TUTORIALS IN
OPERATIONS RESEARCH
INFORMS 2005

ISBN 1-877640-21-2
DOI 10.1287/educ.1053.0020

CBC User Guide*

John Forrest, Robin Lougee-Heimer
Department of Mathematical Sciences, IBM T. J. Watson Research Center, IBM Research,
1101 Kitchawan Road, Yorktown Heights, New York 10598
{jjforre@us.ibm.com, robinlh@us.ibm.com}

Abstract The Computational Infrastructure for Operations Research (COIN-OR) branch-and-cut solver (CBC) is an open-source mixed-integer program (MIP) solver. The performance of branch-and-cut algorithms can vary greatly with problem-specific customization, such as dictating that the order nodes in the search tree are traversed. CBC provides operations research professionals with a well-tested, robust, reusable code base for experimenting with advanced customizations of branch-and-cut algorithms. The CBC design makes the most commonly desired customizations readily possible: (a) dynamically selecting the next node in the search tree for processing, (b) using specialized criteria for determining which variable(s) to branch on, (c) calling tailormade heuristics to generate MIP-feasible solutions quickly, (d) including standard (or user-provided) cut generation in solving the linear program (LP) relaxations of the MIP, and (e) invoking customized subproblem solvers.

CBC is written in C++ and is intended to be used primarily as a callable library. CBC requires a linear program (LP) solver. CBC uses the COIN-OR open solver interface (OSI) to communicate with the user's choice of LP solver. CBC can use any LP solver with an OSI. The LP solver expected to be used most commonly is the freely available COIN-OR LP solver (CLP). CBC can be used as a branch-and-bound solver or as a branch-and-cut solver. For cut generators, CBC relies on the COIN-OR Cut Generation Library (CGL). CBC can use any cut generator written to CGL standards.

CBC is an active open-source project led by John Forrest. The full CBC source code is available under the Common Public License for industrial and academic use at www.coin-or.org.

This chapter introduces CBC and illustrates how to implement a variety of common branch-and-cut customizations in CBC. The chapter assumes familiarity with C++, fundamentals of mixed-integer programming, and basic knowledge of OSI.

Keywords software; branch and bound; cutting planes; mixed-integer programming; open-source software

1. Introduction

The COIN[1] branch-and-cut solver (CBC) is an open-source mixed-integer program (MIP) solver written in C++. CBC is intended to be used primarily as a callable library to create customized branch-and-cut solvers. A basic, standalone executable version is also available. CBC is an active open-source project led by John Forrest at www.coin-or.org.

1.1. Prerequisites

The primary users of CBC are expected to be developers implementing customized branch-and-cut algorithms in C++, using CBC as a library. Consequently, this chapter assumes a

[1] The complete acronym is "COIN-OR" which stands for the Compuational Infrastructure for Operations Research. For simplicity (and in keeping with the directory and function names) we will simply use "COIN."

working knowledge of C++, including basic object-oriented programming terminology, and familiarity with the fundamental concepts of linear programming (LP) and mixed-integer programming (MIP).

CBC relies on other parts of the COIN repository. CBC needs an LP solver and relies on the COIN open solver interface (OSI) to communicate with the user's choice of solver. Any LP solver with an OSI can be used with CBC. The LP solver expected to be used most commonly is COIN's native linear program solver, CLP. For cut generators, CBC relies on the COIN Cut Generation Library (CGL). Any cut generator written to CGL standards can be used with CBC. Some of the cut generators in CGL rely on other parts of COIN, e.g., CGL's Gomory cut generator rely on the factorization functionality of `CoinFactorization`. This chapter assumes basic familiarity with OSI and CGL.

Technically speaking, CBC accesses the solver (and sometimes the model and data it contains) through an `OsiSolverInterface`. For the sake of simplicity, we will refer to the `OsiSolverInterface` as "the solver" in this chapter, rather than "the standard application programming interface to the solver." We hope any confusion caused by blurring this distinction will be mitigated by the shorter sentences.

In summary, readers should have the following prerequisites:

- C++ knowledge,
- LP and MIP fundamentals, and
- OSI familiarity.

Unless otherwise stated, we will assume the problem being optimized is a minimization problem. The terms "model" and "problem" are used synonymously.

1.2. Branch-and-Cut Overview

Before examining CBC in more detail, we tersely describe the basic branch-and-cut algorithm by way of example (which should really be called branch-and-cut-and-bound) and show the major C++ class(es) in CBC related to each step. The major CBC classes, labelled (A) through (F), are described in Table 1.

Step 1: *Bound.* Given an MIP model to minimize where some variables must take on integer values (e.g., 0, 1, or 2), relax the integrality requirements (e.g., consider each "integer"

Table 1. Associated classes.

Note	Class name	Description
(A)	`CbcBranch...`	These classes define the nature of MIP's discontinuity. The simplest discontinuity is a variable that must take an integral value. Other types of discontinuities exist, e.g., lot-sizing variables.
(B)	`CbcNode`	This class decides which variable/entity to branch on next. Even advanced users will probably only interact with this class by setting `CbcModel` parameters (e.g., priorities).
(C)	`CbcTree`	All unsolved models can be thought of as being nodes on a tree where each node (model) can branch two or more times. The user should not need to be concerned with this class.
(D)	`CbcCompare...`	These classes are used to determine which of the unexplored nodes in the tree to consider next. These classes are very small simple classes that can be tailored to suit the problem.
(E)	`CglCutGenerators`	Any cut generator from CGL can be used in CBC. The cut generators are passed to CBC with parameters that modify when each generator will be tried. All cut generators should be tried to determine which are effective. Few users will write their own cut generators.
(F)	`CbcHeuristics`	Heuristics are very important for obtaining valid solutions quickly. Some heuristics are available, but this is an area where it is useful and interesting to write specialized ones.

variable to be continuous with a lower bound of 0.0 and an upper bound of 2.0). Solve the resulting linear model with an LP solver to obtain a lower bound on the MIP's objective function value. If the optimal LP solution has integer values for the MIP's integer variables, we are finished. Any MIP-feasible solution provides an upper bound on the objective value. The upper bound equals the lower bound; the solution is optimal.

Step 2: *Branch.* Otherwise, there exists an "integer" variable with a nonintegral value. Choose one nonintegral variable (e.g., with value 1.3) (A)(B) and branch. Create two nodes, one with the branching variable having an upper bound of 1.0, and the other with the branching variable having a lower bound of 2.0. Add the two nodes to the search tree.

While (search tree is not empty)

Step 3: *Choose Node.* Pick a node off the tree (C)(D).

Step 4: *Reoptimize LP.* Create an LP relaxation and solve.

Step 5: *Bound.* Interrogate the optimal LP solution, and try to prune the node by one of the following.

- LP is infeasible, prune the node.
- Else, the optimal LP solution value of the node exceeds the current upper bound, prune the node.
- Else, the optimal LP solution of the node does not exceed the current upper bound and the solution is feasible to the MIP. Update the upper bound, and the best-known MIP solution, and prune the node by optimality.

Step 6: *Branch.* If we were unable to prune the node, then branch. Choose one nonintegral variable to branch on (A)(B). Create two nodes and add them to the search tree.

This is the outline of a branch-and-bound algorithm. If in optimizing the linear programs, we use cuts to tighten the LP relaxations (E)(F), then we have a branch-and-cut algorithm. (Note, if cuts are only used in Step 1, the method is called a cut-and-branch algorithm.)

There are a number of resources available to help new CBC users get started. This chapter is designed to be used in conjunction with the files in the Samples subdirectory of the main CBC directory (`COIN/Cbc/Samples`). The samples illustrate how to use CBC and may also serve as useful starting points for user projects. In the event that either this chapter or the available Doxygen content conflicts with the observed behavior of the source code, the comments in the header files, found in `COIN/Cbc/include`, are the ultimate reference.

2. The CBC Model Class

The main class in CBC is `CbcModel`. The `CbcModel` class is where most of the parameter setting is done. The absolute minimum number of actions taken with `CbcModel` is two:

- `CbcModel(OsiSolverInterface & linearSolver)` as constructor and
- `branchAndBound()` for solving the problem.

2.1. Simple Branch-and-Bound Example

The first sample program shows how to perform simple branch and bound with CBC. This program is short enough to present in full. Most of the remaining examples will take the form of small code fragments. The complete code for all the examples in this chapter can be found in the CBC Samples directory, `COIN/Cbc/Samples`.

Example 1. `minimum.cpp`

```
// Copyright (C) 2005, International Business Machines
// Corporation and others.  All Rights Reserved.

#include "CbcModel.hpp"

// Using CLP as the solver
#include "OsiClpSolverInterface.hpp"
```

```
int main (int argc, const char *argv[]) {
  OsiClpSolverInterface solver1;

  // Read in example model in MPS file format
  // and assert that it is a clean model
  int numMpsReadErrors = solver1.readMps("../../Mps/Sample/p0033.mps","");
  assert(numMpsReadErrors==0);

  // Pass the solver with the problem to be solved to CbcModel
  CbcModel model(solver1);

  // Do complete search
  model.branchAndBound();

  /* Print the solution.  CbcModel clones the solver so we
     need to get current copy from the CbcModel */
  int numberColumns = model.solver()->getNumCols();

  const double * solution = model.bestSolution();

  for (int iColumn=0;iColumn<numberColumns;iColumn++) {
    double value=solution[iColumn];
    if (fabs(value)>1.0e-7&&model.solver()->isInteger(iColumn))
      printf("%d has value %g\n",iColumn,value);
   }
  return 0;
}
```

The program in Example 1 creates a `OsiClpSolverInterface` solver interface (i.e., `solver1`) and reads an MPS file. If there are no errors, the program passes the problem to `CbcModel`, which solves the problem using the branch-and-bound algorithm. The part of the program that solves the problem is very small—one line!—but before that one line, the LP solver (i.e., `solver1`) had to be created and populated with the problem. After that one line, the results were printed out.

2.2. The Relationship Between OSI and CBC

The program in Example 1 illustrates the dependency of CBC on the `OsiSolverInterface` class. The constructor of `CbcModel` takes a pointer to an `OsiSolverInterface` (i.e., a solver). The `CbcModel` clones the solver, and uses its own instance of the solver. The `CbcModel`'s solver and the original solver (e.g., `solver1`) are not necessarily in sync unless the user synchronizes them. The user can always access the `CbcModel`'s solver through the `model()` class. To synchronize the two solvers, explicitly refreshing the original, e.g.,

```
solver1 = model.solver();
```

`CbcModel`'s method `solver()` returns a pointer to CBC's cloned solver.

For convenience, many of the OSI methods to access problem data have identical method names in `CbcModel`. (It is just more convenient to type `model.getNumCols()` rather than `model.solver()->getNumCols()`.) The `CbcModel` refreshes its solver at certain logical points during the algorithm. At these points, the information from the `CbcModel` model will match the information from the `model.solver()`. Elsewhere the information may vary. For instance, the method `CbcModel::bestSolution()` will contain the best solution so far, while the OSI method `getColSolution()` may not. In this case, it is safer to use `CbcModel::bestSolution()`.

While all the OSI methods used in `minimum.cpp` have equivalent methods in `CbcModel`, there are some OSI methods that do not. For example, if the program produced a lot of

undesired output, one might add the line

```
model.solver()->setHintParam(OsiDoReducePrint,true,OsiHintTry);
```

to reduce the output. There is no `setHintParam()` method in `CbcModel`.

2.3. Getting Solution Information and Impacting the Solution Process

Optimality can be checked through a call to `model.isProvenOptimal()`. Also available are `isProvenInfeasible()`, `isSolutionLimitReached()`, `isNodeLimitReached()`, or the feared `isAbandoned()`. There is also `int status()`, which returns 0 if finished (which includes the case when the algorithm is finished because it has been proved infeasible), 1 if stopped by user, and 2 if difficulties arose.

In addition to these `CbcModel` methods, solution values can be accessed via OSI methods (see Table 2). The OSI methods pick up the current solution in the `CbcModel`. The current solution will match the best solution found so far if called after `branchAndBound()` and a solution was found.

Most of the parameter setting in CBC is done through `CbcModel` methods. The most commonly used set and get methods are listed in Table 3.

`CbcModel` is extremely flexible and customizable. The class structure of CBC is designed to make the most commonly desired customizations of branch and cut possible. These include:

- selecting the next node to consider in the search tree,
- determining which variable to branch on,
- using heuristics to generate MIP-feasible solutions quickly,
- including cut generation when solving the LP-relaxations, and
- invoking customized subproblem solvers.

To enable this flexibility, `CbcModel` uses other classes in CBC (some of which are virtual and may have multiple instances). Not all classes are created equal. Tables 4 and 5 list in alphabetical order the classes used by `CbcModel` that are of most interest and of least interest. There is not much about the classes listed in Table 5 that the average user needs to know about.

3. Selecting the Next Node in the Search Tree

The order in which the nodes of the search tree are explored can strongly influence the performance of branch-and-cut algorithms. CBC gives users complete control over the search order. The search order is controlled via the `CbcCompare...` class. CBC provides an abstract

TABLE 2. Methods for getting solution information from OSI.

Purpose	Name	Notes
Primal-column solution	`const double * getColSolution()`	The OSI method will return the best solution found thus far, unless none has been found. It is safer to use `CbcModel` version, `CbcModel::bestSolution()`.
Dual-row solution	`const double * getRowPrice()`	Identical `CbcModel` version available, `CbcModel::getRowPrice()`.
Primal-row solution	`const double * getRowActivity()`	Identical `CbcModel` version available, `CbcModel::getRowActivity()`.
Dual-column solution	`const double * getReducedCost()`	Identical `CbcModel` version available, `CbcModel::getReducedCost()`.
Number of rows in model	`int getNumRows()`	Identical `CbcModel` version available, `CbcModel::getNumRows()`. Note: the number of rows can change due to cuts.
Number of columns in model	`int getNumCols()`	Identical `CbcModel` version available, `CbcModel::getNumCols()`.

TABLE 3. Useful set and get methods in `CbcModel`.

Method(s)	Description

`bool setMaximumNodes(int value)`
`int getMaximumNodes() const`
`bool setMaximumSeconds(double value)`
`double getMaximumSeconds()`
`bool setMaximumSolutions(double value)`
`double getMaximumSolutions() const`
	These set methods tell CBC to stop after a given number of nodes, seconds, or solutions is reached. The get methods return the corresponding values.

`bool setIntegerTolerance(double value)` `const double getIntegerTolerance() const`	An integer variable is deemed to be at an integral value if it is no further than this *value* (tolerance) away.

`bool setAllowableGap(double value)`
`double getAllowableGap() const`
`bool setAllowablePercentageGap (double value)`
`double getAllowablePercentageGap() const`
`bool setAllowableFractionGap(double value)`
`double getAllowableFractionGap() const`
	`CbcModel` returns if the gap between the best known solution and the best possible solution is less than this *value*, as a percentage or as a fraction.

`void setNumberStrong(double value)` `int numberStrong() const`	These methods set or get the maximum number of candidates at a node to be evaluated for strong branching.
`void setPrintFrequency(int value)` `int printFrequency() const`	Controls the number of nodes evaluated between status prints. Print frequency has a very slight overhead, if *value* is small.
`int getNodeCount() const`	Returns number of nodes evaluated in the search.
`int numberRowsAtContinuous() const`	Returns number of rows in the problem when handed to the solver (i.e., before cuts were added). Commonly used in implementing heuristics.
`int numberIntegers() const` `const int * integerVariable() const`	Returns number of integer variables and an array specifying them.

`bool isBinary(int colIndex) const`
`bool isContinuous(int colIndex) const`
`bool isInteger(int colIndex) const`
	Returns information on variable *colIndex*. OSI methods can be used to set these attributes (before handing the model to `CbcModel`).

`double getObjValue() const`	This method returns the best objective value so far.
`double getCurrentObjValue() const`	This method returns the current objective value.
`const double * getObjCoefficients() const`	This method returns the objective coefficients.

`const double * getRowLower() const`
`const double * getRowUpper() const`
`const double * getColLower() const`
`const double * getColUpper() const`
	These methods return the lower and upper bounds on row and column activities.

`const CoinPackedMatrix * getMatrixByRow() const`	This method returns a pointer to a row copy of matrix stored as a `CoinPackedMatrix`, which can be further examined.
`const CoinPackedMatrix * getMatrixByCol() const`	This method returns a pointer to a column copy of matrix stored as a `CoinPackedMatrix`, which can be further examined.
`CoinBigIndex getNumElements() const`	Returns the number of nonzero elements in the problem matrix.
`void setObjSense(double value)` `double getObjSense() const`	These methods set and get the objective sense. The parameter *value* should be $+1$ to minimize and -1 to maximize.

Notes. The method `numberStrong` (and some of the others) does not follow the "get" convention. The convention has changed over time, and there are still some inconsistencies to be cleaned up. Also, `CoinBigIndex` is a `typedef`, which in most cases is the same as int.

base class, `CbcCompareBase`, and several commonly used instances that are described in Table 6.

It is relatively simple for a user to create new compare class instances. The code in Example 2 describes how to build a new comparison class and the reasoning behind it. The complete source can be found in `CbcCompareUser.hpp` and `CbcCompareUser.cpp`, located in the CBC Samples directory (see §7). The key method in `CbcCompare` is `bool test(CbcNode* x, CbcNode* y))`, which returns *true* if node *y* is preferred over node *x*. In the `test()`

TABLE 4. Classes used by `CbcModel`—Most useful.

Class name	Description	Notes
`CbcCompareBase`	Controls which node on the tree is selected.	The default is `CbcCompareDefault`. Other comparison classes in `CbcCompareActual.hpp` include `CbcCompareDepth` and `CbcCompareObjective`. Experimenting with these classes and creating new compare classes is easy.
`CbcCutGenerator`	A wrapper for `CglCutGenerator` with additional data to control when the cut generator is invoked during the tree search.	Other than knowing how to add a cut generator to `CbcModel`, there is not much the average user needs to know about this class. However, sophisticated users can implement their own cut generators.
`CbcHeuristic`	Heuristic that attempts to generate valid MIP-solutions leading to good upper bounds.	Specialized heuristics can dramatically improve branch-and-cut performance. As many different heuristics as desired can be used in CBC. Advanced users should consider implementing custom heuristics when tackling difficult problems.
`CbcObject`	Defines what it means for a variable to be satisfied.	Used in branching. Virtual class. CBC's concept of branching is based on the idea of an "object." An object has (i) a feasible region, (ii) can be evaluated for infeasibility, (iii) can be branched on, e.g., a method of generating a branching object, which defines an up branch and a down branch, and (iv) allows comparison of the effect of branching. Instances of objects include `CbcSimpleInteger`, `CbcSimpleIntegerPseudoCosts`, `CbcClique`, `CbcSOS` (Type 1 and 2), `CbcFollowOn`, and `CbcLotsize`.
`OsiSolverInterface`	Defines the LP solver being used and the LP model. Normally a pointer to the desired `OsiSolverInterface` is passed to `CbcModel` before branch-and-cut.	Virtual class. The user instantiates the solver interface of their choice, e.g., `OsiClpSolverInterface`.

method, information from `CbcNode` can easily be used. Table 7 lists some commonly used methods to access information at a node.

The node desired in the tree is often a function of how the search is progressing. In the design of CBC, there is no information on the state of the tree. The CBC is designed so that the method `newSolution()` is called whenever a solution is found, and the method `every1000Nodes()` is called every 1,000 nodes. When these methods are called, the user has the opportunity to modify the behavior of `test()` by adjusting their common variables (e.g., `weight_`). Because `CbcNode` has a pointer to the model, the user can also influence the search through actions such as changing the maximum time CBC is allowed, once a solution has been found (e.g., `CbcModel::setMaximumSeconds(double value)`). In `CbcCompareUser.cpp` of the `COIN/Cbc/Samples` directory, four items of data are used:

(1) the number of solutions found so far;

(2) the size of the tree (defined to be the number of active nodes);

(3) a weight, `weight_`, which is initialized to -1.0; and

(4) a saved value of weight, `saveWeight_` (for when weight is set back to -1.0 for a special reason).

TABLE 5. Classes used by `CbcModel`—Least useful.

Class name	Description	Notes
`CbcBranchDecision`	Used in choosing which variable to branch on, however, most of the work is done by the definitions in `CbcObject`.	Defaults to `CbcBranchDefaultDecision`.
`CbcCountRowCut`	Interface to `OsiRowCut`. It counts the usage so cuts can gracefully vanish.	See `OsiRowCut` for more details.
`CbcNode`	Controls which variable/entity is selected to be branched on.	Controlled via `CbcModel` parameters. Information from `CbcNode` can be useful in creating customized node selection rules.
`CbcNodeInfo`	Contains data on bounds, basis, etc., for one node of the search tree.	Header is located in `CbcNode.hpp`.
`CbcTree`	Defines how the search tree is stored.	This class can be changed, but it is not likely to be modified.
`CoinMessageHandler`	Deals with message handling.	The user can inherit from `CoinMessageHandler` to specialize message handling.
`CoinWarmStartBasis`	Basis representation to be used by solver.	

The full code for the `CbcCompareUser::test()` method is given in Example 2.

Example 2. `CbcCompareUser::test()`

```
// Returns true if y better than x
bool CbcCompareUser::test (CbcNode * x, CbcNode * y) {
  if (weight_==-1.0) {
    // before solution
    if (x->numberUnsatisfied() > y->numberUnsatisfied())
      return true;
    else if (x->numberUnsatisfied() < y->numberUnsatisfied())
      return false;
    else
      return x->depth() < y->depth();
  } else {
    // after solution.
    // note: if weight_=0, comparison is based
    //       solely on objective value
    double weight = CoinMax(weight_,0.0);
    return x->objectiveValue()+ weight*x->numberUnsatisfied() >
      y->objectiveValue() + weight*y->numberUnsatisfied();
  }
}
```

Initially, `weight_` is -1.0 and the search is biased toward depth first. In fact, `test()` prefers y if y has fewer unsatisfied variables. In the case of a tie, `test()` prefers the node with the greater depth in the tree.

Once a solution is found, `newSolution()` is called. The method `newSolution()` interacts with `test()` by means of the variable `weight_`. If the solution was achieved by branching, a calculation is made to determine the cost per unsatisfied integer variable to go from the continuous solution to an integer solution. The variable `weight_` is then set to aim at a slightly better solution. From then on, `test()` returns true if it seems that y will lead to a better solution than x. This source for `newSolution()` is given in Example 3.

Table 6. Compare classes provided.

Class name	Description
`CbcCompareDepth`	This will always choose the node deepest in the tree. It gives minimum tree size, but may take a long time to find the best solution.
`CbcCompareObjective`	This will always choose the node with the best objective value. This may give a very large tree. It is likely that the first solution found will be the best and the search should finish soon after the first solution is found.
`CbcCompareDefault`	This is designed to do a mostly depth-first search until a solution has been found. It then uses estimates that are designed to give a slightly better solution. If a reasonable number of nodes have been explored (or a reasonable number of solutions found), then this class will adopt a breadth-first search (i.e., making a comparison based strictly on objective function values) unless the tree is very large, in which case it will revert to depth-first search.
`CbcCompareEstimate`	When pseudo costs are invoked, they can be used to guess a solution. This class uses the guessed solution.

Table 7. Information available from `CbcNode`.

Class name	Description
`double objectiveValue() const`	Value of objective at the node.
`int numberUnsatisfied() const`	Number of unsatisfied integers (assuming branching object is an integer—otherwise it might be number of unsatisfied sets).
`int depth() const`	Depth of the node in the search tree.
`double guessedObjectiveValue() const`	If user was setting this (e.g., if using pseudo costs).
`int way() const`	The way in which branching would next occur from this node (for more advanced use).
`int variable() const`	The branching "variable" (associated with the `CbcBranchingObject`—for more advanced use).

Example 3. `CbcCompareUser::newSolution()`

```
// This allows the test() method to change behavior by resetting weight_.
// It is called after each new solution is found.
void CbcCompareUser::newSolution(CbcModel * model,
                   double objectiveAtContinuous,
                   int numberInfeasibilitiesAtContinuous)
{
  if (model->getSolutionCount()==model->getNumberHeuristicSolutions())
    return; // solution was found by rounding so ignore it.

  // set weight_ to get close to this solution
  double costPerInteger =
    (model->getObjValue()-objectiveAtContinuous)/
    ((double) numberInfeasibilitiesAtContinuous);
  weight_ = 0.98*costPerInteger;
  saveWeight_=weight_;
  numberSolutions_++;
  if (numberSolutions_>5)
    weight_ =0.0; // comparison in test() will be
                  // based strictly on objective value.
}
```

As the search progresses, the comparison can be modified. If many nodes (or many solutions) have been generated, then `weight_` is set to 0.0, leading to a breadth-first search. Breadth-first search can lead to an enormous tree. If the tree size exceeds 10,000, it may be desirable to return to a search biased toward depth first. Changing the behavior in this manner is done by the method `every1000Nodes`, shown in Example 4.

Example 4. `CbcCompareUser::every1000Nodes()`

```
// This allows the test() method to change behavior every 1000 nodes
bool CbcCompareUser::every1000Nodes(CbcModel * model, int
numberNodes) {
  if (numberNodes>10000)
    weight_ =0.0; // compare nodes based on objective value
  // get size of tree
  treeSize_ = model->tree()->size();
  if (treeSize_>10000) {
    // set weight to reduce size most of time
    if (treeSize_>20000)
      weight_=-1.0;
    else if ((numberNodes%4000)!=0)
      weight_=-1.0;
    else
      weight_=saveWeight_;
  }
  return numberNodes==11000; // resort if first time
}
```

4. Getting Good Bounds in CBC

In practice, it is very useful to get a good solution reasonably fast. Any MIP-feasible solution produces an upper bound, and a good bound will greatly reduce the run time. Good solutions can satisfy the user on very large problems where a complete search is impossible. Obviously, heuristics are problem dependent, although some do have more general use. At present there is only one heuristic in CBC itself, `CbcRounding`. Hopefully, the number will grow. Other heuristics are in the `COIN/Cbc/Samples` directory. A heuristic tries to obtain a solution to the original problem so that it only needs to consider the original rows and does not have to use the current bounds. CBC provides an abstract base class `CbcHeuristic` and a rounding heuristic in CBC.

This chapter describes how to build a greedy heuristic for a set-covering problem, e.g., the miplib problem fast0507. A more general (and efficient) version of the heuristic is in `CbcHeuristicGreedy.hpp` and `CbcHeuristicGreedy.cpp`, located in the `COIN/Cbc/Samples` directory (see §7).

The greedy heuristic will leave all variables taking value 1 at this node of the tree at value 1, and will initially set all other variables to value 0. All variables are then sorted in order of their cost divided by the number of entries in rows that are not yet covered. (We may randomize that value a bit so that ties will be broken in different ways on different runs of the heuristic.) The best one is choosen, and set to 1. The process is repeated. Because this is a set-covering problem (i.e., all constraints are $\geq$), the heuristic is guaranteed to find a solution (but not necessarily an improved solution). The speed of the heuristic could be improved by just redoing those affected, but for illustrative purposes we will keep it simple. (The speed could also be improved if all elements equal 1.)

The key `CbcHeuristic` method is `int solution(double & solutionValue, double * betterSolution)`. The `solution()` method returns 0 if no solution found, and returns 1 if a solution is found, in which case it fills in the objective value and primal solution. The code in `CbcHeuristicGreedy.cpp` is a little more complicated than this following example. For instance, the code here assumes all variables are integer. The important bit of data is a copy of the matrix (stored by column) before any cuts have been made. The data used are bounds, objective, and the matrix, plus two work arrays.

Example 5. Data

```
OsiSolverInterface * solver = model_->solver();
  // Get solver from CbcModel
const double * columnLower = solver->getColLower(); // Column Bounds
const double * columnUpper = solver->getColUpper();
const double * rowLower = solver->getRowLower();
      // We know we only need lower bounds
const double * solution = solver->getColSolution();
const double * objective = solver->getObjCoefficients();
      // In code we also use min/max
double integerTolerance =
  model_->getDblParam(CbcModel::CbcIntegerTolerance);
double primalTolerance;
solver->getDblParam(OsiPrimalTolerance,primalTolerance);
int numberRows = originalNumberRows_;
      // This is number of rows when matrix was passed in
// Column copy of matrix (before cuts)
const double * element = matrix_.getElements();
const int * row = matrix_.getIndices();
const CoinBigIndex * columnStart = matrix_.getVectorStarts();
const int * columnLength = matrix_.getVectorLengths();

// Get solution array for heuristic solution
int numberColumns = solver->getNumCols();
double * newSolution = new double [numberColumns];
// And to sum row activities
double * rowActivity = new double[numberRows];
```

The `newSolution` is then initialized to the rounded-down solution.

Example 6. Initialize `newSolution`

```
for (iColumn=0;iColumn<numberColumns;iColumn++) {
  CoinBigIndex j;
  double value = solution[iColumn];
  // Round down integer
  if (fabs(floor(value+0.5)-value)<integerTolerance)
    value=floor(CoinMax(value+1.0e-3,columnLower[iColumn]));
  // make sure clean
  value = CoinMin(value,columnUpper[iColumn]);
  value = CoinMax(value,columnLower[iColumn]);
  newSolution[iColumn]=value;
  if (value) {
    double cost = objective[iColumn];
    newSolutionValue += value*cost;
    for (j=columnStart[iColumn];
         j<columnStart[iColumn]+columnLength[iColumn];j++) {
      int iRow=row[j];
      rowActivity[iRow] += value*element[j];
    }
  }
}
```

At this point, some row activities are below their lower bound. To correct the infeasibility, the variable that is cheapest in reducing the sum of infeasibilities is found and updated, and the process repeats. This is a finite process. (The implementation could be faster, but is kept simple for illustrative purposes.)

Example 7. Create Feasible `newSolution` from Initial `newSolution`

```
while (true) {
  // Get column with best ratio
  int bestColumn=-1;
  double bestRatio=COIN_DBL_MAX;
```

```
  for (int iColumn=0;iColumn<numberColumns;iColumn++) {
    CoinBigIndex j;
    double value = newSolution[iColumn];
    double cost = direction * objective[iColumn];
    // we could use original upper rather than current
    if (value+0.99<columnUpper[iColumn]) {
      double sum=0.0; // Compute how much we will reduce infeasibility by
      for (j=columnStart[iColumn];
           j<columnStart[iColumn]+columnLength[iColumn];j++) {
        int iRow=row[j];
        double gap = rowLower[iRow]-rowActivity[iRow];
        if (gap>1.0e-7) {
          sum += CoinMin(element[j],gap);
        if (element[j]+rowActivity[iRow]<rowLower[iRow]+1.0e-7) {
          sum += element[j];
    }
  }
      if (sum>0.0) {
    double ratio = (cost/sum)*(1.0+0.1*CoinDrand48());
    if (ratio<bestRatio) {
          bestRatio=ratio;
          bestColumn=iColumn;
        }
      }
    }
  }
  if (bestColumn<0)
    break; // we have finished
  // Increase chosen column
  newSolution[bestColumn] += 1.0;
  double cost = direction * objective[bestColumn];
  newSolutionValue += cost;
  for (CoinBigIndex j=columnStart[bestColumn];
       j<columnStart[bestColumn]+columnLength[bestColumn];j++) {
    int iRow = row[j];
    rowActivity[iRow] += element[j];
  }
}
```

A solution value of `newSolution` is compared to the best solution value. If `newSolution` is an improvement, its feasibility is validated.

Example 8. Check Solution Quality of `newSolution`

```
returnCode=0; // 0 means no good solution
if (newSolutionValue<solutionValue) { // minimization
  // check feasible
  memset(rowActivity,0,numberRows*sizeof(double));
  for (iColumn=0;iColumn<numberColumns;iColumn++) {
    CoinBigIndex j;
    double value = newSolution[iColumn];
    if (value) {
  for (j=columnStart[iColumn];
       j<columnStart[iColumn]+columnLength[iColumn];j++) {
    int iRow=row[j];
    rowActivity[iRow] += value*element[j];
      }
    }
  }
  // check was approximately feasible
  bool feasible=true;
  for (iRow=0;iRow<numberRows;iRow++) {
    if(rowActivity[iRow]<rowLower[iRow]) {
```

```
    if (rowActivity[iRow]<rowLower[iRow]-10.0*primalTolerance)
      feasible = false;
      }
    }
    if (feasible) {
      // new solution
      memcpy(betterSolution,newSolution,numberColumns*sizeof(double));
      solutionValue = newSolutionValue;
      // We have good solution
      returnCode=1;
    }
  }
```

5. Branching

CBC's concept of branching is based on the idea of an "object." An object has (i) a feasible region; (ii) can be evaluated for infeasibility; (iii) can be branched on, e.g., a method of generating a branching object, which defines an up branch and a down branch; and (iv) allows comparison of the effect of branching. Instances of objects include

(1) `CbcSimpleInteger`,
(2) `CbcSimpleIntegerPseudoCosts`,
(3) `CbcClique`,
(4) `CbcSOS` (Type 1 and 2),
(5) `CbcFollowOn`, and
(6) `CbcLotsize`.

In this section we give examples of how to use existing branching objects.

5.1. Pseudo Cost Branching

If the user declares variables as integer but does no more, then CBC will treat them as simple integer variables. In many cases, the user would like to do some more fine-tuning. This section shows how to create integer variables with pseudo costs. When pseudo costs are given, then it is assumed that if a variable is at 1.3, then the cost of branching that variable down will be 0.3 times the down pseudo cost and the cost of branching up would be 0.7 times the up pseudo cost. Pseudo costs can be used both for branching and for choosing a node. The full code is in `longthin.cpp`, located in the CBC Samples directory (see §7).

The idea is simple for set-covering problems. Branching up gets us much closer to an integer solution, so we will encourage that direction by branching up if variable value is greater than one-third. The expected cost of going up obviously depends on the cost of the variable. The pseudo costs are chosen to reflect that fact.

Example 9. `CbcSimpleIntegerPseudoCosts`

```
int iColumn;
int numberColumns = solver3->getNumCols();
// do pseudo costs
CbcObject ** objects = new CbcObject * [numberColumns];
// Point to objective
const double * objective = model.getObjCoefficients();
int numberIntegers=0;
for (iColumn=0;iColumn<numberColumns;iColumn++) {
  if (solver3->isInteger(iColumn)) {
    double cost = objective[iColumn];
    CbcSimpleIntegerPseudoCost * newObject =
      new CbcSimpleIntegerPseudoCost(&model,numberIntegers,iColumn,
                                     2.0*cost,cost);
    newObject->setMethod(3);
    objects[numberIntegers++]= newObject;
  }
```

```
}
// Now add in objects (they will replace simple integers)
model.addObjects(numberIntegers,objects);
for (iColumn=0;iColumn<numberIntegers;iColumn++)
  delete objects[iColumn];
delete [] objects;
```

The code in Example 9 also tries to give more importance to variables with more coefficients. Whether this sort of thing is worthwhile should be the subject of experimentation.

5.2. Follow-on Branching

In crew scheduling, the problems are long and thin. A problem may have a few rows, but many thousands of variables. Branching a variable to 1 is very powerful as it fixes many other variables to 0, but branching to 0 is very weak, as thousands of variables can increase from 0. In crew-scheduling problems, each constraint is a flight leg, e.g., JFK airport to DFW airport. From DFW there may be several flights the crew could take next—suppose one flight is the 9:30 flight from DFW to LAX airport. A binary branch is that the crew arriving at DFW either takes the 9:30 flight to LAX or they do not. This "follow-on" branching does not fix individual variables. Instead this branching divides all the variables with entries in the JFK-DFW constraint into two groups—those with entries in the DFW-LAX constraint and those without entries.

The full sample code for follow-on branching is in `crew.cpp`, located in the CBC Samples directory in §7. In this case, the simple integer variables are left, which may be necessary if other sorts of constraints exist. Follow-on branching rules are to be considered first, so the priorities are set to indicate that the follow-on rules take precedence, where Priority 1 is the highest priority.

Example 10. `CbcFollowOn`

```
int iColumn;
int numberColumns = solver3->getNumCols();
/* We are going to add a single follow-on object but we
   want to give low priority to existing integers
   As the default priority is 1000 we don't actually need to give
   integer priorities but it is here to show how.
*/
// Normal integer priorities
int * priority = new int [numberColumns];
int numberIntegers=0;
for (iColumn=0;iColumn<numberColumns;iColumn++) {
  if (solver3->isInteger(iColumn)) {
    priority[numberIntegers++]= 100; // low priority
  }
}
/* Second parameter is set to true for objects,
   and false for integers. This indicates integers */
model.passInPriorities(priority,false);
delete [] priority;
/* Add in objects before we can give them a priority.
   In this case just one object
   - but it shows the general method
*/
CbcObject ** objects = new CbcObject * [1];
objects[0]=new CbcFollowOn(&model);
model.addObjects(1,objects);
delete objects[0];
delete [] objects;
// High priority
int followPriority=1;
model.passInPriorities(&followPriority,true);
```

6. Advance Solver Uses

CBC uses a generic `OsiSolverInterface` and its `resolve` capability. This does not give much flexibility, so advanced users can inherit from their interface of choice. This section illustrates how to implement such a solver for a long thin problem, e.g., fast0507 again. As with the other examples in this chapter, the sample code is not guaranteed to be the fastest way to solve the problem. The main purpose of the example is to illustrate techniques. The full source is in `CbcSolver2.hpp` and `CbcSolver2.cpp` located in the CBC Samples directory (see §7).

The method `initialSolve` is called a few times in CBC, and provides a convenient starting point. The `modelPtr_` derives from `OsiClpSolverInterface`.

Example 11. `initialSolve()`

```
// modelPtr_ is of type ClpSimplex *
modelPtr_->setLogLevel(1); // switch on a bit of printout
modelPtr_->scaling(0); // We don't want scaling for fast0507
setBasis(basis_,modelPtr_); // Put basis into ClpSimplex
// Do long thin by sprint
ClpSolve options;
options.setSolveType(ClpSolve::usePrimalorSprint);
options.setPresolveType(ClpSolve::presolveOff);
options.setSpecialOption(1,3,15); // Do 15 sprint iterations
modelPtr_->initialSolve(options); // solve problem
basis_ = getBasis(modelPtr_); // save basis
modelPtr_->setLogLevel(0); // switch off printout
```

The `resolve()` method is more complicated than `initialSolve()`. The main pieces of data are a counter `count_` (which is incremented each solve), and an integer array `node_` (which stores the last time a variable was active in a solution). For the first few times, the normal dual simplex is called and `node_` array is updated.

Example 12. First Few Solves

```
if (count_<10) {
  OsiClpSolverInterface::resolve(); // Normal resolve
  if (modelPtr_->status()==0) {
    count_++; // feasible - save any nonzero or basic
    const double * solution = modelPtr_->primalColumnSolution();
    for (int i=0;i<numberColumns;i++) {
  if (solution[i]>1.0e-6||modelPtr_->getStatus(i)==ClpSimplex::basic) {
    node_[i]=CoinMax(count_,node_[i]);
    howMany_[i]++;
  }
    }
  } else {
    printf("infeasible early on\n");
  }
}
```

After the first few solves, only those variables that took part in a solution in the last so many solves are used. As fast0507 is a set-covering problem, any rows that are already covered can be taken out.

Example 13. Create Small Subproblem

```
  int * whichRow = new int[numberRows]; // Array to say which rows used
  int * whichColumn = new int [numberColumns]; // Array to say which
      columns used
  int i;
  const double * lower = modelPtr_->columnLower();
  const double * upper = modelPtr_->columnUpper();
  setBasis(basis_,modelPtr_); // Set basis
  int nNewCol=0; // Number of columns in small model
```

```
// Column copy of matrix
const double * element = modelPtr_->matrix()->getElements();
const int * row = modelPtr_->matrix()->getIndices();
const CoinBigIndex * columnStart =
    modelPtr_->matrix()->getVectorStarts();
const int * columnLength = modelPtr_->matrix()->getVectorLengths();

int * rowActivity = new int[numberRows];
            // Number of columns with entries in each row
memset(rowActivity,0,numberRows*sizeof(int));
int * rowActivity2 = new int[numberRows];
            // Lower bound on row activity for each row
memset(rowActivity2,0,numberRows*sizeof(int));
char * mark = (char *) modelPtr_->dualColumnSolution();
            // Get some space to mark columns
memset(mark,0,numberColumns);
for (i=0;i<numberColumns;i++) {
  bool choose = (node_[i]>count_-memory_&&node_[i]>0);
            // Choose if used recently
  // Take if used recently or active in some sense
  if ((choose&&upper[i])
  ||(modelPtr_->getStatus(i)!=ClpSimplex::atLowerBound&&
        modelPtr_->getStatus(i)!=ClpSimplex::isFixed)
  ||lower[i]>0.0) {
    mark[i]=1; // mark as used
whichColumn[nNewCol++]=i; // add to list
    CoinBigIndex j;
    double value = upper[i];
    if (value) {
      for (j=columnStart[i];
           j<columnStart[i]+columnLength[i];j++) {
        int iRow=row[j];
        assert (element[j]==1.0);
        rowActivity[iRow] ++; // This variable can cover this row
      }
      if (lower[i]>0.0) {
        for (j=columnStart[i];
             j<columnStart[i]+columnLength[i];j++) {
          int iRow=row[j];
          rowActivity2[iRow] ++; // This row redundant
        }
      }
    }
  }
}
int nOK=0; // Use to count rows which can be covered
int nNewRow=0; // Use to make list of rows needed
for (i=0;i<numberRows;i++) {
  if (rowActivity[i])
    nOK++;
  if (!rowActivity2[i])
    whichRow[nNewRow++]=i; // not satisfied
  else
    modelPtr_->setRowStatus(i,ClpSimplex::basic); // make slack basic
}
if (nOK<numberRows) {
  // The variables we have do not cover rows - see if we can find
    any that do
  for (i=0;i<numberColumns;i++) {
    if (!mark[i]&&upper[i]) {
      CoinBigIndex j;
      int good=0;
```

```
      for (j=columnStart[i];
           j<columnStart[i]+columnLength[i];j++) {
        int iRow=row[j];
        if (!rowActivity[iRow]) {
          rowActivity[iRow] ++;
          good++;
        }
      }
      if (good) {
        nOK+=good; // This covers - put in list
        whichColumn[nNewCol++]=i;
      }
    }
  }
}
delete [] rowActivity;
delete [] rowActivity2;
if (nOK<numberRows) {
  // By inspection the problem is infeasible - no need to solve
  modelPtr_->setProblemStatus(1);
  delete [] whichRow;
  delete [] whichColumn;
  printf("infeasible by inspection\n");
  return;
}
// Now make up a small model with the right rows and columns
ClpSimplex *temp=
    new ClpSimplex(modelPtr_,nNewRow,whichRow,nNewCol,whichColumn);
```

If the variables cover the rows, then the problem is feasible (no cuts are being used). (If the rows were equality constraints, then this might not be the case. More work would be needed.) After the solution, the reduced costs are checked. If any reduced costs are negative, the code goes back to the full problem and cleans up with primal simplex.

Example 14. Check Optimal Solution

```
temp->setDualObjectiveLimit(1.0e50);
           // Switch off dual cutoff as problem is restricted
temp->dual(); // solve
double * solution = modelPtr_->primalColumnSolution();
    // put back solution
const double * solution2 = temp->primalColumnSolution();
memset(solution,0,numberColumns*sizeof(double));
for (i=0;i<nNewCol;i++) {
  int iColumn = whichColumn[i];
  solution[iColumn]=solution2[i];
  modelPtr_->setStatus(iColumn,temp->getStatus(i));
}
double * rowSolution = modelPtr_->primalRowSolution();
const double * rowSolution2 = temp->primalRowSolution();
double * dual = modelPtr_->dualRowSolution();
const double * dual2 = temp->dualRowSolution();
memset(dual,0,numberRows*sizeof(double));
for (i=0;i<nNewRow;i++) {
  int iRow=whichRow[i];
  modelPtr_->setRowStatus(iRow,temp->getRowStatus(i));
  rowSolution[iRow]=rowSolution2[i];
  dual[iRow]=dual2[i];
}
// See if optimal
double * dj = modelPtr_->dualColumnSolution();
// get reduced cost for large problem
// this assumes minimization
```

```
memcpy(dj,modelPtr_->objective(),numberColumns*sizeof(double));
modelPtr_->transposeTimes(-1.0,dual,dj);
modelPtr_->setObjectiveValue(temp->objectiveValue());
modelPtr_->setProblemStatus(0);
int nBad=0;

for (i=0;i<numberColumns;i++) {
  if (modelPtr_->getStatus(i)==ClpSimplex::atLowerBound
      &&upper[i]>lower[i]&&dj[i]<-1.0e-5)
    nBad++;
}
// If necessary clean up with primal (and save some statistics)
if (nBad) {
  timesBad_++;
  modelPtr_->primal(1);
  iterationsBad_ += modelPtr_->numberIterations();
}
```

The array `node_` is updated for the first few solves. To give some idea of the effect of this tactic, the problem fast0507 has 63,009 variables, but the small problem never has more than 4,000 variables. In only about 10% of solves was it necessary to re-solve, and then the average number of iterations on full problem was less than 20.

Quadratic MIP. To give another example—again only for illustrative purposes—it is possible to do quadratic MIP with CBC. In this case, we make `resolve` the same as `initialSolve`. The full code is in `ClpQuadInterface.hpp` and `ClpQuadInterface.cpp`, located in the CBC Samples directory (see §7).

Example 15. Solving a Quadratic MIP

```
// save cutoff
double cutoff = modelPtr_->dualObjectiveLimit();
modelPtr_->setDualObjectiveLimit(1.0e50);
modelPtr_->scaling(0);
modelPtr_->setLogLevel(0);
// solve with no objective to get feasible solution
setBasis(basis_,modelPtr_);
modelPtr_->dual();
basis_ = getBasis(modelPtr_);
modelPtr_->setDualObjectiveLimit(cutoff);
if (modelPtr_->problemStatus())
  return; // problem was infeasible
// Now pass in quadratic objective
ClpObjective * saveObjective  = modelPtr_->objectiveAsObject();
modelPtr_->setObjectivePointer(quadraticObjective_);
modelPtr_->primal();
modelPtr_->setDualObjectiveLimit(cutoff);
if (modelPtr_->objectiveValue()>cutoff)
  modelPtr_->setProblemStatus(1);
modelPtr_->setObjectivePointer(saveObjective);
```

7. More Samples

The CBC distribution includes a number of `.cpp` sample files. Users are encouraged to use them as starting points for their own CBC projects. The files can be found in the `COIN/Cbc/Samples/` directory. For the latest information on compiling and running these samples, please see the file `COIN/Cbc/Samples/INSTALL`. Most of them can be built by

```
make DRIVER=name,
```

which produces an executable `testit`. Tables 8 and 9 provide lists of some of the most useful sample files with a short description for each file.

8. Messages

Messages and codes passed by CBC are listed in the tables below. For a complete list, see `COIN/Cbc/CbcMessages.cpp`. The notation used is the same as for the `printf` in the C programming language.

- `%s` is a string
- `%d` is an integer
- `%g` or `%f` is a floating point value

There are several log levels. Setting the log level to be i produces the log messages for level i and all levels less than i.

- Log Level 0: Switches off all CBC messages but one (see Table 10).
- Log Level 1: The default (see Table 11).
- Log Level 2: Substantial amount of information, e.g., message 15 is generated once per node. Can be useful when the evaluation at each node is slow (see Table 12).
- Log Level 3: Tremendous amount of information, e.g., multiple messages per node (see Table 13).

Table 8. Basic samples.

Source file	Description
`minimum.cpp`	This is a CBC "Hello, world" program. It reads a problem in MPS file format and solves the problem using simple branch and bound.
`sample2.cpp`	This is designed to be a file that a user could modify to get a useful driver program for his or her project. In particular, it demonstrates the use of CGL's preprocess functionality. It uses `CbcBranchUser.cpp`, `CbcCompareUser.cpp`, and `CbcHeuristicUser.cpp` with corresponding `*.hpp` files.

Table 9. Advanced samples.

Source file	Description
`crew.cpp`	This sample shows the use of advanced branching and a use of priorities. It uses `CbcCompareUser.cpp` with corresponding `*.hpp` files.
`longthin.cpp`	This sample shows the advanced use of a solver. It also has coding for a greedy heuristic. The solver is given in `CbcSolver2.hpp` and `CbcSolver2.cpp`. The heuristic is given in `CbcHeuristicGreedy.hpp` and `CbcHeuristicGreedy.cpp`. It uses `CbcBranchUser.cpp` and `CbcCompareUser.cpp` with corresponding `*.hpp` files.
`qmip.cpp`	This solves a quadratic MIP. It is to show advanced use of a solver. The solver is given in `ClpQuadInterface.hpp` and `ClpQuadInterface.cpp`. It uses `CbcBranchUser.cpp` and `CbcCompareUser.cpp` with corresponding `*.hpp` files.
`sos.cpp`	This artificially creates a special ordered set problem.
`lotsize.cpp`	This artificially creates a lot-sizing problem.

Table 10. CBC messages passed at Log Level 0.

Code	Text and notes
3007	`No integer variables - nothing to do`

TABLE 11. CBC messages passed at or above Log Level 1.

Code	Text and notes
1	`Search completed - best objective %g, took %d iterations, and %d nodes`
3	`Exiting on maximum nodes`
4	`Integer solution of %g found after %d iterations and %d nodes`
5	`Partial search - best objective %g (best possible %g), took %d iterations, and %d nodes`
6	`The LP relaxation is infeasible or too expensive`
9	`Objective coefficients multiple of %g`
10	`After %d nodes, %d on tree, %g best solution, best possible %g`
11	`Exiting as integer gap of %g less than %g or %g%%`
12	`Integer solution of %g found by heuristic after %d iterations and %d nodes`
13	`At root node, %d cuts changed objective from %g to %g in %d passes`
14	`Cut generator %d (%s) - %d row cuts (%d active), %d column cuts %? in %g seconds - new frequency is %d`
16	`Integer solution of %g found by strong branching after %d iterations and %d nodes`
17	`%d solved, %d variables fixed, %d tightened`
18	`After tightenVubs, %d variables fixed, %d tightened`
19	`Exiting on maximum solutions`
20	`Exiting on maximum time`
23	`Cutoff set to %g - equivalent to best solution of %g`
24	`Integer solution of %g found by subtree after %d iterations and %d nodes`
26	`Setting priorities for objects %d to %d inclusive (out of %d)`
3008	`Strong branching is fixing too many variables, too expensively!`

TABLE 12. CBC messages passed at or above Log Level 2.

Code	Text and notes
15	`Node %d Obj %g Unsat %d depth %d`
21	`On closer inspection node is infeasible`
22	`On closer inspection objective value of %g above cutoff of %g`
23	`Allowing solution, even though largest row infeasibility is %g`

TABLE 13. CBC messages passed at or above Log Level 3.

Code	Text and notes
7	`Strong branching on %d (%d), down %g (%d) up %g (%d) value %g`
25	`%d cleanup iterations before strong branching`

Appendix

Frequently Asked Questions

Q: What is CBC?
A: The COIN-OR branch-and-cut code is designed to be a high-quality mixed-integer code provided under the terms of the Common Public License. CBC is written in C++, and is primarily intended to be used as a callable library (though a rudimentary standalone executable exists).

Q: What are some of the features of CBC?
A: CBC allows the use of any CGL cuts and the use of heuristics and specialized branching methods.

Q: How do I obtain and install CBC?
A: Please see the COIN-OR FAQ at www.coin-or.org for details on how to obtain and install COIN-OR modules.

Q: Is CBC reliable?
A: CBC has been tested on many problems, but more testing and improvement is needed before it can get to Version 1.0.

Q: Is there any documentation for CBC?
A: A list of CBC class descriptions generated by Doxygen is available. The latest user guide is available at www.coin-or.org.

Q: Is CBC as fast as CPLEX or Xpress?
A: No. However, its design is much more flexible, so advanced users will be able to tailor CBC to their needs.

Q: When will Version 1.0 of CBC be available?
A: It is expected that Version 1.0 will be released in time for the 2005 INFORMS annual meeting.

Q: What can the community do to help?
A: People from all around the world are already helping. There are probably 10 people who do not always post to the discussion mail list but are constantly "improving" the code by demanding performance or bug fixes or enhancements, and there are others posting questions to discussion groups.

A good start is to join the coin-discuss mailing list where CBC is discussed. See www.coin-or.org/mail.html. Some other possibilities include:

- Comment on the design.
- Give feedback on the documentation and FAQs.
- Break the code, or better yet—mend it.
- Tackle any of the "to-dos" listed in the Doxyen documentation and contribute back to COIN-OR.

Doxygen

There is Doxygen content for CBC available online at http://www.coin-or.org/Doxygen/Cbc/index.html. A local version of the Doxygen content can be generated from the CBC distribution. To do so, in the directory COIN/Cbc, enter `make doc`. The Doxygen content will be created in the directory `COIN/Cbc/Doc/html`. The same can be done for the COIN core, from the `COIN/Coin` directory.