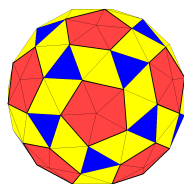# jCbc: JNI for Coin OR Mixed Integer Programming Solver CBC

# A User's Manual

Babak Moazzez

University of California Davis

bmoazzez@ucdavsi.edu

July 1, 2016

**jCbc**

# Contents

# 1 Introduction

jCbc is a Java Native Interface for COIN OR Mixed Integer Linear Programming Solver CBC [4] and also coin OR Linear Programming Solver CLP [3], with some modifications and new capabilities added. jCbc has been created using open source Simplified Wrapper and Interface Generator SWIG [7], which is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages.

Most CBC and CLP objects and functions carry over to a Java environment using jCbc along with several new functions that are useful for different purposes such as warm start, reduce solve time, selective pre-solve and/or pre-process, coefficient/RHS scanning, parallel attempts to solve a single model with different tunings, selective cutting planes and/or branching methods, convenient MPS/LP read and write methods, etc.

jCbc solves Mixed Integer Programming Problems of the form

$$
\begin{aligned}
\min \quad & c^T x \\
\text{s.t.} \quad & Ax \lesseqgtr b \qquad (MILP)\\
& l \le x \le u \\
& x_i \in \mathbb{Z} \ \forall i \in I.
\end{aligned}
$$

where $I$ is a subset of variable indices and $l$ and $u$ can be $\pm\infty$. The model is either constructed using the functions and object that will be described in Section 6, or it is fed to the solver using an LP or MPS file. In either case, the solver applies branch and cut to find an optimal solution or declares that the problem has no solution or it is unbounded. Appendix D gives a brief introduction on MILP and algorithms to solve it. It also describes the steps that these algorithms take to solve the problem.

# 2 Getting jCbc

The latest stable snapshot of jCbc-1.0 can be downloaded from:

<div align="center">

`http://web.cs.ucdavis.edu/~bmoazzez/jCbc.htm`

</div>

and also from

<div align="center">

`https://github.com/babakmoazzez/jCbc`.

</div>

After the tar-ball file is unzipped, a directory `/jCbc` is created which contains the following files and sub-folders:

- `manual.pdf`

- `jCbc.dll` (or `jCbc.so` for Linux version) This is the library that should be loaded into the Java code. See Section 4 for more information on general conventions.

- `jCbc.i` SWIG input file

- `jCbc.cpp` Source file

- `/src` Source folder

- `/example` Contains example Java codes.

- `make.bat` (or `make` for Linux version) make file to compile jCbc (see Section 3).

# 3 Building jCbc

The compilation process is different based on the platform and operating system. We will first describe the compilation steps on Windows 32 and 64 bit platforms and then Linux steps will be described.

## 3.1 Compilation on Windows 32 bit using gcc (g++)

### 3.1.1 Install MinGW 32

1. Download and install MinGW 32 bit installer from `http://www.mingw.org` or `http://sourceforge.net/projects/mingw/files/Installer`.

2. Install MinGW 32 bit by running the installer. Choose all elements of MinGW 32 bit including MSYS and wget libraries.

3. Make sure that `C:\MinGW\bin` is in the PATH and no other gcc/g++ compiler is in the PATH including MinGW64 or MSYS64.

### 3.1.2 Install SWIG

1. Download Swig for Windows (also known as swigwin) from `http://www.swig.org/download.html` or `http://prdownloads.sourceforge.net/swig/swigwin-3.0.7.zip`.

2. No installation is needed for Swig, extract the zip file and put the folder in `C:\Program Files (x86)`.

3. Add `C:\Program Files (x86)\swigwin-3.0.7` to PATH.

### 3.1.3 Install Coin OR CBC

1. Download latest version of Cbc from `http://www.coin-or.org/download/source/Cbc`.

2. Extract the CBC file (using 7zip for example) and put it anywhere (we will call it `<path-to-cbc>`).

3. Run `msys.bat` from `C:\MinGW\msys\1.0\msys.bat`.

4. In the MSYS shell, type `cd /c/.../<path-to-cbc>/`.

5. First you should get third party packages BLAS, LAPACK, METIS and MUMPS. In order to do this, in an MSYS shell, enter `cd /c/.../<path-to-cbc>/ThirdParty/BLAS` and then run `./get.BLAS`. This will download and unpack BLAS. Do the same for other third party packages. See Appendix B for more information on third party Packages.

6. In the MSYS shell, type `cd /c/.../<path-to-cbc>/`

7. Type `./configure --enable-cbc-parallel` and press enter.

8. After configure is finished type `make` and press enter.

9. After make is done, type `make install` and press enter.

This will compile Cbc and create cbc.exe in the `<path-to-cbc>/bin` folder.

### 3.1.4 Compile jCbc

1. Make sure that you have Java Development Kit (JDK) 32 bit installed and in the PATH. This is a 32 bit compilation and requires 32 bit Java.

2. Download and unzip jCbc from `http://web.cs.ucdavis.edu/~bmoazzez/jCbc.htm`. Put it anywhere and we will call this folder `<path-to-jCbc>`.

3. Enter `<path-to-jCbc>` folder by typing `cd /c/<path-to-jCbc>` in a cmd shell and not a MSYS shell.

4. Run the make file which executes the following commands in order:

   ```
   swig -c++ -java -package src -outdir src jCbc.i.
   ```

   ```
   g++ -c -fopenmp jCbc.cpp jCbc_wrap.cxx -static -Wl,--kill-at  -I"C:\Java\jdk1.8.0_65\
   include" -I"C:\Java\jdk1.8.0_65\include\win32" -I"C:\<path-to-cbc>\Cbc-2.9.8\include\
   coin".
   ```

   ```
   g++ -shared -fopenmp jCbc.o jCbc_wrap.o -o jCbc.dll -static -Wl,--kill-at  -I"C:\Java\
   jdk1.8.0_65\include" -I"C:\Java\jdk1.8.0_65\include\win32" -I"C:\<path-to-cbc>\Cbc-2.
   9.8\include\coin"  -LC:\<path-to-cbc>\Cbc-2.9.8\lib -lCbcSolver -lCbc  -lClp -lOsiClp
   -lOsi -lOsiCbc  -lCgl -lCoinUtils -lcoinmumps -lcoinblas -lcoinlapack -lcoinasl
   -lcoinmetis -lgfortran -lmingw32 -lmoldname -lmingwex -lmsvcrt -lquadmath -lm
   -ladvapi32 -lshell32 -luser32 -lkernel32 .
   ```

This will create the libraries and the `jCbc.dll` file.

Note that you need to make a change in the make file to the point to your Java Development kit installation (jdk) where it will find folders `include` and `include\win32` and also `<path-to-cbc>`.

Note: the created package will be usable only with a 32 bit Java. When you are calling jCbc, make sure that `C:\MinGW\bin` (32 bit version) is in PATH.

## 3.2 Compilation on Windows 64 bit using gcc (g++)

### 3.2.1 Install MinGW 64

1. Download MinGW 64 bit installer from `http://sourceforge.net/projects/mingw-w64/?source=typ_redirect`

2. Install MinGW 64 bit by running the installer. Choose Architecture x86_64.

   Note: Install in `c:\` not in `c:\Program Files` (The space in the address will cause errors).

3. Make sure that `C:\mingw-w64\x86_64-5.2.0-posix-seh-rt_v4-rev1\mingw64\bin` is in the PATH and no other gcc/g++ compiler is in the PATH including MinGW32 or MSYS32.

### 3.2.2 Install SWIG

1. Download Swig for Windows (also known as swigwin) from `http://www.swig.org/download.html` or `http://prdownloads.sourceforge.net/swig/swigwin-3.0.7.zip`

2. No installation is needed for Swig, extract the zip file and put the folder in `C:\Program Files`

3. Add `C:\Program Files\swigwin-3.0.7` to PATH

### 3.2.3 Install MSYS 64

1. Download MSYS 64 bit from `https://sourceforge.net/projects/mingw-w64/files/External%20binary%20packages%20%28Win64%20hosted%29/MSYS%20%2832-bit%29/MSYS-20111123.zip/download`

2. Unzip (extract). No installation is needed. Just put the extracted folder (`msys`) in `c:\` (Final installation path should be `C:\mingw -w64\x_86_64-x.y.z-posix-seh-rt_v4-rev0` if you download `x.y.z` version.)

3. IMPORTANT: from `C:\MinGW\msys\1.0\bin` of 32 bit MinGW installation, copy `find.exe` and paste it in `C:\msys\bin`

4. Click on `msys.bat` in `C:\msys` and MSYS window will open. Type `sh /postinstall/pi.sh` and press enter. Then follow the instructions on the window. You have to answer yes to first and second questions and enter the address `c:/mingw-w64/x86_64-x.y.z-posix-seh-rt_v4-rev1/mingw64` for the third question.

This will install MSYS 64 bit.

### 3.2.4 Install Coin OR Cbc

1. Download latest version of Cbc from `http://www.coin-or.org/download/source/Cbc`

2. Extract the CBC file (using 7zip for example) and put it anywhere (we will call it `<path-to-cbc>`)

3. Run msys.bat from `C:\msys\msys.bat`

4. In the MSYS shell, type `cd /c/.../<path-to-cbc>/`.

5. First you should get third party packages BLAS, LAPACK, METIS and MUMPS. In order to do this, in an MSYS shell, enter `cd /c/.../<path-to-cbc>/ThirdParty/BLAS` and then run `./get.BLAS`. This will download and unpack BLAS. Do the same for other third party packages. See Appendix B for more information on third party Packages.

6. In the MSYS shell, type `cd /c/.../<path-to-cbc>/`

7. Type `./configure --enable-cbc-parallel` and press enter.

8. After configure is finished type `make` and press enter.

9. After make is done, type `make install` and press enter.

This will compile Cbc and create cbc.exe in the `<path-to-cbc>/bin` folder.

### 3.2.5 Compile jCbc

1. Make sure that you have Java Development Kit (JDK) 64 bit installed and in the PATH. This is a 64 bit compilation and requires 64 bit Java.

2. Download and unzip jCbc from `http://web.cs.ucdavis.edu/~bmoazzez/jCbc.htm`. Put it anywhere and we will call this folder `<path-to-jCbc>`

3. Enter `<path-to-jCbc>` folder by typing `cd /c/<path-to-jCbc>` in a cmd shell and not a MSYS shell.

4. Run the make file which executes the following commands in order:

   `swig -c++ -java -package src -outdir src jCbc.i`

   `g++ -c -fopenmp jCbc.cpp jCbc_wrap.cxx -static -Wl,--kill-at -I"C:\Java\jdk1.8.0_65_x64\include" -I"C:\Java\jdk1.8.0_65_x64\include\win32" -I"C:\<path-to-cbc>\Cbc-2.9.8\include\coin"`

   `g++ -shared -fopenmp jCbc.o jCbc_wrap.o -o jCbc.dll -static -Wl,--kill-at -I "C:\Java\jdk1.8.0_65_x64\include" -I "C:\Java\jdk1.8.0_65_x64\include\win32" -I "C:\<path-to-cbc>\Cbc-2.9.8\include\coin" -L C:\<path-to-cbc>\Cbc-2.9.8\lib -lCbcSolver -lCbc -lClp -lOsiClp -lOsi -lOsiCbc -lCgl -lCoinUtils -lgfortran -lz -lquadmath`

This will create the libraries and the `jCbc.dll` file.

Note that you need to make a change in the make file to the point to your Java Development kit installation (jdk) where it will find folders `include` and `include\win32` and also `<path-to-cbc>`.

Note: the created package will be usable only with a 64 bit Java. When you call jCbc, make sure that `C:\mingw-w64\x86_64-5.2.0-posix-seh-rt_v4-rev1\mingw64\bin` (64 bit version) is in PATH.

**Note:** Compilation instructions using Intel compiler will be added soon.

## 3.3 Compilation on Linux

Compiling on Linux is very straight forward: To install prerequisites, do (on Ubuntu 14.04)

```
sudo apt-get install swig
```

```
sudo apt-get install gfortran libbz2-dev zlib1g-dev g++
```

Download Cbc-2.9.8 from

```
http://www.coin-or.org/download/source/Cbc
```

and unzip. In order to compile with third party Packages, enter each package folder and run `./get.package_name`. The following will install Cbc:

```
./configure --enable-cbc-parallel CDEFS="-DCBC_THREAD_SAFE -DCBC_NO_INTERRUPT -DHAVE_STRUCT_
TIMESPEC" CXXDEFS="-DCBC_THREAD_SAFE -DCBC_NO_INTERRUPT -DHAVE_STRUCT_TIMESPEC"
make
make install
```

Now run the make file which runs the following commands in order:

```
swig -c++ -java -package src -outdir src jCbc.i

g++ -fopenmp -fPIC -c jCbc.cpp jCbc_wrap.cxx -static -Wl,--kill-at -I /home/jdk1.8.0_91/include/
-I /home/jdk1.8.0_91/include/linux/ -I /home/Cbc-2.9.8/include/coin -L /home/Cbc-2.9.8/lib
-lCbcSolver -lCbc -lClp -lOsiClp -lOsi -lOsiCbc -lCgl -lCoinUtils -lcoinmumps -lcoinblas
  -lcoinlapack -lcoinmetis -lgfortran -lpthread

g++ -fopenmp -fPIC -shared *.o -o libjCbc.so -I /home/jdk1.8.0_91/include/ -I /home/jdk1.
8.0_91/include/linux/ -I /home/Cbc-2.9.8/include/coin -L /home/Cbc-2.9.8/lib -lCbcSolver
  -lCbc -lClp -lOsiClp -lOsi -lOsiCbc -lCgl -lCoinUtils -lcoinmumps -lcoinblas -lcoinlapack
  -lcoinmetis -lgfortran -lpthread
```

Remember to do `export LD_LIBRARY_PATH=/home/Cbc-2.9.8/lib` before compiling.

# 4 jCbc and parallel functions

In order to take advantage of Cbc's parallel options and also in order to use jCbc's `par_solve` function (which is discussed in Section 7), one needs to compile Cbc and jCbc using some extra commands. On a

Windows 32 bit platform, POSIX threads are not supported by MinGW 32, so the user must use another third party package called `Pthread for Windows 32` which is available through

<div align="center">

https://www.sourceware.org/pthreads-win32.

</div>

After downloading and extracting,

1. copy the files `pthread.h, sched.h` and `semaphore.h` from `Pre-built.2\include` and paste into `\MinGW\include`

2. copy `libpthreadGC2.a` from `Pre-built.2\lib\x86` and paste into `\MinGW\lib` and rename it to `libpthread.a`

3. copy `pthreadGC2.dll` from `Pre-built.2\dll\x86` to and paste into `\MinGW\include` and rename it to `libpthread.dll`

This allows Windows 32 bit to use `pthreads` with MinGW 32.

In order to compile Coin OR Cbc (32 or 64) with capability of running in threads, Coin OR Cbc should be compiled in the following way (in msys shell):

```
./configure --enable-cbc-parallel CDEFS="-DCBC_THREAD_SAFE -DCBC_NO_INTERRUPT -DHAVE_STRUCT_TIMESPEC" CXXDEFS="-DCBC_THREAD_SAFE -DCBC_NO_INTERRUPT -DHAVE_STRUCT_TIMESPEC"
make
install
```

This will allow the use of `jCbc.setThreadMode(int a)` and `jCbc.setNumberThread(int a)` which are discussed in Section 6.

In order to be able to use `jCbc.par_solve()` function (Section 7), the make file needs to be changed in the following way:

```
swig -c++ -java -package src -outdir src jCbc.i
```

```
g++ -c -fopenmp jCbc.cpp jCbc_wrap.cxx -static -Wl,--kill-at -DHAVE_STRUCT_TIMESPEC -I"C:\Java\jdk1.8.0_65_x64\include" -I"C:\Java\jdk1.8.0_65_x64\include\win32" -I"C:\<path-to-cbc>\Cbc-2.9.8\include\coin"
```

```
g++ -shared -fopenmp jCbc.o jCbc_wrap.o -o jCbc.dll -static -Wl,--kill-at  -I "C:\Java\jdk1.8.0_65_x64\include" -I "C:\Java\jdk1.8.0_65_x64\include\win32" -I "C:\<path-to-cbc>\Cbc-2.9.8\include\coin"  -L C:\<path-to-cbc>\Cbc-2.9.8\lib -lCbcSolver -lCbc  -lClp -lOsiClp -lOsi -lOsiCbc  -lCgl -lCoinUtils -lgfortran -lz -lquadmath -lpthread
```

# 5   Basic JNI Objects and Routines

In this section, we describe the objects and functions (routines) inside jCbc. Each object/function is described briefly and examples in the next section will make it more clear on how to use them.

## 5.1   Objects

There are several objects that are needed to build a model in jCbc. The following list describes all these objects.

- jCbcModel

  This is the core object to build a model. Use the command `SWIGTYPE_p_CbcModel Model = jCbc.new_jCbcModel()` to create a new `jCbcModel` object.

- jOsiClpSolverInterface

  This is the LP solver that is used inside a `jCbcModel` object. This needs to be created before `jCbcModel` since most of the operations such as read/write or build methods are done using this object. Use the command `SWIGTYPE_p_OsiClpSolverInterface solver = jCbc.new_jOsiClpSolverInterface()` to create a new `jOsiClpSolverInterface` object. It should be assigned later to a `jCbcModel` using the command `jCbc.assignSolver(jCbcModel A,jOsiClpSolverInterface B)`.

- 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`. Use `SWIGTYPE_p_CoinModel Model = jCbc.new_jCoinModel()` to create a new `jCoinModel` object.

- SWIGTYPE_p_double

  This is a swig pointer for doubles.

- 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 `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 `jCbc.jarray_double_setitem(jarray_double A,i,k)`. This will set $A[i] = k$. To get the element do `jCbc.jarray_double_getitem(jarray_double A,i)` to access $A[i]$.

- SWIGTYPE_p_int

  This is a swig pointer for integers.

- 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. Use command `SWIGTYPE_p_int A = jCbc.new_jarray_int(n)` to create a `jarray_int` of size n. To add an element to `jarray_int`, use `jCbc.jarray_int_setitem(jarray_int A,i,k)`. This will set $A[i] = k$. To get the element do `jCbc.jarray_int_getitem(jarray_int A,i)` to access $A[i]$.

- SWIGTYPE_p_std__string

  This is a swig pointer for strings.

- jarray_string

  This is a JNI wrapped object that must be used instead of an array of strings in Java to pass information to objects. Use command `SWIGTYPE_p_string A = jCbc.new_jarray_string(n)` to create a `jarray_string` of size n. To add an element to `jarray_string`, use `jCbc.jarray_string_setitem(jarray_string A,i,"string")`. This will set $A[i] = $ "*string*". To get the element do `jCbc.jarray_string_getitem(jarray_string A,i)` to access $A[i]$.

  To delete any of these objects do `jCbc.delete_Object(A)`, for example `jCbc.delete_jCbcModel(A)`.

## 5.2  Functions

In this section, different jCbc functions and routines are described. Please look at Section 8 of this manual to learn more about the usage of each command. All of these functions must be used within the jCbc wrapper i.e. `jCbc.function(a,b,..)`.

### 5.2.1 Building a Model

- `void addCol(jCoinModel A, double collb, double colub, double obj, String name, bool isInt)`

  Add a column with to a `jCoinModel` object. The upper and lower bounds can be `Double.MAX_VALUE` ($\infty$) and `-Double.MAX_VALUE` ($-\infty$) respectively.

- `void addRow(jCoinModel A, int numberInRow, long int [] index, double [] values, double rowlb, double`

  Add 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. The data structure for matrices in Cbc (and jCbc) is called Compressed row/Column Storage and is described in Appendix A. Basically `numberInRow` is the number of non-zeros in a row, and the coefficient of $k$-th variable in a row is `values[index[k]]`.

- `void addRows(jCbcModel A, jCoinModel B)`

  Add all rows at once from a `jCoinModel` object to a `jCbcModel` object. This works faster than adding rows one by one to `jCbcModel`.

- `void setInteger(jCbcModel A, int i)`

  Set column $i$ in model `A` to be integer type.

- `void addRows(jOsiClpSolverInterface A, jCoinModel B)`

  Add all rows at once from a `jCoinModel` object to a `jOsiClpSolverInterface` object. This works faster than adding rows one by one to `jCbcModel`.

- `void assignSolver(jCbcModel A, jOsiClpSolverInterface B)`

  Assigns `jOsiClpSolverInterface` B as the LP solver for `A`. This is a necessary step. See Examples section.

- `void setInteger(jOsiClpSolverInterface A, int i)`

  Set column $i$ in model `A` to be integer type.

- `void setModelName(jOsiClpSolverInterface A, String name)`

  Sets the model name.

- `void setRowName(jOsiClpSolverInterface A, int i , String name)`

  Sets the name of the column (variable) with index $i$.

### 5.2.2 Basic Solve Methods

- `void branchAndBound(jCbcModel A)`

  Perform pure branch and bound on model `A`.

- `void initialSolve(jOsiClpSolverInterface A)`

  solve LP relaxation of the model.

- `void solve(jCbcModel A, jOsiClpSolverInterface B, int logLevel= 0)`

  Solve model using default parameters (very similar to CBC command line).

### 5.2.3 Problem Status

- `int status(jCbcModel A)`

  Final status of problem. -1 before branchAndBound 0 finished - check `isProvenOptimal` or `isProvenInfeasible` to see if solution found (or check value of best solution) 1 stopped - 2 difficulties so run was abandoned

- `int secondaryStatus(jCbcModel A)`

  Secondary status of problem -1 unset 0 search completed with solution 1 linear relaxation not feasible 2 stopped on gap 3 stopped on node limit 4 stopped on time limit 5 stopped on user event 6 stopped on solution count limit 7 linear relaxation unbounded 8 stopped on iteration limit

- `int isProvenOptimal(jCbcModel A)`

  Is optimality proven?

- `int isProvenInfeasible(jCbcModel A)`

  Is infeasibility proven?

### 5.2.4 Names

- `String getRowName(jOsiClpSolverInterface A, int i)`

  Gets the name of $i$-th row

- `String getColName(jOsiClpSolverInterface A, int i)`

  Gets the name of $i$-th column

- `String getRowName(jCbcModel A, int i)`

  Gets the name of $i$-th row

- `String getColName(jCbcModel A, int i)`

  Gets the name of $i$-th column

### 5.2.5 Read and Write

See Appendix C for LP and MPS files.

- `void readMps(jOsiClpSolverInterface A, String name)`

  Read a model in the current directory from an MPS file into a `jOsiClpSolverInterface` object.

- `void readLp(jOsiClpSolverInterface A, String name)`

  Read a model in the current directory from an LP file into a `jOsiClpSolverInterface` object.

- `void readMps(jCbcModel A, String name)`

  Read a model in the current directory from an MPS file into a `jCbcModel` object.

- `void readLp(jCbcModel A, String name)`

  Read a model in the current directory from an LP file into a `jCbcModel` object.

- `void writeMps(jCbcModel A, String name)`

  Write current model saved inside a `jCbcModel` object, into an MPS file in the current directory.

- `void writeLp(jCbcModel A, String name)`

  Write current model saved inside a `jCbcModel` object, into an LP file in the current directory.

- `void writeMps(jOsiClpSolverInterface A, String name)`

  Write current model saved inside a `jOsiClpSolverInterface` object, into an MPS file in the current directory.

- `void writeLp(jOsiClpSolverInterface A, String name)`

  Write current model saved inside a `jOsiClpSolverInterface` object, into an LP file in the current directory.

- `void writeLp1(jCbcModel A, String name, double epsilon=1e-5, int decimals=5)`

  Same as `writeLp` but with options for epsilon and number of decimal points.

- `void writeMps1(jCbcModel A,String name,int formatType=0,int numberAcross = 2,double objSense= 0)`

  Write the problem in MPS format to the specified file with more control over the output. Row and column names may be null. formatType is 0 - normal 1 - extra accuracy 2 - IEEE hex

### 5.2.6 Getting Solution

- `SWIGTYPE_p_double getSol(jCbcModel A)`

  Get the solution from a `jCbcModel` object in the form of a pointer of doubles.

- `SWIGTYPE_p_double getColSolution(jOsiClpSolverInterface A)`

  Get the solution from a `jOsiClpSolverInterface` object in the form of a pointer of doubles.

- `SWIGTYPE_p_double getRowPrice(jOsiClpSolverInterface A)`

  Get dual prices from a `jOsiClpSolverInterface` object in the form of a pointer of doubles.

- `SWIGTYPE_p_double getRowActivity(jOsiClpSolverInterface A)`

  Get row activity levels from a `jOsiClpSolverInterface` object in the form of a pointer of doubles.

- `SWIGTYPE_p_double getReducedCost(jOsiClpSolverInterface A)`

  Get reduced costs from a `jOsiClpSolverInterface` object in the form of a pointer of doubles.

- `SWIGTYPE_p_double getColSolution(jCbcModel A)`

  Get the solution from a `jCbcModel` object in the form of a pointer of doubles. (Same as `getsol`)

- `SWIGTYPE_p_double getRowPrice(jCbcModel A)`

  Get dual prices from a `jCbcModel` object in the form of a pointer of doubles.

- `SWIGTYPE_p_double getRowActivity(jCbcModel A)`

  Get row activity levels from a `jCbcModel` object in the form of a pointer of doubles.

- `SWIGTYPE_p_double getReducedCost(jCbcModel A)`

  Get reduced costs from a `jCbcModel` object in the form of a pointer of doubles.

- `double getObjValue(jOsiClpSolverInterface A)`

  Get objective value of the current solution.

- `double getObjValue(jCbcModel A)`

  Get objective value of the current solution.

### 5.2.7   Problem Statistics

- `int isInteger(jOsiClpSolverInterface A, int i)`

  Is the $i$-th variable integer type?

- `int isInteger(jCbcModel A, int i)`

  Is the $i$-th variable integer type?

- `int getNumRows(jOsiClpSolverInterface A)`

  Get the number of constraints in a model.

- `int getNumCols(jOsiClpSolverInterface A)`

  Get the number of variables in a model.

- `int getNumRows(jCbcModel A)`

  Get the number of constraints in a model.

- `int getNumCols(jCbcModel A)`

  Get the number of variables in a model.

- `int  numberIntegers(jCbcModel A)`

  Get the number of integer variables in a model.

- `String getModelName(jOsiClpSolverInterface A)`

  Get the name of the model.

- `int isBinary(jOsiClpSolverInterface A, int i)`

  Is the $i$-th variable binary type?

- `int getNumIntegers(jOsiClpSolverInterface A)`

  Get the number of integer variables in a model.

### 5.2.8   Log

- `void setLogLevel(jOsiClpSolverInterface A, int i)`

  Set the level of output printing (in the command line) to integer $i$. It can change from 0 to 10.

- `void setLogLevel(jCbcModel A, int i)`

  Set the level of output printing (in the command line) to integer $i$. It can change from 0 to 10.

### 5.2.9   Tolerances

- `void setPrimalTolerance(jCbcModel A, double a)`

  Assuming that one has a minimization problem, an individual variable is deemed primal feasible if it is less than the tolerance below its lower bound and less than it above its upper bound. This method sets the primal tolerance to `a`.

- `void setDualTolerance(jCbcModel A, double a)`

  Assuming that one has a minimization problem, a variable is deemed to be dual feasible if its reduced cost is greater than minus the tolerance or its distance to the upper bound is less than primal tolerance and the reduced cost is less than plus the tolerance or the distance to lower bound is less than primal tolerance. This method sets the dual tolerance to `a`.

- `void setIntegerTolerance(jCbcModel A, double a)`

  The maximum amount the value of an integer variable can vary from integer and still be considered feasible. This method sets the integer tolerance to `a`.

### 5.2.10  Time

- `double getCoinCpuTime()`
  Return CPU time.

### 5.2.11  Threads

- `void setNumberThreads(jCbcModel model,int a)`

  Tells the solver how many threads to create (or how many are available). The ideal is of course equal to the number of cores on your machine.

- `void setThreadMode(jCbcModel model,int a)`

  `a` determines the following:

    - 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` $> 0$ - do heuristics `numberThreads` at a time
    - 8 set and `numberThreads` $= 0$ - do all heuristics at once

  default is 0

# 6  Advanced JNI Routines

Based on our needs, we have made the following interface routines to use jCbc:

- solve
- solve_1
- solve_2
- solve_3
- solve_whs
- par_solve
- solve_unified

Let us explain briefly what each subroutine does.

1. `solve(jCbcModel A, jOsiClpSolverInterface B, int logLevel=0)`

   This function calls internal jCbc solver with the following tunings: Cutting planes applied only at root node, All heuristics are turned off, and also there is no preprocessing and no presolve used. Primal simplex is used for LP.

2. `solve_1(jCbcModel A, jOsiClpSolverInterface B, int logLevel=0)`

   This function calls pure branch and bound method.

3. `solve_2(jCbcModel A, jOsiClpSolverInterface B, int logLevel=0)`

   This function calls jCbc solver, but first uses CglPreProcess to simplify the model. The following cutting planes are used: Gomory, Probing, GMI, TwoMIR and MIR. Heuristics are turned off. Also the cuts are applied only to the root node. This functions improves the solve time to a great extent in some cases.

4. `solve_3(jCbcModel A, jOsiClpSolverInterface B, int logLevel=0 ,double presolve_tolerance= 1e-07)`

   This function calls jCbc, with RINS and FeasibilityPump heuristics turned off, doing presolve using ClpPresolve before calling jCbc.

5. `solve_whs(jCbcModel A,jOsiClpSolverInterface B,int logLevel=0,String names[],int values[], int intvars,int logLevel= 0,double presolve_tolerance=1e-07)`

   First ClpPresolve is used to simplify the model. The information from a previous model is passed to solve_whs in terms of names and values of integer variables. A feasible solution is calculated for the new simplified model using that information, and the warm start is used along with the solve function.

6. `par_solve(jCbcModel model, jOsiClpSolverInterface solver, String A, names1=null, values1= null,intvars1=0,names2=null, values2=null,intvars2=0,names3=null, values3=null,intvars3= 0)`

   Note:

   (a) right now up to 3 `solve_whs` functions can be called at the same time.

   (b) You can call as many `solve_2` and/or `solve_3` and/or `callCbc` functions as needed.

   (c) The first set of `names`, `values` and `intvars` corresponds to the first `solve_whs`, the second set corresponds to the second `solve_whs` and so on.

   (d) String $A$ determines how many solvers you call and what parameters you want to set for each solver:
   - `pt` is for primal tolerance
   - `it` is for integer tolerance
   - `pst` is for presolve tolerance.

   For example:

   ```
   jCbc.par_solve(model, solver, "5 solve_2 pt 1e-8 solve_3 pt 1e-6 it 1e-8 solve_whs pt 1e-8 solve_
   whs it 1e-9 callCbc -zeroT 1e-10 -presolve off -cuts off",names1, values1, intvars1,
   names2, values2, intvars2)
   ```

   will call 5 solvers:

   - `solve_2` with primalT set to 1e-8
   - `solve_3` with primalT set to 1e-6 and integerT set to 1e-8
   - `solve_whs` with primalT set to 1e-8 with names1, values1 and intvars1
   - `solve_whs` with integerT set to 1e-9 with names2, values2 and intvars2
   - `callCbc` with input arguments `-zeroT 1e-10 -presolve off -cuts off`

   The ideal is to call maximum of $n$ solvers if you have $n$ cores on your machine.

7. `solve_unified(jCbcModel A,jOsiClpSolverInterface B,int logLevel=0,String names[],int values[], int intvars,int logLevel= 0)`

   This function calls `solve_whs` if any warm start solution has been provided. Otherwise it will use `solve_2`. If the problem is reported infeasible, it will try to change the tuning especially tolerances to find a solution.

8. `iis(jOsiClpSolverInterface A)`

   Finds an Irreducible Infeasible Set (IIS) if the problem is infeasible. (LP only)

# 7 Code Samples

In this section we have four Java codes. These codes demonstrate the use of jCbc to solve MILP models. The MILP models used here are from CalLite [1] and CalSim [2] applications of California Department of Water Resources.

# 8 Example 1

The purpose of this example is to test whether the generated `jCbc.dll` file works properly. This example shows a Java code that reads a model from the LP file `model.lp`, corresponding to a MILP problem, solves the model and reports the objective value as well as values for integer variables.

The MILP model is of the form

$$
\begin{aligned}
\min \quad & c^T x \\
\text{s.t.} \quad & Ax \lesseqgtr b \\
& l \le x \le u \\
& x_i \in \mathbb{Z} \ \forall i \in I.
\end{aligned}
$$

where $A, b$ and $c$ have rational elements.

This particular instance has 6914 variables (19 of which are integer variables) and 5770 constraints.

```
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 ex1 {
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);
```

```
int nCols=jCbc.getNumCols(Model);
SWIGTYPE_p_double sol = jCbc.new_jarray_double(nCols);
sol = jCbc.getColSolution(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));
}

jCbc.delete_jCbcModel(Model);
}}
```

To compile and run the code, use the following commands in a cmd or terminal shell:

- `javac -cp . ex1.java`

- `java -cp . ex1`

Note that `jCbc.dll` should be in the Java library path. Otherwise you can use the command `java -cp . -Djava.library.path=<path-to-jCbc.dll> ex1` instead. The output of this code is the following:



Figure 1: Output of the first example code

## 8.1 Example 2

Consider given MILP

$$\min \quad x_1 + 2x_2 + 4x_3$$
$$\text{s.t.}$$
$$x_1 + 3x_3 \geq 1$$
$$-x_1 + 2x_2 - 0.5x_3 \leq 3$$
$$x_1 - x_2 + 2x_3 = \tfrac{7}{2}$$
$$0 \leq x_1 \leq 1$$
$$-\infty < x_2 \leq 10$$
$$2 \leq x_3 \leq +\infty$$
$$x_1, x_3 \in \mathbb{Z}.$$

In order to solve this model, we first need to enter (build) this model using a `jCbcModel` object. The following examples shows how to build a model and input data. Then it solves the model and returns the solution.

```
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 ex2 {
public static void main(String argv[]) {
System.loadLibrary("jCbc"); // load the dll library

//CoinModel for adding Rows and Columns faster

SWIGTYPE_p_CoinModel modelObject = jCbc.new_jCoinModel();

//this is our LP solver!
SWIGTYPE_p_OsiClpSolverInterface solver = jCbc.new_jOsiClpSolverInterface();

// values for objective function (costs)
double objValue[] = {1.0, 2.0, 4.0};
// variable upper bounds
double upper[] = {1.0, 10.0, Double.MAX_VALUE};
// variable lower bounds
double lower[] = {0.0,-Double.MAX_VALUE, 2.0};


//define variables:
jCbc.addCol(modelObject,lower[0],upper[0],objValue[0],"x1",true);
jCbc.addCol(modelObject,lower[1],upper[1],objValue[1],"x2",false);
jCbc.addCol(modelObject,lower[2],upper[2],objValue[2],"x3",true);

// this defines a swig type integer array of length 3
SWIGTYPE_p_int row1Index = jCbc.new_jarray_int(2);
// set the values of this array
jCbc.jarray_int_setitem(row1Index,0,0);
jCbc.jarray_int_setitem(row1Index,1,2);
```

```
SWIGTYPE_p_double row1Value = jCbc.new_jarray_double(2);
jCbc.jarray_double_setitem(row1Value,0,1.0);
jCbc.jarray_double_setitem(row1Value,1,3.0);

//add row to CoinModel
jCbc.addRow(modelObject,2, row1Index, row1Value, 1.0, Double.MAX_VALUE,"r1");

SWIGTYPE_p_int row2Index = jCbc.new_jarray_int(3);
jCbc.jarray_int_setitem(row2Index,0,0);
jCbc.jarray_int_setitem(row2Index,1,1);
jCbc.jarray_int_setitem(row2Index,2,2);


SWIGTYPE_p_double row2Value = jCbc.new_jarray_double(3);
jCbc.jarray_double_setitem(row2Value,0,-1.0);
jCbc.jarray_double_setitem(row2Value,1,2.0);
jCbc.jarray_double_setitem(row2Value,2,-0.5);

jCbc.addRow(modelObject,3, row2Index, row2Value, -Double.MAX_VALUE, 3.0,"r2");

SWIGTYPE_p_int row3Index = jCbc.new_jarray_int(3);
jCbc.jarray_int_setitem(row3Index,0,0);
jCbc.jarray_int_setitem(row3Index,1,1);
jCbc.jarray_int_setitem(row3Index,2,2);

SWIGTYPE_p_double row3Value = jCbc.new_jarray_double(3);
jCbc.jarray_double_setitem(row3Value,0,1.0);
jCbc.jarray_double_setitem(row3Value,1,-1.0);
jCbc.jarray_double_setitem(row3Value,2,2.0);

jCbc.addRow(modelObject,3,row3Index,row3Value,7/2.,7/2.,"r3");

//add all rows to OsiClpSolverInterface at once
jCbc.addRows(solver, modelObject);

// This defines a new empty CbcModel
SWIGTYPE_p_CbcModel Model = jCbc.new_jCbcModel();
// Assign the solver to CbcModel
jCbc.assignSolver(Model,solver);


jCbc.setModelName(solver, "test");

jCbc.solve(Model,solver,0);

// get the solution
SWIGTYPE_p_double sol = jCbc.new_jarray_double(3);
sol = jCbc.getColSolution(Model);

// print solution
System.out.println("Solution:");

System.out.println("Objective_Value=" + jCbc.getObjValue(Model));
for (int j = 0; j < 3; j++){
        System.out.println(jCbc.getColName(solver,j)+"="
```

```
                 +jCbc.jarray_double_getitem(sol,j));
}
//delete
jCbc.delete_jCoinModel(modelObject);
jCbc.delete_jCbcModel(Model);


}
}
```
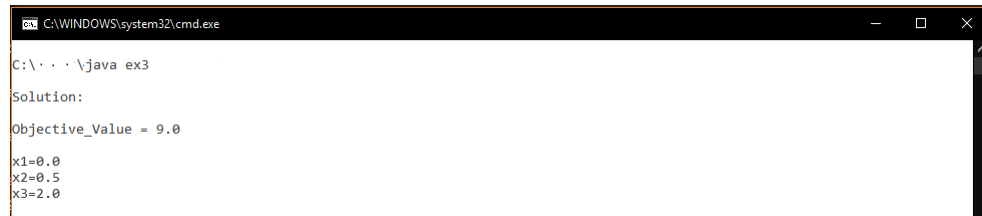
To compile and run the code, use the following commands in a cmd or terminal shell:

- `javac -cp . ex2.java`

- `java -cp . ex2`

Note that `jCbc.dll` should be in the Java library path. Otherwise you can use the command `java -cp . -Djava.library.path=<path-to-jCbc.dll> ex2` instead. The output of this code is the following:

Figure 2: Output of second example code

## 8.2 Example 3

The following example shows a Java code that reads an initial model from LP file `model.lp`, solves it and keeps the integer solution as a warm start solution for later use. Then it reads a second model from another LP file `model1.lp`, and solves it using the warm start solution and reports the objective value.

```
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 ex3 {
 public static void main(String argv[]) {
  System.loadLibrary("jCbc");

SWIGTYPE_p_OsiClpSolverInterface solver = jCbc.new_jOsiClpSolverInterface();
jCbc.readLp(solver,"model.lp");
SWIGTYPE_p_CbcModel Model = jCbc.new_jCbcModel();
jCbc.assignSolver(Model,solver);

jCbc.solve_2(Model,solver,0);

int numIntVars = jCbc.getNumIntegers(solver);
```

19

```
System.out.println("Objective_Value=" + jCbc.getObjValue(Model));
SWIGTYPE_p_std__string names = jCbc.new_jarray_string(numIntVars);
SWIGTYPE_p_int values = jCbc.new_jarray_int(numIntVars);
SWIGTYPE_p_double sol= jCbc.getColSolution(solver);

int k=0;
for (int j = 0; j < jCbc.getNumCols(solver); j++){
        if (jCbc.isInteger(solver,j)==1){
            jCbc.jarray_string_setitem(names,k,jCbc.getColName(solver,j));
            jCbc.jarray_int_setitem(values,k,(int)jCbc.jarray_double_getitem(sol,j));
   k++;
 }}

jCbc.delete_jCbcModel(Model);

SWIGTYPE_p_OsiClpSolverInterface solver1 = jCbc.new_jOsiClpSolverInterface();
jCbc.readLp(solver1,"model1.lp");
SWIGTYPE_p_CbcModel Model1 = jCbc.new_jCbcModel();
jCbc.assignSolver(Model1,solver1);

double time1 = System.currentTimeMillis();
jCbc.solve_whs(Model1,solver1,names,values,k,0);
double time2 = System.currentTimeMillis() - time1;
time_total+= time2;

SWIGTYPE_p_double sol1= jCbc.getColSolution(solver1);
int nCols = jCbc.getNumCols(solver1);


System.out.println("Solution:");
System.out.println("Objective_Value=" + jCbc.getObjValue(Model1));

jCbc.delete_jCbcModel(Model1);
jCbc.delete_jarray_int(values);
jCbc.delete_jarray_string(names);

System.out.println("Total_time="+time_total);

}}
```

To compile and run the code, use the following commands in a cmd or terminal shell:

- `javac -cp . ex3.java`

- `java -cp . ex3`

Note that `jCbc.dll` should be in the Java library path. Otherwise you can use the command `java -cp . -Djava.library.path=<path-to-jCbc.dll> ex3` instead. The output of this code is the following:

Figure 3: Output of third example code

## 8.3 Example 4

The following example shows the use of `par_solve` function. First `model.lp` is read and solved to prepare a warm start solution. Then `par_solve` is called to execute 4 solvers on 4 threads to solve the `model1.lp`. One of the solvers is `solve_whs` which uses the warm start solution.

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

public class ex4 {
public static void main(String argv[]) {
System.loadLibrary("jCbc");

SWIGTYPE_p_OsiClpSolverInterface solver = jCbc.new_jOsiClpSolverInterface();
jCbc.readLp(solver,"model.lp");
SWIGTYPE_p_CbcModel Model = jCbc.new_jCbcModel();
jCbc.assignSolver(Model,solver);

jCbc.solve_2(Model,solver,0);

int numIntVars = jCbc.getNumIntegers(solver);
SWIGTYPE_p_std__string names = jCbc.new_jarray_string(numIntVars);
SWIGTYPE_p_int values = jCbc.new_jarray_int(numIntVars);
SWIGTYPE_p_double sol= jCbc.getColSolution(solver);

int k=0;
for (int j = 0; j < jCbc.getNumCols(solver); j++){
if (jCbc.isInteger(solver,j)==1){
jCbc.jarray_string_setitem(names,k,jCbc.getColName(solver,j));
jCbc.jarray_int_setitem(values,k,(int)jCbc.jarray_double_getitem(sol,j));
k++;
}}

jCbc.delete_jCbcModel(Model);


SWIGTYPE_p_OsiClpSolverInterface solver0 = jCbc.new_jOsiClpSolverInterface();
jCbc.readLp(solver0,"model1.lp");

SWIGTYPE_p_CbcModel Model0 = jCbc.new_jCbcModel();
```

21

```
jCbc.assignSolver(Model0, solver0);

jCbc.par_solve(Model0, solver0, "4_solve_2_solve_3_solve_whs_callCbc_−solve", names, values, k);

System.out.println("ObjVal_=_" + jCbc.getObjValue(Model0));
jCbc.delete_jCbcModel(Model0);

}}
```

To compile and run the code, use the following commands in a cmd or terminal shell:

- `javac -cp . ex4.java`

- `java -cp . ex4`

Note that `jCbc.dll` should be in the Java library path. Otherwise you can use the command `java -cp . -Djava.library.path=<path-to-jCbc.dll> ex4` instead. The output of this code is the following:



Figure 4: Output of fourth example code

# 9 License

jCbc: JNI for Coin OR Mixed Integer Programming Solver CBC
Copyright (C) 2016 Babak Moazzez

This program is free software: you 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.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.

Contact: bmoazzez@ucdavis.edu

# A  Compressed Row Storage

The Compressed Row and Column Storage formats are general: they make absolutely no assumptions about the sparsity structure of the matrix, and they don't store any unnecessary elements. On the other hand, they are not very efficient, needing an indirect addressing step for every single scalar operation in a matrix-vector product or preconditioner solve. The Compressed Row Storage (CRS) format puts the subsequent non-zeros of the matrix rows in contiguous memory locations. Assuming we have a non-symmetric sparse matrix $A$, we create 3 vectors: one for floating-point numbers (`val`), and the other two for integers (`col_ind`, `row_ptr`). The `val` vector stores the values of the nonzero elements of the matrix $A$, as they are traversed in a row-wise fashion. The `col_ind` vector stores the column indexes of the elements in the `val` vector. That is, if $val(k) = a_{ij}$ then $col\_ind(k) = j$. The `row_ptr` vector stores the locations in the `val` vector that start a row, that is, if $val(k) = a_{ij}$ then $row\_prt(i) \le k < row\_ptr(i+1)$. By convention, we define $row\_prt(n+1) = nnz + 1$, where $nnz$ is the number of nonzeros in the matrix $A$. The storage savings for this approach is significant. Instead of storing $n^2$ elements, we need only $2nnz + n + 1$ storage locations. As an example, consider the non-symmetric matrix $A$ defined by

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}$$

The CRS format for this matrix is then specified by the arrays `val, col_ind, row_ptr` given below:

| val | 10 | -2 | 3 | 9 | 3 | 7 | 8 | 7 | 3...9 | 13 | 4 | 2 | -1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| col_ind | 1 | 5 | 1 | 2 | 6 | 2 | 3 | 4 | 1...5 | 6 | 2 | 5 | 6 |

| row_ptr | 1 | 3 | 6 | 9 | 13 | 17 | 20 |
|---|---|---|---|---|---|---|---|

If the matrix $A$ is symmetric, we need only store the upper (or lower) triangular portion of the matrix. The trade-off is a more complicated algorithm with a somewhat different pattern of data access.

# B  Third Party Packages

When compiling Coin OR Cbc, there are five third party packages that should be downloaded manually. The are BLAS[1], LAPACK[2], METIS[3] and MUMPS[4]. These packages are especially useful when the user is dealing with large size problems and the storage and calculation efficiency are important. This manual is written assuming that the user decides to use these packages, otherwise please revise the make file and corresponding commands in the manual.

In order to compile Cbc with third party Packages, you should run the file `get.name` file inside the corresponding package folder. On Windows, you can do this in an MSYS window.

---

[1]Basic Linear Algebra Subprograms, http://www.netlib.org/blas.
[2]Linear Algebra PACKage, http://www.netlib.org/lapack.
[3]Serial Graph Partitioning and Fill-reducing Matrix Ordering, http://glaros.dtc.umn.edu/gkhome/views/metis.
[4]a MUltifrontal Massively Parallel sparse direct Solver, http://mumps.enseeiht.fr.

# C   LP and MPS Files

MPS format was named after an early IBM LP product and has emerged as a de facto standard ASCII medium among most of the commercial LP codes. Essentially all commercial LP codes accept this format, but if you are using public domain software and have MPS files, you may need to write your own reader routine for this.

The main things to know about MPS format are that it is column oriented (as opposed to entering the model as equations), and everything (variables, rows, etc.) gets a name. MPS is a very old format, so it is set up as though you were using punch cards, and is not free format. Fields start in column 1, 5, 15, 25, 40 and 50. Sections of an MPS file are marked by so-called header cards, which are distinguished by their starting in column 1. Although it is typical to use upper-case throughout the file, many MPS-readers will accept mixed-case for anything except the header cards, and some allow mixed-case anywhere. The names that you choose for the individual entities (constraints or variables) are not important to the solver; you should pick names that are meaningful to you, or will be easy for a post-processing code to read.

Here is a little sample model written in MPS format (explained in more detail below):

```
NAME              TESTPROB
ROWS
 N   COST
 L   LIM1
 G   LIM2
 E   MYEQN
COLUMNS
    XONE      COST            1    LIM1            1
    XONE      LIM2            1
    YTWO      COST            4    LIM1            1
    YTWO      MYEQN          -1
    ZTHREE    COST            9    LIM2            1
    ZTHREE    MYEQN           1
RHS
    RHS1      LIM1            5    LIM2            10
    RHS1      MYEQN           7
BOUNDS
 UP BND1      XONE            4
 LO BND1      YTWO           -1
 UP BND1      YTWO            1
ENDATA
```

For comparison, here is the same model written out in an LP format:

```
Optimize
COST:      XONE + 4 YTWO + 9 ZTHREE
Subject To
LIM1:      XONE + YTWO <= 5
LIM2:      XONE + ZTHREE >= 10
MYEQN:    - YTWO + ZTHREE = 7
Bounds
0 <= XONE <= 4
-1 <= YTWO <= 1
End
```

Strangely, there is nothing in MPS format that specifies the direction of optimization. And there really is no standard "default" direction; some LP codes will maximize if you don't specify otherwise, others will minimize, and still others put safety first and have no default and require you to specify it somewhere in a control program or by a calling parameter. If you have a model formulated for minimization and the code you are using insists on maximization (or vice versa), it may be easy to convert: just multiply all the coefficients in your objective function by (-1). The optimal value of the objective function will then be the negative of the true value, but the values of the variables themselves will be correct.

The NAME card can have anything you want, starting in column 15. The ROWS section defines the names of all the constraints; entries in column 2 or 3 are E for equality rows, L for less-than ( $\leq$ ) rows, G for greater-than ( $\geq$ ) rows, and N for non-constraining rows (the first of which would be interpreted as the objective function). The order of the rows named in this section is unimportant.

The largest part of the file is in the COLUMNS section, which is the place where the entries of the A-matrix are put. All entries for a given column must be placed consecutively, although within a column the order of the entries (rows) is irrelevant. Rows not mentioned for a column are implied to have a coefficient of zero.

The RHS section allows one or more right-hand-side vectors to be defined; most people don't bother having more than one. In the above example, the name of the RHS vector is RHS1, and has non-zero values in all 3 of the constraint rows of the problem. Rows not mentioned in an RHS vector would be assumed to have a right-hand-side of zero.

The optional BOUNDS section lets you put lower and upper bounds on individual variables, instead of having to impose bounds through extra rows in the matrix. All the bounds that have a given name in column 5 are taken together as a set. Variables not mentioned in a given BOUNDS set are taken to be non-negative (lower bound zero, no upper bound). A bound of type UP means an upper bound is applied to the variable. A bound of type LO means a lower bound is applied. A bound type of FX ("fixed") means that the variable has upper and lower bounds equal to a single value. A bound type of FR ("free") means the variable has neither lower nor upper bounds.

There is another optional section called RANGES that specifies double-inequalities. This specifies constraints that are restricted to lie in the interval between two values. The final card must be ENDATA (note the spelling).

MPS accepts two commonly used ways of extending the MPS file format to include integer variables: in the COLUMNS section or in the BOUNDS section. In the first way, integer variables are identified within the COLUMNS section of the MPS file by marker lines. A marker line is placed at the beginning and end of a range of integer variables. Multiple sets of marker lines are allowed. Integer marker lines have a field format consisting of Fields 2 through 4.

Field 2: Marker name
Field 3: 'MARKER' (including the single quotation marks)
Field 4: Keyword 'INTORG' and 'INTEND' to mark beginning and end respectively
Fields 5 and 6 are ignored.

The marker name must differ from the preceding and succeeding column names. If no bounds are specified for the variables within markers, bounds of 0 (zero) and 1 (one) are assumed.

In the second way of treating integer variables, integer variables are declared in the BOUNDS section with special bound types in Field 1.

BV: Binary variable - Field 4 must be 1.0 or blank
LI: Integer lower bound - Field 4 is the lower bound value and must be an integer
SC: Semi-continuous variable - Field 4 is the upper bound and must be specified
UI: Integer upper bound - Field 4 is the upper bound value and must be an integer

There are a few special cases of the MPS "standard" that are not consistently handled by implementations. In the BOUNDS section, if a variable is given a non-positive upper bound but no lower bound, its lower bound may default to zero or to minus infinity. If an integer variable has no upper bound specified, its upper bound may default to one rather than to plus infinity.

# D  Mixed Integer Linear Programming Problems

In this section, we first describe the structure of a MILP solver, and then give a brief explanation for each step. The reader can skip these parts if they are already familiar with them.

Consider the following MILP

$$
\begin{aligned}
\min \quad & c^T x \\
\text{s.t.} \quad & Ax = b \qquad\qquad (MILP) \\
& l \leq x \leq u \\
& x_i \in \mathbb{Z} \; \forall i \in I.
\end{aligned}
$$

The integrality constraints allow MILP models to capture the discrete nature of some decisions. For example, a variable whose values are restricted to 0 or 1, called a binary variable, can be used to decide whether or not some action is taken. A MILP solver takes the following steps:

1. Presolve

2. Preprocess

3. Heuristics

4. Initial Solve

5. Warmstart

6. Branch and Cut/Branch and Bound

7. Postsolve/Postprocess

Each step is discussed briefly[5].

Initial solve refers to solving the LP relaxation of the problem using the internal LP solver. Postsolve/-Postprocess is referred to the steps for reversing the effect of presolve/preprocess in the original model.

## D.1  Presolve

Presolve refers to a collection of problem reductions that are typically applied in advance of the start of the branch-and-bound procedure. These reductions are intended to reduce the size of the problem and to tighten its formulation. A simple example of a size-reducing transformation is the following. Suppose a given problem contains the following constraints:

$$
\begin{aligned}
x_1 + x_2 + x_3 &\geq 15 \\
x_1 &\leq 7 \\
x_2 &\leq 3 \\
x_3 &\leq 5
\end{aligned}
$$

Clearly the only way that all of these constraints can be satisfied is if $x_1 = 7, x_2 = 3$, and $x_3 = 5$. In this case we can substitute out these variables, completely removing them from the formulation along with the above four constraints. The list of such possible reductions, of which this is only one, is quite extensive and can have an enormous effect on the overall size of the problem. The above reduction is an LP-presolve reduction, since its validity does not depend on integrality restrictions. An example of an MILP-specific reduction is the following. Suppose that $x_1$ and $x_2$ are non-negative integer variables and that our formulation includes a constraint of the following form:

$$
2x_1 + 2x_2 \leq 1.
$$

---

[5]Some MILP definitions and illustrations are from Gurobi website [5].

Dividing both sides of this constraint by 2 yields:

$$x_1 + x_2 \leq 1/2.$$

Since $x_1$ and $x_2$ are both required to be integral, this inequality clearly implies that $x_1 + x_2 \leq 0$, and so by non-negativity that $x_1 = x_2 = 0$. Hence both of these variables and this constraint can be removed from the formulation.

## D.2  Preprocessing

Preprocess is a collection of methods to analyze each of the inequalities of the system of inequalities defining the feasible region in turn, trying to establish whether the inequality forces the feasible region to be empty, whether the inequality is redundant, whether the inequality can be used to improve the bounds on the variables, whether the inequality can be strengthened by modifying its coefficients, or whether the inequality forces some of the binary variables to either zero or one.

## D.3  Heuristics

Having good incumbents, and finding them as quickly as possible, can be extremely valuable in the MILP search for a number of reasons. First, it may not be possible to solve a problem to provable optimality. For example, the underlying MILP may just be too difficult, or there may be some user imposed restriction on the amount of time that we can allow our MILP algorithm run. In either case, we want to have the best possible feasible solution at termination. Having good feasible solutions also helps the search process prior to termination. The better the objective value of the incumbent, the more likely it is that the value of an LP relaxation will exceed it (in a minimization problem) and hence lead to a node being fathomed. As the above remarks are meant to suggest, it has turned out to be extremely valuable to do a little extra work at some of the nodes of the search tree to see if a good integer feasible solution can be extracted, even though integrality has not yet resulted due to the branching. For example, it may be that many of the integer variables, while not integral, have values that are quite close to integral. We could then consider rounding some of these variables to their nearby values, fixing them to these values, solving the resulting LP relaxation, and repeating this procedure several times in the hopes that all integer variables will fall into line. If they do, and if the resulting feasible has a better objective value than the current incumbent, we can replace that incumbent and proceed.

## D.4  WarmStart

If the solution has been stopped for some reason, then the current information can be stored. When the solution is restarted, this stored information provides a hot start which permits the solver to simply pick up where it left off and continue to the optimum, rather than having to go through the entire set of iterations to reach the current point again. If there have been some minor changes to the LP since the solution process was stopped (could have stopped at the optimum), then you may be able to use a warm start in which the previous solution is initially used to restart. This usually means that you can arrive at the new optimum point in only a few iterations. Warm starts are extremely useful if you are repeatedly solving different versions of the same MILP, each version of which is only very slightly different from the last. This is very useful in mixed-integer linear programming, and also in the algorithms for analyzing infeasibility in linear programs. Depending on the changes that have been made, an expert can select which algorithm to use when the problem is restarted. For example, if a constraint has been added that renders the current point infeasible, it will still be dual-feasible, so that the dual simplex method can be used to resume the solution process. By the same token, adding a new variable will mean that the current solution is still primal-feasible, but it will usually be dual-infeasible, so the primal simplex method should be used to restart the solution process.
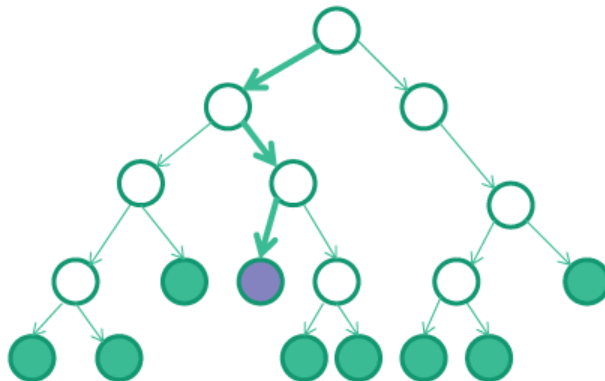
Figure 5: Branch and Bound Tree

## D.5  Branch and Bound

Branch and bound (B&B) is a set of enumerative methods applied to solving discrete optimization problems. The original problem, also referred to as a root problem is bounded from below and above. If the bounds match, the optimal solutions have been found. Otherwise the feasible region i.e., the space in which the argument of the objective function is confined by explicit constraints, is partitioned into subregions. The subregions constitute feasible regions for subproblems, which become children of the root problem in a search tree. The principle behind creating relaxed subproblems (relaxations) of the original problem, the process also known as branching, is that unlike the original problem, the relaxations can be solved within a reasonable amount of time. If a subproblem can be optimally solved, its solution is a feasible, though not necessarily optimal, solution to the original problem. Therefore, it provides a new upper bound for the original problem. Any node of the search tree with a solution that exceeds the global upper bound can be removed from consideration, i.e., the branching procedure will not be applied to that node. The tree is searched until all nodes are either removed or solved. B&B is guaranteed to reach the optimal solution, provided that it exists. See Figure 1.

Mixed Integer Linear Programming problems are generally solved using a linear-programming based branch-and-bound algorithm. Basic LP-based branch-and-bound can be described as follows. We begin with the original MILP. Not knowing how to solve this problem directly, we remove all of the integrality restrictions. The resulting LP is called the linear-programming relaxation of the original MILP. We can then solve this LP. If the result happens to satisfy all of the integrality restrictions, even though these were not explicitly imposed, then this solution is an optimal solution of the original MILP, and we can stop. If not, as is usually the case, then the normal procedure is to pick some variable that is restricted to be integer, but whose value in the LP relaxation is fractional. For the sake of argument, suppose that this variable is $x$ and its value in the LP relaxation is 1.3. We can then exclude this value by, in turn, imposing the restrictions $x \leq 1.0$ and $x \geq 2.0$.

If the original MILP is denoted $P_0$, then we might denote these two new MILPs by $P_1$, where $x \leq 1.0$ is imposed, and $P_2$, where $x \geq 2.0$ is imposed. The variable $x$ is then called a branching variable, and we are said to have branched on $x$, producing the two sub-MILPs $P_1$ and $P_2$. It should be clear that if we can compute optimal solutions for each of $P_1$ and $P_2$, then we can take the better of these two solutions and it will be optimal to the original problem $P_0$. In this way we have replaced $P_0$ by two simpler (or at least more-restricted) MILPs. We now apply the same idea to these two MILPs, solving the corresponding LP relaxations and, if necessary, selecting branching variables. In so doing we generate what is called a search tree. The MILPs generated by the search procedure are called the nodes of the tree, with $P_0$ designated as the root node. The leaves of the tree are all the nodes from which we have not yet branched. In general, if we reach a point at which we can solve or otherwise dispose of all leaf nodes, then we will have solved the

original MILP.

### D.5.1  Fathomed and Incumbent Nodes

We now describe the logic that is applied in processing the nodes of the search tree. Suppose that we have just solved the LP relaxation of some node in the search tree. If it happens that all of the integrality restrictions in the original MILP are satisfied in the solution at this node, then we know we have found a feasible solution to the original MILP. There are two important steps that we then take. First, we designate this node as fathomed. It is not necessary to branch on this node; it is a permanent leaf of the search tree. Second, we analyze the information provided by the feasible solution we have just found, as follows. Let us denote the best integer solution found at any point in the search as the incumbent. At the start of the search, we have no incumbent. If the integer feasible solution that we have just found has a better objective function value than the current incumbent (or if we have no incumbent), then we record this solution as the new incumbent, along with its objective function value. Otherwise, no incumbent update is necessary and we simply proceed with the search. There are two other possibilities that can lead to a node being fathomed. First, it can happen that the branch that led to the current node added a restriction that made the LP relaxation infeasible. Obviously if this node contains no feasible solution to the LP relaxation, then it contains no integer feasible solution. The second possibility is that an optimal relaxation solution is found, but its objective value is bigger than that of the current incumbent. Clearly this node cannot yield a better integral solution and again can be fathomed.

### D.5.2  Best Bound and Gap

Observe that, once we have an incumbent, the objective value for this incumbent, assuming the original MILP is a minimization problem, is a valid upper bound on the optimal solution of the given MILP. That is, we know that we will never have to accept an integer solution of value higher than this value. Somewhat less obvious is that, at any time during the branch-and-bound search we also have a valid lower bound, sometimes call the best bound. This bound is obtained by taking the minimum of the optimal objective values of all of the current leaf nodes. Finally, the difference between the current upper and lower bounds is known as the gap. When the gap is zero we have demonstrated optimality. In general, the better (more constrained) the relaxation and the branching method, the better is the performance of the branch and bound method.

## D.6  Branch and Cut

Branch and Cut is a hybrid method for solving MILP problems. It uses LP based branch and bound and cutting planes. Let us first discuss cutting planes.

The idea of cutting planes is that they tighten the formulation by removing undesirable fractional solutions, as in the case of MILP presolve, but that they do this during the solution process and without the undesirable side-effect of creating additional sub-problems (unlike branching). See Figure 2.

### D.6.1  Cutting planes in optimization

Here is one simple example of a cutting plane. Suppose our formulation includes the following constraint:

$$6x_1 + 5x_2 + 7x_3 + 4x_4 + 5x_5 \leq 15$$

where $x_1$ through $x_5$ are restricted to be binary. Suppose in addition that we have just solved an LP relaxation and that these variables take the following values in this LP relaxation: $x_1 = 0, x_2 = 1, x_3 = x_4 = x_5 = 3/4$. This undesirable solution can be excluded with the following observation: since $7 + 4 + 5 = 16 > 15$, it is not possible that $x_3 = x_4 = x_5 = 1$, and hence that the following new inequality is a valid addition to the given MILP: $x_3 + x_4 + x_5 \leq 2$. Since $3/4 + 3/4 + 3/4 = 9/4 > 2$, the new inequality cuts off the current solution. This inequality is an example of a so-called knapsack cover. But why have we not simply added this new constraint, or cut, at the start? There are several reasons. One is that there are generally an enormous
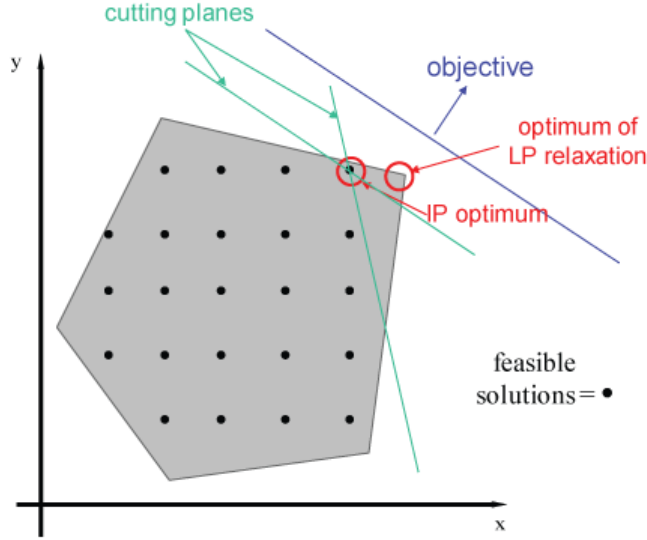
Figure 6: Cutting Planes

number of such additional constraints. It would be too expensive to find them all, and likely impossible to add them all to the model. A second reason is that adding constraints makes the LP relaxations progressively harder to solve. We only want to add these constraints if we know they will help. Judiciously adding such constraints can have an enormously beneficial effect on the solution process.

Now let us show the steps of a branch and cut method which is the core to any MILP solver:

1. Initialization: Denote the initial integer programming problem by $MILP^0$ and set the active nodes to be $L = \{MILP^0\}$. Set the upper bound to be $\bar{z} = +\infty$. Set $\underline{z} = -\infty$ for the one problem $l \in L$.

2. Termination: If $L = \emptyset$, then the solution $x^*$ which yielded the incumbent objective value $\bar{z}$ is optimal. If no such $x^*$ exists (i.e., $\bar{z} = +\infty$) then MILP is infeasible.

3. Problem selection: Select and delete a problem $MILP^l$ from $L$.

4. Relaxation: Solve the linear programming relaxation of $MILP^l$ . If the relaxation is infeasible, set $\underline{z} = +\infty$ and go to Step 6. Let $\underline{z}$ denote the optimal objective value of the relaxation if it is finite and let $x^{lR}$ be an optimal solution; otherwise set $\underline{z} = -\infty$.

5. Add cutting planes: If desired, search for cutting planes that are violated by $x^{lR}$; if any are found, add them to the relaxation and return to Step 4.

6. Fathoming and Pruning:

   (a) If $\underline{z} \geq \bar{z}$ go to Step 2.

   (b) If $\underline{z} \leq \bar{z}$ and $x^{lR}$ is integral feasible, update $\underline{z} = \bar{z}$, delete from $L$ all problems with $\underline{z} \geq \bar{z}$, and go to Step 2.

7. Partitioning: Let $\{S^{lj}\}_{j=1}^{j=k}$ be a partition of the constraint set $S^l$ of problem $MILP^l$. Add problems $\{MILP^{lj}\}_{j=1}^{j=k}$ to $L$, where $MILP^{lj}$ is $MILP^l$ with feasible region restricted to $S^{lj}$ and $\underline{z}_{lj}$ for $j = 1, ..., k$ is set to the value of $\underline{z}_l$ for the parent problem l. Go to Step 2.

See [6] for more details on Mixed Integer Programming theory and applications.

# References

[1] CalLite: Central Valley Water Management Screening Model *California Department of Water Resources*, `http://baydeltaoffice.water.ca.gov/modeling/hydrology/CalLite/index.cfm`

[2] CalSim: The Water Resource Integrated Modeling System (WRIMS model engine or WRIMS) *California Department of Water Resources*, `http://baydeltaoffice.water.ca.gov/modeling/hydrology/CalSim/index.cfm`

[3] J. Forrest and J. Hall *COIN-OR Linear Programming (CLP) v1.15*, Computational Infrastructure for Operations Research (COIN-OR), 2014

[4] COIN-OR Branch and Cut *CBC*, `https://projects.coin-or.org/Cbc`

[5] Gurobi *MILP solver*, `http://www.gurobi.com`

[6] Alexander Schrijver *Theory of linear and integer programming*, Wiley-Interscience Series in Discrete Mathematics, John Wiley & Sons Ltd., Chichester, 1986

[7] Simplified Wrapper and Interface Generator *SWIG*, `http://www.swig.org`