jCbc: An Open Source MILP Solver for use in WRIMS

Babak Moazzez\textsuperscript{1}, Mahdi Ghamkhari\textsuperscript{2}, Kevin Kao\textsuperscript{3}, Junaid As-Salek\textsuperscript{4}, Zhaojun Bai\textsuperscript{5}, Matthias Koepe\textsuperscript{6}, Tariq Kadir\textsuperscript{7}, Prabhjot Sandhu\textsuperscript{8} and Sanjaya Seneviratne\textsuperscript{9}

\textsuperscript{1}Kennesaw State University, bmoazzez@kennesaw.edu
\textsuperscript{2}University of California, Davis, ghamkhari@ucdavis.edu
\textsuperscript{3}California Department of Water Resources, kuo-cheng.kao@water.ca.gov
\textsuperscript{4}United States Bureau of Reclamation, jassalek@usbr.gov
\textsuperscript{5}University of California, Davis, bai@cs.ucdavis.edu
\textsuperscript{6}University of California, Davis, mkoeppe@math.ucdavis.edu
\textsuperscript{7}California Department of Water Resources, Tariq.Kadir@water.ca.gov
\textsuperscript{8}California Department of Water Resources, Prabhjot.Sandhu@water.ca.gov
\textsuperscript{9}California Department of Water Resources, Sanjaya.Seneviratne@water.ca.gov

Abstract

We describe an open-source linear programming (LP) and mixed integer linear programming (MILP) solver, called jCbc, developed for use by CalSim/CalLite models through the WRIMS interface. jCbc is a Java Native Interface (JNI) for the COIN-OR MILP solver Cbc and LP solver Clp with modifications and new capabilities. The development of jCbc is based on a comprehensive review and performance test of commercial and open-source LP and MILP solvers for CalSim/CalLite models. jCbc exploits the domain-specific features for reducing the solution time of MILPs. Two examples are presented to demonstrate the functionality and accuracy of jCbc.

Keywords: Cbc, MILP, solver, Branch-and-Bound, WRIMS, CalSim, Callite

1 Introduction

jCbc is a Java Native Interface for COIN OR Mixed Integer Linear Programming Solver Cbc \cite{3} and Linear Programming Solver Clp \cite{2}, with some modifications and new capabilities added.

A Mixed Integer Programming Problem (MILP) has the following general form

\[ \min \ c^T x \]
\[ \text{s.t.} \]
\[ Ax \leq b \]
\[ Bx = d \]
\[ l \leq x \leq u \]
\[ x_i \in \mathbb{Z} \quad \text{for } i \in I, \]

where \( x \) is the column vector of variables, and \( c, b, d, l \) and \( u \) are column vectors. Also, \( A \) and \( B \) are the matrices of coefficients and \( I \) is a subset of variable indices. The equality and inequality constraints in (1) are element-wise and \( l \) and \( u \) can be \( \pm \infty \). jCbc solves problems that have the general form of (1). An MILP model is either constructed using the functions and objects that are available in jCbc (see Section 4.1), or it is fed to the solver using an LP or MPS format file (see Section 4.2). In either case, the solver applies a Branch-and-Bound algorithm to find an optimal solution or declares that the problem
has no solution or it is unbounded. See [6, Chapter 12] for an introduction on the MILP and algorithms to solve the MILP.

The MILPs in the CalSim and CalLite models are in the form of (1) and therefore can be solved by jCbc. WRIMS is a water resources modeling system for evaluating operational alternatives of large and complex river basins [1], and is the GUI through which CalSim/CalLite models are solved. The development of jCbc is supported in part by a grant from the Bureau of Reclamation, US Department of the Interior. The latest stable version of jCbc can be downloaded from https://github.com/JNICbc/jCbc. jCbc comes with a user manual that can be downloaded from the above link and includes detailed discussions about the existing tools in jCbc for solving MILPs.

The rest of this paper is organized as follows. Section 2 discusses the components that are used in building jCbc, the supported platforms and licensing in using jCbc. The available objects and models in jCbc are discussed in Section 3. Two example Java codes are discussed in Section 4 illustrating how jCbc can be used for solving MILPs. Conclusions and discussion of future works are presented in Section 5.

2 jCbc

jCbc uses an open-source software Simplified Wrapper and Interface Generator (SWIG) [4]. SWIG is a software development tool that connects libraries written in C and C++ with a variety of high-level programming languages. Most of Cbc and Clp objects and functions are carried over to the Java environment using jCbc along with several new functions that are useful for different purposes such as warm starting, reducing solve time, customizing pre-solve and pre-process steps, coefficient scanning, parallel attempts to solve a single model with different tunings, and customizing cutting planes and branching methods. Currently, there are two main solvers implemented in jCbc:

1. Function solve() calls internal jCbc solver with the following tunings: Cutting planes are applied only at root node, all heuristics are turned off, and there is no preprocessing and no presolve steps used. In addition, the Primal Simplex algorithm is used for the LP part of the Branch-and-Bound algorithm.

2. Function solve_whs() first uses ClpPresolve to simplify the model. Then, the information from a previous model is passed to solve_whs() in terms of names and values of integer variables. Using that information, a feasible solution is calculated for the new simplified model which in turn is used for hot starting the Cbc Branch-and-Bound algorithm. As a result, the solving time of solve_whs() function is significantly lower than the solving time of solve() function.

The jCbc github repository contains the following files and sub-folders: manual.pdf, jCbc.dll, jCbc.i, jCbc.cpp, src, examples, make.bat. jCbc.dll is the shared library that is loaded into Java. jCbc.i is the SWIG input file and jCbc.cpp is the C++ source file. src is the source folder and examples contains example Java codes. make.bat is the makefile to compile jCbc. Each one of these files and folders along with detailed process of building and using jCbc are discussed in manual.pdf.

jCbc takes advantages of third party packages ASL [7], BLAS [8], LAPACK [11], METIS [10] and MUMPS [9] to reduce the time in solving an MILP. jCbc also uses the pthread library to support multi-threading in solving an MILP. jCbc is currently supported on Windows 64-bit and Linux 64-bit platforms jCbc is free software, i.e., one can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, version 3.

3 Objects and Functions in jCbc

In this section, we introduce some of jCbc objects and functions that are needed to build a model. See jCbc manual [5] for a complete list of objects and functions in jCbc along with their descriptions.
3.1 Objects

jCbc has different types of objects for defining models, solver interfaces and arrays of doubles and integers:

1. jCbcModel: this is the core object to build a model.
2. jOsiClpSolverInterface: this is the LP solver used inside a jCbcModel object. It needs to be created before jCbcModel since most of the operations such as reading and writing a file or building a model require this object.
3. jCoinModel: it is faster to define variables, bounds, and constraints in a jCoinModel object and then add them at once from this object to jCbcModel.
4. SWIGTYPE_p_double: this is a SWIG pointer for doubles.
5. jarray_double: this is a JNI wrapped object that must be used instead of an array of doubles in Java to pass information to objects. Use command
   \[
   \text{SWIGTYPE_p_double A = jCbc.new_jarray_double(n)}
   \]
   to create a jarray_double of size \( n \). To add an element to jarray_double, use the command
   \[
   \text{jCbc.jarray_double_setitem(jarray_double A, i, k)}
   \]
   to set \( A[i] = k \). Use the command
   \[
   \text{jCbc.jarray_double_getitem(jarray_double A, i)}
   \]
   to access the element \( A[i] \).
6. SWIGTYPE_p_int: this is a SWIG pointer for integers.
7. jarray_int: this is a JNI wrapped object that must be used instead of an array of integers in Java to pass information to objects.

3.2 Functions

The functions in jCbc let a user to set a name for a defined model object, add rows to the model, and assign a solver interface to the model:

1. void addRow(jCoinModel A, int numberInRow, int[] index, double[] values, double rowlb, double rowup, String name): adds a row of the form \( l \leq ax \leq u \) to a jCoinModel object. The upper and lower bounds can be Double.MAX_VALUE (\( \infty \)) and -Double.MAX_VALUE (\( -\infty \)) respectively. numberInRow is the number of variables with non-zero coefficients in the row, and the coefficient of index[k] th variable in the row is values[k]. The elements of index are not required to be sorted.
2. void addRows(jOsiClpSolverInterface A, jCoinModel B): add all rows at once from a jCoinModel B object to a jCbcModel A object. Adding rows one by one to a jCoinModel object and then copying all the resulted rows from the jCoinModel object to a jCbcModel object is faster than adding rows one by one to a jCbcModel. The source in copying is jCoinModel B and the destination is jCbcModel A.
3. void assignSolver(jCbcModel A, jOsiClpSolverInterface B): assigns jOsiClpSolverInterface B as the LP solver interface for jCbcModel A.
4. void setModelName(jOsiClpSolverInterface A, String name): sets the model name.

4 Examples

In this section we provide two examples to show how jCbc can be used for solving an MILP in the general form of (1). The interested readers may refer to the jCbc manual for more illustrative examples. The Java codes in these examples are executed on a dual core personal computer with CPU frequency @2.60GHz and 8Gb RAM.
4.1 Example 1

Consider the following MILP:

\[
\begin{align*}
\text{min} & \quad x_1 + 2x_2 + 4x_3 \\
\text{s.t.} & \quad x_1 + 3x_3 \geq 1 \\
& \quad -x_1 + 2x_2 - 0.5x_3 \leq 3 \\
& \quad x_1 - x_2 + 2x_3 = 1/2 \\
& \quad 0 \leq x_1 \leq 1 \\
& \quad -\infty < x_2 \leq 10 \\
& \quad 2 \leq x_3 \leq +\infty \\
& \quad x_1, x_3 \in \mathbb{Z}.
\end{align*}
\]

The following example code shows how to build a model in jCbc for the MILP in (1), solve the model and finally retrieve the solution.

```java
01: import src.jCbc;
02: import src.SWIGTYPE_p_CbcModel;
03: import src.SWIGTYPE_p_double;
04: import src.SWIGTYPE_p_int;
05: import src.SWIGTYPE_p_CoinModel;
06: import src.SWIGTYPE_p_OsiClpSolverInterface;
07:
08: public class example1 {
09:    public static void main(String argv[]) {
10:        System.loadLibrary("jCbc");
11:
12:        SWIGTYPE_p_CoinModel modelObject = jCbc.new_jCoinModel();
13:
14:        SWIGTYPE_p_OsiClpSolverInterface solver =
15:            jCbc.new_jOsiClpSolverInterface();
16:
17:        double objValue[] = {1.0, 2.0, 4.0};
18:        double upper[] = {1.0, 10.0, Double.MAX_VALUE};
19:        double lower[] = {0.0, -Double.MAX_VALUE, 2.0};
20:
21:        jCbc.addCol(modelObject, lower[0], upper[0], objValue[0], "x1", true);
22:        jCbc.addCol(modelObject, lower[1], upper[1], objValue[1], "x2", false);
23:        jCbc.addCol(modelObject, lower[2], upper[2], objValue[2], "x3", true);
24:
25:        SWIGTYPE_p_int row1Index = jCbc.new_jarray_int(2);
26:        jCbc.jarray_int_setitem(row1Index, 0, 0);
27:        jCbc.jarray_int_setitem(row1Index, 1, 2);
28:
29:        SWIGTYPE_p_double row1Value = jCbc.new_jarray_double(2);
30:        jCbc.jarray_double_setitem(row1Value, 0, 1.0);
31:        jCbc.jarray_double_setitem(row1Value, 1, 3.0);
32:
33:        jCbc.addRow(modelObject, 2, row1Index, row1Value, 1.0,
34:            Double.MAX_VALUE,"r1");
35:
36:        SWIGTYPE_p_int row2Index = jCbc.new_jarray_int(3);
37:        jCbc.jarray_int_setitem(row2Index, 0, 0);
38:        jCbc.jarray_int_setitem(row2Index, 1, 1);
39:        jCbc.jarray_int_setitem(row2Index, 2, 2);
40:
41:        SWIGTYPE_p_double row2Value = jCbc.new_jarray_double(3);
```
The Java code has the following structure:

- Initializing and defining objects (Lines 1-16), in particular,
  - Line 12: defining a CoinModel for adding rows and columns faster
  - Line 14: defining the LP solver interface
- Building the model (Lines 17-31), in particular,
  - Line 17: coefficients of the variables in objective function are set
  - Line 18: upper bounds for variables are set
  - Line 19: lower bounds for variables are set
  - Line 21-23: optimization variables are defined
  - Line 25-27: defining and initializing a SWIG type array of integers
  - Line 29-31: defining and initializing a SWIG type array of doubles
- Solving the model, and getting and printing out the solution (Lines 33-78), in particular,
  - Line 33: adding a row to CoinModel
  - Line 61: adding all rows to OsiClpSolverInterface
Line 63: defining a new empty CbcModel
Line 65: assigning the solver to CbcModel
Line 69: solving the model
Line 71: getting the solution
Line 73-78: printing the solution

Assuming that the above Java code is saved in a file example1.java, the code can be executed by typing the following commands in a Microsoft Windows cmd or Linux terminal:

• javac example1.java
• java example1

The output of the Java code in this example is as follows:

```
Solution: Objective_Value = 9.0
x1 = 0.0
x2 = 0.5
x3 = 2.0
```

We note that both jCbc and the MILP solver Gurobi [12] give the same optimal values for the objective function and variables of the MILP in (1), which shows that jCbc's solution is accurate.

### 4.2 Example 2

This example shows a Java code that reads a CalSim/CalLite model from the model.lp file, corresponding to an MILP in the general form of (2), solves the model and reports the optimal objective value as well as optimal values for integer variables. The model.lp file is generated by WRIMS [1]. This MILP has 6914 variables (19 of which are integer variables) and 5770 constraints.

```java
import src.jCbc;
import src.SWIGTYPE_p_CbcModel;
import src.SWIGTYPE_p_double;
import src.SWIGTYPE_p_int;
import src.SWIGTYPE_p_CoinModel;
import src.SWIGTYPE_p_OsiClpSolverInterface;

public class example2 {
    public static void main(String argv[]) {
        System.loadLibrary("jCbc");

        SWIGTYPE_p_OsiClpSolverInterface solver = jCbc.new_jOsiClpSolverInterface();
        SWIGTYPE_p_CbcModel Model = jCbc.new_jCbcModel();

        jCbc.assignSolver(Model, solver);
        jCbc.readLp(solver, "model.lp");
        jCbc.solve(Model, solver);

        System.out.println("Solution:");
        System.out.println("Objective_Value = " + jCbc.getObjValue(Model));

        for (int j = 0; j < nCols; j++){
            if (jCbc.isInteger(Model,j) == 1)
                System.out.println("x["+j+"]=" + jCbc.jarray_double_getitem(sol,j));
            }
        }
    }
}
```
The Java code in this example has the following structure:

- Initializing and defining the model object (Lines 1-12), in particular,
  - Line 12: defining LP solver interface
  - Line 14: defining CbcModel object
- Solving the model (Lines 16-18), in particular,
  - Line 16: assigning solver to Model
  - Line 17: reading the model from model.lp file
  - Line 18: solving the model
- Getting the solution and printing it out (Lines 20-21), in particular,
  - Line 20-21: getting and printing the optimal objective value
  - Line 23-25: getting and printing the optimal values for the integer variables.

Assuming that the above Java code is in saved in a file example2.java, the code can be executed by typing the following commands in a Microsoft Windows cmd or Linux terminal:

- `javac example2.java`
- `java example2`

The output of executing the Java code in this example is as follows:

```
Solution:
Objective_Value=-1.74658261388485E10
x[2760]=1.0
x[2782]=0.0
x[2785]=0.0
x[2787]=0.0
x[2789]=0.0
x[2790]=0.0
x[2793]=0.0
x[2794]=0.0
x[2796]=0.0
x[2797]=0.0
x[2799]=0.0
x[2801]=0.0
x[2888]=1.0
x[4465]=1.0
x[4518]=1.0
x[5443]=0.0
x[6746]=0.0
x[6747]=0.0
x[6907]=1.0
```

We note that Gurobi [12] gives the same objective value for the model in this example.

5 Concluding Remarks

In this paper, we introduced jCbc, an open source solver for MILP and discussed the objects and tools that come with this solver. In jCbc, the state-of-art MILP algorithm is tuned for CalSim/CalLite models
through the WRIMS interface [13]. There is an on-going effort on full integration of jCBC solvers with
WRIMS, and performance tuning and optimization on various computing platforms. In addition, we are
exploring a number of strategies to improve jCbc’s solution quality and solving time. Finally, we plan
to develop a sensitivity analyzer module in jCbc for so that it will reveal the sensitivity of the optimal
values to the perturbations of key parameters in CalSim and CalLite models.

References

http://baydeltaoffice.water.ca.gov/modeling/hydrology/CalSim
Wiley & Sons Inc., Hoboken, New Jersey, 2010
California Water and Environmental Modeling Forum Annual Meeting, Folsom, CA, March 20-22,
2017