2 copy and delete before (but use virgin generators)*
- void moveInfo (const CbcModel &rhs)

  *Move status, nodes etc etc across.*

### semi-private i.e. users should not use

- int getNodeCount2 () const

  *Get how many Nodes it took to solve the problem.*
- void setPointers (const **OsiSolverInterface** ∗solver)

  *Set pointers for speed.*
- int reducedCostFix ()

  *Perform reduced cost fixing.*
- void synchronizeHandlers (int makeDefault)

  *Makes all handlers same.*
- void saveExtraSolution (const double ∗solution, double objectiveValue)

  *Save a solution to saved list.*
- void saveBestSolution (const double ∗solution, double objectiveValue)

  *Save a solution to best and move current to saved.*
- void deleteSolutions ()

  *Delete best and saved solutions.*
- int resolve (**OsiSolverInterface** ∗solver)

  *Encapsulates solver resolve.*
- int chooseBranch (CbcNode ∗&newNode, int numberPassesLeft, CbcNode ∗oldNode, **OsiCuts** &cuts, bool &resolved, **CoinWarmStartBasis** ∗lastws, const double ∗lowerBefore, const double ∗upperBefore, **OsiSolverBranch** ∗&branches)

  *Encapsulates choosing a variable - anyAction -2, infeasible (-1 round again), 0 done.*
- int **chooseBranch** (CbcNode ∗newNode, int numberPassesLeft, bool &resolved)
- **CoinWarmStartBasis** ∗ getEmptyBasis (int ns=0, int na=0) const

  *Return an empty basis object of the specified size.*
- int takeOffCuts (**OsiCuts** &cuts, bool allowResolve, **OsiCuts** ∗saveCuts, int numberNewCuts=0, const **OsiRowCut** ∗∗newCuts=NULL)

  *Remove inactive cuts from the model.*
- int addCuts (CbcNode ∗node, **CoinWarmStartBasis** ∗&lastws)

  *Determine and install the active cuts that need to be added for the current subproblem.*
- bool addCuts1 (CbcNode ∗node, **CoinWarmStartBasis** ∗&lastws)

  *Traverse the tree from node to root and prep the model.*
- void previousBounds (CbcNode ∗node, CbcNodeInfo ∗where, int iColumn, double &lower, double &upper, int force)

  *Returns bounds just before where - initially original bounds.*
- void setObjectiveValue (CbcNode ∗thisNode, const CbcNode ∗parentNode) const

  *Set objective value in a node.*
- void convertToDynamic ()

  *If numberBeforeTrust >0 then we are going to use CbcBranchDynamic.*
- void synchronizeNumberBeforeTrust (int type=0)

  *Set numberBeforeTrust in all objects.*
- void zapIntegerInformation (bool leaveObjects=true)

  *Zap integer information in problem (may leave object info)*
- int cliquePseudoCosts (int doStatistics)

  *Use cliques for pseudocost information - return nonzero if infeasible.*
- void pseudoShadow (int type)

*Fill in useful estimates.*

- void fillPseudoCosts (double ∗downCosts, double ∗upCosts, int ∗priority=NULL, int ∗numberDown=NULL, int ∗numberUp=NULL, int ∗numberDownInfeasible=NULL, int ∗numberUpInfeasible=NULL) const

  *Return pseudo costs If not all integers or not pseudo costs - returns all zero Length of arrays are numberIntegers() and entries correspond to integerVariable()[i] User must allocate arrays before call.*
- void doHeuristicsAtRoot (int deleteHeuristicsAfterwards=0)

  *Do heuristics at root.*
- void adjustHeuristics ()

  *Adjust heuristics based on model.*
- const double ∗ hotstartSolution () const

  *Get the hotstart solution.*
- const int ∗ hotstartPriorities () const

  *Get the hotstart priorities.*
- CbcCountRowCut ∗∗ addedCuts () const

  *Return the list of cuts initially collected for this subproblem.*
- int currentNumberCuts () const

  *Number of entries in the list returned by addedCuts()*
- CbcRowCuts ∗ globalCuts ()

  *Global cuts.*
- void zapGlobalCuts ()

  *Get rid of global cuts.*
- void setNextRowCut (const **OsiRowCut** &**cut**)

  *Copy and set a pointer to a row cut which will be added instead of normal branching.*
- CbcNode ∗ currentNode () const

  *Get a pointer to current node (be careful)*
- **CglTreeProbingInfo** ∗ probingInfo () const

  *Get a pointer to probing info.*
- **CoinThreadRandom** ∗ randomNumberGenerator ()

  *Thread specific random number generator.*
- void setNumberStrongIterations (int number)

  *Set the number of iterations done in strong branching.*
- int numberStrongIterations () const

  *Get the number of iterations done in strong branching.*
- int maximumNumberIterations () const

  *Get maximum number of iterations (designed to be used in heuristics)*
- void setMaximumNumberIterations (int value)

  *Set maximum number of iterations (designed to be used in heuristics)*
- CbcSymmetry ∗ symmetryInfo () const

  *Symmetry information.*
- void setFastNodeDepth (int value)

  *Set depth for fast nodes.*
- int fastNodeDepth () const

  *Get depth for fast nodes.*
- int continuousPriority () const

  *Get anything with priority >= this can be treated as continuous.*
- void setContinuousPriority (int value)

  *Set anything with priority >= this can be treated as continuous.*
- void **incrementExtra** (int nodes, int iterations, int fathoms=1)
- void zeroExtra ()

  *Zero extra.*
- int numberExtraIterations () const

  *Number of extra iterations.*

- void incrementStrongInfo (int numberTimes, int numberIterations, int numberFixed, bool ifInfeasible)
  *Increment strong info.*
- const int ∗ strongInfo () const
  *Return strong info.*
- int ∗ mutableStrongInfo ()
  *Return mutable strong info.*
- **CglStored** ∗ storedRowCuts () const
  *Get stored row cuts for donor/recipient CbcModel.*
- void setStoredRowCuts (**CglStored** ∗cuts)
  *Set stored row cuts for donor/recipient CbcModel.*
- bool allDynamic () const
  *Says whether all dynamic integers.*
- void generateCpp (FILE ∗fp, int options)
  *Create C++ lines to get to current state.*
- **OsiBranchingInformation** usefulInformation () const
  *Generate an* **OsiBranchingInformation** *object.*
- void setBestSolutionBasis (const **CoinWarmStartBasis** &bestSolutionBasis)
  *Warm start object produced by heuristic or strong branching.*
- void redoWalkBack ()
  *Redo walkback arrays.*

## Solve methods

- void initialSolve ()
  *Solve the initial LP relaxation.*
- void branchAndBound (int doStatistics=0)
  *Invoke the branch & cut algorithm.*
- void addUpdateInformation (const CbcObjectUpdateData &data)
  *Adds an update information object.*
- int doOneNode (CbcModel ∗baseModel, CbcNode ∗&node, CbcNode ∗&newNode)
  *Do one node - broken out for clarity? also for parallel (when baseModel!=this) Returns 1 if solution found node NULL on return if no branches left newNode NULL if no new node created.*
- int resolve (CbcNodeInfo ∗parent, int whereFrom, double ∗saveSolution=NULL, double ∗saveLower=NULL, double ∗saveUpper=NULL)
  *Reoptimise an LP relaxation.*
- void makeGlobalCuts (int numberRows, const int ∗which)
  *Make given rows (L or G) into global cuts and remove from lp.*
- int makeGlobalCut (const **OsiRowCut** ∗**cut**)
  *Make given cut into a global cut.*
- int makeGlobalCut (const **OsiRowCut** &**cut**)
  *Make given cut into a global cut.*
- void makeGlobalCut (const **OsiColCut** ∗**cut**)
  *Make given column cut into a global cut.*
- void makeGlobalCut (const **OsiColCut** &**cut**)
  *Make given column cut into a global cut.*
- void makePartialCut (const **OsiRowCut** ∗**cut**, const **OsiSolverInterface** ∗solver=NULL)
  *Make partial cut into a global cut and save.*
- void makeGlobalCuts ()
  *Make partial cuts into global cuts.*
- const int ∗ whichGenerator () const
  *Which cut generator generated this cut.*

### Multithreading

- CbcThread ∗ masterThread () const

  *Get pointer to masterthread.*
- CbcNodeInfo ∗∗ walkback () const

  *Get pointer to walkback.*
- int getNumberThreads () const

  *Get number of threads.*
- void setNumberThreads (int value)

  *Set number of threads.*
- int getThreadMode () const

  *Get thread mode.*
- void setThreadMode (int value)

  *Set thread mode always use numberThreads for branching 1 set then deterministic 2 set then use numberThreads for root cuts 4 set then use numberThreads in root mini branch and bound 8 set and numberThreads - do heuristics numberThreads at a time 8 set and numberThreads==0 do all heuristics at once default is 0.*
- int parallelMode () const

  *Return -2 if deterministic threaded and main thread -1 if deterministic threaded and serial thread 0 if serial 1 if opportunistic threaded.*
- CbcBaseModel ∗ master () const

  *Thread stuff for master.*
- bool isLocked () const

  *From here to end of section - code in CbcThread.cpp until class changed Returns true if locked.*
- void **lockThread** ()
- void **unlockThread** ()
- void setInfoInChild (int type, CbcThread ∗info)

  *Set information in a child -3 pass pointer to child thread info -2 just stop -1 delete simple child stuff 0 delete opportunistic child stuff 1 delete deterministic child stuff.*
- void moveToModel (CbcModel ∗baseModel, int mode)

  *Move/copy information from one model to another -1 - initialization 0 - from base model 1 - to base model (and reset) 2 - add in final statistics etc (and reset so can do clean destruction)*
- int splitModel (int numberModels, CbcModel ∗∗model, int numberNodes)

  *Split up nodes.*
- void startSplitModel (int numberIterations)

  *Start threads.*
- void mergeModels (int numberModel, CbcModel ∗∗model, int numberNodes)

  *Merge models.*
- static bool haveMultiThreadSupport ()

  *Indicates whether Cbc library has been compiled with multithreading support.*

## 4.73.1 Detailed Description

Simple Branch and bound class.

The initialSolve() method solves the initial LP relaxation of the MIP problem. The branchAndBound() method can then be called to finish using a branch and cut algorithm.

#### Search Tree Traversal

Subproblems (aka nodes) requiring additional evaluation are stored using the CbcNode and CbcNodeInfo objects. Ancestry linkage is maintained in the CbcNodeInfo object. Evaluation of a subproblem within branchAndBound() proceeds as follows:

- The node representing the most promising parent subproblem is popped from the heap which holds the set of subproblems requiring further evaluation.
- Using branching instructions stored in the node, and information in its ancestors, the model and solver are adjusted to create the active subproblem.
- If the parent subproblem will require further evaluation (*i.e.*, there are branches remaining) its node is pushed back on the heap. Otherwise, the node is deleted. This may trigger recursive deletion of ancestors.
- The newly created subproblem is evaluated.
- If the subproblem requires further evaluation, a node is created. All information needed to recreate the subproblem (branching information, row and column cuts) is placed in the node and the node is added to the set of subproblems awaiting further evaluation.

Note that there is never a node representing the active subproblem; the model and solver represent the active subproblem.

#### Row (Constraint) Cut Handling

For a typical subproblem, the sequence of events is as follows:

- The subproblem is rebuilt for further evaluation: One result of a call to addCuts() is a traversal of ancestors, leaving a list of all cuts used in the ancestors in #addedCuts_. This list is then scanned to construct a basis that includes only tight cuts. Entries for loose cuts are set to NULL.
- The subproblem is evaluated: One result of a call to solveWithCuts() is the return of a set of newly generated cuts for the subproblem. #addedCuts_ is also kept up-to-date as old cuts become loose.
- The subproblem is stored for further processing: A call to CbcNodeInfo::addCuts() adds the newly generated cuts to the CbcNodeInfo object associated with this node.

See CbcCountRowCut for details of the bookkeeping associated with cut management.

Definition at line 101 of file CbcModel.hpp.

### 4.73.2 Member Enumeration Documentation

#### 4.73.2.1 enum CbcModel::CbcIntParam

**Enumerator**

***CbcMaxNumNode*** The maximum number of nodes before terminating.

***CbcMaxNumSol*** The maximum number of solutions before terminating.

***CbcFathomDiscipline*** Fathoming discipline. Controls objective function comparisons for purposes of fathoming by bound or determining monotonic variables.

If 1, action is taken only when the current objective is strictly worse than the target. Implementation is handled by adding a small tolerance to the target.

***CbcPrinting*** Adjusts printout 1 does different node message with number unsatisfied on last branch.

***CbcNumberBranches*** Number of branches (may be more than number of nodes as may include strong branching)

***CbcLastIntParam*** Just a marker, so that a static sized array can store parameters.

Definition at line 105 of file CbcModel.hpp.

#### 4.73.2.2 enum CbcModel::CbcDblParam

**Enumerator**

***CbcIntegerTolerance*** The maximum amount the value of an integer variable can vary from integer and still be considered feasible.

***CbcInfeasibilityWeight*** The objective is assumed to worsen by this amount for each integer infeasibility.

***CbcCutoffIncrement*** The amount by which to tighten the objective function cutoff when a new solution is discovered.

***CbcAllowableGap*** Stop when the gap between the objective value of the best known solution and the best bound on the objective of any solution is less than this. This is an absolute value. Conversion from a percentage is left to the client.

***CbcAllowableFractionGap*** Stop when the gap between the objective value of the best known solution and the best bound on the objective of any solution is less than this fraction of of the absolute value of best known solution. Code stops if either this test or CbcAllowableGap test succeeds

***CbcMaximumSeconds*** The maximum number of seconds before terminating. A double should be adequate!

***CbcCurrentCutoff*** Cutoff - stored for speed.

***CbcOptimizationDirection*** Optimization direction - stored for speed.

***CbcCurrentObjectiveValue*** Current objective value.

***CbcCurrentMinimizationObjectiveValue*** Current minimization objective value.

***CbcStartSeconds*** The time at start of model. So that other pieces of code can access

***CbcHeuristicGap*** Stop doing heuristics when the gap between the objective value of the best known solution and the best bound on the objective of any solution is less than this. This is an absolute value. Conversion from a percentage is left to the client.

***CbcHeuristicFractionGap*** Stop doing heuristics when the gap between the objective value of the best known solution and the best bound on the objective of any solution is less than this fraction of of the absolute value of best known solution. Code stops if either this test or CbcAllowableGap test succeeds

***CbcSmallestChange*** Smallest non-zero change on a branch.

***CbcSumChange*** Sum of non-zero changes on a branch.

***CbcLargestChange*** Largest non-zero change on a branch.

***CbcSmallChange*** Small non-zero change on a branch to be used as guess.

***CbcLastDblParam*** Just a marker, so that a static sized array can store parameters.

Definition at line 131 of file CbcModel.hpp.

### 4.73.3 Constructor & Destructor Documentation

#### 4.73.3.1 CbcModel::CbcModel ( const CbcModel & *rhs*, bool *cloneHandler* = `false` )

Copy constructor .

If cloneHandler is true then message handler is cloned

### 4.73.4 Member Function Documentation

#### 4.73.4.1 void CbcModel::initialSolve ( )

Solve the initial LP relaxation.

Invoke the solver's initialSolve() method.

#### 4.73.4.2 void CbcModel::branchAndBound ( int *doStatistics* = 0 )

Invoke the branch & cut algorithm.

The method assumes that initialSolve() has been called to solve the LP relaxation. It processes the root node, then proceeds to explore the branch & cut search tree. The search ends when the tree is exhausted or one of several execution limits is reached. If doStatistics is 1 summary statistics are printed if 2 then also the path to best solution (if found by branching) if 3 then also one line per node

#### 4.73.4.3 int CbcModel::resolve ( CbcNodeInfo * *parent*, int *whereFrom*, double * *saveSolution* = NULL, double * *saveLower* = NULL, double * *saveUpper* = NULL )

Reoptimise an LP relaxation.

Invoke the solver's resolve() method. whereFrom - 0 - initial continuous 1 - resolve on branch (before new cuts) 2 - after new cuts 3 - obsolete code or something modified problem in unexpected way 10 - after strong branching has fixed variables at root 11 - after strong branching has fixed variables in tree

returns 1 feasible, 0 infeasible, -1 feasible but skip cuts

#### 4.73.4.4 CbcModel* CbcModel::findCliques ( bool *makeEquality*, int *atLeastThisMany*, int *lessThanThis*, int *defaultValue* = 1000 )

Identify cliques and construct corresponding objects.

Find cliques with size in the range [atLeastThisMany, lessThanThis] and construct corresponding CbcClique objects. If makeEquality is true then a new model may be returned if modifications had to be made, otherwise this is returned. If the problem is infeasible #numberObjects_ is set to -1. A client must use deleteObjects() before a second call to findCliques(). If priorities exist, clique priority is set to the default.

#### 4.73.4.5 CbcModel* CbcModel::integerPresolve ( bool *weak* = false )

Do integer presolve, creating a new (presolved) model.

Returns the new model, or NULL if feasibility is lost. If weak is true then just does a normal presolve

#### 4.73.4.6 bool CbcModel::integerPresolveThisModel ( OsiSolverInterface * *originalSolver*, bool *weak* = false )

Do integer presolve, modifying the current model.

Returns true if the model remains feasible after presolve.

#### 4.73.4.7 bool CbcModel::tightenVubs ( int *type*, bool *allowMultipleBinary* = false, double *useCutoff* = 1.0e50 )

For variables involved in VUB constraints, see if we can tighten bounds by solving lp's.

Returns false if feasibility is lost. If **CglProbing** is available, it will be tried as well to see if it can tighten bounds. This routine is just a front end for tightenVubs(int,const int*,double).

If `type = -1` all variables are processed (could be very slow). If `type = 0` only variables involved in VUBs are processed. If `type = n > 0`, only the n most expensive VUB variables are processed, where it is assumed that x is at its maximum so delta would have to go to 1 (if x not at bound).

If `allowMultipleBinary` is true, then a VUB constraint is a row with one continuous variable and any number of binary variables.

If `useCutoff < 1.0e30`, the original objective is installed as a constraint with `useCutoff` as a bound.

#### 4.73.4.8 bool CbcModel::tightenVubs ( int *numberVubs*, const int * *which*, double *useCutoff* = `1.0e50` )

For variables involved in VUB constraints, see if we can tighten bounds by solving lp's.

This version is just handed a list of variables to be processed.

#### 4.73.4.9 void CbcModel::addObjects ( int *numberObjects*, OsiObject ** *objects* )

Add in object information.

Objects are cloned; the owner can delete the originals.

#### 4.73.4.10 void CbcModel::addObjects ( int *numberObjects*, CbcObject ** *objects* )

Add in object information.

Objects are cloned; the owner can delete the originals.

#### 4.73.4.11 void CbcModel::findIntegers ( bool *startAgain*, int *type* = `0` )

Identify integer variables and create corresponding objects.

Record integer variables and create an CbcSimpleInteger object for each one. If `startAgain` is true, a new scan is forced, overwriting any existing integer variable information. If type > 0 then 1==PseudoCost, 2 new ones low priority

#### 4.73.4.12 void CbcModel::setCutoff ( double *value* )

Set cutoff bound on the objective function.

When using strict comparison, the bound is adjusted by a tolerance to avoid accidentally cutting off the optimal solution.

#### 4.73.4.13 void CbcModel::setHotstartSolution ( const double * *solution*, const int * *priorities* = `NULL` )

Pass in target solution and optional priorities.

If priorities then >0 means only branch if incorrect while <0 means branch even if correct. +1 or -1 are highest priority

#### 4.73.4.14 int CbcModel::getCurrentPassNumber ( ) const [inline]

Get current cut pass number in this round of cuts.

(1 is first pass)

Definition at line 767 of file CbcModel.hpp.

#### 4.73.4.15 void CbcModel::setCurrentPassNumber ( int *value* ) [inline]

Set current cut pass number in this round of cuts.

(1 is first pass)

Definition at line 772 of file CbcModel.hpp.

#### 4.73.4.16 void CbcModel::setNumberStrong ( int *number* )

Set the maximum number of candidates to be evaluated for strong branching.

A value of 0 disables strong branching.

#### 4.73.4.17 void CbcModel::setNumberBeforeTrust ( int *number* )

Set the number of branches before pseudo costs believed in dynamic strong branching.

A value of 0 disables dynamic strong branching.

#### 4.73.4.18 int CbcModel::numberBeforeTrust ( ) const [inline]

get the number of branches before pseudo costs believed in dynamic strong branching.

Definition at line 820 of file CbcModel.hpp.

#### 4.73.4.19 void CbcModel::setNumberPenalties ( int *number* )

Set the number of variables for which to compute penalties in dynamic strong branching.

A value of 0 disables penalties.

#### 4.73.4.20 int CbcModel::numberPenalties ( ) const [inline]

get the number of variables for which to compute penalties in dynamic strong branching.

Definition at line 831 of file CbcModel.hpp.

#### 4.73.4.21 double CbcModel::penaltyScaleFactor ( ) const [inline]

Get scale factor to make penalties match strong.

Should/will be computed

Definition at line 846 of file CbcModel.hpp.

#### 4.73.4.22 void CbcModel::setPenaltyScaleFactor ( double *value* )

Set scale factor to make penalties match strong.

Should/will be computed

#### 4.73.4.23 void CbcModel::setProblemType ( int *number* ) [inline]

Problem type as set by user or found by analysis.

This will be extended 0 - not known 1 - Set partitioning <= 2 - Set partitioning == 3 - Set covering 4 - all +- 1 or all +1 and odd

Definition at line 859 of file CbcModel.hpp.

#### 4.73.4.24 void CbcModel::setPrintFrequency ( int *number* ) [inline]

Set the print frequency.

Controls the number of nodes evaluated between status prints. If `number <=0` the print frequency is set to 100 nodes for large problems, 1000 for small problems. Print frequency has very slight overhead if small.

Definition at line 893 of file CbcModel.hpp.

#### 4.73.4.25 int CbcModel::status ( ) const [inline]

Final status of problem Some of these can be found out by is......

functions -1 before branchAndBound 0 finished - check isProvenOptimal or isProvenInfeasible to see if solution found (or check value of best solution) 1 stopped - on maxnodes, maxsols, maxtime 2 difficulties so run was abandoned (5 event user programmed event occurred)

Definition at line 954 of file CbcModel.hpp.

#### 4.73.4.26 const char* CbcModel::getRowSense ( ) const [inline]

Get pointer to array[getNumRows()] of row constraint senses.

- 'L': <= constraint
- 'E': = constraint
- 'G': >= constraint
- 'R': ranged constraint
- 'N': free constraint

Definition at line 1060 of file CbcModel.hpp.

#### 4.73.4.27 const double* CbcModel::getRightHandSide ( ) const [inline]

Get pointer to array[getNumRows()] of rows right-hand sides.

- if rowsense()[i] == 'L' then rhs()[i] == rowupper()[i]
- if rowsense()[i] == 'G' then rhs()[i] == rowlower()[i]
- if rowsense()[i] == 'R' then rhs()[i] == rowupper()[i]

- if rowsense()[i] == 'N' then rhs()[i] == 0.0

Definition at line 1072 of file CbcModel.hpp.

#### 4.73.4.28 const double* CbcModel::getRowRange ( ) const [inline]

Get pointer to array[getNumRows()] of row ranges.

- if rowsense()[i] == 'R' then rowrange()[i] == rowupper()[i] - rowlower()[i]
- if rowsense()[i] != 'R' then rowrange()[i] is 0.0

Definition at line 1084 of file CbcModel.hpp.

#### 4.73.4.29 bool CbcModel::isInteger ( int *colIndex* ) const [inline]

Return true if column is integer.

Note: This function returns true if the the column is binary or a general integer.

Definition at line 1123 of file CbcModel.hpp.

#### 4.73.4.30 int* CbcModel::usedInSolution ( ) const [inline]

Array marked whenever a solution is found if non-zero.

Code marks if heuristic returns better so heuristic need only mark if it wants to on solutions which are worse than current

Definition at line 1196 of file CbcModel.hpp.

#### 4.73.4.31 virtual double CbcModel::checkSolution ( double *cutoff,* double * *solution,* int *fixVariables,* double *originalObjValue* ) [virtual]

Call this to really test if a valid solution can be feasible Solution is number columns in size.

If fixVariables true then bounds of continuous solver updated. Returns objective value (worse than cutoff if not feasible) Previously computed objective value is now passed in (in case user does not do solve) virtual so user can override

#### 4.73.4.32 bool CbcModel::feasibleSolution ( int & *numberIntegerInfeasibilities,* int & *numberObjectInfeasibilities* ) const

Test the current solution for feasiblility.

Scan all objects for indications of infeasibility. This is broken down into simple integer infeasibility (`numberIntegerInfeasibilities`) and all other reports of infeasibility (`numberObjectInfeasibilities`).

#### 4.73.4.33 double* CbcModel::currentSolution ( ) const [inline]

Solution to the most recent lp relaxation.

The solver's solution to the most recent lp relaxation.

Definition at line 1235 of file CbcModel.hpp.

#### 4.73.4.34 double CbcModel::getBestPossibleObjValue ( ) const

Get best possible objective function value.

This is better of best possible left on tree and best solution found. If called from within branch and cut may be optimistic.

#### 4.73.4.35 double* CbcModel::bestSolution ( ) const [inline]

The best solution to the integer programming problem.

The best solution to the integer programming problem found during the search. If no solution is found, the method returns null.

Definition at line 1313 of file CbcModel.hpp.

#### 4.73.4.36 void CbcModel::setBestSolution ( const double * *solution*, int *numberColumns*, double *objectiveValue*, bool *check* = false )

User callable setBestSolution.

If check false does not check valid If true then sees if feasible and warns if objective value worse than given (so just set to COIN_DBL_MAX if you don't care). If check true then does not save solution if not feasible

#### 4.73.4.37 int CbcModel::phase ( ) const [inline]

Current phase (so heuristics etc etc can find out).

0 - initial solve 1 - solve with cuts at root 2 - solve with cuts 3 - other e.g. strong branching 4 - trying to validate a solution 5 - at end of search

Definition at line 1357 of file CbcModel.hpp.

#### 4.73.4.38 int CbcModel::numberGlobalViolations ( ) const [inline]

Number of times global cuts violated.

When global cut pool then this should be kept for each cut and type of cut

Definition at line 1400 of file CbcModel.hpp.

#### 4.73.4.39 void CbcModel::passInSubTreeModel ( CbcModel & *model* )

For passing in an CbcModel to do a sub Tree (with derived tree handlers).

Passed in model must exist for duration of branch and bound

#### 4.73.4.40 CbcModel* CbcModel::subTreeModel ( OsiSolverInterface * *solver* = NULL ) const

For retrieving a copy of subtree model with given OsiSolver.

If no subtree model will use self (up to user to reset cutoff etc). If solver NULL uses current

#### 4.73.4.41 int CbcModel::typePresolve ( ) const [inline]

Whether to automatically do presolve before branch and bound (subTrees).

0 - no 1 - ordinary presolve 2 - integer presolve (dodgy)

Definition at line 1487 of file CbcModel.hpp.

#### 4.73.4.42 void CbcModel::setBranchingMethod ( CbcBranchDecision & *method* ) [inline]

Set the branching method.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Definition at line 1515 of file CbcModel.hpp.

#### 4.73.4.43 void CbcModel::setCutModifier ( CbcCutModifier & *modifier* )

Set the cut modifier method.

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

#### 4.73.4.44 void CbcModel::addCutGenerator ( CglCutGenerator * *generator*, int *howOften* = 1, const char * *name* = NULL, bool *normal* = true, bool *atSolution* = false, bool *infeasible* = false, int *howOftenInSub* = -100, int *whatDepth* = -1, int *whatDepthInSub* = -1 )

Add one generator - up to user to delete generators.

howoften affects how generator is used. 0 or 1 means always, >1 means every that number of nodes. Negative values have same meaning as positive but they may be switched off (-> -100) by code if not many cuts generated at continuous. -99 is just done at root. Name is just for printout. If depth >0 overrides how often generator is called (if howOften==-1 or >0).

#### 4.73.4.45 void CbcModel::addHeuristic ( CbcHeuristic * *generator*, const char * *name* = NULL, int *before* = -1 )

Add one heuristic - up to user to delete.

The name is just used for print messages.

#### 4.73.4.46 void CbcModel::passInPriorities ( const int * *priorities*, bool *ifNotSimpleIntegers* )

Pass in branching priorities.

If ifClique then priorities are on cliques otherwise priorities are on integer variables. Other type (if exists set to default) 1 is highest priority. (well actually -INT_MAX is but that's ugly) If hotstart > 0 then branches are created to force the variable to the value given by best solution. This enables a sort of hot start. The node choice should be greatest depth and hotstart should normally be switched off after a solution.

If ifNotSimpleIntegers true then appended to normal integers

This is now deprecated except for simple usage. If user creates Cbcobjects then set priority in them

#### 4.73.4.47 void CbcModel::passInEventHandler ( const CbcEventHandler * *eventHandler* )

Set an event handler.

A clone of the handler passed as a parameter is stored in CbcModel.

#### 4.73.4.48 void CbcModel::setApplicationData ( void * *appData* )

Set application data.

This is a pointer that the application can store into and retrieve from the solver interface. This field is available for the application to optionally define and use.

#### 4.73.4.49 void CbcModel::passInSolverCharacteristics ( OsiBabSolver * *solverCharacteristics* )

For advanced applications you may wish to modify the behavior of Cbc e.g.

if the solver is a NLP solver then you may not have an exact optimum solution at each step. Information could be built into **OsiSolverInterface** but this is an alternative so that that interface does not have to be changed. If something similar is useful to enough solvers then it could be migrated You can also pass in by using solver->setAuxiliaryInfo. You should do that if solver is odd - if solver is normal simplex then use this. NOTE - characteristics are not cloned

#### 4.73.4.50 void CbcModel::setDefaultHandler ( bool *yesNo* ) `[inline]`

Set flag to say if handler_ is the default handler.

The default handler is deleted when the model is deleted. Other handlers (supplied by the client) will not be deleted.

Definition at line 1754 of file CbcModel.hpp.

#### 4.73.4.51 void CbcModel::assignSolver ( OsiSolverInterface *& *solver,* bool *deleteSolver* = `true` )

Assign a solver to the model (model assumes ownership)

On return, `solver` will be NULL. If deleteSolver then current solver deleted (if model owned)

**Note**

Parameter settings in the outgoing solver are not inherited by the incoming solver.

#### 4.73.4.52 void CbcModel::setModelOwnsSolver ( bool *ourSolver* ) `[inline]`

Set ownership of solver.

A parameter of false tells CbcModel it does not own the solver and should not delete it. Once you claim ownership of the solver, you're responsible for eventually deleting it. Note that CbcModel clones solvers with abandon. Unless you have a deep understanding of the workings of CbcModel, the only time you want to claim ownership is when you're about to delete the CbcModel object but want the solver to continue to exist (as, for example, when branchAndBound has finished and you want to hang on to the answer).

Definition at line 1942 of file CbcModel.hpp.

#### 4.73.4.53 bool CbcModel::modelOwnsSolver ( ) [inline]

Get ownership of solver.

A return value of true means that CbcModel owns the solver and will take responsibility for deleting it when that becomes necessary.

Definition at line 1951 of file CbcModel.hpp.

#### 4.73.4.54 void CbcModel::resetToReferenceSolver ( )

Uses a copy of reference solver to be current solver.

Because of possible mismatches all exotic integer information is loat (apart from normal information in **OsiSolverInterface**) so SOS etc and priorities will have to be redone

#### 4.73.4.55 int CbcModel::reducedCostFix ( )

Perform reduced cost fixing.

Fixes integer variables at their current value based on reduced cost penalties. Returns number fixed

#### 4.73.4.56 void CbcModel::synchronizeHandlers ( int *makeDefault* )

Makes all handlers same.

If makeDefault 1 then makes top level default and rest point to that. If 2 then each is copy

#### 4.73.4.57 CoinWarmStartBasis∗ CbcModel::getEmptyBasis ( int *ns* = 0, int *na* = 0 ) const

Return an empty basis object of the specified size.

A useful utility when constructing a basis for a subproblem from scratch. The object returned will be of the requested capacity and appropriate for the solver attached to the model.

#### 4.73.4.58 int CbcModel::takeOffCuts ( OsiCuts & *cuts*, bool *allowResolve*, OsiCuts ∗ *saveCuts*, int *numberNewCuts* = 0, const OsiRowCut ∗∗ *newCuts* = NULL )

Remove inactive cuts from the model.

An **OsiSolverInterface** is expected to maintain a valid basis, but not a valid solution, when loose cuts are deleted. Restoring a valid solution requires calling the solver to reoptimise. If it's certain the solution will not be required, set allowResolve to false to suppress reoptimisation. If saveCuts then slack cuts will be saved On input current cuts are cuts and newCuts on exit current cuts will be correct. Returns number dropped

#### 4.73.4.59 int CbcModel::addCuts ( CbcNode ∗ *node*, CoinWarmStartBasis ∗& *lastws* )

Determine and install the active cuts that need to be added for the current subproblem.

The whole truth is a bit more complicated. The first action is a call to addCuts1(). addCuts() then sorts through the list, installs the tight cuts in the model, and does bookkeeping (adjusts reference counts). The basis returned from addCuts1() is adjusted accordingly.

If it turns out that the node should really be fathomed by bound, addCuts() simply treats all the cuts as loose as it does the bookkeeping.

#### 4.73.4.60 bool CbcModel::addCuts1 ( CbcNode * *node*, CoinWarmStartBasis *& *lastws* )

Traverse the tree from node to root and prep the model.

addCuts1() begins the job of prepping the model to match the current subproblem. The model is stripped of all cuts, and the search tree is traversed from node to root to determine the changes required. Appropriate bounds changes are installed, a list of cuts is collected but not installed, and an appropriate basis (minus the cuts, but big enough to accommodate them) is constructed.

Returns true if new problem similar to old

#### 4.73.4.61 void CbcModel::previousBounds ( CbcNode * *node*, CbcNodeInfo * *where*, int *iColumn*, double & *lower*, double & *upper*, int *force* )

Returns bounds just before where - initially original bounds.

Also sets downstream nodes (lower if force 1, upper if 2)

#### 4.73.4.62 void CbcModel::setObjectiveValue ( CbcNode * *thisNode*, const CbcNode * *parentNode* ) const

Set objective value in a node.

This is separated out so that odd solvers can use. It may look at extra information in solverCharacteriscs_ and will also use bound from parent node

#### 4.73.4.63 void CbcModel::convertToDynamic ( )

If numberBeforeTrust >0 then we are going to use CbcBranchDynamic.

Scan and convert CbcSimpleInteger objects

#### 4.73.4.64 void CbcModel::doHeuristicsAtRoot ( int *deleteHeuristicsAfterwards* = 0 )

Do heuristics at root.

0 - don't delete 1 - delete 2 - just delete - don't even use

#### 4.73.4.65 void CbcModel::setBestSolutionBasis ( const CoinWarmStartBasis & *bestSolutionBasis* ) [inline]

Warm start object produced by heuristic or strong branching.

If get a valid integer solution outside branch and bound then it can take a reasonable time to solve LP which produces clean solution. If this object has any size then it will be used in solve.

Definition at line 2390 of file CbcModel.hpp.

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

- CbcModel.hpp

## 4.74 CbcNauty Class Reference

### Public Member Functions

- std::vector< std::vector< int > > * getOrbits () const

  *Returns the orbits in a "convenient" form.*
- void setWriteAutoms (const std::string &afilename)

  *Methods to classify orbits.*

### Constructors and destructors

- CbcNauty (int n, const size_t *v, const int *d, const int *e)

  *Normal constructor (if dense - NULLS)*
- CbcNauty (const CbcNauty &)

  *Copy constructor.*
- CbcNauty & operator= (const CbcNauty &rhs)

  *Assignment operator.*
- ∼CbcNauty ()

  *Destructor.*

### 4.74.1 Detailed Description

Definition at line 176 of file CbcSymmetry.hpp.

### 4.74.2 Member Function Documentation

#### 4.74.2.1 void CbcNauty::setWriteAutoms ( const std::string & *afilename* )

Methods to classify orbits.

Not horribly efficient, but gets the job done

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

- CbcSymmetry.hpp

## 4.75 CbcNode Class Reference

Information required while the node is live.

`#include <CbcNode.hpp>`

Inheritance diagram for CbcNode:

Collaboration diagram for CbcNode:

## Public Member Functions

- CbcNode ()

  *Default Constructor.*
- CbcNode (CbcModel ∗model, CbcNode ∗lastNode)

  *Construct and increment parent reference count.*
- CbcNode (const CbcNode &)

  *Copy constructor.*
- CbcNode & operator= (const CbcNode &rhs)

  *Assignment operator.*
- ∼CbcNode ()

  *Destructor.*
- void createInfo (CbcModel ∗model, CbcNode ∗lastNode, const **CoinWarmStartBasis** ∗lastws, const double ∗lastLower, const double ∗lastUpper, int numberOldActiveCuts, int numberNewCuts)

  *Create a description of the subproblem at this node.*
- int chooseBranch (CbcModel ∗model, CbcNode ∗lastNode, int numberPassesLeft)

  *Create a branching object for the node.*
- int chooseDynamicBranch (CbcModel ∗model, CbcNode ∗lastNode, **OsiSolverBranch** ∗&branches, int numberPassesLeft)

  *Create a branching object for the node - when dynamic pseudo costs.*
- int chooseOsiBranch (CbcModel ∗model, CbcNode ∗lastNode, **OsiBranchingInformation** ∗usefulInfo, int branchState)

  *Create a branching object for the node.*
- int chooseClpBranch (CbcModel ∗model, CbcNode ∗lastNode)

  *Create a branching object for the node.*
- void decrementCuts (int change=1)

  *Decrement active cut counts.*
- void decrementParentCuts (CbcModel ∗model, int change=1)

  *Decrement all active cut counts in chain starting at parent.*
- void nullNodeInfo ()

  *Nulls out node info.*
- void initializeInfo ()

  *Initialize reference counts in attached CbcNodeInfo.*
- int branch (**OsiSolverInterface** ∗solver)

  *Does next branch and updates state.*
- double checkIsCutoff (double cutoff)

  *Double checks in case node can change its mind! Returns objective value Can change objective etc.*
- int numberBranches () const

  *Number of arms defined for the attached* **OsiBranchingObject**.
- int depth () const

  *Depth in branch-and-cut search tree.*
- void setDepth (int value)

  *Set depth in branch-and-cut search tree.*
- int numberUnsatisfied () const

  *Get the number of objects unsatisfied at this node.*
- void setNumberUnsatisfied (int value)

  *Set the number of objects unsatisfied at this node.*
- double sumInfeasibilities () const

*Get sum of "infeasibilities" reported by each object.*
- void setSumInfeasibilities (double value)

    *Set sum of "infeasibilities" reported by each object.*
- const **OsiBranchingObject** ∗ branchingObject () const

    *Branching object for this node.*
- **OsiBranchingObject** ∗ modifiableBranchingObject () const

    *Modifiable branching object for this node.*
- void setBranchingObject (**OsiBranchingObject** ∗branchingObject)

    *Set branching object for this node (takes ownership)*
- int nodeNumber () const

    *The node number.*
- bool onTree () const

    *Returns true if on tree.*
- void setOnTree (bool yesNo)

    *Sets true if on tree.*
- bool active () const

    *Returns true if active.*
- void setActive (bool yesNo)

    *Sets true if active.*
- int getState () const

    *Get state (really for debug)*
- void setState (int value)

    *Set state (really for debug)*
- void print () const

    *Print.*
- void checkInfo () const

    *Debug.*

### 4.75.1 Detailed Description

Information required while the node is live.

When a subproblem is initially created, it is represented by an CbcNode object and an attached CbcNodeInfo object.

The CbcNode contains information (depth, branching instructions), that's needed while the subproblem remains 'live', *i.e.*, while the subproblem is not fathomed and there are branch arms still be be evaluated. The CbcNode is deleted when the last branch arm has been evaluated.

The CbcNodeInfo object contains the information needed to maintain the search tree and recreate the subproblem for the node. It remains in existence until there are no nodes remaining in the subtree rooted at this node.

Definition at line 49 of file CbcNode.hpp.

### 4.75.2 Member Function Documentation

#### 4.75.2.1 void CbcNode::createInfo ( **CbcModel** ∗ *model,* **CbcNode** ∗ *lastNode,* const **CoinWarmStartBasis** ∗ *lastws,* const double ∗ *lastLower,* const double ∗ *lastUpper,* int *numberOldActiveCuts,* int *numberNewCuts* )

Create a description of the subproblem at this node.

The CbcNodeInfo structure holds the information (basis & variable bounds) required to recreate the subproblem for this node. It also links the node to its parent (via the parent's CbcNodeInfo object).

If lastNode == NULL, a CbcFullNodeInfo object will be created. All parameters except `model` are unused.

If lastNode != NULL, a CbcPartialNodeInfo object will be created. Basis and bounds information will be stored in the form of differences between the parent subproblem and this subproblem. (More precisely, `lastws`, `lastUpper`, `lastLower`, `numberOldActiveCuts`, and `numberNewCuts` are used.)

#### 4.75.2.2 int CbcNode::chooseBranch ( CbcModel * *model*, CbcNode * *lastNode*, int *numberPassesLeft* )

Create a branching object for the node.

The routine scans the object list of the model and selects a set of unsatisfied objects as candidates for branching. The candidates are evaluated, and an appropriate branch object is installed.

The numberPassesLeft is decremented to stop fixing one variable each time and going on and on (e.g. for stock cutting, air crew scheduling)

If evaluation determines that an object is monotone or infeasible, the routine returns immediately. In the case of a monotone object, the branch object has already been called to modify the model.

Return value:

- 0: A branching object has been installed
- -1: A monotone object was discovered
- -2: An infeasible object was discovered

#### 4.75.2.3 int CbcNode::chooseDynamicBranch ( CbcModel * *model*, CbcNode * *lastNode*, OsiSolverBranch *& *branches*, int *numberPassesLeft* )

Create a branching object for the node - when dynamic pseudo costs.

The routine scans the object list of the model and selects a set of unsatisfied objects as candidates for branching. The candidates are evaluated, and an appropriate branch object is installed. This version gives preference in evaluation to variables which have not been evaluated many times. It also uses numberStrong to say give up if last few tries have not changed incumbent. See Achterberg, Koch and Martin.

The numberPassesLeft is decremented to stop fixing one variable each time and going on and on (e.g. for stock cutting, air crew scheduling)

If evaluation determines that an object is monotone or infeasible, the routine returns immediately. In the case of a monotone object, the branch object has already been called to modify the model.

Return value:

- 0: A branching object has been installed
- -1: A monotone object was discovered
- -2: An infeasible object was discovered
- $>$0: Number of quich branching objects (and branches will be non NULL)

#### 4.75.2.4 int CbcNode::chooseOsiBranch ( **CbcModel** * *model*, **CbcNode** * *lastNode*, **OsiBranchingInformation** * *usefulInfo*, int *branchState* )

Create a branching object for the node.

The routine scans the object list of the model and selects a set of unsatisfied objects as candidates for branching. The candidates are evaluated, and an appropriate branch object is installed.

The numberPassesLeft is decremented to stop fixing one variable each time and going on and on (e.g. for stock cutting, air crew scheduling)

If evaluation determines that an object is monotone or infeasible, the routine returns immediately. In the case of a monotone object, the branch object has already been called to modify the model.

Return value:

- 0: A branching object has been installed
- -1: A monotone object was discovered
- -2: An infeasible object was discovered

Branch state:

- -1: start
- -1: A monotone object was discovered
- -2: An infeasible object was discovered

#### 4.75.2.5 int CbcNode::chooseClpBranch ( **CbcModel** * *model*, **CbcNode** * *lastNode* )

Create a branching object for the node.

The routine scans the object list of the model and selects a set of unsatisfied objects as candidates for branching. It then solves a series of problems and a CbcGeneral branch object is installed.

If evaluation determines that an object is infeasible, the routine returns immediately.

Return value:

- 0: A branching object has been installed
- -2: An infeasible object was discovered

#### 4.75.2.6 void CbcNode::initializeInfo ( )

Initialize reference counts in attached CbcNodeInfo.

This is a convenience routine, which will initialize the reference counts in the attached CbcNodeInfo object based on the attached **OsiBranchingObject**.

**See also**

CbcNodeInfo::initializeInfo(int).

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

- CbcNode.hpp

## 4.76 CbcNodeInfo Class Reference

Information required to recreate the subproblem at this node.

`#include <CbcNodeInfo.hpp>`

Inheritance diagram for CbcNodeInfo:

Collaboration diagram for CbcNodeInfo:

### Public Member Functions

- virtual void applyToModel (CbcModel ∗model, **CoinWarmStartBasis** ∗&basis, CbcCountRowCut ∗∗addCuts, int &currentNumberCuts) const =0

  *Modify model according to information at node.*
- virtual int applyBounds (int iColumn, double &lower, double &upper, int force)=0

  *Just apply bounds to one variable - force means overwrite by lower,upper (1=>infeasible)*
- virtual CbcNodeInfo ∗ buildRowBasis (**CoinWarmStartBasis** &basis) const =0

  *Builds up row basis backwards (until original model).*
- virtual CbcNodeInfo ∗ clone () const =0

  *Clone.*
- virtual void allBranchesGone ()

  *Called when number branches left down to zero.*
- void increment (int amount=1)

  *Increment number of references.*
- int decrement (int amount=1)

  *Decrement number of references and return number left.*
- void initializeInfo (int number)

  *Initialize reference counts.*
- int numberBranchesLeft () const

  *Return number of branches left in object.*
- void setNumberBranchesLeft (int value)

  *Set number of branches left in object.*
- int numberPointingToThis () const

  *Return number of objects pointing to this.*
- void setNumberPointingToThis (int number)

  *Set number of objects pointing to this.*
- void incrementNumberPointingToThis ()

  *Increment number of objects pointing to this.*
- int branchedOn ()

  *Say one branch taken.*
- void throwAway ()

  *Say thrown away.*
- CbcNodeInfo ∗ parent () const

  *Parent of this.*
- void nullParent ()

  *Set parent null.*
- void deleteCuts (int numberToDelete, CbcCountRowCut ∗∗cuts)

  *Delete cuts (decrements counts) Slow unless cuts in same order as saved.*

- void deleteCut (int whichOne)

  *Really delete a cut.*
- void decrementCuts (int change=1)

  *Decrement active cut counts.*
- void incrementCuts (int change=1)

  *Increment active cut counts.*
- void decrementParentCuts (CbcModel ∗model, int change=1)

  *Decrement all active cut counts in chain starting at parent.*
- void incrementParentCuts (CbcModel ∗model, int change=1)

  *Increment all active cut counts in parent chain.*
- CbcCountRowCut ∗∗ cuts () const

  *Array of pointers to cuts.*
- int numberCuts () const

  *Number of row cuts (this node)*
- void nullOwner ()

  *Set owner null.*
- int nodeNumber () const

  *The node number.*
- void deactivate (int mode=3)

  *Deactivate node information.*
- bool allActivated () const

  *Say if normal.*
- bool marked () const

  *Say if marked.*
- void mark ()

  *Mark.*
- void unmark ()

  *Unmark.*
- bool symmetryWorked () const

  *Get symmetry value (true worked at this node)*
- void setSymmetryWorked ()

  *Say symmetry worked at this node)*
- const **OsiBranchingObject** ∗ parentBranch () const

  *Branching object for the parent.*
- void unsetParentBasedData ()

  *If we need to take off parent based data.*

#### Constructors & destructors

- CbcNodeInfo ()

  *Default Constructor.*
- CbcNodeInfo (const CbcNodeInfo &)

  *Copy constructor.*
- CbcNodeInfo (CbcNodeInfo ∗parent, CbcNode ∗owner)

  *Construct with parent and owner.*
- virtual ∼CbcNodeInfo ()

  *Destructor.*

### Protected Attributes

- int numberPointingToThis_

  *Number of other nodes pointing to this node.*
- CbcNodeInfo * parent_

  *parent*
- **OsiBranchingObject** * parentBranch_

  *Copy of the branching object of the parent when the node is created.*
- CbcNode * owner_

  *Owner.*
- int numberCuts_

  *Number of row cuts (this node)*
- int nodeNumber_

  *The node number.*
- CbcCountRowCut ** cuts_

  *Array of pointers to cuts.*
- int numberRows_

  *Number of rows in problem (before these cuts).*
- int numberBranchesLeft_

  *Number of branch arms left to explore at this node.*
- int active_

  *Active node information.*

### 4.76.1 Detailed Description

Information required to recreate the subproblem at this node.

When a subproblem is initially created, it is represented by a CbcNode object and an attached CbcNodeInfo object.

The CbcNode contains information needed while the subproblem remains live. The CbcNode is deleted when the last branch arm has been evaluated.

The CbcNodeInfo contains information required to maintain the branch-and-cut search tree structure (links and reference counts) and to recreate the subproblem for this node (basis, variable bounds, cutting planes). A CbcNodeInfo object remains in existence until all nodes have been pruned from the subtree rooted at this node.

The principle used to maintain the reference count is that the reference count is always the sum of all potential and actual children of the node. Specifically,

- Once it's determined how the node will branch, the reference count is set to the number of potential children (*i.e.*, the number of arms of the branch).
- As each child is created by CbcNode::branch() (converting a potential child to the active subproblem), the reference count is decremented.
- If the child survives and will become a node in the search tree (converting the active subproblem into an actual child), increment the reference count.

Notice that the active subproblem lives in a sort of limbo, neither a potential or an actual node in the branch-and-cut tree.

CbcNodeInfo objects come in two flavours. A CbcFullNodeInfo object contains a full record of the information required to recreate a subproblem. A CbcPartialNodeInfo object expresses this information in terms of differences from the parent.

Definition at line 68 of file CbcNodeInfo.hpp.

## 4.76.2 Constructor & Destructor Documentation

### 4.76.2.1 CbcNodeInfo::CbcNodeInfo ( )

Default Constructor.

Creates an empty NodeInfo object.

### 4.76.2.2 CbcNodeInfo::CbcNodeInfo ( CbcNodeInfo * *parent*, CbcNode * *owner* )

Construct with parent and owner.

As for 'construct with parent', and attached to `owner`.

### 4.76.2.3 virtual CbcNodeInfo::∼CbcNodeInfo ( ) [virtual]

Destructor.

Note that the destructor will recursively delete the parent if this nodeInfo is the last child.

## 4.76.3 Member Function Documentation

### 4.76.3.1 virtual void CbcNodeInfo::applyToModel ( CbcModel * *model*, CoinWarmStartBasis *& *basis*, CbcCountRowCut ** *addCuts*, int & *currentNumberCuts* ) const [pure virtual]

Modify model according to information at node.

The routine modifies the model according to bound and basis information at node and adds any cuts to the addCuts array.

Implemented in CbcFullNodeInfo, and CbcPartialNodeInfo.

### 4.76.3.2 virtual CbcNodeInfo* CbcNodeInfo::buildRowBasis ( CoinWarmStartBasis & *basis* ) const [pure virtual]

Builds up row basis backwards (until original model).

Returns NULL or previous one to apply . Depends on Free being 0 and impossible for cuts

Implemented in CbcFullNodeInfo, and CbcPartialNodeInfo.

### 4.76.3.3 void CbcNodeInfo::initializeInfo ( int *number* ) [inline]

Initialize reference counts.

Initialize the reference counts used for tree maintenance.

Definition at line 149 of file CbcNodeInfo.hpp.

### 4.76.3.4 void CbcNodeInfo::deactivate ( int *mode* = 3 )

Deactivate node information.

1 - bounds 2 - cuts 4 - basis! 8 - just marked 16 - symmetry branching worked

### 4.76.4 Member Data Documentation

#### 4.76.4.1 int CbcNodeInfo::numberPointingToThis_ [protected]

Number of other nodes pointing to this node.

Number of existing and potential search tree nodes pointing to this node. 'Existing' means referenced by parent_ of some other CbcNodeInfo. 'Potential' means children still to be created (numberBranchesLeft_ of this CbcNodeInfo).

Definition at line 301 of file CbcNodeInfo.hpp.

#### 4.76.4.2 int CbcNodeInfo::numberRows_ [protected]

Number of rows in problem (before these cuts).

This means that for top of chain it must be rows at continuous

Definition at line 323 of file CbcNodeInfo.hpp.

#### 4.76.4.3 int CbcNodeInfo::numberBranchesLeft_ [protected]

Number of branch arms left to explore at this node.

Definition at line 331 of file CbcNodeInfo.hpp.

#### 4.76.4.4 int CbcNodeInfo::active_ [protected]

Active node information.

1 - bounds 2 - cuts 4 - basis!

Definition at line 337 of file CbcNodeInfo.hpp.

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

- CbcNodeInfo.hpp

## 4.77 CbcNWay Class Reference

Define an n-way class for variables.

`#include <CbcNWay.hpp>`

Inheritance diagram for CbcNWay:

Collaboration diagram for CbcNWay:

### Public Member Functions

- CbcNWay (CbcModel ∗model, int numberMembers, const int ∗which, int identifier)

    *Useful constructor (which are matrix indices)*
- virtual CbcObject ∗ clone () const

    *Clone.*
- CbcNWay & operator= (const CbcNWay &rhs)

*Assignment operator.*
- virtual ~CbcNWay ()

  *Destructor.*
- void setConsequence (int iColumn, const CbcConsequence &consequence)

  *Set up a consequence for a single member.*
- void applyConsequence (int iSequence, int state) const

  *Applies a consequence for a single member.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗info, int &preferredWay) const

  *Infeasibility - large is 0.5 (and 0.5 will give this)*
- virtual void feasibleRegion ()

  *This looks at solution and sets bounds to contain solution.*
- virtual CbcBranchingObject ∗ createCbcBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way)

  *Creates a branching object.*
- int numberMembers () const

  *Number of members.*
- const int ∗ members () const

  *Members (indices in range 0 ... numberColumns-1)*
- virtual void redoSequenceEtc (CbcModel ∗model, int numberColumns, const int ∗originalColumns)

  *Redoes data when sequence numbers change.*

### Protected Attributes

- int numberMembers_

  *data Number of members*
- int ∗ members_

  *Members (indices in range 0 ... numberColumns-1)*
- CbcConsequence ∗∗ consequence_

  *Consequences (normally NULL)*

### 4.77.1 Detailed Description

Define an n-way class for variables.

Only valid value is one at UB others at LB Normally 0-1

Definition at line 15 of file CbcNWay.hpp.

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

- CbcNWay.hpp

## 4.78 CbcNWayBranchingObject Class Reference

N way branching Object class.

`#include <CbcNWay.hpp>`

Inheritance diagram for CbcNWayBranchingObject:

Collaboration diagram for CbcNWayBranchingObject:

## Public Member Functions

- CbcNWayBranchingObject (CbcModel ∗model, const CbcNWay ∗nway, int numberBranches, const int ∗order)
  *Useful constructor - order had matrix indices way_ -1 corresponds to setting first, +1 to second, +3 etc.*
- virtual CbcBranchingObject ∗ clone () const
  *Clone.*
- virtual double branch ()
  *Does next branch and updates state.*
- virtual void print ()
  *Print something about branch - only if log level high.*
- virtual int numberBranches () const
  *The number of branch arms created for this branching object.*
- virtual bool twoWay () const
  *Is this a two way object (-1 down, +1 up)*
- virtual CbcBranchObjType type () const
  *Return the type (an integer identifier) of `this`.*
- virtual int compareOriginalObject (const CbcBranchingObject ∗brObj) const
  *Compare the original object of `this` with the original object of `brObj`.*
- virtual CbcRangeCompare compareBranchingObject (const CbcBranchingObject ∗brObj, const bool replaceIfOverlap=false)
  *Compare the `this` with `brObj`.*

## Additional Inherited Members

### 4.78.1 Detailed Description

N way branching Object class.

Variable is number of set.

Definition at line 81 of file CbcNWay.hpp.

### 4.78.2 Constructor & Destructor Documentation

#### 4.78.2.1 CbcNWayBranchingObject::CbcNWayBranchingObject ( CbcModel ∗ *model*, const CbcNWay ∗ *nway*, int *numberBranches*, const int ∗ *order* )

Useful constructor - order had matrix indices way_ -1 corresponds to setting first, +1 to second, +3 etc.

this is so -1 and +1 have similarity to normal

### 4.78.3 Member Function Documentation

#### 4.78.3.1 virtual int CbcNWayBranchingObject::compareOriginalObject ( const CbcBranchingObject ∗ *brObj* ) const [virtual]

Compare the original object of `this` with the original object of `brObj`.

Assumes that there is an ordering of the original objects. This method should be invoked only if `this` and brObj are of the same type. Return negative/0/positive depending on whether `this` is smaller/same/larger than the argument.

Reimplemented from CbcBranchingObject.

#### 4.78.3.2 virtual CbcRangeCompare CbcNWayBranchingObject::compareBranchingObject ( const CbcBranchingObject ∗ *brObj*, const bool *replaceIfOverlap* = false ) [virtual]

Compare the `this` with `brObj`.

`this` and `brObj` must be os the same type and must have the same original object, but they may have different feasible regions. Return the appropriate CbcRangeCompare value (first argument being the sub/superset if that's the case). In case of overlap (and if `replaceIfOverlap` is true) replace the current branching object with one whose feasible region is the overlap.

Implements CbcBranchingObject.

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

- CbcNWay.hpp

## 4.79 CbcObject Class Reference

Inheritance diagram for CbcObject:

Collaboration diagram for CbcObject:

### Public Member Functions

- virtual CbcObject ∗ clone () const =0

    *Clone.*
- virtual ∼CbcObject ()

    *Destructor.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗, int &preferredWay) const

    *Infeasibility of the object.*
- virtual void feasibleRegion ()=0

    *For the variable(s) referenced by the object, look at the current solution and set bounds to match the solution.*
- virtual double feasibleRegion (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info) const

    *Dummy one for compatibility.*
- virtual double feasibleRegion (**OsiSolverInterface** ∗solver) const

    *For the variable(s) referenced by the object, look at the current solution and set bounds to match the solution.*
- virtual CbcBranchingObject ∗ createCbcBranch (**OsiSolverInterface** ∗, const **OsiBranchingInformation** ∗, int)

    *Create a branching object and indicate which way to branch first.*
- virtual **OsiBranchingObject** ∗ createOsiBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way) const

    *Create an Osibranching object and indicate which way to branch first.*
- virtual **OsiSolverBranch** ∗ solverBranch () const

    *Create an* **OsiSolverBranch** *object.*
- virtual CbcBranchingObject ∗ preferredNewFeasible () const

    *Given a valid solution (with reduced costs, etc.), return a branching object which would give a new feasible point in a good direction.*
- virtual CbcBranchingObject ∗ notPreferredNewFeasible () const

    *Given a valid solution (with reduced costs, etc.), return a branching object which would give a new feasible point in a bad direction.*
- virtual void resetBounds (const **OsiSolverInterface** ∗)

*Reset variable bounds to their original values.*

- virtual void floorCeiling (double &floorValue, double &ceilingValue, double value, double tolerance) const

  *Returns floor and ceiling i.e.*
- virtual CbcObjectUpdateData createUpdateInformation (const **OsiSolverInterface** ∗solver, const CbcNode ∗node, const CbcBranchingObject ∗branchingObject)

  *Pass in information on branch just done and create CbcObjectUpdateData instance.*
- virtual void updateInformation (const CbcObjectUpdateData &)

  *Update object by CbcObjectUpdateData.*
- int id () const

  *Identifier (normally column number in matrix)*
- void setId (int value)

  *Set identifier (normally column number in matrix) but 1000000000 to 1100000000 means optional branching object i.e.*
- bool optionalObject () const

  *Return true if optional branching object i.e.*
- int position () const

  *Get position in object_ list.*
- void setPosition (int position)

  *Set position in object_ list.*
- void setModel (CbcModel ∗model)

  *update model*
- CbcModel ∗ model () const

  *Return model.*
- int preferredWay () const

  *If -1 down always chosen first, +1 up always, 0 normal.*
- void setPreferredWay (int value)

  *Set -1 down always chosen first, +1 up always, 0 normal.*
- virtual void redoSequenceEtc (CbcModel ∗, int, const int ∗)

  *Redoes data when sequence numbers change.*
- virtual void initializeForBranching (CbcModel ∗)

  *Initialize for branching.*

## Protected Attributes

- CbcModel ∗ model_

  *data*
- int id_

  *Identifier (normally column number in matrix)*
- int position_

  *Position in object list.*
- int preferredWay_

  *If -1 down always chosen first, +1 up always, 0 normal.*

### 4.79.1 Detailed Description

Definition at line 67 of file CbcObject.hpp.

### 4.79.2 Member Function Documentation

#### 4.79.2.1 virtual double CbcObject::infeasibility ( const **OsiBranchingInformation** * , int & *preferredWay* ) const [inline], [virtual]

Infeasibility of the object.

This is some measure of the infeasibility of the object. It should be scaled to be in the range [0.0, 0.5], with 0.0 indicating the object is satisfied.

The preferred branching direction is returned in preferredWay,

This is used to prepare for strong branching but should also think of case when no strong branching

The object may also compute an estimate of cost of going "up" or "down". This will probably be based on pseudo-cost ideas

Reimplemented in CbcSimpleInteger, CbcIdiotBranch, CbcClique, CbcSimpleIntegerDynamicPseudoCost, CbcSOS, CbcBranchToFixLots, CbcFollowOn, CbcBranchAllDifferent, CbcNWay, CbcSimpleIntegerPseudoCost, CbcGeneral, CbcBranchCut, and CbcLotsize.

Definition at line 107 of file CbcObject.hpp.

#### 4.79.2.2 virtual double CbcObject::feasibleRegion ( **OsiSolverInterface** * *solver* ) const [virtual]

For the variable(s) referenced by the object, look at the current solution and set bounds to match the solution.

Returns measure of how much it had to move solution to make feasible

Reimplemented from **OsiObject**.

#### 4.79.2.3 virtual **CbcBranchingObject*** CbcObject::createCbcBranch ( **OsiSolverInterface** * , const **OsiBranchingInformation** * , int ) [inline], [virtual]

Create a branching object and indicate which way to branch first.

The branching object has to know how to create branches (fix variables, etc.)

Reimplemented in CbcSimpleInteger, CbcIdiotBranch, CbcClique, CbcSOS, CbcBranchToFixLots, CbcSimpleIntegerDynamicPseudoCost, CbcBranchCut, CbcFollowOn, CbcNWay, CbcLotsize, CbcBranchAllDifferent, CbcGeneral, and CbcSimpleIntegerPseudoCost.

Definition at line 137 of file CbcObject.hpp.

#### 4.79.2.4 virtual **OsiBranchingObject*** CbcObject::createOsiBranch ( **OsiSolverInterface** * *solver*, const **OsiBranchingInformation** * *info*, int *way* ) const [virtual]

Create an Osibranching object and indicate which way to branch first.

The branching object has to know how to create branches (fix variables, etc.)

#### 4.79.2.5 virtual **OsiSolverBranch*** CbcObject::solverBranch ( ) const [virtual]

Create an **OsiSolverBranch** object.

This returns NULL if branch not represented by bound changes

Reimplemented in CbcSimpleIntegerDynamicPseudoCost, and CbcSOS.

#### 4.79.2.6 virtual CbcBranchingObject* CbcObject::preferredNewFeasible ( ) const [inline], [virtual]

Given a valid solution (with reduced costs, etc.), return a branching object which would give a new feasible point in a good direction.

If the method cannot generate a feasible point (because there aren't any, or because it isn't bright enough to find one), it should return null.

Reimplemented in CbcBranchCut, and CbcLotsize.

Definition at line 169 of file CbcObject.hpp.

#### 4.79.2.7 virtual CbcBranchingObject* CbcObject::notPreferredNewFeasible ( ) const [inline], [virtual]

Given a valid solution (with reduced costs, etc.), return a branching object which would give a new feasible point in a bad direction.

If the method cannot generate a feasible point (because there aren't any, or because it isn't bright enough to find one), it should return null.

Reimplemented in CbcBranchCut, and CbcLotsize.

Definition at line 181 of file CbcObject.hpp.

#### 4.79.2.8 virtual void CbcObject::resetBounds ( const OsiSolverInterface * ) [inline], [virtual]

Reset variable bounds to their original values.

Bounds may be tightened, so it may be good to be able to set this info in object.

Reimplemented from **OsiObject**.

Reimplemented in CbcSimpleInteger, and CbcLotsize.

Definition at line 189 of file CbcObject.hpp.

#### 4.79.2.9 virtual void CbcObject::floorCeiling ( double & *floorValue*, double & *ceilingValue*, double *value*, double *tolerance* ) const [virtual]

Returns floor and ceiling i.e.

closest valid points

Reimplemented in CbcLotsize.

#### 4.79.2.10 virtual CbcObjectUpdateData CbcObject::createUpdateInformation ( const OsiSolverInterface * *solver*, const CbcNode * *node*, const CbcBranchingObject * *branchingObject* ) [virtual]

Pass in information on branch just done and create CbcObjectUpdateData instance.

If object does not need data then backward pointer will be NULL. Assumes can get information from solver

Reimplemented in CbcSimpleIntegerDynamicPseudoCost, and CbcSOS.

#### 4.79.2.11 void CbcObject::setId ( int *value* ) [inline]

Set identifier (normally column number in matrix) but 1000000000 to 1100000000 means optional branching object i.e.

code would work without it

Definition at line 214 of file CbcObject.hpp.

#### 4.79.2.12 bool CbcObject::optionalObject ( ) const [inline]

Return true if optional branching object i.e.

code would work without it

Definition at line 220 of file CbcObject.hpp.

### 4.79.3 Member Data Documentation

#### 4.79.3.1 CbcModel* CbcObject::model_ [protected]

data

Model

Definition at line 261 of file CbcObject.hpp.

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

- CbcObject.hpp

## 4.80 CbcObjectUpdateData Class Reference

Collaboration diagram for CbcObjectUpdateData:

### Public Member Functions

- CbcObjectUpdateData ()

  *Default Constructor.*
- CbcObjectUpdateData (CbcObject *object, int way, double change, int status, int intDecrease_, double branchingValue)

  *Useful constructor.*
- CbcObjectUpdateData (const CbcObjectUpdateData &)

  *Copy constructor.*
- CbcObjectUpdateData & operator= (const CbcObjectUpdateData &rhs)

  *Assignment operator.*
- virtual ∼CbcObjectUpdateData ()

  *Destructor.*

### Public Attributes

- CbcObject * object_

  *data*
- int way_

  *Branch as defined by instance of CbcObject.*

- int objectNumber_

  *Object number.*
- double change_

  *Change in objective.*
- int status_

  *Status 0 Optimal, 1 infeasible, 2 unknown.*
- int intDecrease_

  *Decrease in number unsatisfied.*
- double branchingValue_

  *Branching value.*
- double originalObjective_

  *Objective value before branching.*
- double cutoff_

  *Current cutoff.*

### 4.80.1 Detailed Description

Definition at line 14 of file CbcObjectUpdateData.hpp.

### 4.80.2 Member Data Documentation

#### 4.80.2.1 CbcObject* CbcObjectUpdateData::object_

data

Object

Definition at line 43 of file CbcObjectUpdateData.hpp.

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

- CbcObjectUpdateData.hpp

## 4.81 CbcOrbitalBranchingObject Class Reference

Branching object for Orbital branching.

```
#include <CbcSymmetry.hpp>
```

Inheritance diagram for CbcOrbitalBranchingObject:

Collaboration diagram for CbcOrbitalBranchingObject:

### Public Member Functions

- virtual CbcBranchingObject ∗ clone () const

  *Clone.*
- virtual double branch ()

  *Does next branch and updates state.*
- virtual void fix (**OsiSolverInterface** ∗solver, double ∗lower, double ∗upper, int branchState) const

*Update bounds in solver as in 'branch' and update given bounds.*

- virtual void previousBranch ()

  *Reset every information so that the branching object appears to point to the previous child.*
- virtual void print ()

  *Print something about branch - only if log level high.*
- virtual CbcBranchObjType type () const

  *Return the type (an integer identifier) of `this`.*
- virtual int compareOriginalObject (const CbcBranchingObject *brObj) const

  *Compare the original object of `this` with the original object of `brObj`.*
- virtual CbcRangeCompare compareBranchingObject (const CbcBranchingObject *brObj, const bool replaceIf↩ Overlap=false)

  *Compare the `this` with `brObj`.*

## Additional Inherited Members

### 4.81.1 Detailed Description

Branching object for Orbital branching.

Variable_ is the set id number (redundant, as the object also holds a pointer to the set.

Definition at line 285 of file CbcSymmetry.hpp.

### 4.81.2 Member Function Documentation

#### 4.81.2.1 virtual void CbcOrbitalBranchingObject::fix ( OsiSolverInterface * *solver,* double * *lower,* double * *upper,* int *branchState* ) const [virtual]

Update bounds in solver as in 'branch' and update given bounds.

branchState is -1 for 'down' +1 for 'up'

Reimplemented from CbcBranchingObject.

#### 4.81.2.2 virtual void CbcOrbitalBranchingObject::previousBranch ( ) [inline], [virtual]

Reset every information so that the branching object appears to point to the previous child.

This method does not need to modify anything in any solver.

Reimplemented from CbcBranchingObject.

Definition at line 321 of file CbcSymmetry.hpp.

#### 4.81.2.3 virtual int CbcOrbitalBranchingObject::compareOriginalObject ( const CbcBranchingObject * *brObj* ) const [virtual]

Compare the original object of `this` with the original object of `brObj`.

Assumes that there is an ordering of the original objects. This method should be invoked only if `this` and brObj are of the same type. Return negative/0/positive depending on whether `this` is smaller/same/larger than the argument.

Reimplemented from CbcBranchingObject.

#### 4.81.2.4 virtual CbcRangeCompare CbcOrbitalBranchingObject::compareBranchingObject ( const CbcBranchingObject * *brObj,* const bool *replaceIfOverlap* = false ) [virtual]

Compare the `this` with `brObj`.

`this` and `brObj` must be os the same type and must have the same original object, but they may have different feasible regions. Return the appropriate CbcRangeCompare value (first argument being the sub/superset if that's the case). In case of overlap (and if `replaceIfOverlap` is true) replace the current branching object with one whose feasible region is the overlap.

Implements CbcBranchingObject.

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

- CbcSymmetry.hpp

## 4.82 CbcOsiParam Class Reference

Class for control parameters that act on a **OsiSolverInterface** object.

`#include <CbcGenOsiParam.hpp>`

Inheritance diagram for CbcOsiParam:

Collaboration diagram for CbcOsiParam:

### Public Types

#### Subtypes

- enum CbcOsiParamCode
  *Enumeration for parameters that control an* **OsiSolverInterface** *object.*

### Public Member Functions

#### Constructors and Destructors

*Be careful how you specify parameters for the constructors! There's great potential for confusion.*

- CbcOsiParam ()
  *Default constructor.*
- CbcOsiParam (CbcOsiParamCode code, std::string **name**, std::string help, double lower, double upper, double dflt=0.0, bool **display**=true)
  *Constructor for a parameter with a double value.*
- CbcOsiParam (CbcOsiParamCode code, std::string **name**, std::string help, int lower, int upper, int dflt=0, bool **display**=true)
  *Constructor for a parameter with an integer value.*
- CbcOsiParam (CbcOsiParamCode code, std::string **name**, std::string help, std::string firstValue, int dflt, bool **display**=true)
  *Constructor for a parameter with keyword values.*
- CbcOsiParam (CbcOsiParamCode code, std::string **name**, std::string help, std::string dflt, bool **display**=true)
  *Constructor for a string parameter.*
- CbcOsiParam (CbcOsiParamCode code, std::string **name**, std::string help, bool **display**=true)
  *Constructor for an action parameter.*
- CbcOsiParam (const CbcOsiParam &orig)

*Copy constructor.*
- CbcOsiParam ∗ clone ()
  *Clone.*
- CbcOsiParam & operator= (const CbcOsiParam &rhs)
  *Assignment.*
- ∼CbcOsiParam ()
  *Destructor.*

**Methods to query and manipulate a parameter object**

- CbcOsiParamCode paramCode () const
  *Get the parameter code.*
- void setParamCode (CbcOsiParamCode code)
  *Set the parameter code.*
- **OsiSolverInterface** ∗ obj () const
  *Get the underlying* **OsiSolverInterface** *object.*
- void setObj (**OsiSolverInterface** ∗obj)
  *Set the underlying OsiSolverInterace object.*

### 4.82.1 Detailed Description

Class for control parameters that act on a **OsiSolverInterface** object.

Adds parameter type codes and push/pull functions to the generic parameter object.

Definition at line 31 of file CbcGenOsiParam.hpp.

### 4.82.2 Member Enumeration Documentation

#### 4.82.2.1 enum CbcOsiParam::CbcOsiParamCode

Enumeration for parameters that control an **OsiSolverInterface** object.

These are parameters that control the operation of an **OsiSolverInterface** object. CBCOSI_FIRSTPARAM and CBC←OSI_LASTPARAM are markers to allow convenient separation of parameter groups.

Definition at line 46 of file CbcGenOsiParam.hpp.

### 4.82.3 Constructor & Destructor Documentation

#### 4.82.3.1 CbcOsiParam::CbcOsiParam ( CbcOsiParamCode *code,* std::string *name,* std::string *help,* double *lower,* double *upper,* double *dflt* = `0.0`, bool *display* = `true` )

Constructor for a parameter with a double value.

The default value is 0.0. Be careful to clearly indicate that `lower` and `upper` are real (double) values to distinguish this constructor from the constructor for an integer parameter.

#### 4.82.3.2 CbcOsiParam::CbcOsiParam ( CbcOsiParamCode *code,* std::string *name,* std::string *help,* int *lower,* int *upper,* int *dflt* = `0`, bool *display* = `true` )

Constructor for a parameter with an integer value.

The default value is 0.

#### 4.82.3.3 CbcOsiParam::CbcOsiParam ( CbcOsiParamCode *code*, std::string *name*, std::string *help*, std::string *firstValue*, int *dflt*, bool *display* = `true` )

Constructor for a parameter with keyword values.

The string supplied as `firstValue` becomes the first keyword. Additional keywords can be added using **appendKwd()**. Keywords are numbered from zero. It's necessary to specify both the first keyword (`firstValue`) and the default keyword index (`dflt`) in order to distinguish this constructor from the string and action parameter constructors.

#### 4.82.3.4 CbcOsiParam::CbcOsiParam ( CbcOsiParamCode *code*, std::string *name*, std::string *help*, std::string *dflt*, bool *display* = `true` )

Constructor for a string parameter.

The default string value must be specified explicitly to distinguish a string constructor from an action parameter constructor.

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

- CbcGenOsiParam.hpp

## 4.83 CbcOsiSolver Class Reference

This is for codes where solver needs to know about CbcModel Seems to provide only one value-added feature, a CbcModel object.

`#include <CbcFathom.hpp>`

Inheritance diagram for CbcOsiSolver:

Collaboration diagram for CbcOsiSolver:

### Public Member Functions

#### Constructors and destructors

- CbcOsiSolver ()
  *Default Constructor.*
- virtual **OsiSolverInterface** ∗ clone (bool copyData=true) const
  *Clone.*
- CbcOsiSolver (const CbcOsiSolver &)
  *Copy constructor.*
- CbcOsiSolver & operator= (const CbcOsiSolver &rhs)
  *Assignment operator.*
- virtual ∼CbcOsiSolver ()
  *Destructor.*

#### Sets and Gets

- void setCbcModel (CbcModel ∗model)
  *Set Cbc Model.*
- CbcModel ∗ cbcModel () const
  *Return Cbc Model.*

### Protected Attributes

**Private member data**

- CbcModel ∗ cbcModel_

  *Pointer back to CbcModel.*

#### 4.83.1 Detailed Description

This is for codes where solver needs to know about CbcModel Seems to provide only one value-added feature, a CbcModel object.

Definition at line 90 of file CbcFathom.hpp.

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

- CbcFathom.hpp

## 4.84 CbcParam Class Reference

Very simple class for setting parameters.

```
#include <CbcParam.hpp>
```

### Public Member Functions

**Constructor and destructor**

- CbcParam ()

  *Constructors.*
- **CbcParam** (std::string name, std::string help, double lower, double upper, CbcParameterType type, bool display=true)
- **CbcParam** (std::string name, std::string help, int lower, int upper, CbcParameterType type, bool display=true)
- **CbcParam** (std::string name, std::string help, std::string firstValue, CbcParameterType type, int defaultIndex=0, bool display=true)
- **CbcParam** (std::string name, std::string help, CbcParameterType type, int indexNumber=-1, bool display=true)
- CbcParam (const CbcParam &)

  *Copy constructor.*
- CbcParam & operator= (const CbcParam &rhs)

  *Assignment operator. This copies the data.*
- ∼CbcParam ()

  *Destructor.*

### stuff

- void append (std::string keyWord)

  *Insert string (only valid for keywords)*
- void addHelp (std::string keyWord)

  *Adds one help line.*
- std::string name () const

  *Returns name.*

- std::string shortHelp () const
  *Returns short help.*
- int setDoubleParameter (CbcModel &model, double value) const
  *Sets a double parameter (nonzero code if error)*
- double doubleParameter (CbcModel &model) const
  *Gets a double parameter.*
- int setIntParameter (CbcModel &model, int value) const
  *Sets a int parameter (nonzero code if error)*
- int intParameter (CbcModel &model) const
  *Gets a int parameter.*
- int setDoubleParameter (**ClpSimplex** ∗model, double value) const
  *Sets a double parameter (nonzero code if error)*
- double doubleParameter (**ClpSimplex** ∗model) const
  *Gets a double parameter.*
- int setIntParameter (**ClpSimplex** ∗model, int value) const
  *Sets a int parameter (nonzero code if error)*
- int intParameter (**ClpSimplex** ∗model) const
  *Gets a int parameter.*
- int setDoubleParameter (**OsiSolverInterface** ∗model, double value) const
  *Sets a double parameter (nonzero code if error)*
- double doubleParameter (**OsiSolverInterface** ∗model) const
  *Gets a double parameter.*
- int setIntParameter (**OsiSolverInterface** ∗model, int value) const
  *Sets a int parameter (nonzero code if error)*
- int intParameter (**OsiSolverInterface** ∗model) const
  *Gets a int parameter.*
- int checkDoubleParameter (double value) const
  *Checks a double parameter (nonzero code if error)*
- std::string matchName () const
  *Returns name which could match.*
- int parameterOption (std::string check) const
  *Returns parameter option which matches (-1 if none)*
- void printOptions () const
  *Prints parameter options.*
- std::string currentOption () const
  *Returns current parameter option.*
- void setCurrentOption (int value)
  *Sets current parameter option.*
- void setIntValue (int value)
  *Sets int value.*
- int **intValue** () const
- void setDoubleValue (double value)
  *Sets double value.*
- double **doubleValue** () const
- void setStringValue (std::string value)
  *Sets string value.*
- std::string **stringValue** () const

- int matches (std::string input) const

    *Returns 1 if matches minimum, 2 if matches less, 0 if not matched.*
- CbcParameterType type () const

    *type*
- bool displayThis () const

    *whether to display*
- void setLonghelp (const std::string help)

    *Set Long help.*
- void printLongHelp () const

    *Print Long help.*
- void printString () const

    *Print action and string.*
- int indexNumber () const

    *type for classification*

### 4.84.1 Detailed Description

Very simple class for setting parameters.

Definition at line 153 of file CbcParam.hpp.

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

- CbcParam.hpp

## 4.85 CbcGenCtlBlk::cbcParamsInfo_struct Struct Reference

Start and end of CbcModel parameters in parameter vector.

`#include <CbcGenCtlBlk.hpp>`

### 4.85.1 Detailed Description

Start and end of CbcModel parameters in parameter vector.

Definition at line 605 of file CbcGenCtlBlk.hpp.

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

- CbcGenCtlBlk.hpp

## 4.86 CbcPartialNodeInfo Class Reference

Holds information for recreating a subproblem by incremental change from the parent.

`#include <CbcPartialNodeInfo.hpp>`

Inheritance diagram for CbcPartialNodeInfo:

Collaboration diagram for CbcPartialNodeInfo:

### Public Member Functions

- virtual void applyToModel (CbcModel ∗model, **CoinWarmStartBasis** ∗&basis, CbcCountRowCut ∗∗addCuts, int &currentNumberCuts) const

    *Modify model according to information at node.*
- virtual int applyBounds (int iColumn, double &lower, double &upper, int force)

    *Just apply bounds to one variable - force means overwrite by lower,upper (1=>infeasible)*
- virtual CbcNodeInfo ∗ buildRowBasis (**CoinWarmStartBasis** &basis) const

    *Builds up row basis backwards (until original model).*
- virtual CbcNodeInfo ∗ clone () const

    *Clone.*
- const **CoinWarmStartDiff** ∗ basisDiff () const

    *Basis diff information.*
- const int ∗ variables () const

    *Which variable (top bit if upper bound changing)*
- int numberChangedBounds () const

    *Number of bound changes.*

### Protected Attributes

- **CoinWarmStartDiff** ∗ basisDiff_

    *Basis diff information.*
- int ∗ variables_

    *Which variable (top bit if upper bound changing)*
- int numberChangedBounds_

    *Number of bound changes.*

### 4.86.1 Detailed Description

Holds information for recreating a subproblem by incremental change from the parent.

A CbcPartialNodeInfo object contains changes to the bounds and basis, and additional cuts, required to recreate a subproblem by modifying and augmenting the parent subproblem.

Definition at line 39 of file CbcPartialNodeInfo.hpp.

### 4.86.2 Member Function Documentation

#### 4.86.2.1 virtual void CbcPartialNodeInfo::applyToModel ( **CbcModel** ∗ *model,* **CoinWarmStartBasis** ∗& *basis,* **CbcCountRowCut** ∗∗ *addCuts,* int & *currentNumberCuts* ) const [virtual]

Modify model according to information at node.

The routine modifies the model according to bound and basis change information at node and adds any cuts to the addCuts array.

Implements CbcNodeInfo.

#### 4.86.2.2 virtual CbcNodeInfo* CbcPartialNodeInfo::buildRowBasis ( CoinWarmStartBasis & *basis* ) const [virtual]

Builds up row basis backwards (until original model).

Returns NULL or previous one to apply . Depends on Free being 0 and impossible for cuts

Implements CbcNodeInfo.

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

- CbcPartialNodeInfo.hpp

## 4.87 CbcRounding Class Reference

Rounding class.

`#include <CbcHeuristic.hpp>`

Inheritance diagram for CbcRounding:

Collaboration diagram for CbcRounding:

### Public Member Functions

- CbcRounding & operator= (const CbcRounding &rhs)

  *Assignment operator.*
- virtual CbcHeuristic * clone () const

  *Clone.*
- virtual void generateCpp (FILE *fp)

  *Create C++ lines to get to current state.*
- virtual void resetModel (CbcModel *model)

  *Resets stuff if model changes.*
- virtual void setModel (CbcModel *model)

  *update model (This is needed if cliques update matrix etc)*
- virtual int solution (double &objectiveValue, double *newSolution)

  *returns 0 if no solution, 1 if valid solution with better objective value than one passed in Sets solution values if good, sets objective value (only if good) This is called after cuts have been added - so can not add cuts*
- virtual int solution (double &objectiveValue, double *newSolution, double solutionValue)

  *returns 0 if no solution, 1 if valid solution with better objective value than one passed in Sets solution values if good, sets objective value (only if good) This is called after cuts have been added - so can not add cuts Use solutionValue rather than solvers one*
- virtual void validate ()

  *Validate model i.e. sets when_ to 0 if necessary (may be NULL)*
- void setSeed (int value)

  *Set seed.*
- virtual bool shouldHeurRun (int whereFrom)

  *Check whether the heuristic should run at all 0 - before cuts at root node (or from doHeuristics) 1 - during cuts at root 2 - after root node cuts 3 - after cuts at other nodes 4 - during cuts at other nodes 8 added if previous heuristic in loop found solution.*

#### Additional Inherited Members

### 4.87.1 Detailed Description

Rounding class.

Definition at line 408 of file CbcHeuristic.hpp.

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

- CbcHeuristic.hpp

## 4.88 CbcRowCuts Class Reference

### 4.88.1 Detailed Description

Definition at line 134 of file CbcCountRowCut.hpp.

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

- CbcCountRowCut.hpp

## 4.89 CbcSerendipity Class Reference

heuristic - just picks up any good solution found by solver - see **OsiBabSolver**

`#include <CbcHeuristic.hpp>`

Inheritance diagram for CbcSerendipity:

Collaboration diagram for CbcSerendipity:

#### Public Member Functions

- CbcSerendipity & operator= (const CbcSerendipity &rhs)

    *Assignment operator.*
- virtual CbcHeuristic ∗ clone () const

    *Clone.*
- virtual void generateCpp (FILE ∗fp)

    *Create C++ lines to get to current state.*
- virtual void setModel (CbcModel ∗model)

    *update model*
- virtual int solution (double &objectiveValue, double ∗newSolution)

    *returns 0 if no solution, 1 if valid solution.*
- virtual void resetModel (CbcModel ∗model)

    *Resets stuff if model changes.*

## Additional Inherited Members

### 4.89.1 Detailed Description

heuristic - just picks up any good solution found by solver - see **OsiBabSolver**

Definition at line 562 of file CbcHeuristic.hpp.

### 4.89.2 Member Function Documentation

#### 4.89.2.1 virtual int CbcSerendipity::solution ( double & *objectiveValue,* double ∗ *newSolution* ) [virtual]

returns 0 if no solution, 1 if valid solution.

Sets solution values if good, sets objective value (only if good) We leave all variables which are at one at this node of the tree to that value and will initially set all others to zero. We then sort all variables in order of their cost divided by the number of entries in rows which are not yet covered. We randomize that value a bit so that ties will be broken in different ways on different runs of the heuristic. We then choose the best one and set it to one and repeat the exercise.

Implements CbcHeuristic.

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

- CbcHeuristic.hpp

## 4.90 CbcSimpleInteger Class Reference

Define a single integer class.

`#include <CbcSimpleInteger.hpp>`

Inheritance diagram for CbcSimpleInteger:

Collaboration diagram for CbcSimpleInteger:

### Public Member Functions

- virtual CbcObject ∗ clone () const

  *Clone.*
- **OsiSimpleInteger** ∗ osiObject () const

  *Construct an* **OsiSimpleInteger** *object.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗info, int &preferredWay) const

  *Infeasibility - large is 0.5.*
- virtual double feasibleRegion (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info) const

  *Set bounds to fix the variable at the current (integer) value.*
- virtual CbcBranchingObject ∗ createCbcBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way)

  *Create a branching object and indicate which way to branch first.*
- void fillCreateBranch (CbcIntegerBranchingObject ∗branching, const **OsiBranchingInformation** ∗info, int way)

  *Fills in a created branching object.*
- virtual **OsiSolverBranch** ∗ solverBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info) const

*Create an* **OsiSolverBranch** *object.*

- virtual void feasibleRegion ()

  *Set bounds to fix the variable at the current (integer) value.*
- virtual int columnNumber () const

  *Column number if single column object -1 otherwise, so returns >= 0 Used by heuristics.*
- void setColumnNumber (int value)

  *Set column number.*
- virtual void resetBounds (const **OsiSolverInterface** ∗solver)

  *Reset variable bounds to their original values.*
- virtual void resetSequenceEtc (int numberColumns, const int ∗originalColumns)

  *Change column numbers after preprocessing.*
- double originalLowerBound () const

  *Original bounds.*
- double breakEven () const

  *Breakeven e.g 0.7 -> >= 0.7 go up first.*
- void setBreakEven (double value)

  *Set breakeven e.g 0.7 -> >= 0.7 go up first.*

## Protected Attributes

- double originalLower_

  *data*
- double originalUpper_

  *Original upper bound.*
- double breakEven_

  *Breakeven i.e. >= this preferred is up.*
- int columnNumber_

  *Column number in model.*
- int preferredWay_

  *If -1 down always chosen first, +1 up always, 0 normal.*

### 4.90.1 Detailed Description

Define a single integer class.

Definition at line 167 of file CbcSimpleInteger.hpp.

### 4.90.2 Member Function Documentation

#### 4.90.2.1 virtual double CbcSimpleInteger::feasibleRegion ( **OsiSolverInterface** ∗ *solver*, const **OsiBranchingInformation** ∗ *info* ) const [virtual]

Set bounds to fix the variable at the current (integer) value.

Given an integer value, set the lower and upper bounds to fix the variable. Returns amount it had to move variable.

Reimplemented from CbcObject.

#### 4.90.2.2 virtual CbcBranchingObject* CbcSimpleInteger::createCbcBranch ( OsiSolverInterface * *solver*, const OsiBranchingInformation * *info*, int *way* ) [virtual]

Create a branching object and indicate which way to branch first.

The branching object has to know how to create branches (fix variables, etc.)

Reimplemented from CbcObject.

Reimplemented in CbcSimpleIntegerDynamicPseudoCost, and CbcSimpleIntegerPseudoCost.

#### 4.90.2.3 virtual OsiSolverBranch* CbcSimpleInteger::solverBranch ( OsiSolverInterface * *solver*, const OsiBranchingInformation * *info* ) const [virtual]

Create an **OsiSolverBranch** object.

This returns NULL if branch not represented by bound changes

#### 4.90.2.4 virtual void CbcSimpleInteger::feasibleRegion ( ) [virtual]

Set bounds to fix the variable at the current (integer) value.

Given an integer value, set the lower and upper bounds to fix the variable. The algorithm takes a bit of care in order to compensate for minor numerical inaccuracy.

Implements CbcObject.

#### 4.90.2.5 virtual void CbcSimpleInteger::resetBounds ( const OsiSolverInterface * *solver* ) [virtual]

Reset variable bounds to their original values.

Bounds may be tightened, so it may be good to be able to set this info in object.

Reimplemented from CbcObject.

### 4.90.3 Member Data Documentation

#### 4.90.3.1 double CbcSimpleInteger::originalLower_ [protected]

data

Original lower bound

Definition at line 275 of file CbcSimpleInteger.hpp.

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

- CbcSimpleInteger.hpp

## 4.91 CbcSimpleIntegerDynamicPseudoCost Class Reference

Define a single integer class but with dynamic pseudo costs.

`#include <CbcSimpleIntegerDynamicPseudoCost.hpp>`

Inheritance diagram for CbcSimpleIntegerDynamicPseudoCost:

Collaboration diagram for CbcSimpleIntegerDynamicPseudoCost:

## Public Member Functions

- virtual CbcObject ∗ clone () const

  *Clone.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗info, int &preferredWay) const

  *Infeasibility - large is 0.5.*
- virtual CbcBranchingObject ∗ createCbcBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way)

  *Creates a branching object.*
- virtual CbcObjectUpdateData createUpdateInformation (const **OsiSolverInterface** ∗solver, const CbcNode ∗node, const CbcBranchingObject ∗branchingObject)

  *Fills in a created branching object.*
- virtual void updateInformation (const CbcObjectUpdateData &data)

  *Update object by CbcObjectUpdateData.*
- void copySome (const CbcSimpleIntegerDynamicPseudoCost ∗otherObject)

  *Copy some information i.e. just variable stuff.*
- virtual void updateBefore (const **OsiObject** ∗rhs)

  *Updates stuff like pseudocosts before threads.*
- virtual void updateAfter (const **OsiObject** ∗rhs, const **OsiObject** ∗baseObject)

  *Updates stuff like pseudocosts after threads finished.*
- void updateAfterMini (int numberDown, int numberDownInfeasible, double sumDown, int numberUp, int number←UpInfeasible, double sumUp)

  *Updates stuff like pseudocosts after mini branch and bound.*
- virtual **OsiSolverBranch** ∗ solverBranch () const

  *Create an* **OsiSolverBranch** *object.*
- double downDynamicPseudoCost () const

  *Down pseudo cost.*
- void setDownDynamicPseudoCost (double value)

  *Set down pseudo cost.*
- void updateDownDynamicPseudoCost (double value)

  *Modify down pseudo cost in a slightly different way.*
- double upDynamicPseudoCost () const

  *Up pseudo cost.*
- void setUpDynamicPseudoCost (double value)

  *Set up pseudo cost.*
- void updateUpDynamicPseudoCost (double value)

  *Modify up pseudo cost in a slightly different way.*
- double downShadowPrice () const

  *Down pseudo shadow price cost.*
- void setDownShadowPrice (double value)

  *Set down pseudo shadow price cost.*
- double upShadowPrice () const

  *Up pseudo shadow price cost.*
- void setUpShadowPrice (double value)

  *Set up pseudo shadow price cost.*

- double upDownSeparator () const

  *Up down separator.*
- void setUpDownSeparator (double value)

  *Set up down separator.*
- double sumDownCost () const

  *Down sum cost.*
- void setSumDownCost (double value)

  *Set down sum cost.*
- void addToSumDownCost (double value)

  *Add to down sum cost and set last and square.*
- double sumUpCost () const

  *Up sum cost.*
- void setSumUpCost (double value)

  *Set up sum cost.*
- void addToSumUpCost (double value)

  *Add to up sum cost and set last and square.*
- double sumDownChange () const

  *Down sum change.*
- void setSumDownChange (double value)

  *Set down sum change.*
- void addToSumDownChange (double value)

  *Add to down sum change.*
- double sumUpChange () const

  *Up sum change.*
- void setSumUpChange (double value)

  *Set up sum change.*
- void addToSumUpChange (double value)

  *Add to up sum change and set last and square.*
- double sumDownDecrease () const

  *Sum down decrease number infeasibilities from strong or actual.*
- void setSumDownDecrease (double value)

  *Set sum down decrease number infeasibilities from strong or actual.*
- void addToSumDownDecrease (double value)

  *Add to sum down decrease number infeasibilities from strong or actual.*
- double sumUpDecrease () const

  *Sum up decrease number infeasibilities from strong or actual.*
- void setSumUpDecrease (double value)

  *Set sum up decrease number infeasibilities from strong or actual.*
- void addToSumUpDecrease (double value)

  *Add to sum up decrease number infeasibilities from strong or actual.*
- int numberTimesDown () const

  *Down number times.*
- void setNumberTimesDown (int value)

  *Set down number times.*
- void incrementNumberTimesDown ()

  *Increment down number times.*
- int numberTimesUp () const

*Up number times.*

- void setNumberTimesUp (int value)

  *Set up number times.*

- void incrementNumberTimesUp ()

  *Increment up number times.*

- int numberTimesBranched () const

  *Number times branched.*

- int numberTimesDownInfeasible () const

  *Down number times infeasible.*

- void setNumberTimesDownInfeasible (int value)

  *Set down number times infeasible.*

- void incrementNumberTimesDownInfeasible ()

  *Increment down number times infeasible.*

- int numberTimesUpInfeasible () const

  *Up number times infeasible.*

- void setNumberTimesUpInfeasible (int value)

  *Set up number times infeasible.*

- void incrementNumberTimesUpInfeasible ()

  *Increment up number times infeasible.*

- int numberBeforeTrust () const

  *Number of times before trusted.*

- void setNumberBeforeTrust (int value)

  *Set number of times before trusted.*

- void incrementNumberBeforeTrust ()

  *Increment number of times before trusted.*

- virtual double upEstimate () const

  *Return "up" estimate.*

- virtual double downEstimate () const

  *Return "down" estimate (default 1.0e-5)*

- int method () const

  *method - see below for details*

- void setMethod (int value)

  *Set method.*

- void setDownInformation (double changeObjectiveDown, int changeInfeasibilityDown)

  *Pass in information on a down branch.*

- void setUpInformation (double changeObjectiveUp, int changeInfeasibilityUp)

  *Pass in information on a up branch.*

- void setProbingInformation (int fixedDown, int fixedUp)

  *Pass in probing information.*

- void print (int type=0, double value=0.0) const

  *Print - 0 -summary, 1 just before strong.*

- bool same (const CbcSimpleIntegerDynamicPseudoCost ∗obj) const

  *Same - returns true if contents match(ish)*

### Protected Attributes

- double downDynamicPseudoCost_

    *data*
- double upDynamicPseudoCost_

    *Up pseudo cost.*
- double upDownSeparator_

    *Up/down separator If >0.0 then do first branch up if value-floor(value) >= this value.*
- double sumDownCost_

    *Sum down cost from strong or actual.*
- double sumUpCost_

    *Sum up cost from strong or actual.*
- double sumDownChange_

    *Sum of all changes to x when going down.*
- double sumUpChange_

    *Sum of all changes to x when going up.*
- double downShadowPrice_

    *Current pseudo-shadow price estimate down.*
- double upShadowPrice_

    *Current pseudo-shadow price estimate up.*
- double sumDownDecrease_

    *Sum down decrease number infeasibilities from strong or actual.*
- double sumUpDecrease_

    *Sum up decrease number infeasibilities from strong or actual.*
- double lastDownCost_

    *Last down cost from strong (i.e. as computed by last strong)*
- double lastUpCost_

    *Last up cost from strong (i.e. as computed by last strong)*
- int lastDownDecrease_

    *Last down decrease number infeasibilities from strong (i.e. as computed by last strong)*
- int lastUpDecrease_

    *Last up decrease number infeasibilities from strong (i.e. as computed by last strong)*
- int numberTimesDown_

    *Number of times we have gone down.*
- int numberTimesUp_

    *Number of times we have gone up.*
- int numberTimesDownInfeasible_

    *Number of times we have been infeasible going down.*
- int numberTimesUpInfeasible_

    *Number of times we have been infeasible going up.*
- int numberBeforeTrust_

    *Number of branches before we trust.*
- int numberTimesDownLocalFixed_

    *Number of local probing fixings going down.*
- int numberTimesUpLocalFixed_

    *Number of local probing fixings going up.*
- double numberTimesDownTotalFixed_

*Number of total probing fixings going down.*

- double numberTimesUpTotalFixed_

  *Number of total probing fixings going up.*

- int numberTimesProbingTotal_

  *Number of times probing done.*

- int method_

  *Number of times infeasible when tested.*

### 4.91.1 Detailed Description

Define a single integer class but with dynamic pseudo costs.

Based on work by Achterberg, Koch and Martin.

It is wild overkill but to keep design all twiddly things are in each. This could be used for fine tuning.

Definition at line 35 of file CbcSimpleIntegerDynamicPseudoCost.hpp.

### 4.91.2 Member Function Documentation

#### 4.91.2.1 virtual **CbcObjectUpdateData** CbcSimpleIntegerDynamicPseudoCost::createUpdateInformation ( const **OsiSolverInterface** ∗ *solver*, const **CbcNode** ∗ *node*, const **CbcBranchingObject** ∗ *branchingObject* ) [virtual]

Fills in a created branching object.

Pass in information on branch just done and create CbcObjectUpdateData instance. If object does not need data then backward pointer will be NULL. Assumes can get information from solver

Reimplemented from CbcObject.

#### 4.91.2.2 virtual **OsiSolverBranch**∗ CbcSimpleIntegerDynamicPseudoCost::solverBranch ( ) const [virtual]

Create an **OsiSolverBranch** object.

This returns NULL if branch not represented by bound changes

Reimplemented from CbcObject.

### 4.91.3 Member Data Documentation

#### 4.91.3.1 double CbcSimpleIntegerDynamicPseudoCost::downDynamicPseudoCost_ [protected]

data

Down pseudo cost

Definition at line 324 of file CbcSimpleIntegerDynamicPseudoCost.hpp.

#### 4.91.3.2 int CbcSimpleIntegerDynamicPseudoCost::method_ [protected]

Number of times infeasible when tested.

Method - 0 - pseudo costs 1 - probing

Definition at line 381 of file CbcSimpleIntegerDynamicPseudoCost.hpp.

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

- CbcSimpleIntegerDynamicPseudoCost.hpp

## 4.92 CbcSimpleIntegerPseudoCost Class Reference

Define a single integer class but with pseudo costs.

`#include <CbcSimpleIntegerPseudoCost.hpp>`

Inheritance diagram for CbcSimpleIntegerPseudoCost:

Collaboration diagram for CbcSimpleIntegerPseudoCost:

### Public Member Functions

- virtual CbcObject * clone () const

  *Clone.*
- virtual double infeasibility (const **OsiBranchingInformation** *info, int &preferredWay) const

  *Infeasibility - large is 0.5.*
- virtual CbcBranchingObject * createCbcBranch (**OsiSolverInterface** *solver, const **OsiBranchingInformation** *info, int way)

  *Creates a branching object.*
- double downPseudoCost () const

  *Down pseudo cost.*
- void setDownPseudoCost (double value)

  *Set down pseudo cost.*
- double upPseudoCost () const

  *Up pseudo cost.*
- void setUpPseudoCost (double value)

  *Set up pseudo cost.*
- double upDownSeparator () const

  *Up down separator.*
- void setUpDownSeparator (double value)

  *Set up down separator.*
- virtual double upEstimate () const

  *Return "up" estimate.*
- virtual double downEstimate () const

  *Return "down" estimate (default 1.0e-5)*
- int method () const

  *method - see below for details*
- void setMethod (int value)

  *Set method.*

### Protected Attributes

- double downPseudoCost_

  *data*
- double upPseudoCost_

  *Up pseudo cost.*
- double upDownSeparator_

  *Up/down separator If >0.0 then do first branch up if value-floor(value) >= this value.*
- int method_

  *Method - 0 - normal - return min (up,down) 1 - if before any solution return CoinMax(up,down) 2 - if before branched solution return CoinMax(up,down) 3 - always return CoinMax(up,down)*

### 4.92.1 Detailed Description

Define a single integer class but with pseudo costs.

Definition at line 14 of file CbcSimpleIntegerPseudoCost.hpp.

### 4.92.2 Member Data Documentation

#### 4.92.2.1 double CbcSimpleIntegerPseudoCost::downPseudoCost_ [protected]

data

Down pseudo cost

Definition at line 95 of file CbcSimpleIntegerPseudoCost.hpp.

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

- CbcSimpleIntegerPseudoCost.hpp

## 4.93 CbcSolver Class Reference

This allows the use of the standalone solver in a flexible manner.

`#include <CbcSolver.hpp>`

### Public Member Functions

**Solve method**

- int solve (int argc, const char ∗argv[ ], int returnMode)

  *This takes a list of commands, does "stuff" and returns returnMode - 0 model and solver untouched - babModel updated 1 model updated - just with solution basis etc 2 model updated i.e.*
- int solve (const char ∗input, int returnMode)

  *This takes a list of commands, does "stuff" and returns returnMode - 0 model and solver untouched - babModel updated 1 model updated - just with solution basis etc 2 model updated i.e.*

**Constructors and destructors etc**

- CbcSolver ()

*Default Constructor.*
- CbcSolver (const **OsiClpSolverInterface** &)

  *Constructor from solver.*
- CbcSolver (const CbcModel &)

  *Constructor from model.*
- CbcSolver (const CbcSolver &rhs)

  *Copy constructor .*
- CbcSolver & operator= (const CbcSolver &rhs)

  *Assignment operator.*
- ∼CbcSolver ()

  *Destructor.*
- void fillParameters ()

  *Fill with standard parameters.*
- void fillValuesInSolver ()

  *Set default values in solvers from parameters.*
- void addUserFunction (CbcUser ∗function)

  *Add user function.*
- void setUserCallBack (CbcStopNow ∗function)

  *Set user call back.*
- void addCutGenerator (**CglCutGenerator** ∗generator)

  *Add cut generator.*

### miscellaneous methods to line up with old

- int ∗ **analyze** (**OsiClpSolverInterface** ∗solverMod, int &numberChanged, double &increment, bool changeInt, **CoinMessageHandler** ∗generalMessageHandler)
- void updateModel (**ClpSimplex** ∗model2, int returnMode)

  *1 - add heuristics to model 2 - do heuristics (and set cutoff and best solution) 3 - for miplib test so skip some (out model later)*

### useful stuff

- int intValue (CbcOrClpParameterType type) const

  *Get int value.*
- void setIntValue (CbcOrClpParameterType type, int value)

  *Set int value.*
- double doubleValue (CbcOrClpParameterType type) const

  *Get double value.*
- void setDoubleValue (CbcOrClpParameterType type, double value)

  *Set double value.*
- CbcUser ∗ userFunction (const char ∗name) const

  *User function (NULL if no match)*
- CbcModel ∗ model ()

  *Return original Cbc model.*
- CbcModel ∗ babModel ()

  *Return updated Cbc model.*
- int numberUserFunctions () const

  *Number of userFunctions.*
- CbcUser ∗∗ userFunctionArray () const

  *User function array.*
- **OsiClpSolverInterface** ∗ originalSolver () const

  *Copy of model on initial load (will contain output solutions)*

- **CoinModel** ∗ originalCoinModel () const

  *Copy of model on initial load.*
- void setOriginalSolver (**OsiClpSolverInterface** ∗originalSolver)

  *Copy of model on initial load (will contain output solutions)*
- void setOriginalCoinModel (**CoinModel** ∗originalCoinModel)

  *Copy of model on initial load.*
- int numberCutGenerators () const

  *Number of cutgenerators.*
- **CglCutGenerator** ∗∗ cutGeneratorArray () const

  *Cut generator array.*
- double startTime () const

  *Start time.*
- void setPrinting (bool onOff)

  *Whether to print to std::cout.*
- void setReadMode (int value)

  *Where to start reading commands.*

### 4.93.1 Detailed Description

This allows the use of the standalone solver in a flexible manner.

It has an original **OsiClpSolverInterface** and CbcModel which it can use repeatedly, e.g., to get a heuristic solution and then start again.

So I [jjf] will need a primitive scripting language which can then call solve and manipulate solution value and solution arrays.

Also provides for user callback functions. Currently two ideas in gestation, CbcUser and CbcStopNow. The latter seems limited to deciding whether or not to stop. The former seems completely general, with a notion of importing and exporting, and a 'solve', which should be interpreted as 'do whatever this user function does'.

Parameter initialisation is at last centralised in fillParameters().

Definition at line 56 of file CbcSolver.hpp.

### 4.93.2 Member Function Documentation

#### 4.93.2.1 int CbcSolver::solve ( int *argc*, const char ∗ *argv[ ]*, int *returnMode* )

This takes a list of commands, does "stuff" and returns returnMode - 0 model and solver untouched - babModel updated 1 model updated - just with solution basis etc 2 model updated i.e.

as babModel (babModel NULL) (only use without preprocessing)

#### 4.93.2.2 int CbcSolver::solve ( const char ∗ *input*, int *returnMode* )

This takes a list of commands, does "stuff" and returns returnMode - 0 model and solver untouched - babModel updated 1 model updated - just with solution basis etc 2 model updated i.e.

as babModel (babModel NULL) (only use without preprocessing)

#### 4.93.2.3 void CbcSolver::fillValuesInSolver ( )

Set default values in solvers from parameters.

Misleading. The current code actually reads default values from the underlying solvers and installs them as default values for a subset of parameters in #parameters_.

#### 4.93.2.4 void CbcSolver::updateModel ( ClpSimplex * *model2*, int *returnMode* )

1 - add heuristics to model 2 - do heuristics (and set cutoff and best solution) 3 - for miplib test so skip some (out model later)

Updates model_ from babModel_ according to returnMode returnMode - 0 model and solver untouched - babModel updated 1 model updated - just with solution basis etc 2 model updated i.e. as babModel (babModel NULL) (only use without preprocessing)

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

- CbcSolver.hpp

## 4.94 CbcSolverUsefulData Class Reference

The CbcSolver class was taken out at a 9/12/09 meeting This is a feeble replacement.

`#include <CbcSolver.hpp>`

Collaboration diagram for CbcSolverUsefulData:

### Public Member Functions

**Constructors and destructors etc**

- CbcSolverUsefulData ()

  *Default Constructor.*
- CbcSolverUsefulData (const CbcSolverUsefulData &rhs)

  *Copy constructor .*
- CbcSolverUsefulData & operator= (const CbcSolverUsefulData &rhs)

  *Assignment operator.*
- ∼CbcSolverUsefulData ()

  *Destructor.*

### Public Attributes

**Member data**

- double **totalTime_**
- **CbcOrClpParam parameters_** [CBCMAXPARAMETERS]
- bool **noPrinting_**
- bool **useSignalHandler_**
- int **numberParameters_**
- int **initialPumpTune_**

### 4.94.1 Detailed Description

The CbcSolver class was taken out at a 9/12/09 meeting This is a feeble replacement.

At present everything is public

Definition at line 262 of file CbcSolver.hpp.

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

- CbcSolver.hpp

## 4.95 CbcSolverUsefulData2 Struct Reference

Structure to hold useful arrays.

`#include <CbcSolver.hpp>`

### 4.95.1 Detailed Description

Structure to hold useful arrays.

Definition at line 240 of file CbcSolver.hpp.

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

- CbcSolver.hpp

## 4.96 CbcSOS Class Reference

Branching object for Special Ordered Sets of type 1 and 2.

`#include <CbcSOS.hpp>`

Inheritance diagram for CbcSOS:

Collaboration diagram for CbcSOS:

### Public Member Functions

- CbcSOS (CbcModel ∗model, int numberMembers, const int ∗which, const double ∗weights, int identifier, int type=1)

  *Constructor with SOS type and member information.*
- virtual CbcObject ∗ clone () const

  *Clone.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗info, int &preferredWay) const

  *Infeasibility - large is 0.5.*
- virtual void feasibleRegion ()

  *This looks at solution and sets bounds to contain solution.*
- virtual CbcBranchingObject ∗ createCbcBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way)

  *Creates a branching object.*

- virtual CbcObjectUpdateData createUpdateInformation (const **OsiSolverInterface** ∗solver, const CbcNode ∗node, const CbcBranchingObject ∗branchingObject)

  *Pass in information on branch just done and create CbcObjectUpdateData instance.*
- virtual void updateInformation (const CbcObjectUpdateData &data)

  *Update object by CbcObjectUpdateData.*
- virtual **OsiSolverBranch** ∗ solverBranch () const

  *Create an* **OsiSolverBranch** *object.*
- virtual void redoSequenceEtc (CbcModel ∗model, int numberColumns, const int ∗originalColumns)

  *Redoes data when sequence numbers change.*
- **OsiSOS** ∗ osiObject (const **OsiSolverInterface** ∗solver) const

  *Construct an* **OsiSOS** *object.*
- int numberMembers () const

  *Number of members.*
- const int ∗ members () const

  *Members (indices in range 0 ... numberColumns-1)*
- int sosType () const

  *SOS type.*
- int numberTimesDown () const

  *Down number times.*
- int numberTimesUp () const

  *Up number times.*
- const double ∗ weights () const

  *Array of weights.*
- void setNumberMembers (int n)

  *Set number of members.*
- int ∗ mutableMembers () const

  *Members (indices in range 0 ... numberColumns-1)*
- double ∗ mutableWeights () const

  *Array of weights.*
- virtual bool canDoHeuristics () const

  *Return true if object can take part in normal heuristics.*
- void setIntegerValued (bool yesNo)

  *Set whether set is integer valued or not.*

## Additional Inherited Members

### 4.96.1 Detailed Description

Branching object for Special Ordered Sets of type 1 and 2.

SOS1 are an ordered set of variables where at most one variable can be non-zero. SOS1 are commonly defined with binary variables (interpreted as selection between alternatives) but this is not necessary. An SOS1 with all binary variables is a special case of a clique (setting any one variable to 1 forces all others to 0).

In theory, the implementation makes no assumptions about integrality in Type 1 sets. In practice, there are places where the code seems to have been written with a binary SOS mindset. Current development of SOS branching objects is proceeding in **OsiSOS**.

SOS2 are an ordered set of variables in which at most two consecutive variables can be non-zero and must sum to 1 (interpreted as interpolation between two discrete values). By definition the variables are non-integer.

Definition at line 29 of file CbcSOS.hpp.

### 4.96.2 Constructor & Destructor Documentation

#### 4.96.2.1 CbcSOS::CbcSOS ( **CbcModel** * *model*, int *numberMembers*, const int * *which*, const double * *weights*, int *identifier*, int *type* = 1 )

Constructor with SOS type and member information.

Type specifies SOS 1 or 2. Identifier is an arbitrary value.

Which should be an array of variable indices with numberMembers entries. Weights can be used to assign arbitrary weights to variables, in the order they are specified in which. If no weights are provided, a default array of 0, 1, 2, ... is generated.

### 4.96.3 Member Function Documentation

#### 4.96.3.1 virtual **CbcObjectUpdateData** CbcSOS::createUpdateInformation ( const **OsiSolverInterface** * *solver*, const **CbcNode** * *node*, const **CbcBranchingObject** * *branchingObject* ) [virtual]

Pass in information on branch just done and create CbcObjectUpdateData instance.

If object does not need data then backward pointer will be NULL. Assumes can get information from solver

Reimplemented from CbcObject.

#### 4.96.3.2 virtual **OsiSolverBranch*** CbcSOS::solverBranch ( ) const [virtual]

Create an **OsiSolverBranch** object.

This returns NULL if branch not represented by bound changes

Reimplemented from CbcObject.

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

- CbcSOS.hpp

## 4.97 CbcSOSBranchingObject Class Reference

Branching object for Special ordered sets.

`#include <CbcSOS.hpp>`

Inheritance diagram for CbcSOSBranchingObject:

Collaboration diagram for CbcSOSBranchingObject:

### Public Member Functions

- virtual CbcBranchingObject * clone () const
  *Clone.*
- virtual double branch ()
  *Does next branch and updates state.*
- virtual void fix (**OsiSolverInterface** *solver, double *lower, double *upper, int branchState) const
  *Update bounds in solver as in 'branch' and update given bounds.*

- virtual void previousBranch ()

  *Reset every information so that the branching object appears to point to the previous child.*
- virtual void print ()

  *Print something about branch - only if log level high.*
- virtual CbcBranchObjType type () const

  *Return the type (an integer identifier) of `this`.*
- virtual int compareOriginalObject (const CbcBranchingObject ∗brObj) const

  *Compare the original object of `this` with the original object of `brObj`.*
- virtual CbcRangeCompare compareBranchingObject (const CbcBranchingObject ∗brObj, const bool replaceIf↩ Overlap=false)

  *Compare the `this` with `brObj`.*
- void computeNonzeroRange ()

  *Fill out the `firstNonzero_` and `lastNonzero_` data members.*

### Additional Inherited Members

### 4.97.1 Detailed Description

Branching object for Special ordered sets.

Variable_ is the set id number (redundant, as the object also holds a pointer to the set.

Definition at line 191 of file CbcSOS.hpp.

### 4.97.2 Member Function Documentation

#### 4.97.2.1 virtual void CbcSOSBranchingObject::fix ( OsiSolverInterface ∗ *solver,* double ∗ *lower,* double ∗ *upper,* int *branchState* ) const [virtual]

Update bounds in solver as in 'branch' and update given bounds.

branchState is -1 for 'down' +1 for 'up'

Reimplemented from CbcBranchingObject.

#### 4.97.2.2 virtual void CbcSOSBranchingObject::previousBranch ( ) [inline], [virtual]

Reset every information so that the branching object appears to point to the previous child.

This method does not need to modify anything in any solver.

Reimplemented from CbcBranchingObject.

Definition at line 227 of file CbcSOS.hpp.

#### 4.97.2.3 virtual int CbcSOSBranchingObject::compareOriginalObject ( const CbcBranchingObject ∗ *brObj* ) const [virtual]

Compare the original object of `this` with the original object of `brObj`.

Assumes that there is an ordering of the original objects. This method should be invoked only if `this` and brObj are of the same type. Return negative/0/positive depending on whether `this` is smaller/same/larger than the argument.

Reimplemented from CbcBranchingObject.

#### 4.97.2.4 virtual CbcRangeCompare CbcSOSBranchingObject::compareBranchingObject ( const CbcBranchingObject * *brObj*, const bool *replaceIfOverlap* = false ) [virtual]

Compare the `this` with `brObj`.

`this` and `brObj` must be os the same type and must have the same original object, but they may have different feasible regions. Return the appropriate CbcRangeCompare value (first argument being the sub/superset if that's the case). In case of overlap (and if `replaceIfOverlap` is true) replace the current branching object with one whose feasible region is the overlap.

Implements CbcBranchingObject.

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

- CbcSOS.hpp

## 4.98 CbcStatistics Class Reference

For gathering statistics.

`#include <CbcStatistics.hpp>`

### Protected Attributes

- double value_

  *Value.*
- double startingObjective_

  *Starting objective.*
- double endingObjective_

  *Ending objective.*
- int id_

  *id*
- int parentId_

  *parent id*
- int way_

  *way -1 or +1 is first branch -10 or +10 is second branch*
- int sequence_

  *sequence number branched on*
- int depth_

  *depth*
- int startingInfeasibility_

  *starting number of integer infeasibilities*
- int endingInfeasibility_

  *ending number of integer infeasibilities*
- int numberIterations_

  *number of iterations*

### 4.98.1 Detailed Description

For gathering statistics.

Definition at line 13 of file CbcStatistics.hpp.

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

- CbcStatistics.hpp

## 4.99 CbcStopNow Class Reference

Support the use of a call back class to decide whether to stop.

`#include <CbcSolver.hpp>`

### Public Member Functions

**Decision methods**

- virtual int callBack (CbcModel ∗, int)

  *Import.*

**Constructors and destructors etc**

- CbcStopNow ()

  *Default Constructor.*
- CbcStopNow (const CbcStopNow &rhs)

  *Copy constructor .*
- CbcStopNow & operator= (const CbcStopNow &rhs)

  *Assignment operator.*
- virtual CbcStopNow ∗ clone () const

  *Clone.*
- virtual ∼CbcStopNow ()

  *Destructor.*

### 4.99.1 Detailed Description

Support the use of a call back class to decide whether to stop.

Definitely under construction.

Definition at line 399 of file CbcSolver.hpp.

### 4.99.2 Member Function Documentation

#### 4.99.2.1 virtual int CbcStopNow::callBack ( CbcModel ∗ , int ) [inline], [virtual]

Import.

Values for whereFrom:

- 1 after initial solve by dualsimplex etc

- 2 after preprocessing
- 3 just before branchAndBound (so user can override)
- 4 just after branchAndBound (before postprocessing)
- 5 after postprocessing
- 6 after a user called heuristic phase

**Returns**

0 if good nonzero return code to stop

Definition at line 417 of file CbcSolver.hpp.

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

- CbcSolver.hpp

## 4.100 CbcStrategy Class Reference

Strategy base class.

`#include <CbcStrategy.hpp>`

Inheritance diagram for CbcStrategy:

Collaboration diagram for CbcStrategy:

### Public Member Functions

- virtual CbcStrategy * clone () const =0

  *Clone.*
- virtual void setupCutGenerators (CbcModel &model)=0

  *Setup cut generators.*
- virtual void setupHeuristics (CbcModel &model)=0

  *Setup heuristics.*
- virtual void setupPrinting (CbcModel &model, int modelLogLevel)=0

  *Do printing stuff.*
- virtual void setupOther (CbcModel &model)=0

  *Other stuff e.g. strong branching and preprocessing.*
- void setNested (int depth)

  *Set model depth (i.e. how nested)*
- int getNested () const

  *Get model depth (i.e. how nested)*
- void setPreProcessState (int state)

  *Say preProcessing done.*
- int preProcessState () const

  *See what sort of preprocessing was done.*
- **CglPreProcess** * process () const

  *Pre-processing object.*

- void deletePreProcess ()

  *Delete pre-processing object to save memory.*
- virtual CbcNodeInfo * fullNodeInfo (CbcModel *model, int numberRowsAtContinuous) const

  *Return a new Full node information pointer (descendant of CbcFullNodeInfo)*
- virtual CbcNodeInfo * partialNodeInfo (CbcModel *model, CbcNodeInfo *parent, CbcNode *owner, int number←ChangedBounds, const int *variables, const double *boundChanges, const **CoinWarmStartDiff** *basisDiff) const

  *Return a new Partial node information pointer (descendant of CbcPartialNodeInfo)*
- virtual void generateCpp (FILE *)

  *Create C++ lines to get to current state.*
- virtual int status (CbcModel *model, CbcNodeInfo *parent, int whereFrom)

  *After a CbcModel::resolve this can return a status -1 no effect 0 treat as optimal 1 as 0 but do not do any more resolves (i.e.*

### Protected Attributes

- int depth_

  *Model depth.*
- int preProcessState_

  *PreProcessing state - -1 infeasible 0 off 1 was done (so need post-processing)*
- **CglPreProcess** * process_

  *If preprocessing then this is object.*

### 4.100.1 Detailed Description

Strategy base class.

Definition at line 18 of file CbcStrategy.hpp.

### 4.100.2 Member Function Documentation

#### 4.100.2.1 virtual int CbcStrategy::status ( **CbcModel** * *model,* **CbcNodeInfo** * *parent,* int *whereFrom* ) `[virtual]`

After a CbcModel::resolve this can return a status -1 no effect 0 treat as optimal 1 as 0 but do not do any more resolves (i.e.

no more cuts) 2 treat as infeasible

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

- CbcStrategy.hpp

## 4.101 CbcStrategyDefault Class Reference

Default class.

`#include <CbcStrategy.hpp>`

Inheritance diagram for CbcStrategyDefault:

Collaboration diagram for CbcStrategyDefault:

### Public Member Functions

- virtual CbcStrategy * clone () const

  *Clone.*
- virtual void setupCutGenerators (CbcModel &model)

  *Setup cut generators.*
- virtual void setupHeuristics (CbcModel &model)

  *Setup heuristics.*
- virtual void setupPrinting (CbcModel &model, int modelLogLevel)

  *Do printing stuff.*
- virtual void setupOther (CbcModel &model)

  *Other stuff e.g. strong branching.*
- void setupPreProcessing (int desired=1, int passes=10)

  *Set up preProcessing - see below.*
- int desiredPreProcess () const

  *See what sort of preprocessing wanted.*
- int preProcessPasses () const

  *See how many passes wanted.*
- virtual void generateCpp (FILE *fp)

  *Create C++ lines to get to current state.*

### Protected Attributes

- int desiredPreProcess_

  *Desired pre-processing 0 - none 1 - ordinary 2 - find sos 3 - find cliques 4 - more aggressive sos 5 - add integer slacks.*
- int preProcessPasses_

  *Number of pre-processing passes.*

### 4.101.1 Detailed Description

Default class.

Definition at line 131 of file CbcStrategy.hpp.

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

- CbcStrategy.hpp

## 4.102 CbcStrategyDefaultSubTree Class Reference

Default class for sub trees.

`#include <CbcStrategy.hpp>`

Inheritance diagram for CbcStrategyDefaultSubTree:

Collaboration diagram for CbcStrategyDefaultSubTree:

#### Public Member Functions

- virtual CbcStrategy * clone () const

  *Clone.*
- virtual void setupCutGenerators (CbcModel &model)

  *Setup cut generators.*
- virtual void setupHeuristics (CbcModel &model)

  *Setup heuristics.*
- virtual void setupPrinting (CbcModel &model, int modelLogLevel)

  *Do printing stuff.*
- virtual void setupOther (CbcModel &model)

  *Other stuff e.g. strong branching.*

#### Additional Inherited Members

### 4.102.1 Detailed Description

Default class for sub trees.

Definition at line 209 of file CbcStrategy.hpp.

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

- CbcStrategy.hpp

## 4.103 CbcStrategyNull Class Reference

Null class.

`#include <CbcStrategy.hpp>`

Inheritance diagram for CbcStrategyNull:

Collaboration diagram for CbcStrategyNull:

#### Public Member Functions

- virtual CbcStrategy * clone () const

  *Clone.*
- virtual void setupCutGenerators (CbcModel &)

  *Setup cut generators.*
- virtual void setupHeuristics (CbcModel &)

  *Setup heuristics.*
- virtual void setupPrinting (CbcModel &, int)

  *Do printing stuff.*
- virtual void setupOther (CbcModel &)

  *Other stuff e.g. strong branching.*

### Additional Inherited Members

### 4.103.1 Detailed Description

Null class.

Definition at line 95 of file CbcStrategy.hpp.

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

- CbcStrategy.hpp

## 4.104 CbcStrongInfo Struct Reference

Abstract base class for 'objects'.

`#include <CbcObject.hpp>`

Collaboration diagram for CbcStrongInfo:

### 4.104.1 Detailed Description

Abstract base class for 'objects'.

It now just has stuff that **OsiObject** does not have

The branching model used in Cbc is based on the idea of an *object*. In the abstract, an object is something that has a feasible region, can be evaluated for infeasibility, can be branched on (*i.e.*, there's some constructive action to be taken to move toward feasibility), and allows comparison of the effect of branching.

This class (CbcObject) is the base class for an object. To round out the branching model, the class CbcBranchingObject describes how to perform a branch, and the class CbcBranchDecision describes how to compare two CbcBranching↩ Objects.

To create a new type of object you need to provide three methods: #infeasibility(), #feasibleRegion(), and #createCbc↩ Branch(), described below.

This base class is primarily virtual to allow for any form of structure. Any form of discontinuity is allowed.

Definition at line 51 of file CbcObject.hpp.

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

- CbcObject.hpp

## 4.105 CbcSymmetry Class Reference

Class to deal with symmetry.

`#include <CbcSymmetry.hpp>`

Collaboration diagram for CbcSymmetry:

### Public Member Functions

- int orbitalFixing (**OsiSolverInterface** ∗solver)

*Fixes variables using orbits (returns number fixed)*

- void setupSymmetry (const **OsiSolverInterface** &solver)

  *empty if no NTY, symmetry data structure setup otherwise*

**Constructors and destructors**

- CbcSymmetry ()

  *Default constructor.*
- CbcSymmetry (const CbcSymmetry &)

  *Copy constructor.*
- CbcSymmetry & operator= (const CbcSymmetry &rhs)

  *Assignment operator.*
- ∼CbcSymmetry ()

  *Destructor.*

### 4.105.1 Detailed Description

Class to deal with symmetry.

Hacked from Couenne Thanks, but it had been nice to make sure that there are no symbol collisions when building Couenne with this Cbc.

Definition at line 69 of file CbcSymmetry.hpp.

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

- CbcSymmetry.hpp

## 4.106 CbcThread Class Reference

A class to encapsulate thread stuff.

`#include <CbcThread.hpp>`

### 4.106.1 Detailed Description

A class to encapsulate thread stuff.

Definition at line 426 of file CbcThread.hpp.

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

- CbcThread.hpp

## 4.107 CbcTree Class Reference

Using MS heap implementation.

`#include <CbcTree.hpp>`

Inheritance diagram for CbcTree:

Collaboration diagram for CbcTree:

## Public Member Functions

### Constructors and related

- CbcTree ()
  *Default Constructor.*
- CbcTree (const CbcTree &rhs)
  *Copy constructor.*
- CbcTree & operator= (const CbcTree &rhs)
  *= operator*
- virtual ∼CbcTree ()
  *Destructor.*
- virtual CbcTree ∗ clone () const
  *Clone.*
- virtual void generateCpp (FILE ∗)
  *Create C++ lines to get to current state.*

### Heap access and maintenance methods

- void setComparison (CbcCompareBase &compare)
  *Set comparison function and resort heap.*
- virtual CbcNode ∗ top () const
  *Return the top node of the heap.*
- virtual void push (CbcNode ∗x)
  *Add a node to the heap.*
- virtual void pop ()
  *Remove the top node from the heap.*
- virtual CbcNode ∗ bestNode (double cutoff)
  *Gets best node and takes off heap.*
- virtual void rebuild ()
  *Rebuild the heap.*

### Direct node access methods

- virtual bool empty ()
  *Test for an empty tree.*
- virtual int size () const
  *Return size.*
- CbcNode ∗ operator[ ] (int i) const
  *Return a node pointer.*
- CbcNode ∗ nodePointer (int i) const
  *Return a node pointer.*
- void **realpop** ()
- void fixTop ()
  *After changing data in the top node, fix the heap.*
- void **realpush** (CbcNode ∗node)

### Search tree maintenance

- virtual void cleanTree (CbcModel ∗model, double cutoff, double &bestPossibleObjective)
  *Prune the tree using an objective function cutoff.*
- CbcNode ∗ bestAlternate ()
  *Get best on list using alternate method.*

- virtual void endSearch ()

  *We may have got an intelligent tree so give it one more chance.*
- virtual double getBestPossibleObjective ()

  *Get best possible objective function in the tree.*
- void resetNodeNumbers ()

  *Reset maximum node number.*
- int maximumNodeNumber () const

  *Get maximum node number.*
- void setNumberBranching (int value)

  *Set number of branches.*
- int getNumberBranching () const

  *Get number of branches.*
- void setMaximumBranching (int value)

  *Set maximum branches.*
- int getMaximumBranching () const

  *Get maximum branches.*
- unsigned int $*$ branched () const

  *Get branched variables.*
- int $*$ newBounds () const

  *Get bounds.*
- double lastObjective () const

  *Last objective in branch-and-cut search tree.*
- int lastDepth () const

  *Last depth in branch-and-cut search tree.*
- int lastUnsatisfied () const

  *Last number of objects unsatisfied.*
- void addBranchingInformation (const CbcModel $*$model, const CbcNodeInfo $*$nodeInfo, const double $*$currentLower, const double $*$currentUpper)

  *Adds branching information to complete state.*
- void increaseSpace ()

  *Increase space for data.*

## Protected Attributes

- std::vector$<$ CbcNode $*$ $>$ nodes_

  *Storage vector for the heap.*
- CbcCompare comparison_

  *Sort predicate for heap ordering.*
- int maximumNodeNumber_

  *Maximum "node" number so far to split ties.*
- int numberBranching_

  *Size of variable list.*
- int maximumBranching_

  *Maximum size of variable list.*
- double lastObjective_

  *Objective of last node pushed on tree.*
- int lastDepth_

  *Depth of last node pushed on tree.*
- int lastUnsatisfied_

  *Number unsatisfied of last node pushed on tree.*
- unsigned int $*$ branched_

  *Integer variables branched or bounded top bit set if new upper bound next bit set if a branch.*
- int $*$ newBound_

  *New bound.*

### 4.107.1 Detailed Description

Using MS heap implementation.

It's unclear if this is needed any longer, or even if it should be allowed. Cbc occasionally tries to do things to the tree (typically tweaking the comparison predicate) that can cause a violation of the heap property (parent better than either child). In a debug build, Microsoft's heap implementation does checks that detect this and fail. This symbol switched to an alternate implementation of CbcTree, and there are clearly differences, but no explanation as to why or what for.

As of 100921, the code is cleaned up to make it through 'cbc -unitTest' without triggering 'Invalid heap' in an MSVS debug build. The method validateHeap() can be used for debugging if this turns up again.

Controls search tree debugging

In order to have validateHeap() available, set CBC_DEBUG_HEAP to 1 or higher.

- 1 calls validateHeap() after each change to the heap
- 2 will print a line for major operations (clean, set comparison, etc.)
- 3 will print information about each push and pop

#define CBC_DEBUG_HEAP 1

Implementation of the live set as a heap.

This class is used to hold the set of live nodes in the search tree.

Definition at line 53 of file CbcTree.hpp.

### 4.107.2 Member Function Documentation

#### 4.107.2.1 virtual CbcNode* CbcTree::bestNode ( double *cutoff* ) [virtual]

Gets best node and takes off heap.

Before returning the node from the top of the heap, the node is offered an opportunity to reevaluate itself. Callers should be prepared to check that the node returned is suitable for use.

#### 4.107.2.2 virtual void CbcTree::cleanTree ( CbcModel * *model,* double *cutoff,* double & *bestPossibleObjective* ) [virtual]

Prune the tree using an objective function cutoff.

This routine removes all nodes with objective worse than the specified cutoff value. It also sets bestPossibleObjective to the best objective over remaining nodes.

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

- CbcTree.hpp

## 4.108 CbcTreeLocal Class Reference

Inheritance diagram for CbcTreeLocal:

Collaboration diagram for CbcTreeLocal:

## Public Member Functions

- virtual CbcTree ∗ clone () const
  *Clone.*
- virtual void generateCpp (FILE ∗fp)
  *Create C++ lines to get to current state.*

### Heap access and maintenance methods

- virtual CbcNode ∗ top () const
  *Return the top node of the heap.*
- virtual void push (CbcNode ∗x)
  *Add a node to the heap.*
- virtual void pop ()
  *Remove the top node from the heap.*

### Other stuff

- int createCut (const double ∗solution, **OsiRowCut** &**cut**)
  *Create cut - return -1 if bad, 0 if okay and 1 if cut is everything.*
- virtual bool empty ()
  *Test if empty ∗∗∗ note may be overridden.*
- virtual void endSearch ()
  *We may have got an intelligent tree so give it one more chance.*
- void reverseCut (int state, double bias=0.0)
  *Other side of last cut branch (if bias==rhs_ will be weakest possible)*
- void deleteCut (**OsiRowCut** &**cut**)
  *Delete last cut branch.*
- void passInSolution (const double ∗solution, double solutionValue)
  *Pass in solution (so can be used after heuristic)*
- int **range** () const
- void **setRange** (int value)
- int **typeCuts** () const
- void **setTypeCuts** (int value)
- int **maxDiversification** () const
- void **setMaxDiversification** (int value)
- int **timeLimit** () const
- void **setTimeLimit** (int value)
- int **nodeLimit** () const
- void **setNodeLimit** (int value)
- bool **refine** () const
- void **setRefine** (bool yesNo)

## Additional Inherited Members

### 4.108.1 Detailed Description

Definition at line 40 of file CbcTreeLocal.hpp.

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

- CbcTreeLocal.hpp

## 4.109 CbcTreeVariable Class Reference

Inheritance diagram for CbcTreeVariable:

Collaboration diagram for CbcTreeVariable:

### Public Member Functions

- virtual CbcTree ∗ clone () const

  *Clone.*
- virtual void generateCpp (FILE ∗fp)

  *Create C++ lines to get to current state.*

#### Heap access and maintenance methods

- virtual CbcNode ∗ top () const

  *Return the top node of the heap.*
- virtual void push (CbcNode ∗x)

  *Add a node to the heap.*
- virtual void pop ()

  *Remove the top node from the heap.*

#### Other stuff

- int createCut (const double ∗solution, **OsiRowCut** &**cut**)

  *Create cut - return -1 if bad, 0 if okay and 1 if cut is everything.*
- virtual bool empty ()

  *Test if empty ∗∗∗ note may be overridden.*
- virtual void endSearch ()

  *We may have got an intelligent tree so give it one more chance.*
- void reverseCut (int state, double bias=0.0)

  *Other side of last cut branch (if bias==rhs_ will be weakest possible)*
- void deleteCut (**OsiRowCut** &**cut**)

  *Delete last cut branch.*
- void passInSolution (const double ∗solution, double solutionValue)

  *Pass in solution (so can be used after heuristic)*
- int **range** () const
- void **setRange** (int value)
- int **typeCuts** () const
- void **setTypeCuts** (int value)
- int **maxDiversification** () const
- void **setMaxDiversification** (int value)
- int **timeLimit** () const
- void **setTimeLimit** (int value)
- int **nodeLimit** () const
- void **setNodeLimit** (int value)
- bool **refine** () const
- void **setRefine** (bool yesNo)

## Additional Inherited Members

### 4.109.1 Detailed Description

Definition at line 206 of file CbcTreeLocal.hpp.

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

- CbcTreeLocal.hpp

## 4.110 CbcUser Class Reference

A class to allow the use of unknown user functionality.

```
#include <CbcSolver.hpp>
```

Collaboration diagram for CbcUser:

### Public Member Functions

#### import/export methods

- virtual int importData (CbcSolver ∗, int &, char ∗∗)
  *Import - gets full command arguments.*
- virtual void exportSolution (CbcSolver ∗, int, const char ∗=NULL)
  *Export.*
- virtual void exportData (CbcSolver ∗)
  *Export Data (i.e. at very end)*
- virtual void fillInformation (CbcSolver ∗, CbcSolverUsefulData &)
  *Get useful stuff.*

#### usage methods

- **CoinModel** ∗ coinModel () const
  **CoinModel** *if valid.*
- virtual void ∗ stuff ()
  *Other info - needs expanding.*
- std::string name () const
  *Name.*
- virtual void solve (CbcSolver ∗model, const char ∗options)=0
  *Solve (whatever that means)*
- virtual bool canDo (const char ∗options)=0
  *Returns true if function knows about option.*

#### Constructors and destructors etc

- CbcUser ()
  *Default Constructor.*
- CbcUser (const CbcUser &rhs)
  *Copy constructor.*
- CbcUser & operator= (const CbcUser &rhs)
  *Assignment operator.*
- virtual CbcUser ∗ clone () const =0
  *Clone.*
- virtual ∼CbcUser ()
  *Destructor.*

## Protected Attributes

**Private member data**

- **CoinModel** * coinModel_

  **CoinModel**.
- std::string userName_

  *Name of user function.*

### 4.110.1 Detailed Description

A class to allow the use of unknown user functionality.

For example, access to a modelling language (CbcAmpl).

Definition at line 308 of file CbcSolver.hpp.

### 4.110.2 Member Function Documentation

#### 4.110.2.1 virtual int CbcUser::importData ( **CbcSolver** * , int & , char ** ) [inline], [virtual]

Import - gets full command arguments.

**Returns**

- -1 - no action
- 0 - data read in without error
- 1 - errors

Definition at line 320 of file CbcSolver.hpp.

#### 4.110.2.2 virtual void CbcUser::exportSolution ( **CbcSolver** * , int , const char * = NULL ) [inline], [virtual]

Export.

Values for mode:

- 1 OsiClpSolver
- 2 CbcModel
- add 10 if infeasible from odd situation

Definition at line 331 of file CbcSolver.hpp.

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

- CbcSolver.hpp

## 4.111 CglTemporary Class Reference

Stored Temporary Cut Generator Class - destroyed after first use.

`#include <CbcLinked.hpp>`

Inheritance diagram for CglTemporary:

Collaboration diagram for CglTemporary:

### Public Member Functions

#### Generate Cuts

- virtual void generateCuts (const **OsiSolverInterface** &si, **OsiCuts** &cs, const **CglTreeInfo** info=**CglTreeInfo**())
  *Generate Mixed Integer Stored cuts for the model of the solver interface, si.*

#### Constructors and destructors

- CglTemporary ()
  *Default constructor.*
- CglTemporary (const CglTemporary &rhs)
  *Copy constructor.*
- virtual **CglCutGenerator** ∗ clone () const
  *Clone.*
- CglTemporary & operator= (const CglTemporary &rhs)
  *Assignment operator.*
- virtual ∼CglTemporary ()
  *Destructor.*

### 4.111.1 Detailed Description

Stored Temporary Cut Generator Class - destroyed after first use.

Definition at line 1266 of file CbcLinked.hpp.

### 4.111.2 Member Function Documentation

#### 4.111.2.1 virtual void CglTemporary::generateCuts ( const OsiSolverInterface & *si,* OsiCuts & *cs,* const CglTreeInfo *info =* CglTreeInfo`()` ) `[virtual]`

Generate Mixed Integer Stored cuts for the model of the solver interface, si.

Insert the generated cuts into **OsiCut**, cs.

This generator just looks at previously stored cuts and inserts any that are violated by enough

Reimplemented from **CglStored**.

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

- CbcLinked.hpp

## 4.112 CbcGenCtlBlk::chooseStrongCtl_struct Struct Reference

Control variables for a strong branching method.

`#include <CbcGenCtlBlk.hpp>`

### 4.112.1 Detailed Description

Control variables for a strong branching method.

Consult **OsiChooseVariable** and CbcModel for details. An artifact of the changeover from CbcObjects to OsiObjects is that the number of uses before pseudo costs are trusted (numBeforeTrust_) and the number of variables evaluated with strong branching (numStrong_) are parameters of CbcModel.

Definition at line 765 of file CbcGenCtlBlk.hpp.

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

- CbcGenCtlBlk.hpp

## 4.113 ClpAmplObjective Class Reference

Ampl Objective Class.

`#include <ClpAmplObjective.hpp>`

Inheritance diagram for ClpAmplObjective:

Collaboration diagram for ClpAmplObjective:

### Public Member Functions

#### Stuff

- virtual double ∗ gradient (const **ClpSimplex** ∗model, const double ∗solution, double &offset, bool refresh, int includeLinear=2)
  *Returns gradient.*
- virtual double reducedGradient (**ClpSimplex** ∗model, double ∗region, bool useFeasibleCosts)
  *Resize objective.*
- virtual double stepLength (**ClpSimplex** ∗model, const double ∗solution, const double ∗change, double maximumTheta, double &currentObj, double &predictedObj, double &thetaObj)
  *Returns step length which gives minimum of objective for solution + theta ∗ change vector up to maximum theta.*
- virtual double objectiveValue (const **ClpSimplex** ∗model, const double ∗solution) const
  *Return objective value (without any **ClpModel** offset) (model may be NULL)*
- virtual void **resize** (int newNumberColumns)
- virtual void deleteSome (int numberToDelete, const int ∗which)
  *Delete columns in objective.*
- virtual void reallyScale (const double ∗columnScale)
  *Scale objective.*
- virtual int markNonlinear (char ∗which)
  *Given a zeroed array sets nonlinear columns to 1.*
- virtual void newXValues ()
  *Say we have new primal solution - so may need to recompute.*

**Constructors and destructors**

- ClpAmplObjective ()
  *Default Constructor.*
- ClpAmplObjective (void ∗amplInfo)
  *Constructor from ampl info.*
- ClpAmplObjective (const ClpAmplObjective &rhs)
  *Copy constructor .*
- ClpAmplObjective & operator= (const ClpAmplObjective &rhs)
  *Assignment operator.*
- virtual ∼ClpAmplObjective ()
  *Destructor.*
- virtual **ClpObjective** ∗ clone () const
  *Clone.*

**Gets and sets**

- double ∗ linearObjective () const
  *Linear objective.*

### 4.113.1 Detailed Description

Ampl Objective Class.

Definition at line 18 of file ClpAmplObjective.hpp.

### 4.113.2 Member Function Documentation

#### 4.113.2.1 virtual double∗ ClpAmplObjective::gradient ( const **ClpSimplex** ∗ *model,* const double ∗ *solution,* double & *offset,* bool *refresh,* int *includeLinear* = 2 ) [virtual]

Returns gradient.

If Ampl then solution may be NULL, also returns an offset (to be added to current one) If refresh is false then uses last solution Uses model for scaling includeLinear 0 - no, 1 as is, 2 as feasible

Implements **ClpObjective**.

#### 4.113.2.2 virtual double ClpAmplObjective::reducedGradient ( **ClpSimplex** ∗ *model,* double ∗ *region,* bool *useFeasibleCosts* ) [virtual]

Resize objective.

Returns reduced gradient.Returns an offset (to be added to current one).

Implements **ClpObjective**.

#### 4.113.2.3 virtual double ClpAmplObjective::stepLength ( **ClpSimplex** ∗ *model,* const double ∗ *solution,* const double ∗ *change,* double *maximumTheta,* double & *currentObj,* double & *predictedObj,* double & *thetaObj* ) [virtual]

Returns step length which gives minimum of objective for solution + theta ∗ change vector up to maximum theta.

arrays are numberColumns+numberRows Also sets current objective, predicted and at maximumTheta

Implements **ClpObjective**.

#### 4.113.2.4 virtual int ClpAmplObjective::markNonlinear ( char ∗ *which* ) [virtual]

Given a zeroed array sets nonlinear columns to 1.

Returns number of nonlinear columns

Reimplemented from **ClpObjective**.

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

- ClpAmplObjective.hpp

## 4.114 ClpConstraintAmpl Class Reference

Ampl Constraint Class.

`#include <ClpConstraintAmpl.hpp>`

Inheritance diagram for ClpConstraintAmpl:

Collaboration diagram for ClpConstraintAmpl:

### Public Member Functions

#### Stuff

- virtual int gradient (const **ClpSimplex** ∗model, const double ∗solution, double ∗gradient, double &**function**←- **Value**, double &**offset**, bool useScaling=false, bool refresh=true) const
  *Fills gradient.*
- virtual void resize (int newNumberColumns)
  *Resize constraint.*
- virtual void deleteSome (int numberToDelete, const int ∗which)
  *Delete columns in constraint.*
- virtual void reallyScale (const double ∗columnScale)
  *Scale constraint.*
- virtual int markNonlinear (char ∗which) const
  *Given a zeroed array sets nonampl columns to 1.*
- virtual int markNonzero (char ∗which) const
  *Given a zeroed array sets possible nonzero coefficients to 1.*
- virtual void newXValues ()
  *Say we have new primal solution - so may need to recompute.*

#### Constructors and destructors

- ClpConstraintAmpl ()
  *Default Constructor.*
- ClpConstraintAmpl (int row, void ∗amplInfo)
  *Constructor from ampl.*
- ClpConstraintAmpl (const ClpConstraintAmpl &rhs)
  *Copy constructor .*
- ClpConstraintAmpl & operator= (const ClpConstraintAmpl &rhs)
  *Assignment operator.*
- virtual ∼ClpConstraintAmpl ()
  *Destructor.*

- virtual **ClpConstraint** ∗ clone () const
  *Clone.*

**Gets and sets**

- virtual int numberCoefficients () const
  *Number of coefficients.*
- const int ∗ column () const
  *Columns.*
- const double ∗ coefficient () const
  *Coefficients.*

### 4.114.1 Detailed Description

Ampl Constraint Class.

Definition at line 17 of file ClpConstraintAmpl.hpp.

### 4.114.2 Member Function Documentation

#### 4.114.2.1 virtual int ClpConstraintAmpl::gradient ( const **ClpSimplex** ∗ *model*, const double ∗ *solution*, double ∗ *gradient*, double & *functionValue*, double & *offset*, bool *useScaling* = `false`, bool *refresh* = `true` ) const [virtual]

Fills gradient.

If Ampl then solution may be NULL, also returns true value of function and offset so we can use x not deltaX in constraint If refresh is false then uses last solution Uses model for scaling Returns non-zero if gradient udefined at current solution

Implements **ClpConstraint**.

#### 4.114.2.2 virtual int ClpConstraintAmpl::markNonlinear ( char ∗ *which* ) const [virtual]

Given a zeroed array sets nonampl columns to 1.

Returns number of nonampl columns

Implements **ClpConstraint**.

#### 4.114.2.3 virtual int ClpConstraintAmpl::markNonzero ( char ∗ *which* ) const [virtual]

Given a zeroed array sets possible nonzero coefficients to 1.

Returns number of nonzeros

Implements **ClpConstraint**.

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

- ClpConstraintAmpl.hpp

## 4.115 CoinHashLink Struct Reference

Really for Conflict cuts to - a) stop duplicates b) allow half baked cuts The whichRow_ field in **OsiRowCut2** is used for a type 0 - normal 1 - processed cut (conflict) 2 - unprocessed cut i.e.

```
#include <CbcCountRowCut.hpp>
```

### 4.115.1 Detailed Description

Really for Conflict cuts to - a) stop duplicates b) allow half baked cuts The whichRow_ field in **OsiRowCut2** is used for a type 0 - normal 1 - processed cut (conflict) 2 - unprocessed cut i.e.

dual ray computation

Definition at line 131 of file CbcCountRowCut.hpp.

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

- CbcCountRowCut.hpp

## 4.116 CbcGenCtlBlk::debugSolInfo_struct Struct Reference

Array of primal variable values for debugging.

```
#include <CbcGenCtlBlk.hpp>
```

### 4.116.1 Detailed Description

Array of primal variable values for debugging.

Used to provide a known optimal solution to activateRowCutDebugger().

Definition at line 669 of file CbcGenCtlBlk.hpp.

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

- CbcGenCtlBlk.hpp

## 4.117 CbcGenCtlBlk::djFixCtl_struct Struct Reference

Control use of reduced cost fixing prior to B&C.

```
#include <CbcGenCtlBlk.hpp>
```

### 4.117.1 Detailed Description

Control use of reduced cost fixing prior to B&C.

This heuristic fixes variables whose reduced cost for the root relaxtion exceeds the specified threshold. This is purely a heuristic, performed before there's any incumbent solution. It may well fix variables at the wrong bound!

Definition at line 739 of file CbcGenCtlBlk.hpp.

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

- CbcGenCtlBlk.hpp

## 4.118 CbcGenCtlBlk::genParamsInfo_struct Struct Reference

Start and end of cbc-generic parameters in parameter vector.

`#include <CbcGenCtlBlk.hpp>`

### 4.118.1 Detailed Description

Start and end of cbc-generic parameters in parameter vector.

Definition at line 598 of file CbcGenCtlBlk.hpp.

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

- CbcGenCtlBlk.hpp

## 4.119 OsiBiLinear Class Reference

Define BiLinear objects.

`#include <CbcLinked.hpp>`

Inheritance diagram for OsiBiLinear:

Collaboration diagram for OsiBiLinear:

### Public Member Functions

- OsiBiLinear (**OsiSolverInterface** ∗solver, int xColumn, int yColumn, int xyRow, double coefficient, double xMesh, double yMesh, int numberExistingObjects=0, const **OsiObject** ∗∗objects=NULL)

    *Useful constructor - This Adds in rows and variables to construct valid Linked Ordered Set Adds extra constraints to match other x/y So note not const solver.*
- OsiBiLinear (**CoinModel** ∗coinModel, int xColumn, int yColumn, int xyRow, double coefficient, double xMesh, double yMesh, int numberExistingObjects=0, const **OsiObject** ∗∗objects=NULL)

    *Useful constructor - This Adds in rows and variables to construct valid Linked Ordered Set Adds extra constraints to match other x/y So note not const model.*
- virtual **OsiObject** ∗ clone () const

    *Clone.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗info, int &**whichWay**) const

    *Infeasibility - large is 0.5.*
- virtual double feasibleRegion (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info) const

    *Set bounds to fix the variable at the current (integer) value.*
- virtual **OsiBranchingObject** ∗ createBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way) const

    *Creates a branching object.*
- virtual void resetSequenceEtc (int numberColumns, const int ∗originalColumns)

    *Redoes data when sequence numbers change.*
- virtual bool canDoHeuristics () const

    *Return true if object can take part in normal heuristics.*
- virtual bool boundBranch () const

    *Return true if branch should only bound variables.*

- int xColumn () const

    *X column.*
- int yColumn () const

    *Y column.*
- int xRow () const

    *X row.*
- int yRow () const

    *Y row.*
- int xyRow () const

    *XY row.*
- double coefficient () const

    *Coefficient.*
- void setCoefficient (double value)

    *Set coefficient.*
- int firstLambda () const

    *First lambda (of 4)*
- double xSatisfied () const

    *X satisfied if less than this away from mesh.*
- double ySatisfied () const

    *Y satisfied if less than this away from mesh.*
- double xOtherSatisfied () const

    *X other satisfied if less than this away from mesh.*
- double yOtherSatisfied () const

    *Y other satisfied if less than this away from mesh.*
- double xMeshSize () const

    *X meshSize.*
- double yMeshSize () const

    *Y meshSize.*
- double xySatisfied () const

    *XY satisfied if two version differ by less than this.*
- void setMeshSizes (const **OsiSolverInterface** ∗solver, double x, double y)

    *Set sizes and other stuff.*
- int branchingStrategy () const

    *branching strategy etc bottom 2 bits 0 branch on either, 1 branch on x, 2 branch on y next bit 4 set to say don't update coefficients next bit 8 set to say don't use in feasible region next bit 16 set to say - Always satisfied !!*
- int boundType () const

    *Simple quadratic bound marker.*
- void newBounds (**OsiSolverInterface** ∗solver, int way, short xOrY, double separator) const

    *Does work of branching.*
- int updateCoefficients (const double ∗lower, const double ∗upper, double ∗objective, **CoinPackedMatrix** ∗matrix, **CoinWarmStartBasis** ∗basis) const

    *Updates coefficients - returns number updated.*
- double xyCoefficient (const double ∗solution) const

    *Returns true value of single xyRow coefficient.*
- void getCoefficients (const **OsiSolverInterface** ∗solver, double xB[2], double yB[2], double xybar[4]) const

    *Get LU coefficients from matrix.*
- double computeLambdas (const double xB[3], const double yB[3], const double xybar[4], double lambda[4]) const

*Compute lambdas (third entry in each .B is current value) (nonzero if bad)*

- void addExtraRow (int row, double multiplier)

  *Adds in data for extra row with variable coefficients.*

- void getPseudoShadow (const **OsiBranchingInformation** ∗info)

  *Sets infeasibility and other when pseudo shadow prices.*

- double getMovement (const **OsiBranchingInformation** ∗info)

  *Gets sum of movements to correct value.*

## Protected Member Functions

- void computeLambdas (const **OsiSolverInterface** ∗solver, double lambda[4]) const

  *Compute lambdas if coefficients not changing.*

## Protected Attributes

- double coefficient_

  *data*

- double xMeshSize_

  *x mesh*

- double yMeshSize_

  *y mesh*

- double xSatisfied_

  *x satisfied if less than this away from mesh*

- double ySatisfied_

  *y satisfied if less than this away from mesh*

- double xOtherSatisfied_

  *X other satisfied if less than this away from mesh.*

- double yOtherSatisfied_

  *Y other satisfied if less than this away from mesh.*

- double xySatisfied_

  *xy satisfied if less than this away from true*

- double xyBranchValue_

  *value of x or y to branch about*

- int xColumn_

  *x column*

- int yColumn_

  *y column*

- int firstLambda_

  *First lambda (of 4)*

- int branchingStrategy_

  *branching strategy etc bottom 2 bits 0 branch on either, 1 branch on x, 2 branch on y next bit 4 set to say don't update coefficients next bit 8 set to say don't use in feasible region next bit 16 set to say - Always satisfied !!*

- int boundType_

  *Simple quadratic bound marker.*

- int xRow_

  *x row*

- int yRow_

*y row (-1 if x*x)*

- int xyRow_

  *Output row.*
- int convexity_

  *Convexity row.*
- int numberExtraRows_

  *Number of extra rows (coefficients to be modified)*
- double * multiplier_

  *Multiplier for coefficient on row.*
- int * extraRow_

  *Row number.*
- short chosen_

  *Which chosen -1 none, 0 x, 1 y.*

### 4.119.1 Detailed Description

Define BiLinear objects.

This models x*y where one or both are integer

Definition at line 720 of file CbcLinked.hpp.

### 4.119.2 Member Function Documentation

#### 4.119.2.1 virtual double OsiBiLinear::feasibleRegion ( **OsiSolverInterface** * *solver*, const **OsiBranchingInformation** * *info* ) const `[virtual]`

Set bounds to fix the variable at the current (integer) value.

Given an integer value, set the lower and upper bounds to fix the variable. Returns amount it had to move variable.

Implements **OsiObject**.

#### 4.119.2.2 virtual **OsiBranchingObject*** OsiBiLinear::createBranch ( **OsiSolverInterface** * *solver*, const **OsiBranchingInformation** * *info*, int *way* ) const `[virtual]`

Creates a branching object.

The preferred direction is set by `way`, 0 for down, 1 for up.

Reimplemented from **OsiObject**.

#### 4.119.2.3 int OsiBiLinear::boundType ( ) const `[inline]`

Simple quadratic bound marker.

0 no 1 L if coefficient pos, G if negative i.e. value is ub on xy 2 G if coefficient pos, L if negative i.e. value is lb on xy 3 E If bound then real coefficient is 1.0 and coefficient_ is bound

Definition at line 899 of file CbcLinked.hpp.

### 4.119.3 Member Data Documentation

#### 4.119.3.1 double OsiBiLinear::coefficient_ [protected]

data

Coefficient

Definition at line 929 of file CbcLinked.hpp.

#### 4.119.3.2 int OsiBiLinear::boundType_ [protected]

Simple quadratic bound marker.

0 no 1 L if coefficient pos, G if negative i.e. value is ub on xy 2 G if coefficient pos, L if negative i.e. value is lb on xy 3 E If bound then real coefficient is 1.0 and coefficient_ is bound

Definition at line 970 of file CbcLinked.hpp.

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

- CbcLinked.hpp

## 4.120 OsiBiLinearBranchingObject Class Reference

Branching object for BiLinear objects.

`#include <CbcLinked.hpp>`

Inheritance diagram for OsiBiLinearBranchingObject:

Collaboration diagram for OsiBiLinearBranchingObject:

### Public Member Functions

- virtual **OsiBranchingObject** ∗ clone () const

  *Clone.*
- virtual double branch (**OsiSolverInterface** ∗solver)

  *Does next branch and updates state.*
- virtual void print (const **OsiSolverInterface** ∗solver=NULL)

  *Print something about branch - only if log level high.*
- virtual bool boundBranch () const

  *Return true if branch should only bound variables.*

### 4.120.1 Detailed Description

Branching object for BiLinear objects.

Definition at line 991 of file CbcLinked.hpp.

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

- CbcLinked.hpp

## 4.121 OsiBiLinearEquality Class Reference

Define Continuous BiLinear objects for an == bound.

`#include <CbcLinked.hpp>`

Inheritance diagram for OsiBiLinearEquality:

Collaboration diagram for OsiBiLinearEquality:

### Public Member Functions

- OsiBiLinearEquality (**OsiSolverInterface** ∗solver, int xColumn, int yColumn, int xyRow, double rhs, double x←Mesh)

  *Useful constructor - This Adds in rows and variables to construct Ordered Set for x∗y = b So note not const solver.*
- virtual **OsiObject** ∗ clone () const

  *Clone.*
- virtual double improvement (const **OsiSolverInterface** ∗solver) const

  *Possible improvement.*
- double newGrid (**OsiSolverInterface** ∗solver, int type) const

  *change grid if type 0 then use solution and make finer if 1 then back to original returns mesh size*
- int numberPoints () const

  *Number of points.*

### Additional Inherited Members

### 4.121.1 Detailed Description

Define Continuous BiLinear objects for an == bound.

This models x∗y = b where both are continuous

Definition at line 1038 of file CbcLinked.hpp.

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

- CbcLinked.hpp

## 4.122 OsiCbcSolverInterface Class Reference

Cbc Solver Interface.

`#include <OsiCbcSolverInterface.hpp>`

Inheritance diagram for OsiCbcSolverInterface:

Collaboration diagram for OsiCbcSolverInterface:

### Public Member Functions

- virtual void setObjSense (double s)

  *Set objective function sense (1 for min (default), -1 for max,)*

- virtual void setColSolution (const double ∗colsol)

  *Set the primal solution column values.*
- virtual void setRowPrice (const double ∗rowprice)

  *Set dual solution vector.*

### Solve methods

- virtual void initialSolve ()

  *Solve initial LP relaxation.*
- virtual void resolve ()

  *Resolve an LP relaxation after problem modification.*
- virtual void branchAndBound ()

  *Invoke solver's built-in enumeration algorithm.*

### Parameter set/get methods

*The set methods return true if the parameter was set to the given value, false otherwise.*

*There can be various reasons for failure: the given parameter is not applicable for the solver (e.g., refactorization frequency for the cbc algorithm), the parameter is not yet implemented for the solver or simply the value of the parameter is out of the range the solver accepts. If a parameter setting call returns false check the details of your solver.*

*The get methods return true if the given parameter is applicable for the solver and is implemented. In this case the value of the parameter is returned in the second argument. Otherwise they return false.*

- bool **setIntParam** (OsiIntParam key, int value)
- bool **setDblParam** (OsiDblParam key, double value)
- bool **setStrParam** (OsiStrParam key, const std::string &value)
- bool **getIntParam** (OsiIntParam key, int &value) const
- bool **getDblParam** (OsiDblParam key, double &value) const
- bool **getStrParam** (OsiStrParam key, std::string &value) const
- virtual bool **setHintParam** (OsiHintParam key, bool yesNo=true, OsiHintStrength strength=OsiHintTry, void ∗otherInformation=NULL)
- virtual bool getHintParam (OsiHintParam key, bool &yesNo, OsiHintStrength &strength, void ∗&other←Information) const

  *Get a hint parameter.*
- virtual bool getHintParam (OsiHintParam key, bool &yesNo, OsiHintStrength &strength) const

  *Get a hint parameter.*

### Methods returning info on how the solution process terminated

- virtual bool isAbandoned () const

  *Are there a numerical difficulties?*
- virtual bool isProvenOptimal () const

  *Is optimality proven?*
- virtual bool isProvenPrimalInfeasible () const

  *Is primal infeasiblity proven?*
- virtual bool isProvenDualInfeasible () const

  *Is dual infeasiblity proven?*
- virtual bool isPrimalObjectiveLimitReached () const

  *Is the given primal objective limit reached?*
- virtual bool isDualObjectiveLimitReached () const

  *Is the given dual objective limit reached?*
- virtual bool isIterationLimitReached () const

*Iteration limit reached?*

### WarmStart related methods

- virtual **CoinWarmStart** ∗ getEmptyWarmStart () const
  *Get an empty warm start object.*
- virtual **CoinWarmStart** ∗ getWarmStart () const
  *Get warmstarting information.*
- virtual bool setWarmStart (const **CoinWarmStart** ∗warmstart)
  *Set warmstarting information.*

### Hotstart related methods (primarily used in strong branching). <br>

*The user can create a hotstart (a snapshot) of the optimization process then reoptimize over and over again always starting from there.*

***NOTE**: between hotstarted optimizations only bound changes are allowed.*

- virtual void markHotStart ()
  *Create a hotstart point of the optimization process.*
- virtual void solveFromHotStart ()
  *Optimize starting from the hotstart.*
- virtual void unmarkHotStart ()
  *Delete the snapshot.*

### Methods related to querying the input data

- virtual int getNumCols () const
  *Get number of columns.*
- virtual int getNumRows () const
  *Get number of rows.*
- virtual int getNumElements () const
  *Get number of nonzero elements.*
- virtual const double ∗ getColLower () const
  *Get pointer to array[getNumCols()] of column lower bounds.*
- virtual const double ∗ getColUpper () const
  *Get pointer to array[getNumCols()] of column upper bounds.*
- virtual const char ∗ getRowSense () const
  *Get pointer to array[getNumRows()] of row constraint senses.*
- virtual const double ∗ getRightHandSide () const
  *Get pointer to array[getNumRows()] of rows right-hand sides.*
- virtual const double ∗ getRowRange () const
  *Get pointer to array[getNumRows()] of row ranges.*
- virtual const double ∗ getRowLower () const
  *Get pointer to array[getNumRows()] of row lower bounds.*
- virtual const double ∗ getRowUpper () const
  *Get pointer to array[getNumRows()] of row upper bounds.*
- virtual const double ∗ getObjCoefficients () const
  *Get pointer to array[getNumCols()] of objective function coefficients.*
- virtual double getObjSense () const
  *Get objective function sense (1 for min (default), -1 for max)*
- virtual bool isContinuous (int colNumber) const
  *Return true if column is continuous.*

- virtual const **CoinPackedMatrix** ∗ getMatrixByRow () const

    *Get pointer to row-wise copy of matrix.*
- virtual const **CoinPackedMatrix** ∗ getMatrixByCol () const

    *Get pointer to column-wise copy of matrix.*
- virtual double getInfinity () const

    *Get solver's value for infinity.*

### Methods related to querying the solution

- virtual const double ∗ getColSolution () const

    *Get pointer to array[getNumCols()] of primal solution vector.*
- virtual const double ∗ getRowPrice () const

    *Get pointer to array[getNumRows()] of dual prices.*
- virtual const double ∗ getReducedCost () const

    *Get a pointer to array[getNumCols()] of reduced costs.*
- virtual const double ∗ getRowActivity () const

    *Get pointer to array[getNumRows()] of row activity levels (constraint matrix times the solution vector.*
- virtual double getObjValue () const

    *Get objective function value.*
- virtual int getIterationCount () const

    *Get how many iterations it took to solve the problem (whatever "iteration" mean to the solver.*
- virtual std::vector< double ∗ > getDualRays (int maxNumRays, bool fullRay=false) const

    *Get as many dual rays as the solver can provide.*
- virtual std::vector< double ∗ > getPrimalRays (int maxNumRays) const

    *Get as many primal rays as the solver can provide.*

### Methods for row and column names.

*Because OsiCbc is a pass-through class, it's necessary to override any virtual method in order to be sure we catch an override by the underlying solver.*

*See the* **OsiSolverInterface** *class documentation for detailed descriptions.*

- virtual std::string dfltRowColName (char rc, int ndx, unsigned digits=7) const

    *Generate a standard name of the form Rnnnnnnn or Cnnnnnnn.*
- virtual std::string getObjName (unsigned maxLen=std::string::npos) const

    *Return the name of the objective function.*
- virtual void setObjName (std::string name)

    *Set the name of the objective function.*
- virtual std::string getRowName (int rowIndex, unsigned maxLen=std::string::npos) const

    *Return the name of the row.*
- virtual const **OsiNameVec** & getRowNames ()

    *Return a pointer to a vector of row names.*
- virtual void setRowName (int ndx, std::string name)

    *Set a row name.*
- virtual void setRowNames (**OsiNameVec** &srcNames, int srcStart, int len, int tgtStart)

    *Set multiple row names.*
- virtual void deleteRowNames (int tgtStart, int len)

    *Delete len row names starting at index tgtStart.*
- virtual std::string getColName (int colIndex, unsigned maxLen=std::string::npos) const

    *Return the name of the column.*
- virtual const **OsiNameVec** & getColNames ()

    *Return a pointer to a vector of column names.*
- virtual void setColName (int ndx, std::string name)

*Set a column name.*

- virtual void setColNames (**OsiNameVec** &srcNames, int srcStart, int len, int tgtStart)

  *Set multiple column names.*

- virtual void deleteColNames (int tgtStart, int len)

  *Delete len column names starting at index tgtStart.*

### Changing bounds on variables and constraints

- virtual void setObjCoeff (int elementIndex, double elementValue)

  *Set an objective function coefficient.*

- virtual void setColLower (int elementIndex, double elementValue)

  *Set a single column lower bound*
  *Use -DBL_MAX for -infinity.*

- virtual void setColUpper (int elementIndex, double elementValue)

  *Set a single column upper bound*
  *Use DBL_MAX for infinity.*

- virtual void setColBounds (int elementIndex, double lower, double upper)

  *Set a single column lower and upper bound.*

- virtual void setColSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

  *Set the bounds on a number of columns simultaneously*
  *The default implementation just invokes setColLower() and setColUpper() over and over again.*

- virtual void setRowLower (int elementIndex, double elementValue)

  *Set a single row lower bound*
  *Use -DBL_MAX for -infinity.*

- virtual void setRowUpper (int elementIndex, double elementValue)

  *Set a single row upper bound*
  *Use DBL_MAX for infinity.*

- virtual void setRowBounds (int elementIndex, double lower, double upper)

  *Set a single row lower and upper bound.*

- virtual void setRowType (int index, char sense, double rightHandSide, double range)

  *Set the type of a single row*

- virtual void setRowSetBounds (const int ∗indexFirst, const int ∗indexLast, const double ∗boundList)

  *Set the bounds on a number of rows simultaneously*
  *The default implementation just invokes setRowLower() and setRowUpper() over and over again.*

- virtual void setRowSetTypes (const int ∗indexFirst, const int ∗indexLast, const char ∗senseList, const double ∗rhsList, const double ∗rangeList)

  *Set the type of a number of rows simultaneously*
  *The default implementation just invokes setRowType() over and over again.*

### Integrality related changing methods

- virtual void setContinuous (int index)

  *Set the index-th variable to be a continuous variable.*

- virtual void setInteger (int index)

  *Set the index-th variable to be an integer variable.*

- virtual void setContinuous (const int ∗indices, int len)

  *Set the variables listed in indices (which is of length len) to be continuous variables.*

- virtual void setInteger (const int ∗indices, int len)

  *Set the variables listed in indices (which is of length len) to be integer variables.*

### Methods to expand a problem.<br>

*Note that if a column is added then by default it will correspond to a continuous variable.*

- virtual void **addCol** (const **CoinPackedVectorBase** &vec, const double collb, const double colub, const double obj)
- virtual void addCol (int numberElements, const int ∗rows, const double ∗elements, const double collb, const double colub, const double obj)

  *Add a column (primal variable) to the problem.*
- virtual void **addCols** (const int numcols, const **CoinPackedVectorBase** ∗const ∗cols, const double ∗collb, const double ∗colub, const double ∗obj)
- virtual void **deleteCols** (const int num, const int ∗colIndices)
- virtual void **addRow** (const **CoinPackedVectorBase** &vec, const double rowlb, const double rowub)
- virtual void **addRow** (const **CoinPackedVectorBase** &vec, const char rowsen, const double rowrhs, const double rowrng)
- virtual void **addRows** (const int numrows, const **CoinPackedVectorBase** ∗const ∗rows, const double ∗rowlb, const double ∗rowub)
- virtual void **addRows** (const int numrows, const **CoinPackedVectorBase** ∗const ∗rows, const char ∗rowsen, const double ∗rowrhs, const double ∗rowrng)
- virtual void **deleteRows** (const int num, const int ∗rowIndices)
- virtual void applyRowCuts (int numberCuts, const **OsiRowCut** ∗cuts)

  *Apply a collection of row cuts which are all effective.*
- virtual void applyRowCuts (int numberCuts, const **OsiRowCut** ∗∗cuts)

  *Apply a collection of row cuts which are all effective.*

### Methods to input a problem

- virtual void loadProblem (const **CoinPackedMatrix** &matrix, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub)

  *Load in an problem by copying the arguments (the constraints on the rows are given by lower and upper bounds).*
- virtual void assignProblem (**CoinPackedMatrix** ∗&matrix, double ∗&collb, double ∗&colub, double ∗&obj, double ∗&rowlb, double ∗&rowub)

  *Load in an problem by assuming ownership of the arguments (the constraints on the rows are given by lower and upper bounds).*
- virtual void loadProblem (const **CoinPackedMatrix** &matrix, const double ∗collb, const double ∗colub, const double ∗obj, const char ∗rowsen, const double ∗rowrhs, const double ∗rowrng)

  *Load in an problem by copying the arguments (the constraints on the rows are given by sense/rhs/range triplets).*
- virtual void assignProblem (**CoinPackedMatrix** ∗&matrix, double ∗&collb, double ∗&colub, double ∗&obj, char ∗&rowsen, double ∗&rowrhs, double ∗&rowrng)

  *Load in an problem by assuming ownership of the arguments (the constraints on the rows are given by sense/rhs/range triplets).*
- virtual void loadProblem (const int numcols, const int numrows, const CoinBigIndex ∗start, const int ∗index, const double ∗value, const double ∗collb, const double ∗colub, const double ∗obj, const double ∗rowlb, const double ∗rowub)

  *Just like the other loadProblem() methods except that the matrix is given in a standard column major ordered format (without gaps).*
- virtual void loadProblem (const int numcols, const int numrows, const CoinBigIndex ∗start, const int ∗index, const double ∗value, const double ∗collb, const double ∗colub, const double ∗obj, const char ∗rowsen, const double ∗rowrhs, const double ∗rowrng)

  *Just like the other loadProblem() methods except that the matrix is given in a standard column major ordered format (without gaps).*
- virtual int readMps (const char ∗filename, const char ∗extension="mps")

  *Read an mps file from the given filename (defaults to Osi reader) - returns number of errors (see OsiMpsReader class)*
- virtual void writeMps (const char ∗filename, const char ∗extension="mps", double objSense=0.0) const

  *Write the problem into an mps file of the given filename.*
- virtual int writeMpsNative (const char ∗filename, const char ∗∗rowNames, const char ∗∗columnNames, int formatType=0, int numberAcross=2, double objSense=0.0) const

  *Write the problem into an mps file of the given filename, names may be null.*

**Message handling (extra for Cbc messages).**

*Normally I presume you would want the same language.*

*If not then you could use underlying model pointer*

- void newLanguage (**CoinMessages::Language** language)
  *Set language.*
- void **setLanguage** (**CoinMessages::Language** language)

**Cbc specific public interfaces**

- CbcModel ∗ getModelPtr () const
  *Get pointer to Cbc model.*
- **OsiSolverInterface** ∗ getRealSolverPtr () const
  *Get pointer to underlying solver.*
- void setCutoff (double value)
  *Set cutoff bound on the objective function.*
- double getCutoff () const
  *Get the cutoff bound on the objective function - always as minimize.*
- void setMaximumNodes (int value)
  *Set the CbcModel::CbcMaxNumNode maximum node limit.*
- int getMaximumNodes () const
  *Get the CbcModel::CbcMaxNumNode maximum node limit.*
- void setMaximumSolutions (int value)
  *Set the CbcModel::CbcMaxNumSol maximum number of solutions.*
- int getMaximumSolutions () const
  *Get the CbcModel::CbcMaxNumSol maximum number of solutions.*
- void setMaximumSeconds (double value)
  *Set the CbcModel::CbcMaximumSeconds maximum number of seconds.*
- double getMaximumSeconds () const
  *Get the CbcModel::CbcMaximumSeconds maximum number of seconds.*
- bool isNodeLimitReached () const
  *Node limit reached?*
- bool isSolutionLimitReached () const
  *Solution limit reached?*
- int getNodeCount () const
  *Get how many Nodes it took to solve the problem.*
- int status () const
  *Final status of problem - 0 finished, 1 stopped, 2 difficulties.*
- virtual void passInMessageHandler (**CoinMessageHandler** ∗handler)
  *Pass in a message handler.*

**Constructors and destructors**

- OsiCbcSolverInterface (**OsiSolverInterface** ∗solver=NULL, CbcStrategy ∗strategy=NULL)
  *Default Constructor.*
- virtual **OsiSolverInterface** ∗ clone (bool copyData=true) const
  *Clone.*
- OsiCbcSolverInterface (const OsiCbcSolverInterface &)
  *Copy constructor.*
- OsiCbcSolverInterface & operator= (const OsiCbcSolverInterface &rhs)
  *Assignment operator.*
- virtual ∼OsiCbcSolverInterface ()
  *Destructor.*

## Protected Member Functions

### Protected methods

- virtual void applyRowCut (const **OsiRowCut** &rc)
  *Apply a row cut (append to constraint matrix).*
- virtual void applyColCut (const **OsiColCut** &cc)
  *Apply a column cut (adjust one or more bounds).*

## Protected Attributes

### Protected member data

- CbcModel ∗ modelPtr_
  *Cbc model represented by this class instance.*

## Friends

- void OsiCbcSolverInterfaceUnitTest (const std::string &mpsDir, const std::string &netlibDir)
  *A function that tests the methods in the OsiCbcSolverInterface class.*

### 4.122.1 Detailed Description

Cbc Solver Interface.

Instantiation of OsiCbcSolverInterface for the Model Algorithm.

Definition at line 30 of file OsiCbcSolverInterface.hpp.

### 4.122.2 Member Function Documentation

#### 4.122.2.1 virtual CoinWarmStart∗ OsiCbcSolverInterface::getEmptyWarmStart ( ) const [virtual]

Get an empty warm start object.

This routine returns an empty **CoinWarmStartBasis** object. Its purpose is to provide a way to give a client a warm start basis object of the appropriate type, which can resized and modified as desired.

Implements **OsiSolverInterface**.

#### 4.122.2.2 virtual bool OsiCbcSolverInterface::setWarmStart ( const CoinWarmStart ∗ *warmstart* ) [virtual]

Set warmstarting information.

Return true/false depending on whether the warmstart information was accepted or not.

Implements **OsiSolverInterface**.

#### 4.122.2.3 virtual const char∗ OsiCbcSolverInterface::getRowSense ( ) const [virtual]

Get pointer to array[getNumRows()] of row constraint senses.

- 'L' <= constraint
- 'E' = constraint
- 'G' >= constraint
- 'R' ranged constraint
- 'N' free constraint

Implements **OsiSolverInterface**.

#### 4.122.2.4 virtual const double* OsiCbcSolverInterface::getRightHandSide ( ) const [virtual]

Get pointer to array[getNumRows()] of rows right-hand sides.

- if rowsense()[i] == 'L' then rhs()[i] == rowupper()[i]
- if rowsense()[i] == 'G' then rhs()[i] == rowlower()[i]
- if rowsense()[i] == 'R' then rhs()[i] == rowupper()[i]
- if rowsense()[i] == 'N' then rhs()[i] == 0.0

Implements **OsiSolverInterface**.

#### 4.122.2.5 virtual const double* OsiCbcSolverInterface::getRowRange ( ) const [virtual]

Get pointer to array[getNumRows()] of row ranges.

- if rowsense()[i] == 'R' then rowrange()[i] == rowupper()[i] - rowlower()[i]
- if rowsense()[i] != 'R' then rowrange()[i] is undefined

Implements **OsiSolverInterface**.

#### 4.122.2.6 virtual int OsiCbcSolverInterface::getIterationCount ( ) const [virtual]

Get how many iterations it took to solve the problem (whatever "iteration" mean to the solver.

Implements **OsiSolverInterface**.

#### 4.122.2.7 virtual std::vector<double*> OsiCbcSolverInterface::getDualRays ( int *maxNumRays*, bool *fullRay* = false ) const [virtual]

Get as many dual rays as the solver can provide.

(In case of proven primal infeasibility there should be at least one.)

The first getNumRows() ray components will always be associated with the row duals (as returned by getRowPrice()). If `fullRay` is true, the final getNumCols() entries will correspond to the ray components associated with the nonbasic variables. If the full ray is requested and the method cannot provide it, it will throw an exception.

**NOTE for implementers of solver interfaces:**
The double pointers in the vector should point to arrays of length getNumRows() and they should be allocated via new[].

**NOTE for users of solver interfaces:**
It is the user's responsibility to free the double pointers in the vector using delete[].

Implements **OsiSolverInterface**.

#### 4.122.2.8 virtual std::vector<double∗> OsiCbcSolverInterface::getPrimalRays ( int *maxNumRays* ) const [virtual]

Get as many primal rays as the solver can provide.

(In case of proven dual infeasibility there should be at least one.)

**NOTE for implementers of solver interfaces:**
The double pointers in the vector should point to arrays of length getNumCols() and they should be allocated via new[].

**NOTE for users of solver interfaces:**
It is the user's responsibility to free the double pointers in the vector using delete[].

Implements **OsiSolverInterface**.

#### 4.122.2.9 virtual std::string OsiCbcSolverInterface::getRowName ( int *rowIndex*, unsigned *maxLen* = std::string::npos ) const [virtual]

Return the name of the row.

Reimplemented from **OsiSolverInterface**.

#### 4.122.2.10 virtual void OsiCbcSolverInterface::setColLower ( int *elementIndex*, double *elementValue* ) [virtual]

Set a single column lower bound
Use -DBL_MAX for -infinity.

Implements **OsiSolverInterface**.

#### 4.122.2.11 virtual void OsiCbcSolverInterface::setColUpper ( int *elementIndex*, double *elementValue* ) [virtual]

Set a single column upper bound
Use DBL_MAX for infinity.

Implements **OsiSolverInterface**.

#### 4.122.2.12 virtual void OsiCbcSolverInterface::setColSetBounds ( const int ∗ *indexFirst*, const int ∗ *indexLast*, const double ∗ *boundList* ) [virtual]

Set the bounds on a number of columns simultaneously
The default implementation just invokes setColLower() and setColUpper() over and over again.

**Parameters**

| | |
|---|---|
| *index←↩ First,indexLast* | pointers to the beginning and after the end of the array of the indices of the variables whose *either* bound changes |
| *boundList* | the new lower/upper bound pairs for the variables |

Reimplemented from **OsiSolverInterface**.

#### 4.122.2.13 virtual void OsiCbcSolverInterface::setRowLower ( int *elementIndex*, double *elementValue* ) [virtual]

Set a single row lower bound
Use -DBL_MAX for -infinity.

Implements **OsiSolverInterface**.

#### 4.122.2.14 virtual void OsiCbcSolverInterface::setRowUpper ( int *elementIndex*, double *elementValue* ) [virtual]

Set a single row upper bound
Use DBL_MAX for infinity.

Implements **OsiSolverInterface**.

#### 4.122.2.15 virtual void OsiCbcSolverInterface::setRowSetBounds ( const int ∗ *indexFirst*, const int ∗ *indexLast*, const double ∗ *boundList* ) [virtual]

Set the bounds on a number of rows simultaneously
The default implementation just invokes setRowLower() and setRowUpper() over and over again.

**Parameters**

| | |
|---|---|
| *index←↩ First,indexLast* | pointers to the beginning and after the end of the array of the indices of the constraints whose *either* bound changes |
| *boundList* | the new lower/upper bound pairs for the constraints |

Reimplemented from **OsiSolverInterface**.

#### 4.122.2.16 virtual void OsiCbcSolverInterface::setRowSetTypes ( const int ∗ *indexFirst*, const int ∗ *indexLast*, const char ∗ *senseList*, const double ∗ *rhsList*, const double ∗ *rangeList* ) [virtual]

Set the type of a number of rows simultaneously
The default implementation just invokes setRowType() over and over again.

**Parameters**

| | |
|---|---|
| *index←↩ First,indexLast* | pointers to the beginning and after the end of the array of the indices of the constraints whose *any* characteristics changes |
| *senseList* | the new senses |
| *rhsList* | the new right hand sides |
| *rangeList* | the new ranges |

Reimplemented from **OsiSolverInterface**.

#### 4.122.2.17 virtual void OsiCbcSolverInterface::setColSolution ( const double * *colsol* ) [virtual]

Set the primal solution column values.

colsol[numcols()] is an array of values of the problem column variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of colsol() until changed by another call to setColsol() or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements **OsiSolverInterface**.

#### 4.122.2.18 virtual void OsiCbcSolverInterface::setRowPrice ( const double * *rowprice* ) [virtual]

Set dual solution vector.

rowprice[numrows()] is an array of values of the problem row dual variables. These values are copied to memory owned by the solver object or the solver. They will be returned as the result of rowprice() until changed by another call to setRowprice() or by a call to any solver routine. Whether the solver makes use of the solution in any way is solver-dependent.

Implements **OsiSolverInterface**.

#### 4.122.2.19 virtual void OsiCbcSolverInterface::addCol ( int *numberElements,* const int * *rows,* const double * *elements,* const double *collb,* const double *colub,* const double *obj* ) [virtual]

Add a column (primal variable) to the problem.

Reimplemented from **OsiSolverInterface**.

#### 4.122.2.20 virtual void OsiCbcSolverInterface::applyRowCuts ( int *numberCuts,* const OsiRowCut * *cuts* ) [virtual]

Apply a collection of row cuts which are all effective.

applyCuts seems to do one at a time which seems inefficient.

Reimplemented from **OsiSolverInterface**.

#### 4.122.2.21 virtual void OsiCbcSolverInterface::applyRowCuts ( int *numberCuts,* const OsiRowCut ** *cuts* ) [virtual]

Apply a collection of row cuts which are all effective.

applyCuts seems to do one at a time which seems inefficient. This uses array of pointers

Reimplemented from **OsiSolverInterface**.

#### 4.122.2.22 virtual void OsiCbcSolverInterface::loadProblem ( const CoinPackedMatrix & *matrix,* const double * *collb,* const double * *colub,* const double * *obj,* const double * *rowlb,* const double * *rowub* ) [virtual]

Load in an problem by copying the arguments (the constraints on the rows are given by lower and upper bounds).

If a pointer is 0 then the following values are the default:

- `colub`: all columns have upper bound infinity
- `collb`: all columns have lower bound 0
- `rowub`: all rows have upper bound infinity

- `rowlb`: all rows have lower bound -infinity
- `obj`: all variables have 0 objective coefficient

Implements **OsiSolverInterface**.

#### 4.122.2.23 virtual void OsiCbcSolverInterface::assignProblem ( **CoinPackedMatrix** ∗& *matrix*, double ∗& *collb*, double ∗& *colub*, double ∗& *obj*, double ∗& *rowlb*, double ∗& *rowub* ) [virtual]

Load in an problem by assuming ownership of the arguments (the constraints on the rows are given by lower and upper bounds).

For default values see the previous method.
**WARNING**: The arguments passed to this method will be freed using the C++ `delete` and `delete[]` functions.

Implements **OsiSolverInterface**.

#### 4.122.2.24 virtual void OsiCbcSolverInterface::loadProblem ( const **CoinPackedMatrix** & *matrix*, const double ∗ *collb*, const double ∗ *colub*, const double ∗ *obj*, const char ∗ *rowsen*, const double ∗ *rowrhs*, const double ∗ *rowrng* ) [virtual]

Load in an problem by copying the arguments (the constraints on the rows are given by sense/rhs/range triplets).

If a pointer is 0 then the following values are the default:

- `colub`: all columns have upper bound infinity
- `collb`: all columns have lower bound 0
- `obj`: all variables have 0 objective coefficient
- `rowsen`: all rows are >=
- `rowrhs`: all right hand sides are 0
- `rowrng`: 0 for the ranged rows

Implements **OsiSolverInterface**.

#### 4.122.2.25 virtual void OsiCbcSolverInterface::assignProblem ( **CoinPackedMatrix** ∗& *matrix*, double ∗& *collb*, double ∗& *colub*, double ∗& *obj*, char ∗& *rowsen*, double ∗& *rowrhs*, double ∗& *rowrng* ) [virtual]

Load in an problem by assuming ownership of the arguments (the constraints on the rows are given by sense/rhs/range triplets).

For default values see the previous method.
**WARNING**: The arguments passed to this method will be freed using the C++ `delete` and `delete[]` functions.

Implements **OsiSolverInterface**.

#### 4.122.2.26 virtual void OsiCbcSolverInterface::loadProblem ( const int *numcols*, const int *numrows*, const CoinBigIndex ∗ *start*, const int ∗ *index*, const double ∗ *value*, const double ∗ *collb*, const double ∗ *colub*, const double ∗ *obj*, const double ∗ *rowlb*, const double ∗ *rowub* ) [virtual]

Just like the other loadProblem() methods except that the matrix is given in a standard column major ordered format (without gaps).

Implements **OsiSolverInterface**.

#### 4.122.2.27 virtual void OsiCbcSolverInterface::loadProblem ( const int *numcols*, const int *numrows*, const CoinBigIndex * *start*, const int * *index*, const double * *value*, const double * *collb*, const double * *colub*, const double * *obj*, const char * *rowsen*, const double * *rowrhs*, const double * *rowrng* ) [virtual]

Just like the other loadProblem() methods except that the matrix is given in a standard column major ordered format (without gaps).

Implements **OsiSolverInterface**.

#### 4.122.2.28 virtual void OsiCbcSolverInterface::writeMps ( const char * *filename*, const char * *extension* = "mps", double *objSense* = 0.0 ) const [virtual]

Write the problem into an mps file of the given filename.

If objSense is non zero then -1.0 forces the code to write a maximization objective and +1.0 to write a minimization one. If 0.0 then solver can do what it wants

Implements **OsiSolverInterface**.

#### 4.122.2.29 virtual int OsiCbcSolverInterface::writeMpsNative ( const char * *filename*, const char ** *rowNames*, const char ** *columnNames*, int *formatType* = 0, int *numberAcross* = 2, double *objSense* = 0.0 ) const [virtual]

Write the problem into an mps file of the given filename, names may be null.

formatType is 0 - normal 1 - extra accuracy 2 - IEEE hex (later)

Returns non-zero on I/O error

#### 4.122.2.30 virtual void OsiCbcSolverInterface::passInMessageHandler ( CoinMessageHandler * *handler* ) [virtual]

Pass in a message handler.

It is the client's responsibility to destroy a message handler installed by this routine; it will not be destroyed when the solver interface is destroyed.

Reimplemented from **OsiSolverInterface**.

#### 4.122.2.31 virtual void OsiCbcSolverInterface::applyRowCut ( const OsiRowCut & *rc* ) [protected], [virtual]

Apply a row cut (append to constraint matrix).

Implements **OsiSolverInterface**.

#### 4.122.2.32 virtual void OsiCbcSolverInterface::applyColCut ( const OsiColCut & *cc* ) [protected], [virtual]

Apply a column cut (adjust one or more bounds).

Implements **OsiSolverInterface**.

### 4.122.3 Friends And Related Function Documentation

#### 4.122.3.1 void OsiCbcSolverInterfaceUnitTest ( const std::string & *mpsDir*, const std::string & *netlibDir* ) [friend]

A function that tests the methods in the OsiCbcSolverInterface class.

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

- OsiCbcSolverInterface.hpp

## 4.123 OsiChooseStrongSubset Class Reference

This class chooses a variable to branch on.

`#include <CbcLinked.hpp>`

Inheritance diagram for OsiChooseStrongSubset:

Collaboration diagram for OsiChooseStrongSubset:

### Public Member Functions

- OsiChooseStrongSubset ()

  *Default Constructor.*
- OsiChooseStrongSubset (const **OsiSolverInterface** ∗solver)

  *Constructor from solver (so we can set up arrays etc)*
- OsiChooseStrongSubset (const OsiChooseStrongSubset &)

  *Copy constructor.*
- OsiChooseStrongSubset & operator= (const OsiChooseStrongSubset &rhs)

  *Assignment operator.*
- virtual **OsiChooseVariable** ∗ clone () const

  *Clone.*
- virtual ∼OsiChooseStrongSubset ()

  *Destructor.*
- virtual int setupList (**OsiBranchingInformation** ∗info, bool initialize)

  *Sets up strong list and clears all if initialize is true.*
- virtual int chooseVariable (**OsiSolverInterface** ∗solver, **OsiBranchingInformation** ∗info, bool fixVariables)

  *Choose a variable Returns - -1 Node is infeasible 0 Normal termination - we have a candidate 1 All looks satisfied - no candidate 2 We can change the bound on a variable - but we also have a strong branching candidate 3 We can change the bound on a variable - but we have a non-strong branching candidate 4 We can change the bound on a variable - no other candidates We can pick up branch from* **bestObjectIndex()** *and* **bestWhichWay()** *We can pick up a forced branch (can change bound) from* **firstForcedObjectIndex()** *and* **firstForcedWhichWay()** *If we have a solution then we can pick up from* **goodObjectiveValue()** *and* **goodSolution()** *If fixVariables is true then 2,3,4 are all really same as problem changed.*
- int numberObjectsToUse () const

  *Number of objects to use.*
- void setNumberObjectsToUse (int value)

  *Set number of objects to use.*

### Protected Attributes

- int numberObjectsToUse_

  *Number of objects to be used (and set in solver)*

### 4.123.1 Detailed Description

This class chooses a variable to branch on.

This is just as **OsiChooseStrong** but it fakes it so only first so many are looked at in this phase

Definition at line 1203 of file CbcLinked.hpp.

### 4.123.2 Member Function Documentation

#### 4.123.2.1 virtual int OsiChooseStrongSubset::setupList ( OsiBranchingInformation * *info*, bool *initialize* ) [virtual]

Sets up strong list and clears all if initialize is true.

Returns number of infeasibilities. If returns -1 then has worked out node is infeasible!

Reimplemented from **OsiChooseStrong**.

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

- CbcLinked.hpp

## 4.124 OsiLink Class Reference

Define Special Linked Ordered Sets.

`#include <CbcLinked.hpp>`

Inheritance diagram for OsiLink:

Collaboration diagram for OsiLink:

### Public Member Functions

- OsiLink (const **OsiSolverInterface** *solver, int yRow, int yColumn, double meshSize)

  *Useful constructor -.*
- virtual **OsiObject** * clone () const

  *Clone.*
- virtual double infeasibility (const **OsiBranchingInformation** *info, int &**whichWay**) const

  *Infeasibility - large is 0.5.*
- virtual double feasibleRegion (**OsiSolverInterface** *solver, const **OsiBranchingInformation** *info) const

  *Set bounds to fix the variable at the current (integer) value.*
- virtual **OsiBranchingObject** * createBranch (**OsiSolverInterface** *solver, const **OsiBranchingInformation** *info, int way) const

  *Creates a branching object.*
- virtual void resetSequenceEtc (int numberColumns, const int *originalColumns)

  *Redoes data when sequence numbers change.*
- int numberLinks () const

  *Number of links for each member.*
- virtual bool canDoHeuristics () const

  *Return true if object can take part in normal heuristics.*
- virtual bool boundBranch () const

  *Return true if branch should only bound variables.*

### 4.124.1 Detailed Description

Define Special Linked Ordered Sets.

New style

members and weights may be stored in SOS object

This is for y and x∗f(y) and z∗g(y) etc

Definition at line 600 of file CbcLinked.hpp.

### 4.124.2 Constructor & Destructor Documentation

#### 4.124.2.1 OsiLink::OsiLink ( const **OsiSolverInterface** ∗ *solver*, int *yRow*, int *yColumn*, double *meshSize* )

Useful constructor -.

### 4.124.3 Member Function Documentation

#### 4.124.3.1 virtual double OsiLink::feasibleRegion ( **OsiSolverInterface** ∗ *solver*, const **OsiBranchingInformation** ∗ *info* ) const [virtual]

Set bounds to fix the variable at the current (integer) value.

Given an integer value, set the lower and upper bounds to fix the variable. Returns amount it had to move variable.

Reimplemented from **OsiSOS**.

#### 4.124.3.2 virtual **OsiBranchingObject**∗ OsiLink::createBranch ( **OsiSolverInterface** ∗ *solver*, const **OsiBranchingInformation** ∗ *info*, int *way* ) const [virtual]

Creates a branching object.

The preferred direction is set by `way`, 0 for down, 1 for up.

Reimplemented from **OsiSOS**.

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

- CbcLinked.hpp

## 4.125 OsiLinkBranchingObject Class Reference

Branching object for Linked ordered sets.

`#include <CbcLinked.hpp>`

Inheritance diagram for OsiLinkBranchingObject:

Collaboration diagram for OsiLinkBranchingObject:

### Public Member Functions

- virtual **OsiBranchingObject** ∗ clone () const

*Clone.*
- virtual double branch (**OsiSolverInterface** ∗solver)

  *Does next branch and updates state.*
- virtual void print (const **OsiSolverInterface** ∗solver=NULL)

  *Print something about branch - only if log level high.*

### 4.125.1 Detailed Description

Branching object for Linked ordered sets.

Definition at line 678 of file CbcLinked.hpp.

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

- CbcLinked.hpp

## 4.126 OsiLinkedBound Class Reference

List of bounds which depend on other bounds.

`#include <CbcLinked.hpp>`

### Public Member Functions

#### Action methods

- void updateBounds (**ClpSimplex** ∗solver)

  *Update other bounds.*

#### Constructors and destructors

- OsiLinkedBound ()

  *Default Constructor.*
- OsiLinkedBound (**OsiSolverInterface** ∗model, int variable, int numberAffected, const int ∗positionL, const int ∗positionU, const double ∗multiplier)

  *Useful Constructor.*
- OsiLinkedBound (const OsiLinkedBound &)

  *Copy constructor.*
- OsiLinkedBound & operator= (const OsiLinkedBound &rhs)

  *Assignment operator.*
- ∼OsiLinkedBound ()

  *Destructor.*

#### Sets and Gets

- int variable () const

  *Get variable.*
- void addBoundModifier (bool upperBoundAffected, bool useUpperBound, int whichVariable, double multiplier=1.0)

  *Add a bound modifier.*

### 4.126.1 Detailed Description

List of bounds which depend on other bounds.

Definition at line 300 of file CbcLinked.hpp.

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

- CbcLinked.hpp

## 4.127 OsiOldLink Class Reference

Inheritance diagram for OsiOldLink:

Collaboration diagram for OsiOldLink:

### Public Member Functions

- OsiOldLink (const **OsiSolverInterface** ∗solver, int **numberMembers**, int numberLinks, int first, const double ∗**weights**, int setNumber)

  *Useful constructor - A valid solution is if all variables are zero apart from k∗numberLink to (k+1)∗numberLink-1 where k is 0 through numberInSet-1.*
- OsiOldLink (const **OsiSolverInterface** ∗solver, int **numberMembers**, int numberLinks, int typeSOS, const int ∗which, const double ∗**weights**, int setNumber)

  *Useful constructor - A valid solution is if all variables are zero apart from k∗numberLink to (k+1)∗numberLink-1 where k is 0 through numberInSet-1.*
- virtual **OsiObject** ∗ clone () const

  *Clone.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗info, int &**whichWay**) const

  *Infeasibility - large is 0.5.*
- virtual double feasibleRegion (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info) const

  *Set bounds to fix the variable at the current (integer) value.*
- virtual **OsiBranchingObject** ∗ createBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way) const

  *Creates a branching object.*
- virtual void resetSequenceEtc (int numberColumns, const int ∗originalColumns)

  *Redoes data when sequence numbers change.*
- int numberLinks () const

  *Number of links for each member.*
- virtual bool canDoHeuristics () const

  *Return true if object can take part in normal heuristics.*
- virtual bool boundBranch () const

  *Return true if branch should only bound variables.*

### 4.127.1 Detailed Description

Definition at line 434 of file CbcLinked.hpp.

### 4.127.2 Constructor & Destructor Documentation

#### 4.127.2.1 OsiOldLink::OsiOldLink ( const **OsiSolverInterface** * *solver*, int *numberMembers*, int *numberLinks*, int *first*, const double * *weights*, int *setNumber* )

Useful constructor - A valid solution is if all variables are zero apart from k*numberLink to (k+1)*numberLink-1 where k is 0 through numberInSet-1.

The length of weights array is numberInSet. For this constructor the variables in matrix are the numberInSet*numberLink starting at first. If weights null then 0,1,2..

#### 4.127.2.2 OsiOldLink::OsiOldLink ( const **OsiSolverInterface** * *solver*, int *numberMembers*, int *numberLinks*, int *typeSOS*, const int * *which*, const double * *weights*, int *setNumber* )

Useful constructor - A valid solution is if all variables are zero apart from k*numberLink to (k+1)*numberLink-1 where k is 0 through numberInSet-1.

The length of weights array is numberInSet. For this constructor the variables are given by list - grouped. If weights null then 0,1,2..

### 4.127.3 Member Function Documentation

#### 4.127.3.1 virtual double OsiOldLink::feasibleRegion ( **OsiSolverInterface** * *solver*, const **OsiBranchingInformation** * *info* ) const `[virtual]`

Set bounds to fix the variable at the current (integer) value.

Given an integer value, set the lower and upper bounds to fix the variable. Returns amount it had to move variable.

Reimplemented from **OsiSOS**.

#### 4.127.3.2 virtual **OsiBranchingObject*** OsiOldLink::createBranch ( **OsiSolverInterface** * *solver*, const **OsiBranchingInformation** * *info*, int *way* ) const `[virtual]`

Creates a branching object.

The preferred direction is set by `way`, 0 for down, 1 for up.

Reimplemented from **OsiSOS**.

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

- CbcLinked.hpp

## 4.128 OsiOldLinkBranchingObject Class Reference

Branching object for Linked ordered sets.

`#include <CbcLinked.hpp>`

Inheritance diagram for OsiOldLinkBranchingObject:

Collaboration diagram for OsiOldLinkBranchingObject:

### Public Member Functions

- virtual **OsiBranchingObject** ∗ clone () const

  *Clone.*
- virtual double branch (**OsiSolverInterface** ∗solver)

  *Does next branch and updates state.*
- virtual void print (const **OsiSolverInterface** ∗solver=NULL)

  *Print something about branch - only if log level high.*

### 4.128.1 Detailed Description

Branching object for Linked ordered sets.

Definition at line 518 of file CbcLinked.hpp.

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

- CbcLinked.hpp

## 4.129 OsiOneLink Class Reference

Define data for one link.

`#include <CbcLinked.hpp>`

Collaboration diagram for OsiOneLink:

### Public Member Functions

- OsiOneLink (const **OsiSolverInterface** ∗solver, int xRow, int xColumn, int xyRow, const char ∗functionString)

  *Useful constructor -.*

### Public Attributes

- int xRow_

  *data*
- int xColumn_

  *Column which defines x.*
- int xyRow

  *Output row.*
- std::string function_

  *Function.*

### 4.129.1 Detailed Description

Define data for one link.

Definition at line 558 of file CbcLinked.hpp.

### 4.129.2 Constructor & Destructor Documentation

#### 4.129.2.1 OsiOneLink::OsiOneLink ( const **OsiSolverInterface** ∗ *solver,* int *xRow,* int *xColumn,* int *xyRow,* const char ∗ *functionString* )

Useful constructor -.

### 4.129.3 Member Data Documentation

#### 4.129.3.1 int OsiOneLink::xRow_

data

Row which defines x (if -1 then no x)

Definition at line 583 of file CbcLinked.hpp.

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

- CbcLinked.hpp

## 4.130 CbcGenCtlBlk::osiParamsInfo_struct Struct Reference

Start and end of **OsiSolverInterface** parameters in parameter vector.

`#include <CbcGenCtlBlk.hpp>`

### 4.130.1 Detailed Description

Start and end of **OsiSolverInterface** parameters in parameter vector.

Definition at line 614 of file CbcGenCtlBlk.hpp.

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

- CbcGenCtlBlk.hpp

## 4.131 OsiSimpleFixedInteger Class Reference

Define a single integer class - but one where you keep branching until fixed even if satisfied.

`#include <CbcLinked.hpp>`

Inheritance diagram for OsiSimpleFixedInteger:

Collaboration diagram for OsiSimpleFixedInteger:

### Public Member Functions

- OsiSimpleFixedInteger ()

  *Default Constructor.*
- OsiSimpleFixedInteger (const **OsiSolverInterface** ∗solver, int iColumn)

*Useful constructor - passed solver index.*

- OsiSimpleFixedInteger (int iColumn, double lower, double upper)

  *Useful constructor - passed solver index and original bounds.*
- OsiSimpleFixedInteger (const **OsiSimpleInteger** &)

  *Useful constructor - passed simple integer.*
- OsiSimpleFixedInteger (const OsiSimpleFixedInteger &)

  *Copy constructor.*
- virtual **OsiObject** ∗ clone () const

  *Clone.*
- OsiSimpleFixedInteger & operator= (const OsiSimpleFixedInteger &rhs)

  *Assignment operator.*
- virtual ~OsiSimpleFixedInteger ()

  *Destructor.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗info, int &**whichWay**) const

  *Infeasibility - large is 0.5.*
- virtual **OsiBranchingObject** ∗ createBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way) const

  *Creates a branching object.*

### 4.131.1 Detailed Description

Define a single integer class - but one where you keep branching until fixed even if satisfied.

Definition at line 1089 of file CbcLinked.hpp.

### 4.131.2 Member Function Documentation

#### 4.131.2.1 virtual OsiBranchingObject∗ OsiSimpleFixedInteger::createBranch ( OsiSolverInterface ∗ *solver,* const OsiBranchingInformation ∗ *info,* int *way* ) const [virtual]

Creates a branching object.

The preferred direction is set by `way`, 0 for down, 1 for up.

Reimplemented from **OsiSimpleInteger**.

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

- CbcLinked.hpp

## 4.132 OsiSolverLinearizedQuadratic Class Reference

This is to allow the user to replace initialSolve and resolve.

`#include <CbcLinked.hpp>`

Inheritance diagram for OsiSolverLinearizedQuadratic:

Collaboration diagram for OsiSolverLinearizedQuadratic:

## Public Member Functions

### Solve methods

- virtual void initialSolve ()
  *Solve initial LP relaxation.*

### Constructors and destructors

- OsiSolverLinearizedQuadratic ()
  *Default Constructor.*
- OsiSolverLinearizedQuadratic (**ClpSimplex** *quadraticModel)
  *Useful constructor (solution should be good)*
- virtual **OsiSolverInterface** * clone (bool copyData=true) const
  *Clone.*
- OsiSolverLinearizedQuadratic (const OsiSolverLinearizedQuadratic &)
  *Copy constructor.*
- OsiSolverLinearizedQuadratic & operator= (const OsiSolverLinearizedQuadratic &rhs)
  *Assignment operator.*
- virtual ~OsiSolverLinearizedQuadratic ()
  *Destructor.*

### Sets and Gets

- double bestObjectiveValue () const
  *Objective value of best solution found internally.*
- const double * bestSolution () const
  *Best solution found internally.*
- void setSpecialOptions3 (int value)
  *Set special options.*
- int specialOptions3 () const
  *Get special options.*
- **ClpSimplex** * quadraticModel () const
  *Copy of quadratic model if one.*

## Protected Attributes

### Private member data

- double bestObjectiveValue_
  *Objective value of best solution found internally.*
- **ClpSimplex** * quadraticModel_
  *Copy of quadratic model if one.*
- double * bestSolution_
  *Best solution found internally.*
- int specialOptions3_
  *0 bit (1) - don't do mini B&B 1 bit (2) - quadratic only in objective*

### 4.132.1 Detailed Description

This is to allow the user to replace initialSolve and resolve.

Definition at line 1318 of file CbcLinked.hpp.

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

- CbcLinked.hpp

## 4.133 OsiSolverLink Class Reference

This is to allow the user to replace initialSolve and resolve This version changes coefficients.

`#include <CbcLinked.hpp>`

Inheritance diagram for OsiSolverLink:

Collaboration diagram for OsiSolverLink:

### Public Member Functions

#### Solve methods

- virtual void initialSolve ()
  *Solve initial LP relaxation.*
- virtual void resolve ()
  *Resolve an LP relaxation after problem modification.*
- virtual int fathom (bool allFixed)
  *Problem specific Returns -1 if node fathomed and no solution 0 if did nothing 1 if node fathomed and solution allFixed is true if all LinkedBound variables are fixed.*
- double * nonlinearSLP (int numberPasses, double deltaTolerance)
  *Solves nonlinear problem from* **CoinModel** *using SLP - may be used as crash for other algorithms when number of iterations small.*
- double linearizedBAB (**CglStored** ***cut**)
  *Solve linearized quadratic objective branch and bound.*
- double * heuristicSolution (int numberPasses, double deltaTolerance, int mode)
  *Solves nonlinear problem from* **CoinModel** *using SLP - and then tries to get heuristic solution Returns solution array mode - 0 just get continuous 1 round and try normal bab 2 use defaultBound_ to bound integer variables near current solution.*
- int doAOCuts (CglTemporary *cutGen, const double *solution, const double *solution2)
  *Do OA cuts.*

#### Constructors and destructors

- OsiSolverLink ()
  *Default Constructor.*
- OsiSolverLink (**CoinModel** &modelObject)
  *This creates from a coinModel object.*
- void **load** (**CoinModel** &modelObject, bool **tightenBounds**=false, int logLevel=1)
- virtual **OsiSolverInterface** * clone (bool copyData=true) const
  *Clone.*
- OsiSolverLink (const OsiSolverLink &)
  *Copy constructor.*
- OsiSolverLink & operator= (const OsiSolverLink &rhs)
  *Assignment operator.*
- virtual ∼OsiSolverLink ()
  *Destructor.*

#### Sets and Gets

- void addBoundModifier (bool upperBoundAffected, bool useUpperBound, int whichVariable, int which←VariableAffected, double multiplier=1.0)
  *Add a bound modifier.*

- int updateCoefficients (**ClpSimplex** ∗solver, **CoinPackedMatrix** ∗matrix)

  *Update coefficients - returns number updated if in updating mode.*
- void analyzeObjects ()

  *Analyze constraints to see which are convex (quadratic)*
- void addTighterConstraints ()

  *Add reformulated bilinear constraints.*
- double bestObjectiveValue () const

  *Objective value of best solution found internally.*
- void setBestObjectiveValue (double value)

  *Set objective value of best solution found internally.*
- const double ∗ bestSolution () const

  *Best solution found internally.*
- void setBestSolution (const double ∗solution, int numberColumns)

  *Set best solution found internally.*
- void setSpecialOptions2 (int value)

  *Set special options.*
- void sayConvex (bool convex)

  *Say convex (should work it out) - if convex false then strictly concave.*
- int specialOptions2 () const

  *Get special options.*
- **CoinPackedMatrix** ∗ cleanMatrix () const

  *Clean copy of matrix So we can add rows.*
- **CoinPackedMatrix** ∗ originalRowCopy () const

  *Row copy of matrix Just genuine columns and rows Linear part.*
- **ClpSimplex** ∗ quadraticModel () const

  *Copy of quadratic model if one.*
- **CoinPackedMatrix** ∗ quadraticRow (int rowNumber, double ∗linear) const

  *Gets correct form for a quadratic row - user to delete.*
- double defaultMeshSize () const

  *Default meshSize.*
- void **setDefaultMeshSize** (double value)
- double defaultBound () const

  *Default maximumbound.*
- void **setDefaultBound** (double value)
- void setIntegerPriority (int value)

  *Set integer priority.*
- int integerPriority () const

  *Get integer priority.*
- int objectiveVariable () const

  *Objective transfer variable if one.*
- void setBiLinearPriority (int value)

  *Set biLinear priority.*
- int biLinearPriority () const

  *Get biLinear priority.*
- const **CoinModel** ∗ coinModel () const

  *Return* **CoinModel***.*
- void setBiLinearPriorities (int value, double meshSize=1.0)

  *Set all biLinear priorities on x-x variables.*
- void setBranchingStrategyOnVariables (int strategyValue, int priorityValue=-1, int mode=7)

  *Set options and priority on all or some biLinear variables 1 - on I-I 2 - on I-x 4 - on x-x or combinations.*
- void setMeshSizes (double value)

  *Set all mesh sizes on x-x variables.*
- void setFixedPriority (int priorityValue)

  *Two tier integer problem where when set of variables with priority less than this are fixed the problem becomes an easier integer problem.*

## Protected Member Functions

### functions

- void gutsOfDestructor (bool justNullify=false)
  *Do real work of initialize.*
- void gutsOfCopy (const OsiSolverLink &rhs)
  *Do real work of copy.*

## Protected Attributes

### Private member data

- **CoinPackedMatrix** * matrix_
  *Clean copy of matrix Marked coefficients will be multiplied by L or U.*
- **CoinPackedMatrix** * originalRowCopy_
  *Row copy of matrix Just genuine columns and rows.*
- **ClpSimplex** * quadraticModel_
  *Copy of quadratic model if one.*
- int numberNonLinearRows_
  *Number of rows with nonLinearities.*
- int * startNonLinear_
  *Starts of lists.*
- int * rowNonLinear_
  *Row number for a list.*
- int * convex_
  *Indicator whether is convex, concave or neither -1 concave, 0 neither, +1 convex.*
- int * whichNonLinear_
  *Indices in a list/row.*
- **CoinModel** coinModel_
  *Model in* **CoinModel** *format.*
- int numberVariables_
  *Number of variables in tightening phase.*
- OsiLinkedBound * info_
  *Information.*
- int specialOptions2_
  *0 bit (1) - call fathom (may do mini B&B) 1 bit (2) - quadratic only in objective (add OA cuts) 2 bit (4) - convex 3 bit (8) - try adding OA cuts 4 bit (16) - add linearized constraints*
- int objectiveRow_
  *Objective transfer row if one.*
- int objectiveVariable_
  *Objective transfer variable if one.*
- double bestObjectiveValue_
  *Objective value of best solution found internally.*
- double defaultMeshSize_
  *Default mesh.*
- double defaultBound_
  *Default maximum bound.*
- double * bestSolution_
  *Best solution found internally.*
- int integerPriority_
  *Priority for integers.*
- int biLinearPriority_
  *Priority for bilinear.*
- int numberFix_
  *Number of variables which when fixed help.*
- int * fixVariables_
  *list of fixed variables*

### 4.133.1 Detailed Description

This is to allow the user to replace initialSolve and resolve This version changes coefficients.

Definition at line 29 of file CbcLinked.hpp.

### 4.133.2 Constructor & Destructor Documentation

#### 4.133.2.1 OsiSolverLink::OsiSolverLink ( CoinModel & *modelObject* )

This creates from a coinModel object.

if errors.then number of sets is -1

This creates linked ordered sets information. It assumes -

for product terms syntax is yy*f(zz) also just f(zz) is allowed and even a constant

modelObject not const as may be changed as part of process.

### 4.133.3 Member Function Documentation

#### 4.133.3.1 double* OsiSolverLink::nonlinearSLP ( int *numberPasses*, double *deltaTolerance* )

Solves nonlinear problem from **CoinModel** using SLP - may be used as crash for other algorithms when number of iterations small.

Also exits if all problematical variables are changing less than deltaTolerance Returns solution array

#### 4.133.3.2 double OsiSolverLink::linearizedBAB ( CglStored * *cut* )

Solve linearized quadratic objective branch and bound.

Return cutoff and OA cut

#### 4.133.3.3 void OsiSolverLink::setBranchingStrategyOnVariables ( int *strategyValue*, int *priorityValue* = `-1`, int *mode* = `7` )

Set options and priority on all or some biLinear variables 1 - on I-I 2 - on I-x 4 - on x-x or combinations.

-1 means leave (for priority value and strategy value)

#### 4.133.3.4 void OsiSolverLink::gutsOfDestructor ( bool *justNullify* = `false` ) `[protected]`

Do real work of initialize.

Do real work of delete

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

- CbcLinked.hpp

## 4.134 OsiUsesBiLinear Class Reference

Define a single variable class which is involved with OsiBiLinear objects.

`#include <CbcLinked.hpp>`

Inheritance diagram for OsiUsesBiLinear:

Collaboration diagram for OsiUsesBiLinear:

### Public Member Functions

- OsiUsesBiLinear ()

  *Default Constructor.*
- OsiUsesBiLinear (const **OsiSolverInterface** ∗solver, int iColumn, int type)

  *Useful constructor - passed solver index.*
- OsiUsesBiLinear (int iColumn, double lower, double upper, int type)

  *Useful constructor - passed solver index and original bounds.*
- OsiUsesBiLinear (const **OsiSimpleInteger** &rhs, int type)

  *Useful constructor - passed simple integer.*
- OsiUsesBiLinear (const OsiUsesBiLinear &rhs)

  *Copy constructor.*
- virtual **OsiObject** ∗ clone () const

  *Clone.*
- OsiUsesBiLinear & operator= (const OsiUsesBiLinear &rhs)

  *Assignment operator.*
- virtual ∼OsiUsesBiLinear ()

  *Destructor.*
- virtual double infeasibility (const **OsiBranchingInformation** ∗info, int &**whichWay**) const

  *Infeasibility - large is 0.5.*
- virtual **OsiBranchingObject** ∗ createBranch (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info, int way) const

  *Creates a branching object.*
- virtual double feasibleRegion (**OsiSolverInterface** ∗solver, const **OsiBranchingInformation** ∗info) const

  *Set bounds to fix the variable at the current value.*
- void addBiLinearObjects (OsiSolverLink ∗solver)

  *Add all bi-linear objects.*

### Protected Attributes

- int numberBiLinear_

  *data Number of bilinear objects (maybe could be more general)*
- int type_

  *Type of variable - 0 continuous, 1 integer.*
- **OsiObject** ∗∗ objects_

  *Objects.*

### 4.134.1 Detailed Description

Define a single variable class which is involved with OsiBiLinear objects.

This is used so can make better decision on where to branch as it can look at all objects.

This version sees if it can re-use code from **OsiSimpleInteger** even if not an integer variable. If not then need to duplicate code.

Definition at line 1139 of file CbcLinked.hpp.

### 4.134.2 Member Function Documentation

#### 4.134.2.1 virtual **OsiBranchingObject**∗ OsiUsesBiLinear::createBranch ( **OsiSolverInterface** ∗ *solver,* const **OsiBranchingInformation** ∗ *info,* int *way* ) const [virtual]

Creates a branching object.

The preferred direction is set by `way`, 0 for down, 1 for up.

Reimplemented from **OsiSimpleInteger**.

#### 4.134.2.2 virtual double OsiUsesBiLinear::feasibleRegion ( **OsiSolverInterface** ∗ *solver,* const **OsiBranchingInformation** ∗ *info* ) const [virtual]

Set bounds to fix the variable at the current value.

Given an current value, set the lower and upper bounds to fix the variable. Returns amount it had to move variable.

Reimplemented from **OsiSimpleInteger**.

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

- CbcLinked.hpp

## 4.135 CbcHeuristicDive::PriorityType Struct Reference

Array of priorities.

`#include <CbcHeuristicDive.hpp>`

### 4.135.1 Detailed Description

Array of priorities.

Definition at line 160 of file CbcHeuristicDive.hpp.

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

- CbcHeuristicDive.hpp

## 4.136 PseudoReducedCost Struct Reference

### 4.136.1 Detailed Description

Definition at line 12 of file CbcHeuristicDive.hpp.

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

- CbcHeuristicDive.hpp

# Chapter 5

# File Documentation

## 5.1 CbcEventHandler.hpp File Reference

Event handling for cbc.

```
#include <cstddef>
#include <map>
```

Include dependency graph for CbcEventHandler.hpp:

## 5.2 CbcGenMessages.hpp File Reference

This file contains the enum that defines symbolic names for for cbc-generic messages.

This graph shows which files directly or indirectly include this file:

### Enumerations

- enum CbcGenMsgCode

  *Symbolic names for cbc-generic messages.*

### 5.2.1 Detailed Description

This file contains the enum that defines symbolic names for for cbc-generic messages.

### 5.2.2 Enumeration Type Documentation

#### 5.2.2.1 enum **CbcGenMsgCode**

Symbolic names for cbc-generic messages.

These are the 'internal IDs' for cbc-generic messages.

Definition at line 36 of file CbcGenMessages.hpp.

## 5.3 CbcSolver.hpp File Reference

Defines CbcSolver, the proposed top-level class for the new-style cbc solver.

```
#include <string>
#include <vector>
#include "CoinMessageHandler.hpp"
#include "OsiClpSolverInterface.hpp"
#include "CbcModel.hpp"
#include "CbcOrClpParam.hpp"
```

Include dependency graph for CbcSolver.hpp:

### Classes

- class CbcSolver

  *This allows the use of the standalone solver in a flexible manner.*
- struct CbcSolverUsefulData2

  *Structure to hold useful arrays.*
- class CbcSolverUsefulData

  *The CbcSolver class was taken out at a 9/12/09 meeting This is a feeble replacement.*
- class CbcUser

  *A class to allow the use of unknown user functionality.*
- class CbcStopNow

  *Support the use of a call back class to decide whether to stop.*

### Functions

- void CbcMain0 (CbcModel &babSolver, CbcSolverUsefulData &solverData)

  *And this uses it.*

### 5.3.1 Detailed Description

Defines CbcSolver, the proposed top-level class for the new-style cbc solver.

This class is currently an orphan. With the removal of all code flagged with the NEW_STYLE_SOLVER, this class is never instantiated (and cannot be instantiated). It is available to be coopted as a top-level object wrapping the current CbcMain0 and CbcMain1, should that appear to be a desireable path forward. – lh, 091211 –

## 5.4 CbcSolverAnalyze.hpp File Reference

Look to see if a constraint is all-integer (variables & coeffs), or could be all integer.

### 5.4.1 Detailed Description

Look to see if a constraint is all-integer (variables & coeffs), or could be all integer.

## 5.5 CbcSolverExpandKnapsack.hpp File Reference

Expanding possibilities of x∗y, where x∗y are both integers, constructing a knapsack constraint.

### 5.5.1 Detailed Description

Expanding possibilities of x∗y, where x∗y are both integers, constructing a knapsack constraint.

Results in a tighter model.

## 5.6 CbcSolverHeuristics.hpp File Reference

Routines for doing heuristics.

### Functions

- int doHeuristics (CbcModel ∗model, int type, **CbcOrClpParam** ∗parameters_, int numberParameters_, int no↩ Printing_, int initialPumpTune)

  *1 - add heuristics to model 2 - do heuristics (and set cutoff and best solution) 3 - for miplib test so skip some (out model later)*

### 5.6.1 Detailed Description

Routines for doing heuristics.

# Index