User's and programmer's manual of the RCTA package (v.2)

Jordi Castro José Antonio González Daniel Baena
Dept. of Statistics and Operations Research
Universitat Politècnica de Catalunya
Jordi Girona 1–3, 08034 Barcelona, Catalonia
{jordi.castro,j.antonio.gonzalez,daniel.baena}@upc.edu
Technical Report DR 2009-01
January 2009; updated April 2010

Report available from http://www-eio.upc.es/~jcastro

User's and programmer's manual of the RCTA package (v.2) *

Jordi Castro† José Antonio González Daniel Baena
Dept. of Statistics and Operations Research
Universitat Politècnica de Catalunya
Jordi Girona 1–3, 08034 Barcelona, Catalonia
{jordi.castro,j.antonio.gonzalez,daniel.baena}@upc.edu
Technical Report DR 2009-01

Abstract

The package for restricted controlled tabular adjustment (RCTA) was developed in the scope of the Eurostat Framework project 22100.2006.002-2006.532, initially under the specific contract 22100.2006.002-2007.787 and later extended within the specific contract 22100.2006.002-2008.647. It was also supported by the Spanish MEC project MTM2006-05550 and MICINN project MTM2009-0874. It implements a package for the protection of statistical tabular data based on the RCTA method [1]. This document shows the main features of the package. It also describes the package interface and how to embed it within the user's application.

Key words: C/C++ programming languages, restricted controlled tabular adjustment (RCTA), linear programming, mixed integer linear programming, optimization solvers.

*Work supported by Eurostat specific projects 22100.2006.002-2007.787 and 22100.2006.002-2008.647, Eurostat Framework project 22100.2006.002-2006.532, Spanish MEC MTM2006-05550 and MICINN MTM2009-08747 projects, and Catalan SGR-2009-1122 grant.

†Corresponding author

Contents

1 Introduction **6**

2 Two versions of the package **7**

2.1 Input format 7

2.1.1 Example 8

2.2 Standalone application for RCTA 9

2.2.1 Example 12

2.3 Standalone application for TCTA 16

2.3.1 Example 16

2.4 Callable library 18

3 Package options **20**

3.1 Conditional compilation 20

3.2 Guidelines for difficult CTA instances 20

4 Package extensions **21**

4.1 Non-additive tables 22

4.1.1 Example 22

4.2 Negative protection levels 24

4.2.1 Example 24

4.3 Repair infeasibilities tool 26

4.3.1 Example 27

5 Interface routines **28**

5.1 Creating and removing tables 29

CTA_create_table 29

CTA_create_table_from_file 29

CTA_delete_table 29

5.2 Entering table information 30

CTA_put_ncells 30

CTA_put_npcells 30

CTA_put_cellvalue 30

CTA_put_cellperturbation_up 31

CTA_put_cellperturbation_down 31

CTA_put_cellweight 31

CTA_put_lowbound 32

CTA_put_upbound 32

CTA_put_modifupbound 32
CTA_put_index_sensitive_cell 33
CTA_put_info_sensitive_cell 33
CTA_put_typetable 33
CTA_put_K 33
CTA_put_typeconstraints 34
CTA_put_nnz 34
CTA_put_nconstraints 34
CTA_put_begconstraints 35
CTA_put_begconstraints_rowwise 35
CTA_put_begconstraints_columnwise 35
CTA_put_coefconstraints 36
CTA_put_coefconstraints_rowwise 36
CTA_put_coefconstraints_columnwise 36
CTA_put_xcoefconstraints 37
CTA_put_xcoefconstraints_rowwise 37
CTA_put_xcoefconstraints_columnwise 37
CTA_put_rhsconstraints 37
CTA_put_solver 38
CTA_put_optim_gap 38
CTA_put_max_time 38
CTA_put_preprocessSC 39
CTA_put_eprhs 39
CTA_put_epint 39
CTA_put_mipemphasis 39
CTA_put_heurmip 40
CTA_put_varsel 40
CTA_put_objective_fun 40
CTA_put_lowbnd_fobj 41
CTA_set_gap 41
CTA_put_BigM 41
CTA_put_final_status 42
CTA_put_logfile_solver 42
CTA_put_instance_name 42
CTA_put_firstfeas 43
CTA_put_make_additive 43
CTA_put_opt_model 43

CTA_put_repair_infeas . . . 44
CTA_put_repair_inputfile . . . 44
5.3 Retrieving table information . . . 44
CTA_get_ncells . . . 44
CTA_get_npcells . . . 45
CTA_get_cellvalue . . . 45
CTA_get_cellperturbation_up . . . 45
CTA_get_cellperturbation_down . . . 45
CTA_get_cellweight . . . 46
CTA_get_lowbound . . . 46
CTA_get_upbound . . . 46
CTA_get_modifupbound . . . 47
CTA_get_index_sensitive_cell . . . 47
CTA_get_index_cell . . . 47
CTA_get_typetable . . . 48
CTA_get_K . . . 48
CTA_get_typeconstraints . . . 48
CTA_get_nnz . . . 48
CTA_get_nconstraints . . . 49
CTA_get_begconstraints . . . 49
CTA_get_begconstraints_rowwise . . . 49
CTA_get_begconstraints_columnwise . . . 50
CTA_get_coefconstraints . . . 50
CTA_get_coefconstraints_rowwise . . . 50
CTA_get_coefconstraints_columnwise . . . 50
CTA_get_xcoefconstraints . . . 51
CTA_get_xcoefconstraints_rowwise . . . 51
CTA_get_xcoefconstraints_columnwise . . . 51
CTA_get_rhsconstraints . . . 52
CTA_get_solver . . . 52
CTA_get_optim_gap . . . 52
CTA_get_max_time . . . 53
CTA_get_preprocessSC . . . 53
CTA_get_eprhs . . . 53
CTA_get_epint . . . 53
CTA_get_mipemphasis . . . 54
CTA_get_heurmip . . . 54

CTA_get_varsel 54
CTA_get_objective_fun 55
CTA_get_lowbnd_fobj 55
CTA_get_gap 55
CTA_get_BigM 56
CTA_get_final_status 56
CTA_get_logfile_solver 56
CTA_get_instance_name 57
CTA_get_firstfeas 57
CTA_get_make_additive 57
CTA_get_opt_model 57
CTA_get_repair_infeas 58
CTA_get_repair_inputfile 58
5.4 Solving CTA 58
CTA_Find_Solution 58

References **59**

Appendix **60**

A Global information **60**

B List of files (alphabetical order) **60**

C List of routines **61**

D Routines description **66**

1 Introduction

The RCTA package implements the restricted controlled tabular adjustment (RCTA) method for the protection of statistical tabular data. Details about CTA can be found in [1, 2]. This package is used and was motivated for the Protection of Structural Business Statistics by Eurostat [6]; it was later applied to the protection of Balance of Payment data again by Eurostat; it was finally extended for the protection of animal production statistics by Eurostat. It can be used in other applications developing ad-hoc main programs that interface with the RCTA callable library.

The current version of the RCTA package is linked with two state of the art solvers: CPLEX [4] and XPRESS [3]. The package was tested with CPLEX releases 9.0 and 11, so it will likely work with CPLEX 10.0 and new releases if the interface routines are the same than for version 11.0. For XPRESS the 2007 release was used.

The CTA formulation solved in the package is as follows. Given (i) a set of cells $a_i, i = 1, \dots, n$, that satisfy m linear relations $Aa = b$ (a being the vector of a_i's); (ii) a lower and upper bound for each cell $i = 1, \dots, n$, respectively l_{a_i} and u_{a_i}, which are considered to be known by any attacker; (iii) a set $\mathcal{P} = \{i_1, i_2, \dots, i_p\} \subseteq \{1, \dots, n\}$ of indices of sensitive cells; (iv) and a lower and upper protection level for each sensitive cell $i \in \mathcal{P}$, respectively lpl_i and upl_i, such that the released values satisfy either $x_i \geq a_i + upl_i$ or $x_i \leq a_i - lpl_i$; the purpose of CTA is to find the closest safe values $x_i, i = 1, \dots, n$, according to some distance L, that makes the released table safe. This involves the solution of the following optimization problem:

$$\begin{array}{ll} \min\limits_{x} & ||x - a||_L \\ \text{s. to} & Ax = b \\ & l_{a_i} \leq x_i \leq u_{a_i} \quad i = 1, \dots, n \\ & x_i \leq a_i - lpl_i \text{ or } x_i \geq a_i + upl_i \quad i \in \mathcal{P}. \end{array} \tag{1}$$

If we allow $l_{a_i} = u_{a_i}$ for some subset of cells, the values of these cells are preserved. This stronger variant of CTA is named Restricted CTA (RCTA). Problem (1) can also be formulated in terms of deviations from the current cell values. Defining $z_i = x_i - a_i, \quad i = 1, \dots, n$ —and similarly $l_{z_i} = l_{x_i} - a_i$ and $u_{z_i} = u_{x_i} - a_i$—, (1) can be recast as:

$$\begin{array}{ll} \min\limits_{z} & ||z||_L \\ \text{s. to} & Az = 0 \\ & l_{z_i} \leq z_i \leq u_{z_i} \quad i = 1, \dots, n \\ & z_i \leq -lpl_i \text{ or } z_i \geq upl_i \quad i \in \mathcal{P}, \end{array} \tag{2}$$

$z \in \mathbb{R}^n$ being the vector of deviations. The CTA package implements the L_1 distance. Using this distance, after some manipulation, (2) can be written as

$$\begin{array}{ll} \min\limits_{z^+, z^-, y} & \sum_{i=1}^{n} w_i (z_i^+ + z_i^-) \\ \text{s. to} & A(z^+ - z^-) = 0 \\ & 0 \leq z_i^+ \leq \quad u_{z_i} \quad i = 1, \dots, n \\ & 0 \leq z_i^- \leq -l_{z_i} \quad i = 1, \dots, n \\ & upl_i \; y_i \leq z_i^+ \leq u_{z_i} \; y_i \quad i \in \mathcal{P} \\ & lpl_i (1 - y_i) \leq z_i^- \leq -l_{z_i} (1 - y_i) \quad i \in \mathcal{P}, \end{array} \tag{3}$$

$w \in \mathbb{R}^n$ being the vector of cell weights, $z^+ \in \mathbb{R}^n$ and $z^- \in \mathbb{R}^n$ the vector of positive and negative deviations in absolute value, and $y \in \mathbb{R}^p$ being the vector of binary variables associated to protections senses. When $y_i = 1$ the constraints mean $upl_i \leq z_i^+ \leq u_{z_i}$ and $z_i^- = 0$, thus the protection sense is "upper"; when $y_i = 0$ we get $z_i^+ = 0$ and $lpl_i \leq z_i^- \leq -l_{z_i}$, thus protection sense is "lower". Model (3) is a (difficult) mixed integer linear problem (MILP).

The structure of the document is as follows. Section 2 presents the two versions of the package: standalone and callable library, including a simple program that shows how to use RCTA from the user's application. Section 3 describes some of the main options and features of the package. Section 4 shows three extensions that were added to the second version of the package (within the specific project for animal production statistics by Eurostat). In Section 5 we present the set of routines to interface with RCTA, grouped by functional categories. A final Appendix lists all the files and routines of RCTA.

2 Two versions of the package

The package is provided as two standalone applications (one for RCTA, another for TCTA, to be discussed below), and as a set of routines that can be called from the user's application (callable library). Before describing both versions we first show the required instance input format.

2.1 Input format

The package reads instances in CSP format, already used in other methods implemented in the τ-Argus package [5]. Briefly, this format accepts two types of input tables:

- **Format for k-dimensional tables**.

  The structure of a file with this format is:

$$
\begin{array}{l}
k \\
n_1 \; n_2 \; \ldots \; n_k \\
\vdots \\
i_1 \; i_2 \; \ldots \; i_k \quad a_i \quad w_i \quad type \quad l_{a_i} \quad u_{a_i} \quad lpl_i \quad upl_i \quad spl_i \\
\vdots
\end{array}
$$

  k is the table dimension (categorical variables crossed for the table), and $n_1, \ldots, n_k$ the number of categories of each dimension. Unlike in the default CSP format (where $1 \leq k \leq 4$), the package admits any $k \geq 1$. For each combination of categories (including marginals, which are denoted by index/category 0) there is one row with the information of cell i: cell coordinates $i_1, \ldots, i_k$ ($i_j \in \{0, \ldots, n_j\}$; if 0 it means is the marginal for dimension j); cell value a_i; cell weight w_i; cell type (one character, which is `'u'` if cell is sensible, `'s'` if cell may perturbed in the solution, or `'z'` if cell value must be preserved in the solution); lower and upper known cell bounds l_{a_i} and u_{a_i}; and lower and upper protection levels lpl_i and upl_i; last parameter spl_i is not used in CTA.

- **Format for general tables**.

  The structure of a file with this format is:

$$
\begin{array}{l}
0 \\
n \\
\vdots \\
i \;\; a_i \;\; w_i \;\; type \;\; l_{a_i} \;\; u_{a_i} \;\; lpl_i \;\; upl_i \;\; spl_i \\
\vdots \\
m \\
\vdots \\
b_j \;\; l_j \;\; : \; i_{j1} \; (c_{j1}) \; i_{j2} \; (c_{j2}) \; \ldots \; i_{jl_j} \; (c_{jl_j}) \\
\vdots
\end{array}
$$

The 0 in first line means file format is for a general table, and n of second line gives the number of cells. Next n lines, one for each cell, provide the cell information: cell number i (from 0 to $n-1$); cell value a_i; cell weight w_i; cell type (one character, which is `'u'` if cell is sensible, `'s'` if cell may perturbed in the solution, or `'z'` if cell value must be preserved in the solution); lower and upper known cell bounds l_{a_i} and u_{a_i}; and lower and upper protection levels lpl_i and upl_i; last parameter spl_i is not used in CTA. The m of line $n+3$ gives the number of table linear relations or constraints. Next m lines, one for relation, provides the right-hand-side value b_j; number of coefficients l_j of this constraint; and l_j entries giving the cells involved in this j-th linear relation (i_{jk}, $1 \leq k \leq l_j$) and their particular coefficients (c_{jk}, $1 \leq k \leq l_j$).

2.1.1 Example

For instance, for the particular 4×5 2D table of Figure 1, which will be used as test instance in next subsections, the (two-dimensional) input format would be

```
2
4 5
1   1   3     0.3333   s   0     10000   0    0    0
1   2   336   0.0030   s   0     10000   0    0    0
1   3   309   0.0032   z   309   309     0    0    0
1   4   484   0.0021   s   0     10000   0    0    0
1   5   397   0.0025   s   0     10000   0    0    0
1   0   1529  0.0007   s   0     10000   0    0    0
2   1   25    0.0400   s   0     10000   0    0    0
2   2   3     0.3333   s   0     10000   0    0    0
2   3   393   0.0025   u   0     10000   40   30   0
2   4   48    0.0208   s   0     10000   0    0    0
2   5   15    0.0667   s   0     10000   0    0    0
2   0   484   0.0021   s   0     10000   0    0    0
3   1   1     1.0000   z   1     1       0    0    0
3   2   2     0.5000   z   2     2       0    0    0
3   3   137   0.0073   u   0     10000   14   14   0
3   4   145   0.0069   s   0     10000   0    0    0
3   5   107   0.0093   s   0     10000   0    0    0
3   0   392   0.0026   s   0     10000   0    0    0
```

Figure 1: Example instance table, with primaries in boldface

3	336	309	484	397	1529
25	3	**393**	48	15	484
1	2	**137**	145	107	392
55	**291**	91	166	**212**	815
84	632	930	843	731	3220

```
4   1   55     0.0182   s    0       10000   0    0    0
4   2   291    0.0034   u    0       10000   15   30   0
4   3   91     0.0110   s    0       10000   0    0    0
4   4   166    0.0060   s    0       10000   0    0    0
4   5   212    0.0047   u    0       10000   21   21   0
4   0   815    0.0012   s    0       10000   0    0    0
0   1   84     0.0119   s    0       10000   0    0    0
0   2   632    0.0016   s    0       10000   0    0    0
0   3   930    0.0011   s    0       10000   0    0    0
0   4   843    0.0012   s    0       10000   0    0    0
0   5   731    0.0014   s    0       10000   0    0    0
0   0   3220   0.0003   z    3220    3220    0    0    0
```

Note that in this example only the overall total of 3220 is to be preserved in the resulting adjusted table (the other totals are allowed to change, as it will happen in next subsections). If desired, one can force all the totals to be preserved.

2.2 Standalone application for RCTA

The standalone application for the release 1 of RCTA (for extensions in release 2 see Section 4) is called through:

```
main_CTA filename out_dir [-s s] [-f f] [-g g] [-t t] [-p p] [-e e] [-b b] [-m m]
         [-v v] [-c c] [-a a]
```

The first two parameters are mandatory, the remaining ones are optional, and can be entered in any order. Calling this main program with no parameters provides the following usage message:

```
usage: main_CTA filename out_dir [-s s] [-g g] [-t t] [-p p] [-e e] [-b b]
                [-m m] [-v v] [-c c] [-a a]
where    filename: instance file in csp format
         outdir: directory for output files (must exist!)
         s: solver  s= 'c' (CPLEX) or 'x' (XPRESS) (default 'x')
         f: stop at first feasible solution (y='n' (no) or 'y' (yes) (default 'n')
         g: % optimality gap (default g= 5%)')
         t: initial limit time in seconds for optimization (default t= 86400)
         p: preprocess sensitive cells   p='n' (no) or 'y' (yes) (default 'n')
         e: feasibility tolerance (e >= 1.0e-9, default e=1.0e-6)
         i: integrality tolerance (1>=i>=0, default is i= -1: solver default;
```

```
   i>=e in XPRESS)
b: big value to be used, at most, for bounds on deviations
   (default b=Infinity; b= -1: automatically set by the code; if
   problems, set a decent big value as 1.0e+8)
h: emphasis for XPRESS (h=-1,0,1,2,3, default is -1; quality 0--speed 3)
m: mipemphasis for CPLEX (m=0,1,2,3,4, default is 0= balanced)
v: variable selection criteria in CPLEX (v=-1,0,1,2,3,4, default is 0)
c: check input table and solution c= 'n' (no) or 'y' (yes) (default 'y')
a: make table additive if not originally a= 'n' (no) or 'y' (yes)
   (default 'y')
```

A short explanation of the main different options/parameters follows:

f: If yes, the package will stop once the first feasible solution has been found, and it will ask for more CPU time (if 0 is entered, it will definitely stop).

t: CPU time limit in seconds. The optimization will be stopped once this limit has been reached, and the package will ask for more CPU time (if 0 is entered, it will definitely stop).

g: Optimality gap measures the quality of the solution as a relative distance from the current solution to a known lower bound of the optimal solution. Setting g=0% asks for the real optimal solution, but it may be very expensive. Increasing g from the default 5% (to, e.g., 50%) helps in producing a feasible sub-optimal solution quickly.

p: When this preprocessing is active, any sensible cell with a zero lower protection level and a positive upper protection level will be automatically considered as "cell to be protected upwards" (since, otherwise, the original value would be safe since the lower protection level is zero). Similarly, any sensible cell with a zero upper protection level and a positive lower protection level will be automatically considered as "cell to be protected downwards".

e: Feasibility tolerance, i.e., the degree in constraints/bounds violations allowed by the optimization procedure. In CPLEX it must be greater or equal than 1.0e−9; in XPRESS it must be greater or equal than 0. If it is too tight (e.g., 1.0e−9) the solver may falsely conclude the problem is infeasible. By default 1.0e−6 is used. If the problem is reported as infeasible, then you may try to increase it a bit (e.g., 1.0e−5, or 5.0e−5). However, this may affect the quality of the solution: the solver may finish at a solution reported as optimal, that may lead to underprotection of some cells (see Subsection 3.2 for details).

i: Integrality tolerance, i.e., the amount by which the binary variables in the RCTA model can be different from 0 or 1, and still be considered 0 or 1. The CPLEX default is 1.0e−5; the XPRESS default is 5.0e−6. In CPLEX it must be a value greater or equal than 0; in XPRESS it must be greater or equal than the feasibility tolerance. Due to this non-zero integrality tolerance and the bad scaling of RCTA (because of the presence of very large and small values in a table), the solution provided by the solver may violate the protection levels of some cells. In this case it may help to decrease this integrality tolerance (e.g., 1.0e−10). However, this may significantly increase the solution time. Moreover, in XPRESS you are forced to decrease the feasibility tolerance too, and then the solver may falsely conclude the problem is infeasible. Indeed the above feasibility and this integrality tolerances may need to be fine tuned for particular tables. No unique set of values were able to solve all the tables tested; the default values in main_CTA are just reasonable ones. See Subsection 3.2 for guidelines for solving difficult instances.

b: Big value for bounds on allowed (either positive or negative) deviations from current original cell values. The default huge value of 1.0e+120 ($\approx \infty$) guarantees that the bounds given by the user in the input file will be used). This may cause problems with feasibility and integrality tolerances (see comments on these parameters). Tightening the bounds in the input file is a good practice to avoid numerical problems in the solver. Otherwise, a smaller "b" big value may be given (e.g., b=1.0e+5 would be fine). However, be aware that if "b" is set to a too small value, then the problem may become infeasible.

m: CPLEX MIP emphasis parameter (similar to XPRESS heurdivespeedup). It controls the tradeoff between speed, feasibility, and optimality in the MILP algorithm. The meaning is :

m=0: Balance optimality and feasibility.

m=1: Emphasize feasibility over optimality.

m=2: Emphasize optimality over feasibility.

m=3: Emphasize moving best bound.

m=4: Emphasize finding hidden feasible solutions.

The default value in main_CTA is m=0. If the problem is wrongly reported as infeasible, m=1 may be tried. If the solution time is too large, m= 2 may be tried.

h: XPRESS MIP heurdivespeedup parameter (similar to CPLEX mipemphasis). It controls the tradeoff between solution quality and diving speed in the MILP algorithm. The meaning is:

h=-1: Automatic selection.

m= 0,1,2,3: Emphasis bias from emphasis on quality (0) to speed (3).

The default value in main_CTA is h=-1. If the problem is wrongly reported as infeasible, h=0 may be tried. If the solution time is too large, h= 3 may be tried.

c: If this parameter is "y" some simple checks about the input table and the solution obtained is performed and reported on the screen. These checks include feasibility of linear table relations, protection of sensible cells, lower and upper bounds of adjusted table values, and quality of internal optimization model variables (i.e., that no both the positive and negative variables z_i^+ and z_i^- of cell i are positive in the solution of the mathematical programming model (3)).

a: This parameter is described in Subsection 4.1.

When solving an instance, `main_CTA` provides three types of output.

- Output on screen, with minimum information about the instance features, and checks performed (it this option was not deactivated by the user).
- A file named `instance_solver.log`, where `instance` is the instance file and `solver` is either `cplex` or `xpress`, generated by the solver with a summary of the optimization procedure. In a long run, this file may be used to check the progress of the branch-and-cut algorithm. The output depends on the solver—and the version of the solver—used; but in general, the three main values to be checked are: the current best solution, the best lower bound, and the optimality gap (as a percentage). The optimality gap is defined as

$$gap = \frac{best - lb}{1 + |best|} \cdot 100\%,$$

$best$ being the best current solution, and lb the best current lower bound.

Table 1: Return codes

Name	Value	meaning
CTA_OUT_OF_MEMORY	-50	not enough memory
CTA_UNDEFINED	-1	undefined error: to be coded yet
CTA_INTERNAL_ERROR	-2	internal error: should never happen
CTA_TABLE_NOT_EXISTS	-3	attempt to manipulate not existing table
CTA_FILE_NOT_FOUND	-4	input file with table not found
CTA_CPLEX_ERROR	-5	internal CPLEX error
CTA_XPRESS_ERROR	-6	internal XPRESS error
CTA_CPLEX_LICENSE_ERROR	-7	error opening CPLEX license
CTA_XPRESS_LICENSE_ERROR	-8	error opening XPRESS license
CTA_CPLEX_NOT_AVAILABLE	-9	CPLEX not linked in the application
CTA_XPRESS_NOT_AVAILABLE	-10	XPRESS not linked in the application
CTA_OK	0	table successfully created, but CTA not yet solved
CTA_OPTIMAL_SOLUTION	1	optimal solution (within tolerance) found
CTA_TIME_LIMIT_INFEAS	2	time limit exhausted with no feasible solution
CTA_TIME_LIMIT_FEAS	3	time limit exhausted with feasible solution
CTA_INFEASIBLE	4	optimization terminated (not by time limit) with no feasible solution
CTA_FEASIBLE	5	feasible solution found, likely not optimal
CTA_FIRST_FEASIBLE	6	first feasible solution found, likely not optimal
CTA_OTHERWISE	10	other situations from solver with no feasible solution

- A file named `instance_solver.sol`, where `instance` is the instance file, and `solver` is either `cplex` or `xpress`, with the CTA solution table (if the optimization procedure finished successfully). The format of this file is: one line for each cell, providing 4 values i, a_i, x_i and p_i; i is the cell number, a_i the original cell value, x_i the CTA cell value, and p_i is 1 is this cell is sensible, and 0 otherwise.

The different return codes of `main_CTA` (defined in file `cta_table.h` of the package distribution) are listed in Table 1.

2.2.1 Example

For instance, for a file named `example_2D.in` containing the two-dimensional example table of Subsection 2.1, we could type:

```
main_CTA {path_of_instance}/example_2D.in {path_of_output_directory}
```

for using XPRESS, or

```
main_CTA {path_of_instance}/example_2D.in {path_of_output_directory} -s c
```

if CPLEX wants to be used. The output on screen would be:

```
CTA instance:                 example_2D
Number of cells:              30
Number of sensitive cells:    4
```

```
Number of constraints:          11
Solver:                         XPRESS
XPRESS MIP emphasis:            -1
MIP optimality gap:             0.05
MIP time limit (seconds):       86400
Stop at first feasible:         n
Feasibility tolerance:          1e-06
Integrality tolerance:          solver default
Big-M:                          1e+120


Checking table relations for ORIGINAL values.
0 constraints not satisfied within provided tolerance.


At optimum:      Objective F.: 0.5461    Lower bound: 0.544175   Optimality gap: 0.124535%

Checking table relations for CTA values.
0 constraints not satisfied within provided tolerance.


Checking cell protections.
0 unprotected sensitive cells in CTA solution.


Checking cell bounds.
0 violated cell bounds in CTA solution.


Checking cell perturbations.
0 wrong perturbations in CTA solution.

Optimal CTA table found (optimal within tolerances)
Total CPU time: 0.05
```

File `example_2D_xpress.log` with the log of the XPRESS branch-and-cut procedure for this small example is:

```
Reading Problem example_2D
Problem Statistics
          27 (      0 spare) rows
          64 (      0 spare) structural columns
         152 (      0 spare) non-zero elements
Global Statistics
           4 entities        0 sets        0 set members
Presolved problem has:     25 rows      54 cols     152 non-zeros
LP relaxation tightened

   Its         Obj Value      S   Ninf  Nneg        Sum Inf  Time
     0         -3.979600      D     10     0    11168.43750     0
```

```
    17           .354614      D      0     0        .000000     0
Optimal solution found

Starting root cutting and heuristics.

 Its Type    BestSoln    BestBound   Sols    Add    Del     Gap     GInf   Time
+            .687900      .354614      1                  48.45%       0      0
+            .576500      .354614      2                  38.49%       0      0
   1  K      .576500      .451929      2     20      0   21.61%       3      0
   2  K      .576500      .494864      2     14     15   14.16%       2      0
   3  K      .576500      .512832      2     15     13   11.04%       4      0
   4  K      .576500      .525414      2     19     11    8.86%       2      0
   5  K      .576500      .525574      2      3     19    8.83%       3      0
   6  K      .576500      .525754      2     10      2    8.80%       3      0
   7  K      .576500      .526500      2      3      9    8.67%       1      0
   8  K      .576500      .526961      2      3      1    8.59%       2      0
   9  K      .576500      .527318      2      4      4    8.53%       3      0
  10  K      .576500      .528435      2      3      4    8.34%       3      0
  11  K      .576500      .529768      2      5      3    8.11%       3      0
  12  K      .576500      .531636      2     17      4    7.78%       3      0
  13  K      .576500      .531749      2      5     19    7.76%       3      0
  14  K      .576500      .531824      2      3      4    7.75%       3      0
  15  K      .576500      .531900      2      1      3    7.74%       3      0
  16  K      .576500      .531900      2      0      2    7.74%       3      0
  17  G      .576500      .535499      2      3      0    7.11%       4      0
  18  G      .576500      .542468      2     18      2    5.90%       3      0
  19  G      .576500      .544175      2     19     39    5.61%       2      0
+            .546100      .544175      3                   0.35%       0      0
 *** Relative MIP gap less than MIPRELSTOP ***

Cuts in the matrix         : 11
Cut elements in the matrix : 114
 *** Search completed ***     Time:     0 Nodes:     1
Number of integer feasible solutions found is 3
Best integer solution found is      .546100
Best bound is      .544175
Uncrunching matrix
```

Instead, if CPLEX was used, the following file `example_2D_cplex.log` is generated:

```
Tried aggregator 1 time.
MIP Presolve eliminated 0 rows and 8 columns.
MIP Presolve modified 4 coefficients.
Reduced MIP has 27 rows, 56 columns, and 136 nonzeros.
Presolve time =    0.00 sec.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: none, using 1 thread.
Root relaxation solution time =    0.00 sec.
```

```
        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer    Best Node    ItCnt     Gap

*     0+    0                         677.6028        0.3546       12   99.95%
*     0+    0                           0.5765        0.3546       12   38.49%
      0     0        0.5004     3       0.5765     Cuts: 16       27   13.19%
*     0+    0                           0.5461        0.5004       27    8.36%
      0     0        0.5130     3       0.5461      Cuts: 6       30    6.06%

Implied bound cuts applied:  1
Flow cuts applied:  11
Gomory fractional cuts applied:  4
```

Finally the CTA table solution obtained is provided in file `example_2D_xpress.sol` (the same solution is obtained in `example_2D_cplex.sol` if CPLEX is the chosen solver):

```
0                         3220                3220  0
1                           84                  84  0
2                          632                 616  0
3                          930                 946  0
4                          843                 843  0
5                          731                 731  0
6                         1529                1508  0
7                            3                   3  0
8                          336                 336  0
9                          309                 309  0
10                         484                 484  0
11                         397                 376  0
12                         484                 514  0
13                          25                  25  0
14                           3                   3  0
15                         393                 423  1
16                          48                  48  0
17                          15                  15  0
18                         392                 378  0
19                           1                   1  0
20                           2                   2  0
21                         137                 123  1
22                         145                 145  0
23                         107                 107  0
24                         815                 820  0
25                          55                  55  0
26                         291    274.999999999999  1
27                          91                  91  0
28                         166                 166  0
29                         212                 233  1
```

This solution corresponds to table (b) of Figure 2; table (a) of Figure 2 shows the original values.

Figure 2: (a) Original table, with primaries in boldface; (b) Adjusted table after CTA

3	336	309	484	397	1529
25	3	**393**	48	15	484
1	2	**137**	145	107	392
55	**291**	91	166	**212**	815
84	632	930	843	731	3220

(a)

3	336	309	484	376	1508
25	3	423	48	15	514
1	2	123	145	107	378
55	275	91	166	233	820
84	616	946	843	731	3220

(b)

2.3 Standalone application for TCTA

The main_TCTA executable for RCTA is the program to be used for sequential protection of a list of single cells. The problem in the sequence are (reasonably simple) LPs, unlike for main_CTA, that solves a MILP model. Calling this main program with no parameters provides the following usage message:

```
usage: main_TCTA instfile listfile out_dir [-s s] [-c c] [-a a]
where    instfile: instance file in csp format
         listfile: file with list of cells
         outdir: directory for output files (must exist!)
         s: solver       s= 'c' (CPLEX) or 'x' (XPRESS) (default 'x')
         c: check input table and solution c= 'n' (no) or 'y' (yes) (default 'y')
         a: make table additive if not originally a= 'n' (no) or 'y' (yes)
            (default 'y')
```

The "instfile" is the same file used for main_CTA. The additional "listfile" provides the list of cells to be single-protected by CTA. The format of this file is, first, a line with the number of cells to be dealt with, and the list of cells. The program produces a summary of information on the screen, and a `instance_solver.sol` file with the solution (minimum and maximum adjusted values for all the cells after the sequence of single-CTA runs).

2.3.1 Example

For instance, for the two-dimensional example table of Subsection 2.2, if the list of sensitive files are the first two (of values 393 and 137, and coordinates (2,3) and (3,3)), the "listfile" would be

```
2
15
21
```

If the `main_TCTA` code is applied to this instance by typing (for instance, for XPRESS)

```
main_TCTA example_2D.in example_2D_list.in {path_of_output_directory}
```

(where `example_2D_list.in` is the file above shown with the two first sensitive cells), the output on screen would be

```
T-CTA instance:                example_2D
Number of cells:               30
Number of sensitive cells:     4
Number of single cells:        2
Number of constraints:         11
Solver:                        XPRESS


[0] Protecting single cell 15
At optimum:      Objective F.: 0.291
Optimal CTA table found (optimal within tolerances)

[1] Protecting single cell 21
At optimum:      Objective F.: 0.203
Optimal CTA table found (optimal within tolerances)
Total CPU time: 0.03
```

The solution file `example_2D_xpress.sol` would be in this case:

```
0                        3220                      3220
1                          84                        84
2                         632                       632
3                         930                       960
4                         813                       843
5                         731                       731
6                        1499                      1529
7                           3                         3
8                         336                       336
9                         309                       309
10                        454                       484
11                        397                       397
12                        470                       514
13                         25                        25
14                          3                         3
15                        379                       423
16                         48                        48
17                         15                        15
18                        392                       406
19                          1                         1
20                          2                         2
21                        137                       151
22                        145                       145
23                        107                       107
24                        815                       815
25                         55                        55
26                        291                       291
27                         91                        91
28                        166                       166
29                        212                       212
```

which corresponds to the TCTA table of Figure 3:

Figure 3: TCTA table after sequential single-protection of first two sensitive cells

3	336	309	[454,484]	397	[1499,1529]
25	3	[379,423]	48	15	[470,514]
1	2	[137,151]	145	107	[392,406]
55	291	91	166	212	815
84	632	[930,960]	[813,843]	731	3220

2.4 Callable library

The callable library provides a set of routines that can be embedded in a user's application. They provide full control over the package. The example program of pages 19–19 illustrates the main steps that need to be performed to protect any table (e.g., that of Figure 1) with RCTA. This sample code is a (very reduced) subset of the standalone code `main_CTA`. Some of the main routines of the RCTA callable library used in the code are discussed in next items. For a full list of the available routines in the callable library, see Section 5.

- Any code that uses RCTA has to include the header file `cta_table.h`, as in line 8 of the example program. This file contains all the declarations (data structures and routines) needed to interface with RCTA.
- We first need to declare a TABLE* variable (pointer to TABLE structure). In the code we named it `tab` (line 15). It will store all the required information for the table, both before and after its protection.
- After the declaration, we must create the real space for the table. This is done at line 19, calling `CTA_create_table_from_file()`. The first parameter is the TABLE structure, the second is the instance file name, and the last of type TYPE_CONSTRAINTS tells how to internally store the table constraints (by rows, columns, or both); COLUMNS is the preferred choice both for CPLEX and XPRESS. Routine `CTA_create_table_from_file()` returns 0 if successful, and then we can proceed with the protection the table; otherwise the code writes and error message and does not protect the table.
- Routines `CTA_put_logfile_solver()` and `CTA_put_instance_name()` of lines 24–25 provide the name of the log file with the solver output, and the instance name.
- Routine `CTA_Find_Solution()` of line 26 protects the table, with default parameters in this example, since they were not changed in previous calls. The user has to check the return code to see if either a feasible or optimal solution was found (as in lines 27–28); otherwise, no solution will be stored in the TABLE structure. See Table 1 for the list of return codes.
- If a solution to the CTA problem is found, then lines 30–34 write a minimal output with the solution: cell number (`k`), original cell value (`a`) and adjusted cell value (`a+xp-xn`). The number of cells, cell value, upwards and downwards deviations are retrieved by respectively calling routines `CTA_get_ncells()`, `CTA_get_cellvalue()`, `CTA_get_cellperturbation_up()` and `CTA_get_cellperturbation_down()` of lines 30–33.
- Finally, the memory space of the table is freed at line 38, calling `CTA_delete_table()`.

We next display the full example program in C/C++.

Example program using the callable library

```
/***************************************************/
// Simple main program for the CTA callable library
/***************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string>
#include "cta_table.h"


using namespace std;

int main(int argc, char *argv[])
{
    TABLE *tab= NULL;
    int ret_stat;

    // create and read table in file example_2D.in
    ret_stat= CTA_create_table_from_file(&tab, "example_2D.in", COLUMNS);
    if (ret_stat < 0)
        cout << "Error creating table\n";
    else {
        // if no error creating table, solve CTA
        CTA_put_logfile_solver(tab,"logfile.log");
        CTA_put_instance_name(tab,"example_2D");
        ret_stat= CTA_Find_Solution(tab);
        if (ret_stat== CTA_OPTIMAL_SOLUTION || ret_stat== CTA_FEASIBLE ||
            ret_stat== CTA_TIME_LIMIT_FEAS || ret_stat==CTA_FIRST_FEASIBLE) {
            // write original and CTA cell values  to standard output
            for (int k=0; k<CTA_get_ncells(tab);k++) {
                double a= CTA_get_cellvalue(tab, k);
                double xp= CTA_get_cellperturbation_up(tab, k);
                double xn= CTA_get_cellperturbation_down(tab, k);
                cout << k << "\t" << a << "\t" << a+xp-xn<<endl;
            }
        }
    }
    CTA_delete_table(tab);
    return(ret_stat);
}
```

If the above code is applied to, e.g., the table of Figure 1 we obtain the adjusted table reported in Subsection 2.2.

3 Package options

3.1 Conditional compilation

The package has been successfully compiled an tested in both Linux (using gcc 4.2) and MS-Windows XP (using MS-Visual C++ 6.0, MSVC6 for short). It should also work in any other Unix or MS-Windows system.

Three symbols are available for conditional compilation depending on the environment. This is done through /D*symbol_name* in MSVC6 and -D*symbol_name* in gcc. The last two of these symbols are only required for compiling the package, whereas the first one needs also to be defined for compiling the user's application, as explained below. The three symbols are:

- `WIN32`. This symbol must be defined for compiling the RCTA package and the main program with MSVC6 in a MS-Windows system. In Linux systems, this paragraph can be skipped. The symbol is also needed for the user's routines that interface with the RCTA package, again only in MS-Windows systems. When `WIN32` is defined, the additional symbol `CTA_BC_EXPORTS` is required. It allows exporting the interface functions in the .dll libraries. The distribution of the package already includes those symbols, and the user/programmer does not have to care about them. This export symbol **DOES NOT** has to be defined for compiling the user's application, otherwise it will fail to interface with the package.
- `CPLEX_`. This symbol is required if one has a CPLEX license and plans to use it. It is not needed for compiling the user's application. If the symbol is defined, either symbols `CPLEX9_` or `CPLEX11_` must also be defined for the particular CPLEX release to be used (releases 9 and 11 were the only ones tested for the application). If `CPLEX_` is not defined and RCTA is asked to use CPLEX, it will return an error.
- `XPRESS_`. This symbol is required if one has a XPRESS license and plans to use it. It is not needed for compiling the user's application. No symbol with XPRESS release version is needed; the package was developed for release 2007. If `XPRESS_` is not defined and RCTA is asked to use XPRESS, it will return an error.

3.2 Guidelines for difficult CTA instances

Several package options allow the user to control the solution of the mathematical programming model of CTA of (3). These options were listed in Subsection 2.2 Unfortunately, CTA is a difficult problem and no set of default options is valid for any CTA instance. This applies to both solvers, CPLEX and XPRESS. The main parameters to be adjusted, if difficulties appear in the solution of some instance, are the following:

- **Feasibility tolerance.** This is the degree in constraints/bounds violations allowed by the optimization procedure. In CPLEX it must be greater or equal than 1.0e−9; in XPRESS it must be greater or equal than 0. If it is too tight (e.g., 1.0e−9) the solver may falsely conclude the problem is infeasible. By default 1.0e−6 is used. If the problem is reported as infeasible, and you believe it is feasible, then try to increase the feasibility tolerance a bit (e.g., to 1.0e−5, or 5.0e−5). However, this may affect the quality of the solution: the solver may finish at a solution reported as optimal, that may lead to underprotection of some cells. The explanation is the following: Model (3) includes constraints

$$0 \leq z_i^+ \leq u_{z_i}\, y_i \qquad 0 \leq z_i^- \leq -l_{z_i}(1-y_i),$$

where u_{z_i} and $-l_{z_i}$ are the maximum cell deviations upwards and downwards, respectively. If the cell bounds are large, u_{z_i} and $-l_{z_i}$ may be large as well. The above constraints force that when $y_i = 1$ (protection sense is "upper") the downwards deviation must satisfy $z_i^- \leq -l_{z_i}(1 - y_i) = 0$, thus it is 0. However, in practice, because of the feasibility tolerance, we may have for instance $y_i = 1 - \epsilon$, and thus if $-l_{z_i} = M$, and M is a big-value, the constraint imposes $z_i^- \leq -l_{z_i}(1 - y_i) = M(1 - (1 - \epsilon)) = M\epsilon > 0$. Therefore we allow a downwards deviation is a cell that was "upper" protected, leading to an underprotection. A similar reasoning applies for "lower" protected cells (i.e., $y_i = \epsilon$ instead of $y_i = 0$).

Decreasing the feasibility tolerance, we reduce the above ϵ value, but we make the problem much harder, and the solver may report it is infeasible. A best option, if possible, would be to avoid big-values M for cell deviations, but this means the real cell bounds (lower and upper bounds) should be small. If they were about 1.0e+4 or 1.0e+5, the above underprotection issue would not appear. However, in practice, real tables contain very big cell values, and the above "small" bounds are not possible. The user may try to tight them, if she/he has information about the data. Unfortunately, if the imposed bounds are too tight, the problem may become a "real" infeasible problem. The package includes an option (option "b" of `main_CTA`) to play with, which automatically sets a maximum bound for all deviations.

- **Integrality tolerance**. This is the amount by which the binary variables in the RCTA model can be different from 0 or 1, and still be considered 0 or 1. The CPLEX default is 1.0e−5; the XPRESS default is 5.0e−6. In CPLEX it must be a value greater or equal than 0; in XPRESS it must be greater or equal than the feasibility tolerance. This parameter is related with the above feasibility tolerance. Indeed combining both of them we may try to obtain feasible/optimal solutions with no underprotected cells. We discussed in previous item how to avoid underprotections by tuning the feasibility tolerance. The integrality tolerance provides a new possibility: if it is set to a very small value, e.g., 1.0e−10, we are asking for binary solutions that are far from 0 or 1 at most 1.0e−10. Therefore the problem with constraints $z_i^+ \leq u_{z_i}\, y_i$ and $z_i^- \leq -l_{z_i}(1-y_i)$, explained above, may be avoided. Unfortunately, there are two drawbacks of this approach. The first is that this may significantly increase the solution time of the branch-and-cut procedure (very significantly, indeed). The second is that (unlike CPLEX) in XPRESS, as said above, the integrality tolerance must be greater or equal than the feasibility tolerance. Then if we reduce the integrality tolerance, we must reduce the feasibility tolerance as well, and then the algorithm may falsely conclude the problem is infeasible.

- **Infeasible problems**. If a not-too-small (or the default) feasibility tolerance is being used yet, and the solver is still reporting the problem as infeasible, it may help to tune the MIP emphasis parameters. They change the behaviour of the solver in the branch-and-cut tree, and may lead to feasible solutions. This behaviour may be changed with parameters "m" and "h" of `main_CTA`. A more detailed description of how these parameters affect the branch-and-cut procedure must be found in the CPLEX and XPRESS user's manuals [3, 4].

4 Package extensions

Three package extensions were added in the second release of the CTA package (needed for the protection of European animal production statistics). These three extensions are, described in next subsections, are:

- Treatment of non-additive tables (i.e., the deviations should be able to both protect and make the table additive).

- New model for RCTA with negative protection levels (needed for dealing with correlated tables).
- A tool for pseudo-automatic analysis for infeasible instances.

4.1 Non-additive tables

If the original cell values of the tables do not satisfy $Aa = b$ (i.e., the table is non-additive), the deviations may be asked to make the resulting table both protected and additive. By default, when the package detects a non-additive table, it will make the resulting table additive, unless stated by the user. This is controlled by option `-a a` of main_CTA and main_TCTA: by default a='y', i.e. the deviations will make the table additive; otherwise, the user may set a='n'. Internally, RCTA computes the possible infeasibilities of the table as $b - Aa$, and then the constraints of the CTA model are $Az = b - Aa$ (instead of $Az = 0$, as in (2)). Indeed, note that if the original table does not satisfy $Aa = b$ then such a z makes the resulting table feasible:

$$A(a + z) = Aa + Az = Az + (b - Az) = b.$$

If the original table is already additive, then $b - Az = 0$ and thus $Az = b - Az = 0$ is equivalent to the constraints of (2).

4.1.1 Example

For instance, Figure 4(a) shows a non-additive version of the two-dimensional example table of Subsection 2.2. The original marginal values of 1529 and 930 were replaced by 1550 and 950, making the table nonadditive (four infeasible constraints in $Aa = b$). Running

```
main_CTA {path_of_instance}/example_2D_nonadd.in {path_of_output_directory} -g 0
```

the following output is obtained on screen:

```
CTA instance:                   example_2D_nonadd
Number of cells:                30
Number of sensitive cells:      4
Number of constraints:          11
Solver:                         XPRESS
XPRESS MIP emphasis:            -1
MIP optimality gap:             1e-05
MIP time limit (seconds):       86400
Stop at first feasible:         n
Feasibility tolerance:          1e-06
Integrality tolerance:          solver default
Big-M:                          1e+120


Checking table relations for ORIGINAL values.
n. const.              LHS             RHS
0               21              0
```

Figure 4: (a) Original nonadditive table, with primaries in boldface, and non-additive totals in red; (b) Adjusted safe and additive table after CTA

3	336	309	484	397	1550
25	3	**393**	48	15	484
1	2	**137**	145	107	392
55	**291**	91	166	**212**	815
84	632	950	843	731	3220

(a)

3	336	309	484	376	1508
25	3	423	48	15	514
1	2	123	145	107	378
55	275	91	166	233	820
84	616	946	843	731	3220

(b)

```
1                20                0
4                -20               0
7                -21               0
4 constraints not satisfied within provided tolerance.


Optimization performed with CLASSICAL model


At optimum:      Objective F.: 0.5476    Lower bound: 0.5476     Optimality gap: 0%

Checking table relations for CTA values.
0 constraints not satisfied within provided tolerance.


Checking cell protections.
0 unprotected sensitive cells in CTA solution.


Checking cell bounds.
0 violated cell bounds in CTA solution.


Checking cell perturbations.
0 wrong perturbations in CTA solution.

Optimal CTA table found (optimal within tolerances)
Total CPU time: 0.02
```

It is observed that the check for table relations for the original values detects the four infesiable constraints of $Aa = b$, while the perturbed CTA values $a + z$ satisfy the additivity of the table. The solution obtained can be seen in Figure 4(b)

4.2 Negative protection levels

If the problem has negative protection levels (which can be useful for the sequential protection of correlated tables), the optimization model (3) is no longer valid (let us name it the "classical" model). When negative protection levels are detected by the RCTA package, it automatically switches to the alternative model

$$\begin{array}{ll} \min\limits_{z^+,z^-,y} & \sum\limits_{i=1}^{n} w_i(z_i^+ + z_i^-) \\ \text{subject to} & A(z^+ - z^-) = b - Aa \\ & l_z \leq z^+ - z^- \leq u_z \\ & z_i^+ - z_i^- \geq upl_i y_i + l_{z_i}(1-y_i) \quad i \in \mathcal{P} \\ & z_i^+ - z_i^- \leq -lpl_i(1-y_i) + u_{z_i} y_i \quad i \in \mathcal{P} \\ & (z^+, z^-) \geq 0 \\ & y_i \in \{0,1\} \quad i \in \mathcal{P}. \end{array} \tag{4}$$

This model, that will be referred as the "new" model, is valid for any kind of instance, with either positive or negative protection levels. However, it is less efficient than the classical model, and then, for problems with only positive protection levels, the classical model should be the best option. If the classical model has some difficulty (e.g., it is not able to obtain an optimal, even a feasible solution, by, e.g., numerical tolerances or any other cause) then the new model could be tried.

The model to be used is controlled by option `-o o` of main_CTA: by default o='a', i.e. automatic selection of the model: if there is a negative protection level, the new model is used, otherwise the classical model will be applied. Setting o='n' (new model) or o='c' (classical model), the user may set a particular model. Note that for instances with negative protection levels, the option o='c' is forbidden. For problems with only positive protection levels, either model can be used. The output on screen clearly states which was the model used for the optimization stage. Recently, a new model, named the "hybrid" model, was obtained, mixing the new and classical models. This model can be used for both negative and positive protection levels, and it is as efficient as the classical model if there are few negative protection levels. This hybrid model will be included in new releases of the RCTA package.

4.2.1 Example

For instance, Figure 5(a) shows the non-additive table of Figure 4(a) including the lower (subscripts) and upper (superscripts) protection levels of sensitive cells. Note cell a_{23} has a negative upper protection level.

Running

```
main_CTA {path_instance}/example_2D_nonadd_negprotlev.in {output_directory} -g 0
```

the following output is obtained on screen:

```
CTA instance:                   example_2D_nonadd_negprotlev
Number of cells:                30
Number of sensitive cells:      4
Number of constraints:          11
Solver:                         XPRESS
```

Figure 5: (a) Original table (nonadditive and with negative protection levels), with primaries in boldface, lower protections levels as subscripts, and upper protection levels as superscripts; (b) Adjusted safe and additive table after CTA

3	336	309	484	397	1550
25	3	$\mathbf{393}_{\mathbf{40}}^{\mathbf{-30}}$	48	15	484
1	2	$\mathbf{137}_{\mathbf{14}}^{\mathbf{14}}$	145	107	392
55	$\mathbf{291}_{\mathbf{15}}^{\mathbf{30}}$	91	166	$\mathbf{212}_{\mathbf{21}}^{\mathbf{21}}$	815
84	632	950	843	732	3220

(a)

3	336	309	484	377	1509
25	3	393	48	15	484
1	2	151	145	107	406
55	276	91	166	233	821
84	617	944	843	731	3220

(b)

```
XPRESS MIP emphasis:            -1
MIP optimality gap:             1e-05
MIP time limit (seconds):       86400
Stop at first feasible:         n
Feasibility tolerance:          1e-06
Integrality tolerance:          solver default
Big-M:                          1e+120
Make additive table:            yes
Optimization model:             automatic selection
Repair infeasibility:           no


Checking table relations for ORIGINAL values.
n. const.                LHS              RHS
0               21              0
1               20              0
4               -20             0
7               -21             0
4 constraints not satisfied within provided tolerance.

Optimization performed with NEW model

At optimum:      Objective F.: 0.4062    Lower bound: 0.40619    Optimality gap: 0.000711136%

Checking table relations for CTA values.
0 constraints not satisfied within provided tolerance.


Checking cell protections.
0 unprotected sensitive cells in CTA solution.


Checking cell bounds.
0 violated cell bounds in CTA solution.
```

```
Checking cell perturbations.
0 wrong perturbations in CTA solution.

Optimal CTA table found (optimal within tolerances)
Total CPU time: 0.05
```

4.3 Repair infeasibilities tool

If the problem is found to be infeasible by the solver, that is: there is no solution that can satisfy both $Aa = b$ and the protection levels, the case should be stated in another way such that it is possible to protect the table. Finding out which parameters (usually, variable bounds or protection levels) are too tight for the problem is not an easy task, since the infeasibility could come from the interaction among several parameters, which individually may seem not troubling but jointly lead to an insoluble situation.

The code of main_cta includes now a tool for pseudo-automatic analysis of infeasible instances. The options controlling the feature are:

- `-r r`: apply repair infeasibility procedure: `r`='n' (no, by default) or `r`='y' (yes);
- `-x x`: file name with information for repair infeasibility tool (if needed).

The procedure tries to find a quasi-solution by relaxing the variable and the constraint bounds. Though the protected table provided is infeasible, it is intended to include minimal violations at specific cells, which could be enlightening to find a suitable way to protect the table.

If the user wants to relax only either a set of cells or a subset of $Aa = b$, and not any variable or constraint, a file can be used to tell the tool which units may be violated, through the control `-x x`, where `x` is the name of the file. The structure of the file should be like:

```
nr, number of constraints allowed to be violated (may be 0)
constraint number 1
...
constraint number nr
nx, number of cells allowed to be violated (may be 0)
cell number 1
...
cell number nx
ns, number of sensitive cells allowed to violate their protection levels (may be 0)
cell number 1
...
cell number ns
```

It should be taken into account that the numbers for the constraints have to be in the range from 0 to $m - 1$, both included, and the numbers for the cells have to be in the range from 0 to $n - 1$, both included. Besides, the `ns` cell numbers have to correspond to sensitive cells. The procedure stops if any inconsistency is found.

When a cell (sensitive or not) is included in the second section, its upper bound is relaxed, but not its lower bound. In turn, when a sensitive cell is included in the third section, the constraints:

$$\begin{aligned} upl_i\ y_i \leq z_i^+ &\leq u_{z_i}\ y_i \quad i \in \mathcal{P} \\ lpl_i(1-y_i) \leq z_i^- &\leq -l_{z_i}(1-y_i) \quad i \in \mathcal{P}, \end{aligned}$$

or, if negative protection levels are present (see section 4.2), the constraints:

$$\begin{aligned} z_i^+ - z_i^- &\geq upl_i y_i + l_{z_i}(1-y_i) \quad i \in \mathcal{P} \\ z_i^+ - z_i^- &\leq -lpl_i(1-y_i) + u_{z_i} y_i \quad i \in \mathcal{P} \end{aligned}$$

may be relaxed, which in practice means that both protection levels can be violated.

If there exists a solution for the relaxed problem, the program main_CTA writes in an output file information about the reached table, specifically the information related to infeasible constraints or cells. The name of the file is the name of the case, with the extension `'inf'`.

4.3.1 Example

The table shown in Figure 6(a), with upper bounds in Figure 6(b), cannot be protected with the current parameters. If the program is run without optional arguments we get the following output:

```
Problem reported as infeasible: optimization terminated (and not by time limit)
with no feasible CTA table
Total CPU time: 0.05
```

Instead, if the option `-r` is set to *yes* we obtain this:

```
Repair infeasibility procedure successfully finished, see information file.
Total CPU time: 0.1
```

And the output file contains the information related to still infeasible situations:

```
Constraints.
Const. num.     left-hand side  right-hand side
0 infeasibilities detected.

Cells.
0 infeasibilities detected among the variables.

Sensitive cells.
Cell 0 (25.996) under UPL (30)
```

meaning that all the linear relations stay equal to zero (as the table was additive), all the cells remain between bounds, and there is one sensitive cell that could not be set over its upper protection level of 30. Note that the value appearing in the file refers to the deviation from the initial cell value of 300, so the proposed value for the first cell would be 325.996 (rounded to 326 in Figure 6(c), for convenience), suggesting that if the protection level could be 26 instead of 30 the table would have been satisfactorily protected.

The use of the input file for restricted operation of the repair infeasibility tool can be demonstrated with the following example. Suppose that the linear relations and the variable bounds should kept unrelaxed, and only three sensitive cells (all of them, but the first one) are permitted to violate their protection levels. The input file, named `example_infeastool.repair`, is shown below:

Figure 6: (a) Original infeasible table, with primaries in boldface, lower protections levels as subscripts, and upper protection levels as superscripts; (b) upper bounds for cells, table margins are fixed; (c) Adjusted, nonsafe table after repair infeasibility procedure

$\mathbf{300^{30}_{40}}$	8	5	5	5	$\mathbf{38^{4}_{10}}$	361
8	$\mathbf{68^{6}_{10}}$	40	76	29	31	252
11	33	20	60	35	44	203
7	28	$\mathbf{36^{3}_{9}}$	41	22	63	197
326	137	101	182	91	176	1013

(a)

345	15	10	16	13	44
12	78	46	87	33	36
18	38	23	69	40	51
15	32	41	47	25	72

(b)

326	0	0	5	5	25	361
0	74	40	76	29	33	252
0	35	22	60	35	51	203
0	28	39	41	22	67	197
326	137	101	182	91	176	1013

(c)

```
0
0
3
5
8
23
```

The program is called with the following command:

```
main_CTA {path_instance}/example_infeastool.csp . -r y -x example_infeastool.repair
```

Then, the program shows the following message:

```
Repair infeasibility procedure reported relaxed problem is infeasible.
Total CPU time: 0.04
```

That is: if we don't allow to relax every variable and constraint of the model, we can still have an infeasible problem. In this case, the first sensitive cell has to be relaxed: if not, the remainder cells do not allow the first cell to accomplish with its protection levels.

5 Interface routines

This section describes the user's interface routines to the RCTA callable library. They are grouped by the type of manipulation performed to a table.

5.1 Creating and removing tables

- **Function:** `int CTA_create_table (TABLE **ptab, int ncells, int BLKSIZE, TYPE_CONSTRAINTS type_constraints)`

  Purpose: Allocates and initializes table of ncells.

  Returns: 0 if everything goes fine return CTA_OUT_OF_MEMORY if not enough memory.

  Input arguments: `ncells` is the number of cells; `BLKSIZE` is the block size for memory allocation increments; `type_constraints` is the type of constraints (ROWS, COLUMNS or BOTH).

  Output arguments: `*ptab` is a pointer to the newly created table.

  Input/Output arguments: None

  Example:
  ```
  TABLE *ptab=NULL;
  int ret_stat;
  int ncells = 400; //number of cells
  TYPE_CONSTRAINTS tc= COLUMNS;// {ROWS, COLUMNS, BOTH}
  const int BLKSIZE= 100; // block size for memory allocation increments

  ret_stat = CTA_create_table(&ptab,ncells,BLKSIZE,type_constraints);
  ```

- **Function:** `int CTA_create_table_from_file (TABLE **ptab, char *file, TYPE_CONSTRAINTS type_constraints)`

  Purpose: Creates table for CTA from file in csp format.

  Returns: returns 0 if everything goes fine.
  returns CTA_OUT_OF_MEMORY if not enough memory.
  returns CTA_FILE_NOT_FOUND if file not found.

  Input arguments: `file` is the name of the file in csp format; `type_constraints` is the type of constraints (ROWS,COLUMNS or BOTH).

  Output arguments: `*ptab` is a pointer to the newly created table from file.

  Input/Output arguments: None

  Example:
  ```
  TABLE *ptab=NULL;
  int ret_stat;
  TYPE_CONSTRAINTS tc= COLUMNS;// {ROWS, COLUMNS, BOTH}
  char *name= "targus.csp"
  ret_stat = CTA_create_table(&ptab,name,type_constraints);
  ```

- **Function:** `int CTA_delete_table (TABLE *tab)`

  Purpose: Deletes a non-empty table, freeing its memory space.

  Returns: returns 0 if everything goes fine.
  returns CTA_TABLE_NOT_EXISTS if table not exists.

Input arguments: None

Output arguments: None

Input/Output arguments: `tab` on input is a table (possibly empty); on output, is an empty table.

Example:

```
TABLE *tab;
...
CTA_delete_table(tab);
```

5.2 Entering table information

- **Function:** `void CTA_put_ncells (TABLE *tab, int ncells)`

  Purpose: Put number of cells (ncells) of the table.

  Returns: Nothing.

  Input arguments: `tab` is the table; `ncells` is the number of cells.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_ncells(tab,200); // number of cells is 200
  ```

- **Function:** `void CTA_put_npcells (TABLE *tab, int npcells)`

  Purpose: Put sensitive cells (npcells).

  Returns: Nothing.

  Input arguments: `tab` is the table; `ncells` is the number of sensitive cells.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_npcells(tab,50); // number of sensitive cells is 50
  ```

- **Function:** `void CTA_put_cellvalue (TABLE *tab, int pos, double value)`

  Purpose: Put cell value.

  Returns: Nothing.

  Input arguments: `tab` is the table; `pos` is the position of the cell; `value` is the value of the vector cells[pos].

  Output arguments: None.

Input/Output arguments: `tab` is the table to be updated.

Example:
```
TABLE *tab;
...
CTA_put_cellvalue(tab,2,40); // value of vector cells [2] = 40
```

- **Function:** `void CTA_put_cellperturbation_up (TABLE *tab, int pos, double perturbation)`

  Purpose: Put cell perturbation up value.

  Returns: Nothing.

  Input arguments: `tab` is the table; `pos` is the position of the cell; `perturbation` is the perturbation up of the vector cells[pos].

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_cellperturbation_up(tab,2,5); // perturbation up of vector cells [2]
  = 5
  ```

- **Function:** `void CTA_put_cellperturbation_down (TABLE *tab, int pos, double perturbation)`

  Purpose: Put cell perturbation down value.

  Returns: Nothing.

  Input arguments: `tab` is the table; `pos` is the position of the cell; `perturbation` is the perturbation down of the vector cells[pos].

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_cellperturbation_down(tab,2,5); // perturbation down of vector cells
  [2] = 5
  ```

- **Function:** `void CTA_put_cellweight (TABLE *tab, int pos, double weight)`

  Purpose: Put cell weight.

  Returns: Nothing.

  Input arguments: `tab` is the table; `pos` is the position of the cell; `weight` is the weight of the vector cells[pos].

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

Example:

```
TABLE *tab;
...
CTA_put_cellweight(tab,2,1); // weight of vector cells [2] = 1
```

- **Function:** `void CTA_put_lowbound (TABLE *tab, int pos, double lb`

  Purpose: Put cell lower bound.

  Returns: Nothing.

  Input arguments: `tab` is the table; `pos` is the position of the cell; `lb` is the lower bound of the vector cells[pos].

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_lowbound(tab,2,5); // lower bound of vector cells [2] = 5
  ```

- **Function:** `void CTA_put_upbound (TABLE *tab, int pos, double ub)`

  Purpose: Put cell upper bound.

  Returns: Nothing.

  Input arguments: `tab` is the table; `pos` is the position of the cell; `up` is the upper bound of the vector cells[pos].

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_upbound(tab,2,5); // upper bound of vector cells [2] = 5
  ```

- **Function:** `void CTA_put_modifupbound (TABLE *tab, int pos, double modif_ub)`

  Purpose: Put cell modified upper bound.

  Returns: Nothing.

  Input arguments: `tab` is the table; `pos` is the position of the cell; `modif_ub` is the modified upper bound of the vector cells[pos].

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_upmodifbound(tab,2,5); // modified upper bound of vector cells [2]
  = 5
  ```

- **Function:** `void CTA_put_index_sensitive_cell (TABLE *tab, int index, int pos)`

  Purpose: Put index in array of sensitives (0..npcells-1) of cell 'pos'.

  Returns: Nothing.

  Input arguments: `tab` is the table; `index` index in array of sensitives (0..npcells-1); `pos` is the position of the cell.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_index_sensitive_cell(tab,2,3);
  ```

- **Function:** `void CTA_put_info_sensitive_cell (TABLE *tab, int pos, int index, double plpl, double pupl)`

  Purpose: Put basic information sensitive cell:
  - position of this sensitive cell in array of cells
  - lower protection limit
  - upper protection limit.

  Returns: Nothing.

  Input arguments: `tab` is the table;`pos` position of this sensitive cell in array of sensitive cell; `index` position of this sensitive cell in array of cells; `plpl` is the lower protection limit; `pupl` is the upper protection limit.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_info_sensitive_cell(tab,35,3,5,5);
  ```

- **Function:** `void CTA_put_typetable (TABLE *tab, TYPE_TABLE t)`

  Purpose: Put type of table.

  Returns: Nothing.

  Input arguments: `tab` is the table; `t` type of table (GENERAL,K_DIM,HD).

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_typetable(tab,GENERAL); // Type of table=General.
  ```

- **Function:** `void CTA_put_K (TABLE *tab, int K).`

  Purpose: Put K (table dimension) if type_table=k-dim.

  Returns: Nothing.

  Input arguments: `tab` is the table; `K` table dimension.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_K(tab,2); // table dimension = 2 if type_table=k-dim.
  ```

- **Function:** `void CTA_put_typeconstraints (TABLE *tab, TYPE_CONSTRAINTS type_c)`

  Purpose: Put type of constraints (ROWS, COLUMNS, BOTH).

  Returns: Nothing.

  Input arguments: `tab` is the table;`type_c` is the type of constraints (ROWS, COLUMNS, BOTH).

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_typeconstraints(tab,ROWS); // type of constraints = ROWS.
  ```

- **Function:** `void CTA_put_nnz (TABLE *tab, int nnz)`

  Purpose: Put number of nonzeros in tad constraints.

  Returns: Nothing.

  Input arguments: `tab` is the table; `nnz` is the number of nonzeros in tad constraints.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_nnz(tab,10); // number of nonzeros = 10.
  ```

- **Function:** `void CTA_put_nconstraints (TABLE *tab, int nconstraints)`

  Purpose: Put number of constraints in tad constraints.

  Returns: Nothing.

  Input arguments: `tab` is the table;`nconstraints` number of constraints in tad constraints.

Output arguments: None.

Input/Output arguments: `tab` is the table to be updated.

Example:

```
TABLE *tab;
...
CTA_put_nconstraints(tab,10); // number of constraints = 10.
```

- **Function:** `void CTA_put_begconstraints (TABLE *tab, int i, int ctcoef, TYPE_CONSTRAINTS type_cons)`

  Purpose: Actualize pointer to begin of constraints coefficients rowwise/columnwise.

  Returns: Nothing.

  Input arguments: `tab` is the table; `i` position in vector begconst_row or begconst_col; `ctcoef` begin of constraints coefficients; `type_cons` type_of_constraints to actualize begconst_row or begconst_col.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_begconstraints(tab,1,1,ROWS);
  ```

- **Function:** `void CTA_put_begconstraints_rowwise (TABLE *tab, int i, int ctcoef)`

  Purpose: Actualize pointer to begin of constraints coefficients rowwise.

  Returns: Nothing.

  Input arguments: `tab` is the table; `i` position in vector begconst_row; `ctcoef` begin of constraints coefficients.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_begconstraints_rowwise(tab,1,1);
  ```

- **Function:** `void CTA_put_begconstraints_columnwise (TABLE *tab, int i, int ctcoef)`

  Purpose: Actualize pointer to begin of constraints coefficients columnwise.

  Returns: Nothing.

  Input arguments: `tab` is the table; `i` position in vector begconst_col; `ctcoef` begin of constraints coefficients.

  Output arguments: None.

Input/Output arguments: `tab` is the table to be updated.

Example:

```
TABLE *tab;
...
CTA_put_begconstraints_columnwise(tab,1,1);
```

- **Function:** `void CTA_put_coefconstraints (TABLE *tab, int i, double coef, TYPE_-CONSTRAINTS type_cons)`

  Purpose: Put coef value for all constraints (actualize) rowwise/columnwise.

  Returns: Nothing.

  Input arguments: `tab` is the table;`i` position in vector coef_row or coef_col; `coef` coef value; `type_cons` type of constraints (ROWS,COLUMNS)

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_coefconstraints(tab,1,10,ROWS);
  ```

- **Function:** `void CTA_put_coefconstraints_rowwise (TABLE *tab, int i, double coef)`

  Purpose: Put coef value for all constraints (actualize) rowwise.

  Returns: Nothing.

  Input arguments: `tab` is the table; `i` position in vector coef_row; `coef` coef value.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_coefconstraints_rowwise(tab,1,10);
  ```

- **Function:** `void CTA_put_coefconstraints_columnwise (TABLE *tab, int i, double coef)`

  Purpose: Put coef value for all constraints (actualize) columnwise.

  Returns: Nothing.

  Input arguments: `tab` is the table; `i` position in vector coef_col; `coef` coef value.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_coefconstraints_columnwise(tab,1,10);
  ```

- **Function:** `void CTA_put_xcoefconstraints (TABLE *tab, int i, int xcoef, TYPE_CONSTRAINTS type_cons)`

  Purpose: Put index of each coeficient (actualize) rowwise/columnwise.

  Returns: Nothing.

  Input arguments: `tab` is the table; `i` position in index coefficient vector (row/col); `xcoef` index coefficient; `type_cons` Type of constraints (ROWS, COLUMNS).

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_xcoefconstraints(tab,1,1,ROWS);
  ```

- **Function:** `void CTA_put_xcoefconstraints_rowwise (TABLE *tab, int i, int xcoef)`

  Purpose: Put index of each coeficient (actualize) rowwise.

  Returns: Nothing.

  Input arguments: `tab` is the table; `i` position in index coefficient vector (rows); `xcoef` index coefficient.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_xcoefconstraints_rowwise(tab,1,1);
  ```

- **Function:** `void CTA_put_xcoefconstraints_columnwise (TABLE *tab, int i, int xcoef)`

  Purpose: Put index of each coeficient (actualize) columnwise.

  Returns: Nothing.

  Input arguments: `tab` is the table; `i` position in index coefficient vector (cols); `xcoef` index coefficient.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_xcoefconstraints_columnwise(tab,1,1);
  ```

- **Function:** `void CTA_put_rhsconstraints (TABLE *tab, int i, double b)`

Purpose: Put right side of each constraint rowwise/columnwise.

Returns: Nothing.

Input arguments: `tab` is the table;`i` number constraint; `b` right side value of constraint.

Output arguments: None.

Input/Output arguments: `tab` is the table to be updated.

Example:

```
TABLE *tab;
...
CTA_put_rhsconstraints(tab,1,10); // in the first constraint b=10;
```

- **Function:** `void CTA_put_solver (TABLE *tab, SOLVER solver)`

Purpose: Put solver (CPLEX, XPRESS) in order to solve CTA problem.

Returns: Nothing.

Input arguments: `tab` is the table; `solver` solver (CPLEX, XPRESS).

Output arguments: None.

Input/Output arguments: `tab` is the table to be updated.

Example:

```
TABLE *tab;
...
CTA_put_solver(tab,CPLEX);
```

- **Function:** `void CTA_put_optim_gap (TABLE *tab, double optim_gap)`

Purpose: Put optim_gap to solve CTA problem.

Returns: Nothing.

Input arguments: `tab` is the table; `optim_gap` is the optim_gap choosen.

Output arguments: None.

Input/Output arguments: `tab` is the table to be updated.

Example:

```
TABLE *tab;
double optgap = 5.0; // a percentage.  To be divided by 100 ...
CTA_put_optim_gap(tab, optgap);
```

- **Function:** `void CTA_put_max_time (TABLE *tab, double max_time)`

Purpose: Put max_time to solve CTA problem.

Returns: Nothing.

Input arguments: `tab` is the table; `max_time` is the maxime time.

Output arguments: None.

Input/Output arguments: `tab` is the table to be updated.

Example:

```
TABLE *tab;
double maxT= 86400.0; // in seconds ...
CTA_put_max_time(tab, maxT);
```

- **Function:** `void CTA_put_preprocessSC (TABLE *tab, int ppsc)`

  Purpose: Put preprocess sensitive cells option.

  Returns: Nothing.

  Input arguments: `tab` is the table; `ppsc` preprocess sensitive cells option.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  int ppsc = 0; //default ...
  CTA_put_preprocessSC(tab, ppsc);
  ```

- **Function:** `void CTA_put_eprhs (TABLE *tab, double eprhs)`

  Purpose: Put parameter eprhs (feasibility tolerance).

  Returns: Nothing.

  Input arguments: `tab` is the table; `eprhs` is the feasibility tolerance.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  double eprhs= 1.0e-6; // small default feasibility tolerance ...
  CTA_put_eprhs(tab, eprhs);
  ```

- **Function:** `void CTA_put_epint (TABLE *tab, double epint)`

  Purpose: Put parameter epint (integrality tolerance).

  Returns: Nothing.

  Input arguments: `tab` is the table; `epint` is the integrality tolerance.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  double epint= -1; // -1 means default integrality tolerance of solver ...
  CTA_put_epint(tab, epint);
  ```

- **Function:** `void CTA_put_mipemphasis (TABLE *tab, MIPEMPHASIS mipemphasis)`

  Purpose: Put parameter mipemphasis (emphasis parameter of CPLEX MIP optimization).

  Returns: Nothing.

  Input arguments: `tab` is the table; `mipemphasis` is the emphasis parameter of CPLEX MIP optimization (MIPEMPHASIS_BALANCED, MIPEMPHASIS_FEASIBILITY, MIPEMPHASIS_OPTIMALITY, MIPEMPHASIS_BESTBOUND, MIPEMPHASIS_HIDDENFEAS).

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  MIPEMPHASIS mipemphasis= MIPEMPHASIS_BALANCED; //default ...
  CTA_put_mipemphasis(tab, mipemphasis);
  ```

- **Function:** `int CTA_put_heurmip (TABLE *tab, int h)`

  Purpose: Put parameter heurmip (heurdivespeedup parameter of XPRESS MIP optimization).

  Returns: 0 if h is -1, 0,1,2,3;
  otherwise -1, and heurmip is not set.

  Input arguments: `tab` is the table; `h` is the heurdivespeedup parameter of XPRESS MIP optimization.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  int heurmip= -1; // xpress emphasis; default is -1 ...
  CTA_put_mipemphasis(tab, heurmip);
  ```

- **Function:** `void CTA_put_varsel (TABLE *tab, VARSEL varsel)`

  Purpose: Put parameter varsel (variable selection parameter of CPLEX MIP optimization).

  Returns: Nothing.

  Input arguments: `tab` is the table; `varsel` is the variable selection parameter of CPLEX MIP optimization (VARSEL_MININFEAS, VARSEL_DEFAULT, VARSEL_MAXINFEAS, VARSEL_PSEUDO, VARSEL_STRONG, VARSEL_PSEUDOREDUCED) .

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  VARSEL varsel= VARSEL_DEFAULT; //default ...
  CTA_put_mipemphasis(tab, varsel);
  ```

- **Function:** `void CTA_put_objective_fun (TABLE *tab, double fobj)`

  Purpose: Put value of incumbent or final solution.

  Returns: Nothing.

  Input arguments: `tab` is the table; `fobj` is the value of incumbent or final solution .

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_objective_fun(tab, 100);
  ```

- **Function:** `void CTA_put_lowbnd_fobj (TABLE *tab, double lowbnd_fobj)`

  Purpose: Put value of lower bound of objective function.

  Returns: Nothing.

  Input arguments: `tab` is the table; `lowbnd_fobj` is the value of lower bound of objective function.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_lowbnd_fobj(tab, 80);
  ```

- **Function:** `void CTA_set_gap (TABLE *tab)`

  Purpose: Compute gap (in percentage) from objective function and its lower bound.

  Returns: Nothing.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_set_gap(tab);
  ```

- **Function:** `void CTA_put_BigM (TABLE *tab, double bigm)`

  Purpose: Put BigM of constraints $z^+ <= M * y, z^- <= M(1 - y)$.

  Returns: Nothing.

Input arguments: `tab` is the table; `bigm` is the BigM of constraints $z^+ <= M * y, z^- <= M(1 - y)$.

Output arguments: None.

Input/Output arguments: `tab` is the table to be updated.

Example:

```
TABLE *tab;
double bigm= 1.0e+120; // default is Infintity, so real bounds on deviations
will be used ...
CTA_put_BigM(tab,bigm);
```

- **Function:** `void CTA_put_final_status (TABLE *tab, SOLVER_STATUS s)`

  Purpose: Put final status after optimization.

  Returns: Nothing.

  Input arguments: `tab` is the table; `s` is the final status after optimization.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

```
TABLE *tab;
...
CTA_put_final_status(tab,CTA_OPTIMAL_SOLUTION); //find a optimal solution.
```

- **Function:** `int CTA_put_logfile_solver (TABLE *tab, const char *logfile)`

  Purpose: Put name of file with log of solver; if logfile is NULL no output is printed (neither by file nor to screen).

  Returns: 0 if successful, or CTA_OUT_OF_MEMORY if no free space for copying the name.

  Input arguments: `tab` is the table; `logfile` is the name of file with log of solver.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

```
TABLE *tab;
...
CTA_put_logfile_solver(tab,"log_file"); //a file "log_file" with log of solver
is created.
```

- **Function:** `int CTA_put_instance_name (TABLE *tab, const char *instname)`

  Purpose: Put name of instance.

  Returns: 0 if successful, or CTA_OUT_OF_MEMORY if no free space for copying the name.

  Input arguments: `tab` is the table; `instname` is the name of file with the table to protect.

  Output arguments: None.

Input/Output arguments: `tab` is the table to be updated.

Example:

```
TABLE *tab;
...
CTA_put_instance_name(tab,"table2D"); //a file "table2D" with any table is
opened in order to protect it.
```

- **Function:** `void CTA_put_firstfeas (TABLE *tab, bool ff)`

  Purpose: Put boolean first_feasible.

  Returns: Nothing.

  Input arguments: `tab` is the table; `ff` is the boolean first feasible.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_firstfeas(tab,TRUE); //first_feasible=TRUE;
  ```

- **Function:** `void CTA_put_make_additive (TABLE *tab, bool madd)`

  Purpose: Put value of make_additive for making additive nonadditive tables if requested.

  Returns: Nothing.

  Input arguments: `tab` is the table; `madd` is the boolean make_additive.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_put_make_additive(tab,TRUE); //make_additive=TRUE;
  ```

- **Function:** `void CTA_put_opt_model (TABLE *tab, OPTMODEL opt_mod)`

  Purpose: Put value of optimization model to be used.

  Returns: Nothing.

  Input arguments: `tab` is the table; `opt_mod` is the optimization model to be used (NEW, CLASSICAL, AUTOMATIC).

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

Example:
```
TABLE *tab;
OPTMODEL opt_mod=CLASSICAL
...
CTA_put_opt_model(tab,opt_mod);
```

- **Function:** `void CTA_put_repair_infeas (TABLE *tab, bool repair_infeas)`

  Purpose: Put value of repair_infeas for applying repeair infeasibility if requested.

  Returns: Nothing.

  Input arguments: `tab` is the table; `repair_infeas` is the boolean for applying repair infeasibility.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_repair_infeas(tab,TRUE); //repair_infeas=TRUE;
  ```

- **Function:** `int CTA_put_repair_inputfile (TABLE *tab, const char *repfile)`

  Purpose: Put name of file with input information for repair infeas tool.

  Returns: 0 if successful, or CTA_OUT_OF_MEMORY if no free space for copying the name.

  Input arguments: `tab` is the table; `repfile` is the name of file with input information for repair.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  CTA_put_repair_inputfile(tab,"repfile"); //input information for repair infesibility
  tool will be read from file "repfile".
  ```

5.3 Retrieving table information

- **Function:** `int CTA_get_ncells (TABLE *tab)`

  Purpose: Get number of cells (ncells) of the table.

  Returns: The number of cells.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

Example:
```
TABLE *tab;
...
int ncells=CTA_get_ncells(tab); // number of cells.
```

- **Function:** `int CTA_get_npcells (TABLE *tab)`

  Purpose: Get sensitive cells (npcells).

  Returns: Number of sensitive cells.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:
  ```
  TABLE *tab;
  ...
  int npcells = CTA_get_npcells(tab); // number of sensitive cells.
  ```

- **Function:** `double CTA_get_cellvalue (TABLE *tab, int pos)`

  Purpose: Get cell value.

  Returns: Value of the vector cells[pos].

  Input arguments: `tab` is the table; `pos` is the position of the cell.

  Output arguments: None.

  Input/Output arguments: None.

  Example:
  ```
  TABLE *tab;
  ...
  double cellvalue=CTA_put_cellvalue(tab,2); // value of vector cells [2]
  ```

- **Function:** `double CTA_get_cellperturbation_up (TABLE *tab, int pos)`

  Purpose: Get cell perturbation up value.

  Returns: The perturbation up of the vector cells[pos].

  Input arguments: `tab` is the table; `pos` is the position of the cell.

  Output arguments: None.

  Input/Output arguments: None.

  Example:
  ```
  TABLE *tab;
  ...
  double perturbation = CTA_get_cellperturbation_up(tab,2); // perturbation up
  of vector cells [2]
  ```

- **Function:** `double CTA_get_cellperturbation_down (TABLE *tab, int pos)`

  Purpose: Get cell perturbation down value.

  Returns: The perturbation down of the cell.

  Input arguments: `tab` is the table; `pos` is the position of the cell.

  Output arguments: None.

  Input/Output arguments: None.

  Example:
  ```
  TABLE *tab;
  ...
  double perturbation = CTA_put_cellperturbation_down(tab,2); // perturbation
  down of vector cells[2]
  ```

- **Function:** `double CTA_get_cellweight (TABLE *tab, int pos)`

  Purpose: Get cell weight.

  Returns: The weight of the cell.

  Input arguments: `tab` is the table; `pos` is the position of the cell.

  Output arguments: None.

  Input/Output arguments: None.

  Example:
  ```
  TABLE *tab;
  ...
  double weight = CTA_get_cellweight(tab,2); // weight of vector cells [2]
  ```

- **Function:** `double CTA_get_lowbound (TABLE *tab, int pos)`

  Purpose: Get cell lower bound.

  Returns: The lower bound of the cell.

  Input arguments: `tab` is the table; `pos` is the position of the cell.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  double lb = CTA_get_lowbound(tab,2); // lower bound of vector cells [2]
  ```

- **Function:** `double CTA_get_upbound (TABLE *tab, int pos)`

  Purpose: Get cell upper bound.

  Returns: The upper bound of the cell.

  Input arguments: `tab` is the table; `pos` is the position of the cell.

Output arguments: None.

Input/Output arguments: None.

Example:

```
TABLE *tab;
...
double ub = CTA_get_upbound(tab,2); // upper bound of vector cells [2];
```

- **Function:** `double CTA_get_modifupbound (TABLE *tab, int pos)`

  Purpose: Get cell modified upper bound.

  Returns: The modified upper bound.

  Input arguments: `tab` is the table; `pos` is the position of the cell.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  double mub = CTA_get_upmodifbound(tab,2,5); // modified upper bound of vector
  cells [2]
  ```

- **Function:** `int CTA_get_index_sensitive_cell (TABLE *tab,int pos)`

  Purpose: Get index sensitive cell.

  Returns: Index in array of sensitives (0..npcells-1) of cell 'pos'.

  Input arguments: `tab` is the table; `pos` is the position of the cell.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  int index = CTA_get_index_sensitive_cell(tab,2);
  ```

- **Function:** `int CTA_get_index_cell (TABLE *tab, int pos)`

  Purpose: Get index cell.

  Returns: index (0..ncells-1) of sensitive cell 'pos'.

  Input arguments: `tab` is the table; `pos` is the position of the sensitive cell.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  int index = CTA_get_index_cell(tab,2);
  ```

- **Function:** `TYPE_TABLE CTA_get_typetable (TABLE *tab, TYPE_TABLE t)`

  Purpose: Get type of table (GENERAL,K_DIM,HD).

  Returns: The type of table ().

  Input arguments: `tab` is the table; `t` type of table (GENERAL,K_DIM,HD).

  Output arguments: None.

  Input/Output arguments: None.

  Example:
  ```
  TABLE *tab;
  ...
  TYPE_TABLE t = CTA_get_typetable(tab); // Return type of table.
  ```

- **Function:** `int CTA_get_K (TABLE *tab, int K)`

  Purpose: Get K (table dimension) if type_table=k-dim.

  Returns: Table dimension.

  Input arguments: `tab` is the table;

  Output arguments: None.

  Input/Output arguments: None.

  Example:
  ```
  TABLE *tab;
  ...
  int K = CTA_get_K(tab,2);
  ```

- **Function:** `TYPE_CONSTRAINTS CTA_get_typeconstraints (TABLE *tab)`

  Purpose: Get type of constraints (ROWS, COLUMNS, BOTH).

  Returns: The type of the constraints (ROWS, COLUMNS, BOTH).

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:
  ```
  TABLE *tab;
  ...
  TYPE_CONSTRAINTS tc = CTA_get_typeconstraints(tab); // type of constraints.
  ```

- **Function:** `int CTA_get_nnz (TABLE *tab)`

  Purpose: Get number of nonzeros in tad constraints.

  Returns: Nothing.

  Input arguments: `tab` is the table.

Output arguments: None.

Input/Output arguments: None.

Example:

```
TABLE *tab;
...
int nnz = CTA_get_nnz(tab,10); // number of nonzeros.
```

- **Function:** `int CTA_get_nconstraints (TABLE *tab )`

  Purpose: Get number of constraints in tad constraints.

  Returns: The number of constraints.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  int nconstraints = CTA_get_nconstraints(tab,10); // number of constraints.
  ```

- **Function:** `int CTA_get_begconstraints (TABLE *tab, int i, TYPE_CONSTRAINTS type_cons)`

  Purpose: Get pointer to begin of constraints coefficients rowwise/columnwise.

  Returns: The pointer to begin of constraints coefficients rowwise/columnwise.

  Input arguments: `tab` is the table; `i` number of the constraint; `type_cons` type_of_constraints to check begconst_row or begconst_col.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  int begconst = CTA_get_begconstraints(tab,1);
  ```

- **Function:** `int CTA_get_begconstraints_rowwise (TABLE *tab, int i)`

  Purpose: Get pointer to begin of constraints coefficients rowwise.

  Returns: The pointer to begin of constraints coefficients rowwise.

  Input arguments: `tab` is the table; `i` number of the constraint.

  Output arguments: None.

  Input/Output arguments: None.

Example:

```
TABLE *tab;
...
int begconst = CTA_get_begconstraints_rowwise(tab,1);
```

- **Function:** `int CTA_get_begconstraints_columnwise (TABLE *tab, int i)`

  Purpose: Get pointer to begin of constraints coefficients columnwise.

  Returns: The pointer to begin of constraints coefficients columnwise.

  Input arguments: `tab` is the table; `i` number of the constraint.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  CTA_get_begconstraints_columnwise(tab,1);
  ```

- **Function:** `double CTA_get_coefconstraints (TABLE *tab, int i)`

  Purpose: Get coef constraints in cell 'i'.

  Returns: The coef constraints in cell 'i'.

  Input arguments: `tab` is the table; `i` position in vector coef_row or coef_col.)

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  double coef = CTA_get_coefconstraints(tab,1);
  ```

- **Function:** `double CTA_get_coefconstraints_rowwise (TABLE *tab, int i)`

  Purpose: Get the coef constraints row in cell i.

  Returns: The coef constraints row in cell i.

  Input arguments: `tab` is the table; `i` position in vector coef_row.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  double coef = CTA_get_coefconstraints_rowwise(tab,1);
  ```

- **Function:** `double CTA_get_coefconstraints_columnwise (TABLE *tab, int i)`

  Purpose: Get the coef constraints column in cell i.

  Returns: The coef constraints column in cell i.

  Input arguments: `tab` is the table; `i` position in vector coef_col.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  double coef = CTA_get_coefconstraints_columnwise(tab,1);
  ```

- **Function:** `int CTA_get_xcoefconstraints (TABLE *tab, int i, TYPE_CONSTRAINTS type_cons)`

  Purpose: Get index of each coefficient (rowwise/columnwise).

  Returns: The xcoef constraints in cell 'i'.

  Input arguments: `tab` is the table; `i` position in index coefficient vector (row/col); `type_cons` Type of constraints (ROWS, COLUMNS).

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  int xcoef = CTA_get_xcoefconstraints(tab,1,ROWS);
  ```

- **Function:** `int CTA_get_xcoefconstraints_rowwise (TABLE *tab, int i)`

  Purpose: Get index of each coeficient rowwise.

  Returns: The xcoef constraints in cell 'i'.

  Input arguments: `tab` is the table; `i` position in index coefficient vector (rows).

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  int xcoef = CTA_get_xcoefconstraints_rowwise(tab,1);
  ```

- **Function:** `int CTA_get_xcoefconstraints_columnwise (TABLE *tab, int i)`

  Purpose: Get index of each coefficient columnwise.

  Returns: The xcoef constraints in cell 'i'.

Input arguments: `tab` is the table;`i` position in index coefficient vector (cols).

Output arguments: None.

Input/Output arguments: None.

Example:

```
TABLE *tab;
...
int xcoef = CTA_get_xcoefconstraints_columnwise(tab,1);
```

- **Function:** `double CTA_get_rhsconstraints (TABLE *tab, int i)`

  Purpose: Get right side of constraints.

  Returns: Right side of constraint 'i'.

  Input arguments: `tab` is the table;`i` number of constraint.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  double rhs = CTA_get_rhsconstraints(tab,1);
  ```

- **Function:** `SOLVER CTA_get_solver (TABLE *tab)`

  Purpose: Get solver (CPLEX, XPRESS) in order to solve CTA problem.

  Returns: The solver (CPLEX,XPRESS) to solve CTA problem.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  SOLVER solver = CTA_get_solver(tab);
  ```

- **Function:** `double CTA_get_optim_gap (TABLE *tab)`

  Purpose: Get optim_gap to solve CTA problem.

  Returns: The optim_gap to solve CTA problem.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

Example:

```
TABLE *tab;
...
double optgap = CTA_get_optim_gap(tab);
```

- **Function:** `double CTA_get_max_time (TABLE *tab)`

  Purpose: Get max_time to solve CTA problem.

  Returns: The max_time to solve CTA problem.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  double maxT = CTA_put_max_time(tab);
  ```

- **Function:** `int CTA_get_preprocessSC (TABLE *tab)`

  Purpose: Get preprocess sensitive cells option.

  Returns: The preprocess sensitive cells option.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  int ppsc = CTA_get_preprocessSC(tab);
  ```

- **Function:** `double CTA_put_eprhs (TABLE *tab)`

  Purpose: Get parameter eprhs (feasibility tolerance).

  Returns: The parameter eprhs (feasibility tolerance).

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  double eprhs = CTA_get_eprhs(tab, eprhs);
  ```

- **Function:** `double CTA_get_epint (TABLE *tab)`

  Purpose: Get parameter epint (integrality tolerance).

  Returns: The integrality tolerance (epint).

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:
  ```
  TABLE *tab;
  ...
  double epint = CTA_get_epint(tab);
  ```

- **Function:** `MIPEMPHASIS CTA_get_mipemphasis (TABLE *tab)`

  Purpose: Get parameter mipemphasis (emphasis parameter of CPLEX MIP optimization).

  Returns: The parameter mipemphasis (MIPEMPHASIS_BALANCED, MIPEMPHASIS_FEASIBILITY, MIPEMPHASIS_OPTIMALITY, MIPEMPHASIS_BESTBOUND, MIPEMPHASIS_HIDDENFEAS).

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:
  ```
  TABLE *tab;
  ...
  MIPEMPHASIS mipemphasis = CTA_get_mipemphasis(tab);
  ```

- **Function:** `int CTA_get_heurmip (TABLE *tab)`

  Purpose: Get parameter heurmip (heurdivespeedup parameter of XPRESS MIP optimization).

  Returns: Return parameter heurmip.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:
  ```
  TABLE *tab;
  ...
  int heurmip = CTA_get_mipemphasis(tab);
  ```

- **Function:** `VARSEL CTA_get_varsel (TABLE *tab)`

  Purpose: Get parameter varsel (variable selection parameter of CPLEX MIP optimization).

Returns: The parameter varsel (VARSEL_MININFEAS, VARSEL_DEFAULT, VARSEL_MAXINFEAS, VARSEL_PSEUDO, VARSEL_STRONG, VARSEL_PSEUDOREDUCED).

Input arguments: `tab` is the table.

Output arguments: None.

Input/Output arguments: None.

Example:

```
TABLE *tab;
...
VARSEL varsel = CTA_get_mipemphasis(tab);
```

- **Function:** `double CTA_get_objective_fun (TABLE *tab)`

  Purpose: Get value of incumbent or final solution.

  Returns: The value of incumbent or final solution.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  in fun = CTA_get_objective_fun(tab);
  ```

- **Function:** `double CTA_get_lowbnd_fobj (TABLE *tab)`

  Purpose: Get value of lower bound of objective function.

  Returns: The value of lower bound of objective function.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: `tab` is the table to be updated.

  Example:

  ```
  TABLE *tab;
  ...
  double lowbnd_fobj = CTA_get_lowbnd_fobj(tab);
  ```

- **Function:** `double CTA_get_gap (TABLE *tab)`

  Purpose: Get gap (in percentage) from objective function and its lower bound.

  Returns: Return gap from objective function and its lower bound.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

Example:

```
TABLE *tab;
...
double gap = CTA_get_gap(tab);
```

- **Function:** `double CTA_get_BigM (TABLE *tab)`

  Purpose: Get BigM of constraints $z^+ <= M * y, z^- <= M(1 - y)$.

  Returns: The value of BigM.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  double bigm= 1.0e+120; // default is Infintity, so real bounds on deviations
  will be used ...
  double bigm = CTA_get_BigM(tab);
  ```

- **Function:** `SOLVER_STATUS CTA_get_final_status (TABLE *tab)`

  Purpose: Get final status after optimization. The possible values are listed in Table 1 of page 12.

  Returns: The final status after optimization.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  SOLVER_STATUS s = CTA_get_final_status(tab);
  ```

- **Function:** `char* CTA_get_logfile_solver (TABLE *tab)`

  Purpose: Get name of file with log of solver.

  Returns: The name of file with log of solver.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  char* logfile = CTA_get_logfile_solver(tab).
  ```

- **Function:** `char* CTA_get_instance_name (TABLE *tab)`

  Purpose: Get name of instance with a table to protect.

  Returns: The name of the instance.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  char* instance = CTA_get_instance_name(tab).
  ```

- **Function:** `bool CTA_get_firstfeas (TABLE *tab)`

  Purpose: Get boolean first_feasible.

  Returns: The boolean first_feasible.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  bool first_feasible = CTA_get_firstfeas(tab);
  ```

- **Function:** `bool CTA_get_make_additive (TABLE *tab)`

  Purpose: Get value of make_additive for making additive nonadditive tables if requested.

  Returns: The boolean make_additive.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  bool make_additive = CTA_get_make_additive(tab);
  ```

- **Function:** `OPTMODEL CTA_get_opt_model (TABLE *tab)`

  Purpose: Get value of optimization model to be used.

  Returns: The value of optimization model(NEW, CLASSICAL, AUTOMATIC).

  Input arguments: `tab` is the table.

Output arguments: None.

Input/Output arguments: None.

Example:

```
TABLE *tab;
...
OPTMODEL opt_model = CTA_get_opt_model(tab);
```

- **Function:** `bool CTA_get_repair_infeas (TABLE *tab)`

  Purpose: Get value of repair_infeas for applying repair infeasibility procedure if requested.

  Returns: The boolean repair_infeas.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  bool repair_infeas = CTA_get_repair_infeas(tab);
  ```

- **Function:** `char* CTA_get_repair_inputfile (TABLE *tab)`

  Purpose: Get name of file with input information.

  Returns: The name of file with input information.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

  ```
  TABLE *tab;
  ...
  char* repair_inputfile = CTA_get_repair_inputfile(tab).
  ```

5.4 Solving CTA

- **Function:** `void CTA_Find_Solution (TABLE *tab)`

  Purpose: Solves CTA problem.

  Returns: exit conditions, see cta_table.h.

  Input arguments: `tab`: tab: table to be protected; on exit, optimal protections stored in tab.

  Output arguments: None.

  Input/Output arguments: `tab`: is the table to be updated with optimal protections stored in tab.

  Example:

  ```
  TABLE *tab;
  ...
  ret_stat= CTA_Find_Solution(tab);
  ```

References

[1] J. Castro, Minimum-distance controlled perturbation methods for large-scale tabular data protection, *European Journal of Operational Research*, 171 (2006) 39-52.

[2] J. Castro and S. Giessing, Testing variants of minimum distance controlled tabular adjustment, in Monographs of Official Statistics. Work session on Statistical Data Confidentiality, Eurostat-Office for Official Publications of the European Communities, Luxembourg, 2006, 333-343. ISBN 92-79-01108-1.

[3] Dash Optimization, *XPRESS Optimizer Reference Manual*, DASH, (2007).

[4] ILOG CPLEX, *ILOG CPLEX 11.0 Reference Manual*, ILOG, (2007).

[5] A. Hundepool, A. van de Wetering, R. Ramaswamy, P.P de Wolf, S. Giessing, M. Fischetti, J.J. Salazar, J. Castro and P. Lowthian, *τ-ARGUS Users's Manual, version 3.0*, (2004).

[6] S. Giessing, A. Hundepool and J. Castro, Rounding methods for protecting EU-aggregates , Joint UNECE/Eurostat Work Session on Statistical Data Confidentiality, Manchester (United Kingdom), December 2007.

APPENDIX

The information of this appendix was generated from the code, which can be object of future revisions. It can then present some inaccuracies or be out of date. Look at the code for full details. The location of files and routines corresponds to the MS-Windows distribution of the package.

A Global information

Package Name	RCTA Package
Package Owner	Dept. of Statistics and Operations Research Universitat Politècnica de Catalunya Barcelona
Contact Person	Jordi Castro, `jordi.castro@upc.edu`
Starting Date	January 2008
Ending Date	April 2010
Programming Environment	Linux and gcc, ported to Windows and MS-Visual C++

B List of files (alphabetical order)

	File Name	Location	Lines	Bytes
1	cta_model.h	C_CTA\libCTA_bc\src\	32	843
2	cta_solve.cpp	C_CTA\libCTA_bc\src\	480	13451
3	cta_solve_cplex.cpp	C_CTA\libCTA_bc\src\	813	27252
4	cta_solve_cplex.h	C_CTA\libCTA_bc\src\	25	695
5	cta_solve_xpress.cpp	C_CTA\libCTA_bc\src\	1107	37937
6	cta_solve_xpress.h	C_CTA\libCTA_bc\src\	25	709
7	cta_table.cpp	C_CTA\libCTA_bc\src\	1058	31531
8	cta_table.h	C_CTA\libCTA_bc\src\	899	27418
9	**TOTAL**		4439	139836

C List of routines

1. void **CTA_Dump_X_table** (**TABLE** ∗tab, double ∗**x**)
2. int **CTA_lowering_ub** (**TABLE** ∗tab)
3. int **CTA_Close_Solver** (**TABLE** ∗tab, MODEL ∗mod)
4. int **CTA_Clear_Model** (**TABLE** ∗tab, MODEL ∗mod)
5. int **CTA_Preprocess** (**TABLE** ∗tab, MODEL ∗mod)
6. int **CTA_Init_Solver** (**TABLE** ∗tab, MODEL ∗mod)
7. int **CTA_Load_Model** (**TABLE** ∗tab, MODEL ∗mod)
8. int **CTA_Run_Solver** (**TABLE** ∗tab, MODEL ∗mod, int restart)
9. int **CTA_Find_Solution** (**TABLE** ∗tab)
10. void **CTA_show_cpx_status** (int status)
11. int **CTA_Solution_cpx** (MODEL ∗mod, double ∗∗X)
12. int **CTA_Close_Solver_cpx** (MODEL ∗mod)
13. int **CTA_Clear_Model_cpx** (MODEL ∗mod)
14. int **CTA_Init_Solver_cpx** (**TABLE** ∗tab, MODEL ∗mod)
15. int **CTA_Load_original_matrix_cpx** (**TABLE** ∗tab, MODEL ∗mod, double ∗∗MatV, int ∗∗MatI)
16. int **CTA_Load_Model_cpx** (**TABLE** ∗tab, MODEL ∗mod)
17. int **CTA_Run_Solver_cpx** (**TABLE** ∗tab, MODEL ∗mod, int restart)
18. int CPXPUBLIC **infocallback** (CPXCENVptr **env**, void ∗cbdata, int wherefrom, void ∗cbhandle)
19. void **CTA_show_xprs_errormsg** (const char ∗sSubName, int nLineNo, int nErrCode)
20. int **CTA_Solution_xprs** (MODEL ∗mod, double ∗∗X)
21. int **CTA_Close_Solver_xprs** (MODEL ∗mod)
22. int **CTA_Clear_Model_xprs** (MODEL ∗mod)
23. int **CTA_Init_Solver_xprs** (**TABLE** ∗tab, MODEL ∗mod)
24. int **CTA_Load_original_matrix_xprs** (**TABLE** ∗tab, MODEL ∗mod, double ∗∗MatV, int ∗∗MatI)
25. int **CTA_Load_Model_xprs** (**TABLE** ∗tab, MODEL ∗mod)
26. int **CTA_Run_Solver_xprs** (**TABLE** ∗tab, MODEL ∗mod, int restart)
27. int **CTA_reallocate** (**TABLE** ∗tab, int size)

28. int **CTA_ijkl2n** (vector< int > &ijkl, vector< int > &**d**, int **ndim**)
29. void **CTA_n2ijkl** (int n, vector< int > &ijkl, vector< int > &**d**, int **ndim**)
30. int **CTA_allocate_struct_TABLE** (**TABLE** ∗∗ptab, int BLKSIZE)
31. int **CTA_allocate_struct_CELL** (**TABLE** ∗∗ptab, int BLKSIZE)
32. int **CTA_allocate_struct_SENSITIVECELL** (**TABLE** ∗∗ptab, int BLKSIZE)
33. int **CTA_allocate_struct_CONSTRAINTS** (**TABLE** ∗∗ptab, int BLKSIZE)
34. int **CTA_allocate_struct_B** (**TABLE** ∗∗ptab, int BLKSIZE)
35. int **CTA_allocate_struct_BEGCONSTROW** (**TABLE** ∗∗ptab, int BLKSIZE)
36. int **CTA_allocate_struct_COEFROW** (**TABLE** ∗∗ptab, int BLKSIZE)
37. int **CTA_allocate_struct_XCOEFROW** (**TABLE** ∗∗ptab, int BLKSIZE)
38. int **CTA_allocate_struct_BEGCONSTCOL** (**TABLE** ∗∗ptab, int BLKSIZE)
39. int **CTA_allocate_struct_COEFCOL** (**TABLE** ∗∗ptab, int BLKSIZE)
40. int **CTA_allocate_struct_XCOEFCOL** (**TABLE** ∗∗ptab, int BLKSIZE)
41. int **CTA_create_table** (**TABLE** ∗∗ptab, int **ncells**, int BLKSIZE, **TYPE_CONSTRAINTS type_constraints**)
42. int **CTA_delete_table** (**TABLE** ∗tab)
43. int **CTA_delete_constraints** (**TABLE** ∗tab, int type)
44. int **CTA_create_table_from_file** (**TABLE** ∗∗ptab, char ∗file, **TYPE_CONSTRAINTS type_constraints**)
45. int **CTA_generate_columnwise_matrix** (**TABLE** ∗∗ptab)
46. int **CTA_check_relations_table** (**TABLE** ∗tab, **TYPE_VALUES** val, double reltol, int outlevel)
47. int **CTA_check_protections** (**TABLE** ∗tab, double reltol, int outlevel)
48. int **CTA_check_bounds** (**TABLE** ∗tab, double reltol, int outlevel)
49. int **CTA_check_perturbations** (**TABLE** ∗tab, double abstol, int outlevel)
50. double **CTA_get_cellvalue** (**TABLE** ∗tab, int pos)
51. double **CTA_get_cellperturbation_up** (**TABLE** ∗tab, int pos)
52. double **CTA_get_cellperturbation_down** (**TABLE** ∗tab, int pos)
53. double **CTA_get_lowbound** (**TABLE** ∗tab, int pos)
54. double **CTA_get_upbound** (**TABLE** ∗tab, int pos)
55. double **CTA_get_modifupbound** (**TABLE** ∗tab, int pos)
56. double **CTA_get_weight** (**TABLE** ∗tab, int pos)

57. int **CTA_get_ncells** (**TABLE** *tab)

58. int **CTA_get_npcells** (**TABLE** *tab)

59. int **CTA_get_nconstraints** (**TABLE** *tab)

60. int **CTA_get_nnz** (**TABLE** *tab)

61. int **CTA_get_index_sensitive_cell** (**TABLE** *tab, int pos)

62. int **CTA_get_index_cell** (**TABLE** *tab, int pos)

63. double **CTA_get_plpl** (**TABLE** *tab, int pos)

64. double **CTA_get_pupl** (**TABLE** *tab, int pos)

65. int **CTA_get_begconstraints** (**TABLE** *tab, int i, **TYPE_CONSTRAINTS** type_cons)

66. int **CTA_get_begconstraints_rowwise** (**TABLE** *tab, int i)

67. int **CTA_get_begconstraints_columnwise** (**TABLE** *tab, int i)

68. double **CTA_get_coefconstraints** (**TABLE** *tab, int i, **TYPE_CONSTRAINTS** type_cons)

69. double **CTA_get_coefconstraints_rowwise** (**TABLE** *tab, int i)

70. double **CTA_get_coefconstraints_columnwise** (**TABLE** *tab, int i)

71. int **CTA_get_xcoefconstraints** (**TABLE** *tab, int i, **TYPE_CONSTRAINTS** type_cons)

72. int **CTA_get_xcoefconstraints_rowwise** (**TABLE** *tab, int i)

73. int **CTA_get_xcoefconstraints_columnwise** (**TABLE** *tab, int i)

74. double **CTA_get_rhsconstraints** (**TABLE** *tab, int i)

75. **TYPE_CONSTRAINTS CTA_get_typeconstraints** (**TABLE** *tab)

76. **SOLVER CTA_get_solver** (**TABLE** *tab)

77. double **CTA_get_optim_gap** (**TABLE** *tab)

78. double **CTA_get_max_time** (**TABLE** *tab)

79. int **CTA_get_preprocessSC** (**TABLE** *tab)

80. double **CTA_get_eprhs** (**TABLE** *tab)

81. double **CTA_get_epint** (**TABLE** *tab)

82. **MIPEMPHASIS CTA_get_mipemphasis** (**TABLE** *tab)

83. int **CTA_get_heurmip** (**TABLE** *tab)

84. **VARSEL CTA_get_varsel** (**TABLE** *tab)

85. double **CTA_get_objective_fun** (**TABLE** *tab)

86. double **CTA_get_lowbnd_fobj** (**TABLE** ∗tab)

87. double **CTA_get_gap** (**TABLE** ∗tab)

88. double **CTA_get_BigM** (**TABLE** ∗tab)

89. **SOLVER_STATUS CTA_get_final_status** (**TABLE** ∗tab)

90. char ∗ **CTA_get_logfile_solver** (**TABLE** ∗tab)

91. char ∗ **CTA_get_instance_name** (**TABLE** ∗tab)

92. bool **CTA_get_firstfeas** (**TABLE** ∗tab)

93. void **CTA_put_ncells** (**TABLE** ∗tab, int **ncells**)

94. void **CTA_put_npcells** (**TABLE** ∗tab, int **npcells**)

95. void **CTA_put_cellvalue** (**TABLE** ∗tab, int pos, double **value**)

96. void **CTA_put_cellperturbation_up** (**TABLE** ∗tab, int pos, double perturbation)

97. void **CTA_put_cellperturbation_down** (**TABLE** ∗tab, int pos, double perturbation)

98. void **CTA_put_cellweight** (**TABLE** ∗tab, int pos, double **weight**)

99. void **CTA_put_lowbound** (**TABLE** ∗tab, int pos, double **lb**)

100. void **CTA_put_upbound** (**TABLE** ∗tab, int pos, double **ub**)

101. void **CTA_put_modifupbound** (**TABLE** ∗tab, int pos, double **modif_ub**)

102. void **CTA_put_index_sensitive_cell** (**TABLE** ∗tab, int **index**, int pos)

103. void **CTA_put_info_sensitive_cell** (**TABLE** ∗tab, int pos, int **index**, double **plpl**, double **pupl**)

104. void **CTA_put_typetable** (**TABLE** ∗tab, **TYPE_TABLE t**)

105. void **CTA_put_K** (**TABLE** ∗tab, int K)

106. void **CTA_put_typeconstraints** (**TABLE** ∗tab, **TYPE_CONSTRAINTS** type_c)

107. void **CTA_put_nnz** (**TABLE** ∗tab, int **nnz**)

108. void **CTA_put_nconstraints** (**TABLE** ∗tab, int nconstraints)

109. void **CTA_put_begconstraints** (**TABLE** ∗tab, int i, int ctcoef, **TYPE_CONSTRAINTS** type_cons)

110. void **CTA_put_begconstraints_rowwise** (**TABLE** ∗tab, int i, int ctcoef)

111. void **CTA_put_begconstraints_columnwise** (**TABLE** ∗tab, int i, int ctcoef)

112. void **CTA_put_coefconstraints** (**TABLE** ∗tab, int i, double coef, **TYPE_CONSTRAINTS** type_cons)

113. void **CTA_put_coefconstraints_rowwise** (**TABLE** ∗tab, int i, double coef)

114. void **CTA_put_coefconstraints_columnwise** (**TABLE** ∗tab, int i, double coef)

115. void **CTA_put_xcoefconstraints** (**TABLE** ∗tab, int i, int xcoef, **TYPE_CONSTRAINTS** type_cons)

116. void **CTA_put_xcoefconstraints_rowwise** (**TABLE** ∗tab, int i, int xcoef)

117. void **CTA_put_xcoefconstraints_columnwise** (**TABLE** ∗tab, int i, int xcoef)

118. void **CTA_put_rhsconstraints** (**TABLE** ∗tab, int i, double **b**)

119. void **CTA_put_solver** (**TABLE** ∗tab, **SOLVER solver**)

120. void **CTA_put_optim_gap** (**TABLE** ∗tab, double **optim_gap**)

121. void **CTA_put_max_time** (**TABLE** ∗tab, double **max_time**)

122. void **CTA_put_preprocessSC** (**TABLE** ∗tab, int **ppsc**)

123. void **CTA_put_eprhs** (**TABLE** ∗tab, double **eprhs**)

124. void **CTA_put_epint** (**TABLE** ∗tab, double **epint**)

125. void **CTA_put_mipemphasis** (**TABLE** ∗tab, **MIPEMPHASIS mipemphasis**)

126. int **CTA_put_heurmip** (**TABLE** ∗tab, int h)

127. void **CTA_put_varsel** (**TABLE** ∗tab, **VARSEL varsel**)

128. void **CTA_put_objective_fun** (**TABLE** ∗tab, double **fobj**)

129. void **CTA_put_lowbnd_fobj** (**TABLE** ∗tab, double **lowbnd_fobj**)

130. void **CTA_set_gap** (**TABLE** ∗tab)

131. void **CTA_put_BigM** (**TABLE** ∗tab, double bigm)

132. void **CTA_put_final_status** (**TABLE** ∗tab, **SOLVER_STATUS** s)

133. int **CTA_put_logfile_solver** (**TABLE** ∗tab, const char ∗logfile)

134. int **CTA_put_instance_name** (**TABLE** ∗tab, const char ∗instname)

135. void **CTA_put_firstfeas** (**TABLE** ∗tab, bool ff)

136. int **CTA_Repair_Infeas_xprs** (**TABLE** ∗tab, MODEL ∗mod, char∗ fname)

137. bool **CTA_get_make_additive** (**TABLE** ∗tab)

138. int **CTA_put_make_additive** (**TABLE** ∗tab, bool madd)

139. OPTMODEL **CTA_get_opt_model** (**TABLE** ∗tab)

140. void **CTA_put_opt_model** (**TABLE** ∗tab, **OPTMODEL** opt_mod)

141. bool **CTA_get_repair_infeas** (**TABLE** ∗tab)

142. void **CTA_put_repair_infeas** (**TABLE** ∗tab, bool repair_infeas)

143. char ∗ **CTA_get_repair_inputfile** (**TABLE** ∗tab)

144. int **CTA_put_repair_inputfile** (**TABLE** ∗tab, const char ∗repfile)

D Routines description

Routine Name	void CTA_Dump_X_table(TABLE *tab, double *x)
Routine Location	C_CTA\libCTA_bc\src\cta_solve.cpp
Routine Comment	
	write solution in table.

Routine Name	int CTA_lowering_ub(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_solve.cpp
Routine Comment	
	Upper bound for cell a[i] is X times the cell value, being X depending on how large a[i] is. If a[i] is very large, X is NubMAX (e.g. 1.2, 20% more); if small, X (NubMIN) is larger (e.g. 20, 2000% more). X is not linear on a[], but linear on log(a[]).

Routine Name	int CTA_Close_Solver(TABLE *tab, MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve.cpp
Routine Comment	
	close solver returns 0 if successful, or any error found closing the solver.

Routine Name	int CTA_Clear_Model(TABLE *tab, MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve.cpp
Routine Comment	
	frees model memory returns 0 if successful (should not be error, other than internal error).

Routine Name	int CTA_Preprocess(TABLE *tab, MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve.cpp
Routine Comment	
	if CTA_get_preprocessSC()is !=0, then preprocess sensitive cells, such that if some protection level (plpl or pupl is 0) the other sense is fixed (otherwise 0 protection levels means current value already protects the table). returns 0 is there is no error; otherwise returns != 0 (CTA_OUT_-OF_MEMORY).

Routine Name	int CTA_Init_Solver(TABLE *tab, MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve.cpp
Routine Comment	
	Init Solver (CPLEX or XPRESS).

Routine Name	int CTA_Load_Model(TABLE *tab, MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve.cpp
Routine Comment	
	Creates model (calling either the cplex or xpress particular routine). For both solvers, variables are ordered as: 1st (z^+): positive dir (all of them); ncells variables 2nd 2 (z-): negative dir (all of them); ncells variables 3rd (y): binary vars; npb variables For both solvers constraints are ordered as: 1st table constraints (Az=0) ncnstr constraints 2nd (z^+) - pupl y >= 0 npb constraints 3rd (z^+) - (ub-a) y <= 0 npb constraints 4rt (z^-) + plpl y >= plpl npb constraints 5th (z^-) + (a-lb) y <= (a-lb) npb constraints The criterion is thus: y=1 => (z^-)=0 and (z^+) >= pupl) y=0 => (z^+)=0 and (z^-) >= plpl Note that $\lvert x \rvert$ = (z^+) + (z^-) and x = (z^+) - (z^-) this lowering strategy is not used (could cause infeasibility problems) cout << "Warning: upper bounds have been modified.\n"; CTA_lowering_ub(tab);

Routine Name	int CTA_Run_Solver(TABLE *tab, MODEL *mod, int restart)
Routine Location	C_CTA\libCTA_bc\src\cta_solve.cpp
Routine Comment	
	calls solver for optimization, and checks later if a feasible or optimal solution was found returns either the status of the solution, if exits (see cta_table.h for a description of exit status) or some error code, if not enough memory, or solver error retrieving solution (see again cta_table.h for a description of error codes) .

Routine Name	int CTA_Find_Solution(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_solve.cpp
Routine Comment	
	Solves CTA problem Input/output parameters: tab: table to be protected; on exit, optimal protections stored in tab Returns: exit conditions, see cta_table.h.

Routine Name	void CTA_show_cpx_status(int status)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_cplex.cpp
Routine Comment	
	Show CPLEX status.

Routine Name	int CTA_Solution_cpx(MODEL *mod, double **X)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_cplex.cpp
Routine Comment	
	retrieves CPLEX solution after optimizaton and stores in X. X is allocated in that routine This routine is only called once a feasible/optimal solution has been found. returns 0 if successful; otherwise CTA_OUT_OF_MEMORY or solver error if problems retrieving solution.

Routine Name	int CTA_Close_Solver_cpx(MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_cplex.cpp
Routine Comment	
	frees CPLEX memory and closes problem returns CTA_XPRESS_ERROR if any error; otherwise 0.

Routine Name	int CTA_Clear_Model_cpx(MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_cplex.cpp
Routine Comment	
	frees model memory no expected error here, always return 0.

Routine Name	int CTA_Init_Solver_cpx(TABLE *tab, MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_cplex.cpp
Routine Comment	
	returns !=0 if there are problems opening CPLEX (licensing problems; otherwise it returns 0.

Routine Name	int CTA_Load_original_matrix_cpx(TABLE *tab, MODEL *mod, double **MatV, int **MatI)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_cplex.cpp
Routine Comment	
	read data from constraints table and fill intermediate structure MatV and MatI returns CTA_OUT_OF_MEMORY if not enough memory for intermediate structure; 0, if successful.

Routine Name	int CTA_Load_Model_cpx(TABLE *tab, MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_cplex.cpp
Routine Comment	
	creates model; see CTA_load_model() for details about order of variables and constraints returns CTA_OUT_OF_MEMORY if not enough memory for intermediate structure; 0, if successful.

Routine Name	int CTA_Run_Solver_cpx(TABLE *tab, MODEL *mod, int restart)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_cplex.cpp
Routine Comment	
	run CPLEX, loading problem if first run after optimization, performs translation from CPLEX to CTA exit codes returns one of CTA exit codes (see cta_table.h for the list).

Routine Name	int CPXPUBLIC infocallback (CPXCENVptr env, void *cbdata, int wherefrom, void *cbhandle)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_cplex.cpp
Routine Comment	
	CPLEX info callback to retrieve current incumbent ant best lower bound.

Routine Name	void CTA_show_xprs_errormsg(const char *sSubName,int nLineNo,int nErrCode)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_xpress.cpp
Routine Comment	
	Display error information about XPress errors. Arguments: const char ∗sSubName Subroutine name int nLineNo Line number int nErrCode Error code.

Routine Name	int CTA_Solution_xprs(MODEL *mod, double **X)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_xpress.cpp
Routine Comment	
	retrieves XPRESS solution after optimizaton and stores in X X is allocated in that routine This routine is only called once a feasible/optimal solution has been found. returns 0 if successful; otherwise CTA_OUT_OF_MEMORY or solver error if problems retrieving solution.

Routine Name	int CTA_Close_Solver_xprs(MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_xpress.cpp
Routine Comment	
	frees XPRESS memory and closes problem returns CTA_XPRESS_ERROR if any error; otherwise 0.

Routine Name	int CTA_Clear_Model_xprs(MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_xpress.cpp
Routine Comment	
	frees model memory no expected error here, always return 0.

Routine Name	int CTA_Init_Solver_xprs(TABLE *tab, MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_xpress.cpp
Routine Comment	
	returns !=0 if there are problems opening XPRESS (licensing problems) otherwise it returns 0.

Routine Name	int CTA_Load_original_matrix_xprs(TABLE *tab, MODEL *mod, double **MatV, int **MatI)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_xpress.cpp
Routine Comment	
	read data from constraints table and fill intermediate structure MatV and MatI returns CTA_OUT_OF_MEMORY if not enough memory for intermediate structure; 0, if successful.

Routine Name	int CTA_Load_Model_xprs(TABLE *tab, MODEL *mod)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_xpress.cpp
Routine Comment	
	creates model; see CTA_load_model() for details about order of variables and constraints returns CTA_OUT_OF_MEMORY if not enough memory for intermediate structure; 0, if successful.

Routine Name	int CTA_Run_Solver_xprs(TABLE *tab, MODEL *mod, int restart)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_xpress.cpp
Routine Comment	
	run XPRESS, loading problem if first run after optimization, performs translation from XPRESS to CTA exit codes returns one of CTA exit codes (see cta_table.h for the list).

Routine Name	int CTA_reallocate(TABLE *tab, int size)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	to increase or to fit size of memory.

Routine Name	int CTA_ijkl2n(vector¡int¿& ijkl,vector¡int¿& d, int ndim)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	given cell (i,j,k,l) return its position n.

Routine Name	void CTA_n2ijkl(int n, vector¡int¿& ijkl,vector¡int¿& d, int ndim)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	given position n returns cell (i,j,k,l).

Routine Name	int CTA_allocate_struct_TABLE (TABLE **ptab, int BLKSIZE)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	Allocate (initialize) struct table to CTA.

Routine Name	int CTA_allocate_struct_CELL (TABLE **ptab, int BLKSIZE)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	Allocate (initialize) struct CELL of the table to CTA.

Routine Name	int CTA_allocate_struct_SENSITIVECELL (TABLE **ptab, int BLKSIZE)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	initial prevision of BLKSIZE sensitive cells.

Routine Name	int CTA_allocate_struct_CONSTRAINTS (TABLE **ptab, int BLKSIZE)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	allocates struct constraints.

Routine Name	int CTA_allocate_struct_B (TABLE **ptab, int BLKSIZE)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	initial prevision of BLKSIZE b (right side of constraints).

Routine Name	int CTA_allocate_struct_BEGCONSTROW (TABLE **ptab, int BLKSIZE)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	pointer to begin of const. coefs.

Routine Name	int CTA_allocate_struct_COEFROW (TABLE **ptab, int BLKSIZE)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	pointer to coeficient value.

Routine Name	int CTA_allocate_struct_XCOEFROW (TABLE **ptab, int BLKSIZE)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	pointer to index of each coeficient.

Routine Name	int CTA_allocate_struct_BEGCONSTCOL (TABLE **ptab, int BLKSIZE)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	pointer to begin of const. coefs.

Routine Name	int CTA_allocate_struct_COEFCOL (TABLE **ptab, int BLKSIZE)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	pointer to coeficient value.

Routine Name	int CTA_allocate_struct_XCOEFCOL(TABLE **ptab, int BLKSIZE)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	pointer to index of each coeficient.

Routine Name	int CTA_create_table(TABLE **ptab, int ncells, int BLK-SIZE,TYPE_CONSTRAINTS type_constraints)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	allocates and initializes table of ncells returns 0 if everything goes fine return CTA_OUT_OF_MEMORY if not enough memory.

Routine Name	int CTA_delete_table(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	deletes a non-empty table.

Routine Name	int CTA_delete_constraints(TABLE *tab, int type)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	deletes a non-empty constraints struct.

Routine Name	int CTA_create_table_from_file(TABLE **ptab, char *file, TYPE_CONSTRAINTS type_constraints)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	creates table for CTA from file in csp format returns 0 if everything goes fine returns CTA_OUT_OF_MEMORY if not enough memory returns CTA_FILE_NOT_FOUND if file not found.

Routine Name	int CTA_generate_columnwise_matrix (TABLE **ptab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	create a structure columwise for constraints.

Routine Name	int CTA_check_relations_table(TABLE *tab, TYPE_VALUES val, double reltol, int outlevel)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	Check that table values (original or CTA ones) satisfy table linear constraints Parameters are: tab: input table. val: either ORIG_VALUES or CTA_VALUES (to know which table has to be checked, the original or the perturbed one). reltol: constraint violated if abs((lhs-rhs)/(1+rhs)) > reltol outlevel: if 0, nothing printed if 1, a message with the number of violated constraints is printed if 2, lhs and rhs of violated constraints are printed Returns: n: number of violated relations (n>=0) CTA_OUT_OF_MEMORY: if not enough memory.

Routine Name	int CTA_check_protections(TABLE *tab, double reltol, int outlevel)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	Check that sensitive cells perturbations after optimization satisfy protection levels Parameters are: tab: input table. reltol: cell unprotected if ((ctav > (v-lpl)) + (1+abs(ctav))*reltol) and ((ctav < (v+lpl)-(1+abs(ctav))*reltol) outlevel: if 0, nothing printed if 1, a message with the number of unprotected sensitive cells is printed if 2, protection levels and perturbation of unprotected cells are printed Returns: n: number of sensitive unprotected cells after optimization (n>=0).

Routine Name	int CTA_check_bounds(TABLE *tab, double reltol, int outlevel)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	Check that CTA cell values satisfy cell lower and upper bounds Parameters are: tab: input table. reltol: bounds violated if ((ctav < lb - (1+abs(ctav))*reltol) or ((ctav > ub + (1+abs(ctav))*reltol) outlevel: if 0, nothing printed if 1, a message with the number of violated cell bounds is printed if 2, bounds and CTA values of violated cells are printed Returns: n: number of cell bounds violated after optimization (n>=0).

Routine Name	int CTA_check_perturbations(TABLE *tab, double abstol, int outlevel)
Routine Location	C_CTA\libCTA_bc\src\cta_table.cpp
Routine Comment	
	Check that CTA perturbatations satisfy that only z^+ or z^- are positive, but not both. Constraints impose upl*y <= z^+ <= ub_u*y lpl*(1-y) <= z^- <= ub_l*(1-y) y in {0,1}, so only one of z^+,z^- may be >0. However due to big ub_u or ub_l values together with y=epsilon or y=1-epsilon, then ub_u*y>0 or ub_l*(1-y)>0. This would mean a wrong solution. This routine checks for this situation. Parameters are: tab: input table. abstol: wrong perturbation if (z^+ > abstol and z^- > abstol) outlevel: if 0, nothing printed if 1, a message with the number of cells with wrong perturbations if 2, cells, and wrong perturbations of violated cells are printed Returns: n: number of cells with wrong perturbations after optimization (n>=0).

Routine Name	double CTA_get_cellvalue(TABLE *tab,int pos)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return cell value in array[pos].

Routine Name	double CTA_get_cellperturbation_up(TABLE *tab,int pos)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return cell perturbation up in array[pos].

Routine Name	double CTA_get_cellperturbation_down(TABLE *tab,int pos)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return cell perturbation down in array[pos].

Routine Name	double CTA_get_lowbound(TABLE *tab,int pos)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return lower bound cell in array[pos]..

Routine Name	double CTA_get_upbound(TABLE *tab,int pos)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return upper bound cell in array[pos].

Routine Name	double CTA_get_modifupbound(TABLE *tab,int pos)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return modified upper bound cell in array[pos].

Routine Name	double CTA_get_weight(TABLE *tab,int pos)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return cell weight in array[pos].

Routine Name	int CTA_get_ncells(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get number of cells.

Routine Name	int CTA_get_npcells(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get number of sensitive cells.

Routine Name	int CTA_get_nconstraints (TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get number of cell linear relations.

Routine Name	int CTA_get_nnz(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get number of nonzeros in constraints.

Routine Name	int CTA_get_index_sensitive_cell (TABLE *tab,int pos)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return index in array of sensitives (0..npcells-1) of cell 'pos'.

Routine Name	int CTA_get_index_cell(TABLE *tab,int pos)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return index (0..ncells-1) of sensitive cell 'pos'.

Routine Name	double CTA_get_plpl(TABLE *tab,int pos)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return lower protection limit.

Routine Name	double CTA_get_pupl(TABLE *tab,int pos)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return upper protection limit.

Routine Name	int CTA_get_begconstraints(TABLE *tab,int i,TYPE_CONSTRAINTS type_cons)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return pointer to begin of constraints coefficients row/column.

Routine Name	int CTA_get_begconstraints_rowwise(TABLE *tab,int i)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return pointer to begin of constraints coefficients rowwise.

Routine Name	int CTA_get_begconstraints_columnwise (TABLE *tab,int i)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return pointer to begin of constraints coefficients columnwise.

Routine Name	double CTA_get_coefconstraints(TABLE *tab,int i,TYPE_CONSTRAINTS type_cons)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return the coef constraints in cell i.

Routine Name	double CTA_get_coefconstraints_rowwise(TABLE *tab,int i)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return the coef constraints row in cell i.

Routine Name	double CTA_get_coefconstraints_columnwise(TABLE *tab,int i)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return the coef constraints column in cell i.

Routine Name	int CTA_get_xcoefconstraints(TABLE *tab,int i,TYPE_CONSTRAINTS type_cons)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return the xcoef constraints in cell i.

Routine Name	int CTA_get_xcoefconstraints_rowwise(TABLE *tab,int i)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return the xcoef constraints row in cell i.

Routine Name	int CTA_get_xcoefconstraints_columnwise(TABLE *tab,int i)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	return the xcoef constraints col in cell i.

Routine Name	double CTA_get_rhsconstraints(TABLE *tab,int i)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get right hand side of each constraint.

Routine Name	TYPE_CONSTRAINTS CTA_get_typeconstraints(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get type of constraints.

Routine Name	SOLVER CTA_get_solver(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get solver (CPLEX, XPRESS) in order to solve CTA problem.

Routine Name	double CTA_get_optim_gap (TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get optim_gap to solve CTA problem.

Routine Name	double CTA_get_max_time (TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get max_time to solve CTA problem.

Routine Name	int CTA_get_preprocessSC (TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get preprocess sensitive cells option.

Routine Name	double CTA_get_eprhs(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	returns parameter eprhs (feasibility tolerance).

Routine Name	double CTA_get_epint(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	returns parameter epint (integrality tolerance).

Routine Name	MIPEMPHASIS CTA_get_mipemphasis(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	returns parameter mipemphasis (emphasis parameter of CPLEX MIP optimization).

Routine Name	int CTA_get_heurmip(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	returns parameter heurmip (heurdivespeedup parameter of XPRESS MIP optimization).

Routine Name	VARSEL CTA_get_varsel (TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	returns parameter varsel (variable selection parameter of CPLEX MIP optimization).

Routine Name	double CTA_get_objective_fun (TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get value of objective function.

Routine Name	double CTA_get_lowbnd_fobj(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get value of lower bound of objective function.

Routine Name	double CTA_get_gap(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get current gap (in percentage).

Routine Name	double CTA_get_BigM (TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get BigM of constraints z^+ <= M*y, z^- <= M(1-y)

Routine Name	SOLVER_STATUS CTA_get_final_status (TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get status at the end of solution.

Routine Name	char * CTA_get_logfile_solver (TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get name of file with log of solver; pay attention: the pointer to the string is sent, don't change it, jut use it for printing of copying!.

Routine Name	char * CTA_get_instance_name (TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get instance_name pay attention: the pointer to the string is sent, don't change it, jut use it for printing of copying!.

Routine Name	bool CTA_get_firstfeas(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get boolean first_feasible.

Routine Name	void CTA_put_ncells(TABLE *tab, int ncells)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put ncells of the table.

Routine Name	void CTA_put_npcells(TABLE *tab, int npcells)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put npcells (sensitive cells).

Routine Name	void CTA_put_cellvalue(TABLE *tab,int pos,double value)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put cell value.

Routine Name	void CTA_put_cellperturbation_up(TABLE *tab,int pos,double perturbation)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put cell perturbation up value.

Routine Name	void CTA_put_cellperturbation_down(TABLE *tab,int pos,double perturbation)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put cell perturbation down value.

Routine Name	void CTA_put_cellweight(TABLE *tab,int pos,double weight)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put cell weight.

Routine Name	void CTA_put_lowbound(TABLE *tab,int pos,double lb)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put cell lower bound.

Routine Name	void CTA_put_upbound(TABLE *tab,int pos,double ub)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put cell upper bound.

Routine Name	void CTA_put_modifupbound(TABLE *tab,int pos,double modif_ub)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put cell modified upper bound.

Routine Name	void CTA_put_index_sensitive_cell (TABLE *tab,int index,int pos)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put index of each sensitive cell.

Routine Name	void CTA_put_info_sensitive_cell(TABLE *tab,int pos,int index, double plpl,double pupl)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put basic information sensitive cell: - position of this sensitive cell in array of cells - lower protection limit - upper protection limit.

Routine Name	void CTA_put_typetable(TABLE *tab,TYPE_TABLE t)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put type of table.

Routine Name	void CTA_put_K(TABLE *tab,int K)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put K (table dimension).

Routine Name	void CTA_put_typeconstraints(TABLE *tab,TYPE_CONSTRAINTS type_c)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put type of constraints.

Routine Name	void CTA_put_nnz(TABLE *tab, int nnz)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put number of nonzeros in tad constraints.

Routine Name	void CTA_put_nconstraints(TABLE *tab,int nconstraints)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put number of constraints in tad constraints.

Routine Name	void CTA_put_begconstraints(TABLE *tab,int i,int ctcoef,TYPE_CONSTRAINTS type_cons)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	actualize pointer to begin of constraints coefficients rowwise/columnwise.

Routine Name	void CTA_put_begconstraints_rowwise(TABLE *tab,int i,int ctcoef)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	actualize pointer to begin of constraints coefficients rowwise.

Routine Name	void CTA_put_begconstraints_columnwise(TABLE *tab,int i,int ctcoef)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	actualize pointer to begin of constraints coefficients columnwise.

Routine Name	void CTA_put_coefconstraints(TABLE *tab,int i,double coef,TYPE_CONSTRAINTS type_cons)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put coef value for all constraints (actualize) rowwise/columnwise.

Routine Name	void CTA_put_coefconstraints_rowwise(TABLE *tab,int i,double coef)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put coef value for all constraints (actualize) rowwise.

Routine Name	void CTA_put_coefconstraints_columnwise(TABLE *tab,int i,double coef)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put coef value for all constraints (actualize) columnwise.

Routine Name	void CTA_put_xcoefconstraints(TABLE *tab,int i,int xcoef,TYPE_CONSTRAINTS type_cons)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put index of each coeficient (actualize) rowwise/columnwise.

Routine Name	void CTA_put_xcoefconstraints_rowwise(TABLE *tab,int i,int xcoef)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put index of each coeficient (actualize) rowwise.

Routine Name	void CTA_put_xcoefconstraints_columnwise(TABLE *tab,int i,int xcoef)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put index of each coeficient (actualize) columnwise.

Routine Name	void CTA_put_rhsconstraints(TABLE *tab,int i,double b)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put right side of each constraint rowwise/columnwise.

Routine Name	void CTA_put_solver(TABLE *tab, SOLVER solver)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put solver (CPLEX, XPRESS) in order to solve CTA problem.

Routine Name	void CTA_put_optim_gap (TABLE *tab, double optim_gap)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put optim_gap to solve CTA problem.

Routine Name	void CTA_put_max_time (TABLE *tab, double max_time)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put max_time to solve CTA problem.

Routine Name	void CTA_put_eprhs(TABLE *tab, double eprhs)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put parameter eprhs (feasibility tolerance).

Routine Name	void CTA_put_epint(TABLE *tab, double epint)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put parameter epint (integrality tolerance).

Routine Name	void CTA_put_mipemphasis(TABLE *tab,MIPEMPHASIS mipemphasis)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put parameter mipemphasis (emphasis parameter of CPLEX MIP optimization) .

Routine Name	int CTA_put_heurmip(TABLE *tab,int h)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put parameter heurmip (heurdivespeedup parameter of XPRESS MIP optimization) returns 0 if h is -1, 0,1,2,3; otherwise returns -1, and heurmip is not set.

Routine Name	void CTA_put_varsel (TABLE *tab,VARSEL varsel)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put parameter varsel (variable selection parameter of CPLEX MIP optimization).

Routine Name	void CTA_put_objective_fun (TABLE *tab, double fobj)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put value of incumbent or final solution.

Routine Name	void CTA_put_lowbnd_fobj(TABLE *tab, double lowbnd_fobj)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put value of lower bound of objective function.

Routine Name	void CTA_set_gap(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	compute gap (in percentage) from objective function and its lower bound.

Routine Name	void CTA_put_BigM (TABLE *tab, double bigm)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put BigM of constraints z^+ <= M*y, z^- <= M(1-y).

Routine Name	void CTA_put_final_status (TABLE *tab, SOLVER_STATUS s)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put final status after optimization.

Routine Name	int CTA_put_logfile_solver (TABLE *tab, const char *logfile)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put name of file with log of solver; if logfile is NULL no output is printed (neither by file nor to screen). returns 0 if successful, or CTA_OUT_OF_MEMORY if no free space for copying the name.

Routine Name	int CTA_put_instance_name (TABLE *tab, const char *instname)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put name of instance returns 0 if successful, or CTA_OUT_OF_MEMORY if no free space for copying the name.

Routine Name	void CTA_put_firstfeas(TABLE *tab, bool ff)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put boolean first_feasible.

Routine Name	int CTA_Repair_Infeas_xprs(TABLE *tab, MODEL *mod, char* fname)
Routine Location	C_CTA\libCTA_bc\src\cta_solve_xpress.cpp
Routine Comment	
	run XPRESS to obtain the vaiables or constraints which make infeasible the problem. Param: input: tab, mod; fname: name of file output. after optimizations, performs translation from XPRESS to CTA exit codes returns one of CTA exit codes (see cta_table.h for the list).

Routine Name	bool CTA_get_make_additive (TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get value of make_additive for making additive nonadditive tables if requested.

Routine Name	void CTA_put_make_additive (TABLE *tab, bool madd)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put value of make_additive for making additive nonadditive tables if requested.

Routine Name	OPTMODEL CTA_get_opt_model (TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get value of optimization model to be used.

Routine Name	void CTA_put_opt_model (TABLE *tab, OPTMODEL opt_mod)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put value of optimization model to be used.

Routine Name	bool CTA_get_repair_infeas(TABLE *tab, bool repair_infeas)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get value of repair_infeas for applying repair infeasibility procedure if requested.

Routine Name	void CTA_put_repair_infeas(TABLE *tab, bool repair_infeas)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put value of repair_infeas for applying repair infeasibility procedure if requested.

Routine Name	char * CTA_get_repair_inputfile(TABLE *tab)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	get name of file with input information for repair tool; returns 0 if successful, or CTA_OUT_OF_MEMORY if no free space for copying the name.

Routine Name	int CTA_put_repair_inputfile(TABLE *tab, const char *repfile)
Routine Location	C_CTA\libCTA_bc\src\cta_table.h
Routine Comment	
	put name of file with input information for repair tool pay attention: the pointer to the string is sent, don't change it, just use it for printing of copying.