CBC vs CPLEX

Setud guide and first steps

Filippo Checchinato
Davide De Pieri

Padova, May 5, 2019

Introduction

In this paper we are going to describe the setup of *Coin-Or Branch and Cut* (CBC), an open source mixed integer programming solver that represents an alternative to licensed software like CPLEX and GUROBI.

We tested this tool on Windows through Visual Studio Community 2017 and on Linux through CMAKE (with CLion). We will explain all the necessary steps to make the solver work properly. We found that on Windows it is easier to just download the precompiled binaries available on the CBC site, while on Linux it is also possible to compile CBC through a script called CoinBrew.

The COIN-OR Foundation

The Computational Infrastructure for Operations Research (COIN-OR) Foundation, Inc., is a no-profit educational and scientific foundation that was established in March 2004 to manage the COIN-OR project. The mission of the foundation is to create and disseminate knowledge related to all aspects of computational Operations Research by promoting and supporting community-driven developments of open-source software that exploit state-of-the-art research in OR. A comprehensive list of the projects in development can be found on the COIN-OR website under the project section[1].

COIN-OR Branch and Cut (CBC)

CBC[2] is an open-source (distributed under the Eclipse Public License) mixed integer linear programming solver written in C++, and can be used as a callable library or using a stand-alone executable. CBC links a number of other COIN-OR projects for additional functionality, including: Coin-Or Linear Programming (CLP, the default solver for LP relaxations), Coin-Or Cut Generation Library (CGL, for cut generation) and CoinUtils (for various utilities). The software way originally developed by John Forrest, nowadays a retired IBM researcher, and his team.

[1] https://www.coin-or.org/projects
[2] https://github.com/coin-or/Cbc

Setup on Windows

This setup will use the precompiled libraries; binaries for Windows and other platforms are available for download from the CBC bintray[3]. Scrolling down the page until the "Download section" we can choose our preferred version: as we will use Visual Studio 2017, we pick `Cbc-2.10-win32-msvc15.zip`. This is the archive that contains the binaries, headers and libraries for Windows environments.

Once the archive is downloaded we unzip the content in a folder of our choice. Let us take a look to the content in order to understand the role of the directories:

- `/path/to/Cbc-2.10-win32-msvc15/bin`: contains the files `cbc.exe` and `clp.exe` which can be used through command line with additional parameters to read models and solve them.
- `/path/to/Cbc-2.10-win32-msvc15/include`: this folder contains the definitions of all classes, methods and structures, both for C++ and C, that we can use inside our code to build and solve our models.
- `/path/to/Cbc-2.10-win32-msvc15/lib`: contains the compiled black-box libraries that allow to run the procedures defined in the header files of the `include` directory.

Now let us configure the Visual Studio environment; as it is common for external libraries we want to define a custom property to load CBC files. This is very handy, as it requires a setup only the first time: when a new project is created, the property file can be loaded to immediately use the libraries.

First of all we open a new Visual Studio project by clicking on `File>new>Project`. From the different models proposed we choose a C++ empty project as shown on Figure 1. We open the Property Window (either through the menu View or by pressing F4) and open the configuration debug and release, in accord to the architecture (x64 in one case), to show the current properties; right click on one of the folders (we assume Debug x64) and select `'Add new property window'` (Figure 2). On the popup window just appeared we choose the name of the new property (e.g. CBC_Debug.props) and confirm. In the list of the Property Window now we can see the configuration just created. We right-click on it and select 'properties'. From the list of settings on the left we select `C/C++`. On the corresponding panel that appears on the right we edit the field `Additional include directories` (Figure 3). Here we set a

[3]https://bintray.com/coin-or/download/Cbc

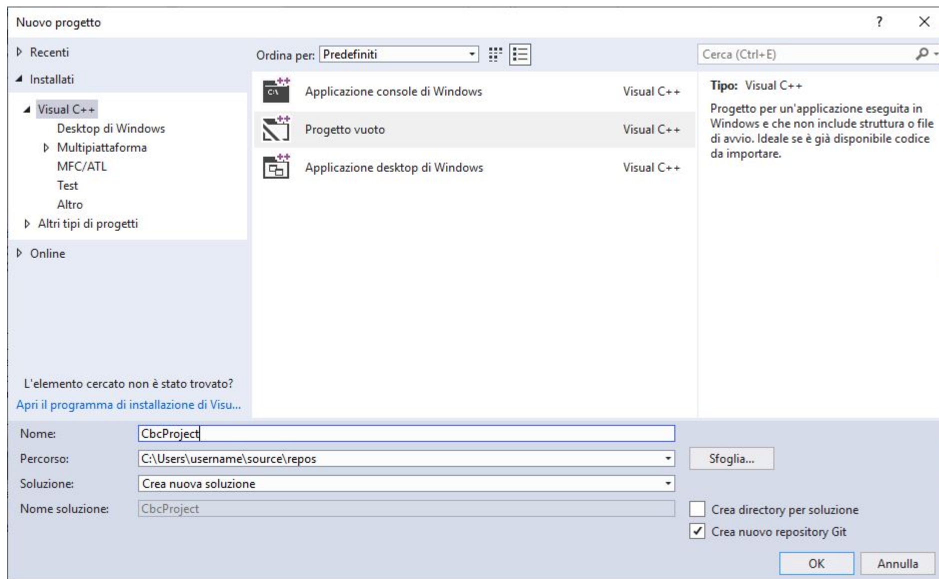

Figure 1: Empty project window

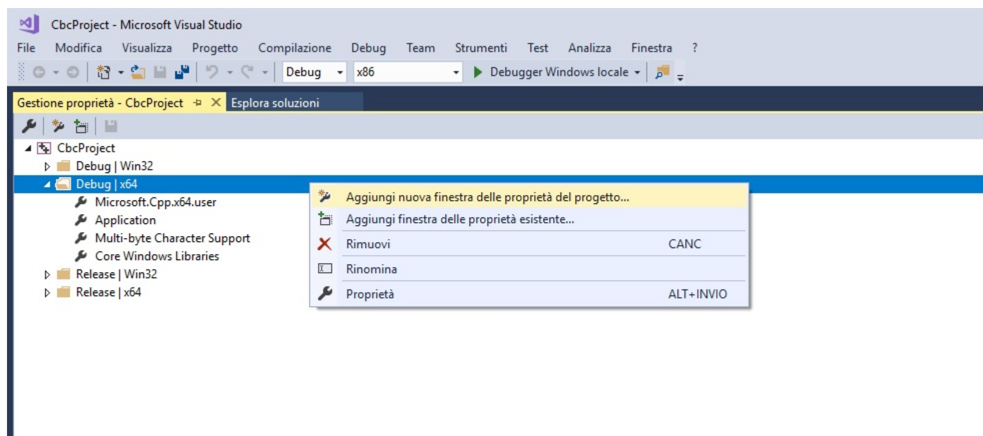

Figure 2: Add new property configuration

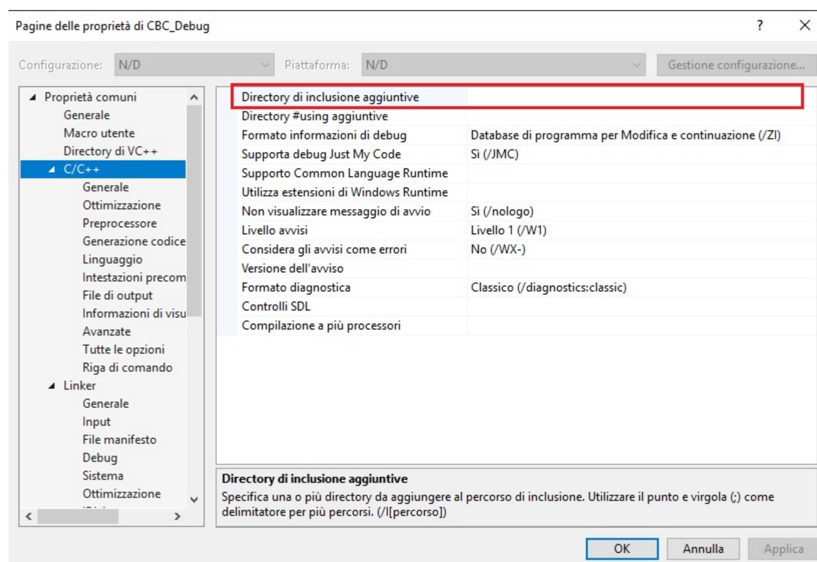

Figure 3: Additional include directories C/C++

new value corresponding to the path to the header files of Cbc, specifically `/path/to/Cbc-2.10-win32-msvc15/include/coin`.
Then we move to `Linker` and edit the field `Additional libraries directories` (Figure 4). As before, we set a new value corresponding to the path to the libraries to import in our project, specifically `/path/to/Cbc-2.10-win32-msvc15/lib`.
Now, under `Linker` section we move to `Input` and edit the field `Additional dependencies`(Figure 5). Here we manually add the name of the libraries we want to import in our project (without path) and that are located inside the directory specified in the previously step. The list of these files should be:

```
libCbc.lib
libCbcSolver.lib
libCgl.lib
libClp.lib
libClpSolver.lib
libCoinUtils.lib
libOsi.lib
libOsiCbc.lib
libOsiClp.lib
libOsiCommonTests.lib
```

Finally we apply the changes and select ‘Ok’. Now, all we have to do to work with Cbc in our C code is to include the <Cbc_C_Interface.h> interface

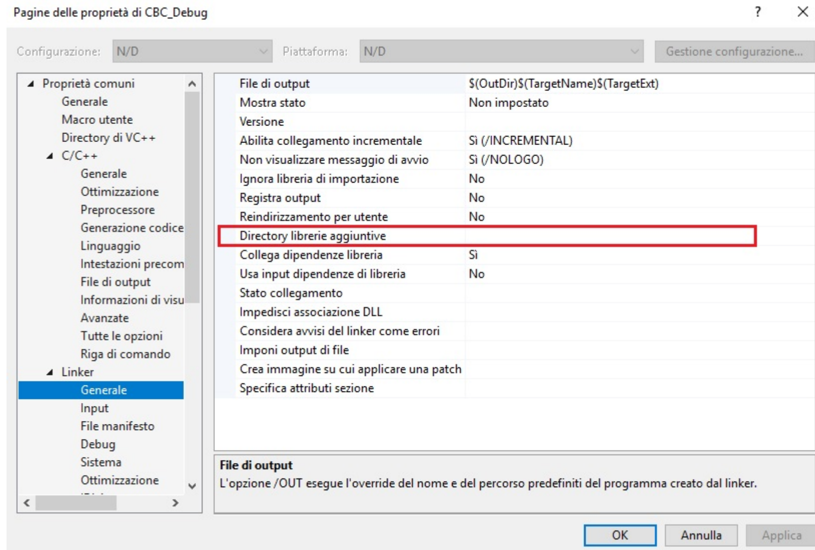

Figure 4: Additional libraries directories

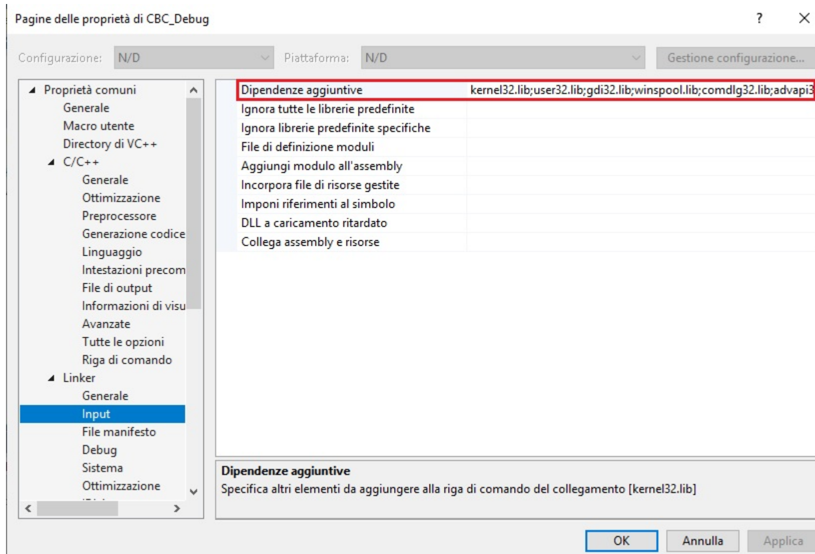

Figure 5: Additional dependencies

at the beginning of the code. The directives we applied to the property file allow the environment to recognize the new interface and all the methods used to build and solve our models.

Setup on Linux

In this setup we will compile CBC from scratch using a script called CoinBrew which will also download all the dependencies directly from the official git repository; it is also possible to use the precompiled libraries, but this case is not covered in this section.

We create a directory of our choice for our library and we save on it the CoinBrew[4] script available in the download section (by right-clicking the link and saving it with extension `.sh`). We will now follow the instructions provided in the CBC git repository[5]; to download all the required files to build CBC in a terminal we use:

```
/path/to/coinbrew.sh fetch --main-proj=Cbc
```

Before proceeding with the compilation one has to be sure to have `gcc` and `g++` installed (you can check by typing them in a terminal); you may have to manually install a fortran compiler through your packet manager of choice (e.g. apt, pacman); the packet is usually called `gfortran` or `gcc-fortran`. If you are missing some other dependency, this next command will stop with an error and will tell what package is required.

```
/path/to/coinbrew.sh build --main-proj=Cbc --test
```

Type `y` when prompted to fetch the files. The compilation might take a while but if it was successful the script will terminate without any error message and you can proceed with (answer `y` when prompted like before):

```
/path/to/coinbrew.sh install --main-proj=Cbc
```

If the process was successful we can see a folder `build` in our original directory, inside which there are many other folders. We are interested only to three of them: `bin`, `include` and `lib`; these serve the same purpose as their Windows counterpart, just check the previous section for a detailed explanation.

A simple way to configure CMAKE is to follow the following structure for the `CMakeList.txt` file:

[4] https://coin-or.github.io/coinbrew/
[5] https://github.com/coin-or/Cbc

```
# CMAKE declarations e.g. version, standard...
...

# Set a variable to your build directory
set(CBC_ROOT_DIR path/to/cbc/build)

# Other settings and flags
...

# CBC headers directory together with
# (eventual) other include directories
include_directories(project_name
                    ...
                    ${CBC_ROOT_DIR}/include/coin)

# Other settings e.g. add_executables()
...

# CBC libraries with extension .so and
# (eventual) other libraries
target_link_libraries(project_name
        ...
        ${CBC_ROOT_DIR}/lib/name_of_the_library.so)
```

Since there are many libraries with extension `.so`, a simple way to list them without wasting too much time is to open the `lib` directory in a terminal and type `ls -1 *.so`, this will output a nice and tidy list, then each file must be formatted like in the example.

Finally, in the code (assuming it is a C project) we include `<Cbc_C_interface.h>`.

Code Examples

In this section we focus on the basic instructions to build and solve a model using structures and functions provided by the Cbc C interface. To do this we compare each piece of code with the corresponding instructions used in CPLEX, in order to ease the learning for those who are familiar with this environment. Both sections are written in C.

Initialization

CBC

```
#include <Cbc_C_Interface.h>

int main(int argc, char **argv){

Cbc_Model *mod;
mod = Cbc_newModel();
Cbc_setProblemName(mod, "MODEL_NAME");
...
}
```

CPLEX

```
#include <cplex.h>

int main(int argc, char **argv){

int error;
CPXENVptr env = CPXopenCPLEX(&error);
CPXLPptr lp = CPXcreateprob(env, &error, "MODEL_NAME");
...
}
```

To initialize an environment for our model we simply define a pointer to a `Cbc_Model` structure which can be initialized by the `Cbc_newModel()` method. Differently from CPLEX, we don't need to define a structure for the data (`lp`) and one for the parameters (`env`), as the model handles everything internally.

Add new variables

To add a new variable to the CBC model, function `Cbc_addCol()` is used. From the following examples we can see how the syntax is slightly different between the two solvers. CPLEX function returns an integer value giving the success or the failure of the operation, while the similar method in Cbc is `void` and does not provide this information. Also, Cbc allows for the insertion of one variable at time, while with CPLEX we can handle more variables through an array of parameters and then call `CPXnewcols()` once to add them all to the model.

FUNCTION

```
void Cbc_addCol(
Cbc_Model* model,
const char* name,
double lb,
double ub,
double obj,
char isInteger,
int nz,
int* rows,
double* coefs
)
```

`model`	pointer to a Cbc_Model
`name`	variable name
`lb`	column lower bound
`ub`	column upper bound
`obj`	objective function coefficient
`isInteger`	1 if variable is integral, 0 otherwise
`nz`	number of rows (constraints) where this column appears, can be 0 if constraints will be added later.
`rows`	index of rows where this column appears, NULL if rows will be added later
`coefs`	coefficients that this column appears in its rows, NULL if rows will be added later

CBC

```
void myAddNewVariable(Cbc_Model *mod){
const double lower_bound_x = 0.0;
const double upper_bound_x = 1.0;
char integral = '1';
char *cname = (char *)calloc(100, sizeof(char));
sprintf(cname, "myNewVariable");
double coeffCost = 12.5;

Cbc_addCol(mod, cname, lower_bound_x, upper_bound_x, coeffCost,
                integral, 0, NULL, NULL);
...
}
```

CPLEX

```
void myAddNewVariable(CPXENVptr env, CPXLPptr lp){
const double lower_bound_x = 0.0;
const double upper_bound_x = 1.0;
const char binary = 'B';
char **cname = (char **)calloc(1, sizeof(char *));
cname[0] = (char *)calloc(100, sizeof(char));
sprintf(cname[0], "myNewVariable");
double coeffCost = 12.5;

if ( CPXnewcols(env, lp, 1, &coeffCost, &lower_bound_x, &upper_bound_x,
 &binary, cname) ) printf(" column creation failed");
...
}
```

Add new constraints

To add a new constraint (row), we can use the function `Cbc_addRow()`. While CPLEX allows to add an "empty" constraint (with `CPXnewrows()`) and then change the values of the considered variables (with `CPXchgcoef()`), Cbc only allows to add a single "filled" constraint in a way similar to `CPXaddrows()` or `CPXaddlazyconstraints()`. Considerations on the return type are similar those made in the previous section.

FUNCTION

```
Cbc_addRow(
Cbc_Model* model,
const char* name,
int nz,
const int* cols,
const double* coefs,
char sense,
double rhs
)
```

model	problem object
name	constraint name, will raise errors if it contains spaces, forbidden characters or it is a duplicate
nz	number of variables with non-zero coefficients in this row
cols	index of variables that appear in this row
coefs	cofficients that that variables appear
sense	constraint sense: L if $\leq$, G if $\geq$, E if $=$, R if ranged and N if free
rhs	right hand size

CBC

```
void myAddNewConstraint(Cbc_Model *mod){
const double rhs = 2.0;
const char sense = 'E';
int nz = 2;
int *cols = (int*)calloc(nz, sizeof(int));
double *coefs = (double*)calloc(nz, sizeof(double));
char *cname = (char *)calloc(100, sizeof(char));
sprintf(cname, "myNewConstraint");
...
Cbc_addCol(mod, cname, nz, cols, coefs, sense, rhs);
}
```

CPLEX

```
void myAddNewConstraint(CPXENVptr env, CPXLPptr lp){
double rhs = 2.0;
const char sense = 'E';
int nz = 2;
const int rcnt = 1;
const int rmatbeg = 0;
int *cols = (int*)calloc(nz, sizeof(int))
double *coefs = (double*)calloc(nz, sizeof(double))
char **cname = (char **)calloc(1, sizeof(char *));
cname[0] = (char *)calloc(100, sizeof(char));
sprintf(cname[0], "myNewConstraint");
...
if ( CPXaddrows(env, lp, 0, rcnt, nz, &rhs, &sense, &rmatbeg, cols,
        coefs, NULL, cname) ) printf(" constraint creation failed");
}
```

Solve a model and retrieve the solutions

To solve a model previously built we can invoke function `Cbc_solve()`. At the end of the computation it will return an `int` informing if the optimal solution was found or not. To retrieve the value of the objective function we call `Cbc_getObjValue()`, while if we want the solution for our variables the function to call is `Cbc_getColSolution()`. Note that the allocation in memory of the solution vector is handled internally, therefore neither `malloc()` nor `free()` is required.

CBC

```
void solveMyModel(Cbc_Model* mod){
if(!Cbc_solve(mod)) {
        printf("No optimal solution found");
        exit(1);
}
double obj = Cbc_getObjValue(mod);
double* variables = Cbc_getColSolution(mod);
}
```

CPLEX

```
void solveMyModel(CPXENVptr env, CPXLPptr lp){
if(!CPXmipopt(env, lp)) {
printf("The model can't be solved");
exit(1);
}
double obj;
CPXgetobjval(env, lp, &obj);
double* variables = (double*) calloc(CPXgetnumcols(env, lp), sizeof(double));
CPXgetx(env, lp, variables, 0, CPXgetnumcols(env, lp)-1);
}
```

Save and Load models

Once our model is built, we may have to export it to a file in order to solve it through command line or run it on a different machine. To export a model we simply call the method `Cbc_writeLp()` which creates a file `filename` with `.lp` extension. The same extension is used in CPLEX and Gurobi. It may be useful if we want to compare the performance of these different solvers. To import a model in our code we call `Cbc_readLp()` which implies the previous creation of an empty model to be filled by the one written in the file.

CBC

```
void exportMyModel(Cbc_Model* mod){
Cbc_writeLp(mod, "Cbc_model"); // Save to "Cbc_model.lp"
}

void importMyModel(const char* filename){
Cbc_Model* mod = Cbc_newModel();
Cbc_readLp(mod, filename); // Requires .lp extension
}
```

CPLEX

```
void exportMyModel(CPXENVptr env, CPXLPptr lp){
CPXwriteprob(env, lp, "CPLEX_model.lp", NULL);
}

void importMyModel(const char* filename){
int error;
CPXENVptr env = CPXopenCPLEX(&error);
CPXLPptr lp = CPXcreateprob(env, &error, "CPLEX_model");
CPXreadcopyprob(env, lp, filename, NULL);
}
```

Delete model

To close a model we use `Cbc_deleteModel()`. This function releases all the resources which were allocated internally by Cbc including the reference to the array returned by `Cbc_getColSolution()`.

CBC

```
void closeMyModel(Cbc_Model* mod){
        ...
        Cbc_deleteModel(mod);
}
```

CPLEX

```
void closeMyModel(CPXENVptr env, CPXLPptr lp){
        ...
        CPXfreeprob(env, &lp);
        CPXcloseCPLEX(&env);
}

```

Command line

As already discussed in the Setup section, after downloading the Cbc binaries (or, eventually, after compiling them), we have access to three folders, namely `bin`, `include` and `lib`. To setup the environment and link our code to the right libraries and headers we use `include` and `lib` . Inside the `bin` folder we can find two executables, `cbc` and `clp`, which can be used to solve models through command line. In particular, `clp` solves only the relaxation of the problem, while `cbc` takes into account the integrabily constraints (if any).

To use these tools we have to open a terminal and move to the `bin` directory. Alternatively, we can add the executables in the PATH of the system and use them from any location. The model we want to solve must be in `.lp` format. Then we simply type:

```
cbc.exe myCbcModel.lp
```

or, in Linux environment:

```
cbc myCbcModel.lp
```

The model will be then read and solved. During the process the program outputs real-time information about the solution process and, at the end, the value of the optimal solution along with information on the used cuts and elapsed time.

Documentation

Currently, the Cbc project lacks a satisfactory documentation: the information is lacking and quite scattered. In the last few years however, the community has grown and a number of release versions of the source code has been released. This improved the performances of the solver, but at the same time parts of the documentation became outdated or incomplete, hence it is often a good idea to take a look directly at the source files to have some insight.

We list here the references to the webpages that we used to move our first steps in Cbc and that we consider reliable sources of information.

- **Official Site** - `https://github.com/coin-or/Cbc`: The official site from which all the following links can be found. Besides the source code, the github page provides an overview of the entire project.
- **Binaries** - `https://bintray.com/coin-or/download/Cbc`: This is where we can find the latest binary packages for Windows and Linux Environments.
- **User Guide** - `https://coin-or.github.io/Cbc/`: An User guide that provide a general overview of the software, providing practical examples of code (C++ only).
- **Doxygen** - `https://www.coin-or.org/Doxygen/Cbc/hierarchy.html`: The standard C++ documentation generated from the source code. This is the main reference to understand methods and parameters. (NOTE: The C documentation is summarized at `https://www.coin-or.org/Doxygen/Cbc/Cbc__C__Interface_8h.html`)
- **Command Line** - `https://projects.coin-or.org/CoinBinary/export/1059/OptimizationSuite/trunk/Installer/files/doc/cbcCommandLine.pdf`: An in-depth guide for the use of Cbc through command line, although it is the official reference it is quite incomplete, for a more complete explanation of the available parameters see: `https://www.gams.com/latest/docs/S_CBC.html`.