User's and programmer's manual of the network flows heuristics package for cell suppression in 2D tables

Jordi Castro
Dept. of Statistics and Operations Research
Universitat Politècnica de Catalunya
Pau Gargallo 5, 08028 Barcelona (Catalonia, Spain)
jcastro@eio.upc.es

Technical Report DR 2003-07
February 2003

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

User's and programmer's manual of the network flows heuristics package for cell suppression in 2D tables *

Jordi Castro
Dept. of Statistics and Operations Research
Universitat Politècnica de Catalunya
Pau Gargallo 5, 08028 Barcelona (Catalonia, Spain)
jcastro@eio.upc.es

Abstract

The network flows heuristics package for cell suppression was developed in the scope of the 5th European Union program IST-2000-25069 CASC project. It implements two heuristics based on network flows for secondary cell suppression in two dimensional tables. This document shows the main features of the package. It also describes the package interface and how to connect it with the user's application.

Key words: C/C++ programming languages, complementary cell suppression problem, linear programming, shortest-paths, minimum-cost network flows.

*Work supported by the IST-2000-25069 CASC project. This document corresponds to CASC deliverable 4.1-D5.

Contents

1 Introduction

The network flows package for cell suppression (NF_CSP) implements two heuristics for the protection of statistical data in two dimensional tables. The heuristics are improved versions (i.e., faster) of those originally presented in [5] and [7] for the secondary cell suppression problem. General details about the algorithms implemented in NF_CSP can be found in [3]; a thorough description will be provided in a future paper.

In the first heuristic, derived from [5], only flows 0 or 1 are sent through the network. We'll refer it as the "0-1-flows" heuristic. The second will be denoted as the "n-flows" heuristics, since the network can transport any positive flow.

The current package is linked with three network flows solvers: CPLEX7.5 [6], PPRN [4], and an efficient implementation of the bidirectional Dijkstra's algorithm for shortest-paths (that will be denoted as "Dijkstra") [1]. Later releases of CPLEX will also work if the interface routines are the same than for version 7.5. The 0-1-flows heuristic can use any of the three solvers. The network flows problems formulated by the n-flows heuristic can only be solved with PPRN and CPLEX. PPRN and Dijkstra were implemented at the Dept. of Statistics and Operations Research of the Universitat Politècnica de Catalunya, and are included in NF_CSP. PPRN was originally developed during 1992–1995, but it had to be significantly improved within the CASC project to work with NF_CSP. Dijkstra was completely developed in the scope of CASC.

The third solver, CPLEX7.5, is a commercial tool, and requires purchasing a license. However, PPRN is a fairly good replacement—although not so robust— for the network flows routines of CPLEX7.5. Therefore, in principle, there is no need for an external commercial solver. Moreover, solver Dijkstra is by far the most efficient option, although it can only deal with problems formulated by the 0-1 flows heuristics. It should be used whenever possible for efficiency reasons.

Even though two of the three solvers are included in the distribution of NF_CSP, this document only describes the features of the heuristics, and from the user's point of view. A detailed description of PPRN and Dijkstra's solvers can be found in [2, 4] and [1], respectively.

The structure of the document is as follows. Section 2 introduces a simple program that shows how to use NF_CSP from the user's application. Section 3 describes the main options and features of the package. In Section 4 we present the set of routines to interface with NF_CSP, grouped by functional categories. A final Appendix lists all the files and routines of NF_CSP.

2 Example program

Consider we want to protect with NF_CSP the 7×9 table shown in Table 1, where rows and columns are numbered respectively from 0 to 6 and from 0 to 8, last column and row being the marginals. This is the standard numbering used by the package. This table has four primary cells, (1,1), (1,2), (4,3), (4,6), highlighted in boldface in Table 1.

The example program of pages 6–8 illustrates the main steps that need to be performed to protect Table 1 with NF_CSP. In this example the information about the cell values and primaries is defined within the code. In a large application this would likely be read from a file or computed from some raw data. Moreover, in the user's application instead of a main program it would surely be a routine.

Any code that uses NF_CSP has to include the header file `csp_table2D.h`, as in line 4 of the example program. This file contains all the declarations (data structures and routines) needed to interface with NF_CSP.

Table 1: Original table, with primaries in boldface

	0	1	2	3	4	5	6	7	8
0	418	730	930	85	762	48	986	744	4703
1	727	**94**	**253**	530	507	952	177	545	3785
2	962	64	911	862	974	647	304	221	4945
3	377	354	897	507	998	144	894	447	4618
4	584	616	109	**48**	208	94	**49**	192	1900
5	986	934	243	820	214	737	896	783	5613
6	4054	2792	3343	2852	3663	2622	3306	2932	25564

In the example code, the one-dimension arrays `values` and `pos_primary`, defined between lines 9–24, contain the table information. `values` stores the table cell values by rows. `pos_primary` stores sequentially the (row,column) position of the four primary cells. This is clearly not the best way to organize the information, but it is simple enough for an example program. Note that numbering of rows and columns starts at 0.

We first need to declare a TABLE2D* variable (pointer to TABLE2D structure). In the code we named it `tab` (line 27). 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 32, calling `create_table(&tab,m,n,p)`. `m` is the number of rows, `n` is the number of columns, and `p` is the number of primary cells (7, 9 and 4 respectively in the example). `create_table()` also sets to default values the several options and parameters to be used for protecting the table (as the heuristic and solver, type of weights for each cell, etc.). These default values can be changed later through several manipulation routines (described in Section 4).

The next step is to fill the table with the cell values (lines 35–43). Routine `put_cellvalue2D(tab, row,col,value)` is used for this purpose. It fills cell (`row`,`col`) with `value`. Routines `get_nrows2D(tab)` and `get_ncolumns2D(tab)` (lines 39 and 41) provide respectively the number of rows and columns of the table. They clearly return 7 and 9 in this example. However, for portability reasons, it is better to call these routines, instead of typing 7 and 9. The user should use in his application the interface routines to NF_CSP whenever possible.

Information of primary cells is given at lines 45–60. `get_npcells2D(tab)` returns the number of primaries of `tab`. The information of each primary cell is provided calling `put_pcell2D(tab,i,row, col,lpl,upl)`: cell (`row`,`col`) is the `i`-th primary, and requires lower and upper protection levels of `lpl` and `upl`, respectively. In the example the lower and upper protections are computed as a 10% and 15% of the cell value. The value of cell (`row`,`col`) is recovered through `get_cellvalue2D(tab,row,col)` (line 56).

The above is the minimum information required to protect the table. We can now proceed with the protection, calling `csp_heur2D(tab)` (line 67). Since we did not modify them, the default settings will be used. These are:

- 0-1 flows heuristic.
- Solver Dijkstra.
- The cell weight (used in the objective function to be minimized) is the cell value.
- Primary cells are processed in NORMAL order, i.e., the order provided by the user through `put_pcell2D()`.

- A lower bound of the optimal weight suppressed (i.e., weight of the suppressed cells in the optimal solution). Although this can be useful to know how far the solution provided by the heuristic is from the optimal one, that procedure can be fairly time consuming for large tables (in fact, it can take more time than the heuristic).

If the table is successfully protected, `csp_heur2D(tab)` returns 0. Otherwise, a nonzero value will be returned. See Section 4 for the list of possible return status.

Once the protection is performed, we can retrieve the solution obtained through several routines. What to be done at this stage is specific of each application. In our example, we just show some general information about the solution obtained (lines 77–87) and list the primary and secondary cells (lines 89–99). The general information displayed is the total value suppressed (`get_valuesuppressed2D(tab)`), total weight suppressed (`get_weightsuppressed2D(tab)`)—in this case is equal to the total value—, and total number of suppressed cells (primary plus secondary, computed as `get_npcells2D(tab)+get_nseccells2D(tab)`). The list of primary and secondary cells is obtained by calling for all the cells `get_cellstatus2D(tab,row,col)`. This routine returns the current status of cell (`row`,`col`), which can take the values PRIMARY, SECONDARY, PERMANENT or NONREMOVED (described in detail in Section 3. If only secondary cells were needed, we could have used routine `is_cell_secondary2D(tab,row,col)`, which is a shortcut for `get_cellstatus2D(tab,row,column)==SECONDARY`. Analogous routines `is_cell_primary`, `is_cell_permanent()` and `is_cell_nonremoved()` are provided.

Finally, the memory space of the table is freed at line 101, calling `delete_table(&tab)`.

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

Example program for the protection of Table 1

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  // to use the package we only need to include the csp_table2D.h
 4  #include "csp_table2D.h"
 5
 6  main(int argc, char *argv[])
 7  {
 8      // Values of the table to be protected, stored row-wise
 9      double values[]= {
10      418,    730,   930,     85,    762,     48,    986,    744,   4703,
11      727,     94,   253,    530,    507,    952,    177,    545,   3785,
12      962,     64,   911,    862,    974,    647,    304,    221,   4945,
13      377,    354,   897,    507,    998,    144,    894,    447,   4618,
14      584,    616,   109,     48,    208,     94,     49,    192,   1900,
15      986,    934,   243,    820,    214,    737,    896,    783,   5613,
16     4054,   2792,  3343,   2852,   3663,   2622,   3306,   2932,  25564
17      };
18
19      // Position of primay cells, each pair gives (i,j) in table
20      int pos_primary[]={ 1,1,
21                          1,2,
22                          4,3,
23                          4,6
```

```
24      };
25
26      // tab is the TABLE2D to be protected
27      TABLE2D *tab= NULL;
28
29      // Initializes tab as a 7x9 two dimensional table with 4 primary cells
30      // default settings are considered when initializing the table. They
31      // can be changed later through several table manipulation routines
32      create_table(&tab,7,9,4);
33
34      // Fill the table with the cell values from array values[],
35      // calling put_cellvalue2D() for each cell. In a general application
36      // the cell values will likely be read from file(s)
37      int i,j,l= 0;
38      // get_nrows2D() provides the number of table rows
39      for(i= 0;i<get_nrows2D(tab);i++)
40      // get_ncolumns2D() provides the number of table columns
41          for (j=0;j<get_ncolumns2D(tab);j++) {
42              put_cellvalue2D(tab,i,j,values[l++]);
43          }
44
45      // Provide position, and upper and lower protections of primary cells
46      // calling put_pcell2D() for each primary cell. In this example we
47      // consider 15% an 10% of the cell values for lower and upper protection.
48      // In a general application the information of primary cells will likely
49      // be read from file(s)
50      l= 0;
51      // get_npcells2D() provides the number of primary cells
52      for(i= 0; i<get_npcells2D(tab);i++) {
53          int pos_i= pos_primary[l];
54          int pos_j= pos_primary[l+1];
55          l += 2;
56          double cell_value= get_cellvalue2D(tab,pos_i,pos_j);
57          double lower_prot= 0.10*cell_value;
58          double upper_prot= 0.15*cell_value;
59          put_pcell2D(tab,i,pos_i,pos_j,lower_prot,upper_prot);
60      }
61
62      // Call heuristic procedure (default one is 0-1 flows heuristic).
63      // To use the n-flows heuristic we may call
64      // put_heuristic2D(tab,FLOWS_N) and put_solver2D(tab,PPRN)
65      // before csp_heur2D(tab)
66      int return_status;
67      if ((return_status= csp_heur2D(tab))) {
68          fprintf(stderr,"error %d reported by csp_heur2D\n",return_status);
69          delete_table(&tab);
70          exit(-1);
71      }
72
73      // Show results. In a general application that information will likely
```

```
74      // not be displayed (e.g., it will be sent to a file, or to some
75      // data structure)
76
77      // First some general information about the protected table
78
79      // Total value suppressed
80      printf("value suppressed \t= %g\n",get_valuesuppressed2D(tab));
81      // Total weight suppressed; weight is used in the objective function to
82      // be minimized instead of value (weight can be equal to value, log of
83      // value, binary (0 or 1), or a custom value set by the user)
84      printf("weight suppressed \t= %g\n",get_weightsuppressed2D(tab));
85      // Total number of suppressed cells: primary plus secondary
86      printf("# suppressed cells \t= %d\n\n",
87             get_npcells2D(tab)+get_nseccells2D(tab));
88
89      // Second, the list of removed cells.
90      // get_cellstatus2D() returns the status of a cell, which can take
91      // values PRIMARY, SECONDARY, PERMANENT or NONREMOVED
92      for(i= 0;i<get_nrows2D(tab);i++)
93          for (int j=0;j<get_ncolumns2D(tab);j++) {
94              STATUS_CELL st_cell= get_cellstatus2D(tab,i,j);
95              if (st_cell == PRIMARY)
96                  printf("Cell (%d,%d) is PRIMARY\n");
97              else if (st_cell == SECONDARY)
98                  printf("Cell (%d,%d) is SECONDARY\n");
99          }
100
101     delete_table(&tab);
102     return(0);
103 }
```

The output of the code is

```
value suppressed        = 1148
weight suppressed       = 1148
# suppressed cells      = 10

Cell (0,3) is SECONDARY
Cell (0,5) is SECONDARY
Cell (1,1) is PRIMARY
Cell (1,2) is PRIMARY
Cell (2,1) is SECONDARY
Cell (2,6) is SECONDARY
Cell (4,2) is SECONDARY
Cell (4,3) is PRIMARY
Cell (4,5) is SECONDARY
Cell (4,6) is PRIMARY
```

The suppression pattern is shown in Table 2, with primaries in boldface and secondaries underlined.

If a lower bound for the optimal solution wants to be computed, we can call `put_comp_low-`

Table 2: Protected table, with primaries in boldface and secondaries underlined

	0	1	2	3	4	5	6	7	8
0	418	730	930	85	762	48	986	744	4703
1	727	**94**	**253**	530	507	952	177	545	3785
2	962	64	911	862	974	647	304	221	4945
3	377	354	897	507	998	144	894	447	4618
4	584	616	109	**48**	208	94	**49**	192	1900
5	986	934	243	820	214	737	896	783	5613
6	4054	2792	3343	2852	3663	2622	3306	2932	25564

`bound2D(tab,TRUE)` before `csp_heur2D(tab)`, at line 66 for instance. Computing a lower bound means solving a linear program, and a CPLEX7.5 license is needed for this. In our example the lower bound obtained is 1148, i.e., the optimal solution is greater or equal than 1148. Since the solution provided by the heuristic has a total weight of exactly 1148, it is optimal. For large tables (millions of cells) computing the lower bound can be prohibitive, both in time and memory requirements. In these cases we must be confident that the heuristic solution is not too far from the optimal one.

NF_CSP also includes an auditing phase for computing the lower and upper bounds than an external attacker could derive for the primary cells after the publication of the table. For two dimensional tables, these values can be computed by solving two minimum-cost network flows problems. This is done in NF_CSP either by CPLEX7.5 or PPRN. NF_CSP first attempts to use CPLEX7.5; if no license is available then it switches to PPRN. Note that the auditing phase of NF_CSP was just developed for testing purposes, and that it is not the most efficient procedure. It may be significantly improved by computing the two bounds through specialized maximum-flows algorithms. In our example, to compute the lower and upper protection provided by the suppression pattern, we could insert at line 100 the following code:

```
/* perform auditing phase--just for testing purposes */
 if ((return_status= csp_auditing2D(tab,0,get_npcells2D(tab)-1))) {
     fprintf(stderr,"error %d reported by csp_auditing2D\n",return_status);
     delete_table(&tab);
     exit(-1);
 };
 int pi,pj= 0;
 double lpl,upl,plpl,pupl;
 for(i=0;i<get_npcells2D(tab);i++) {
     get_pcell2D(tab,i,&pi,&pj,&lpl,&upl);
     plpl= get_pcelllpl2D(tab,i);
     pupl= get_pcellupl2D(tab,i);
     printf("primary (%d,%d):\t lower %f > %f \t upper %f > %f\n",
            pi,pj,plpl,lpl,pupl,upl);
 }
```

`csp_auditing2D(tab,first,last)` performs the auditing of primary cells between `first` and `last`. In our code, we perform the auditing for all the 4 primary cells (i.e., `first` is 0 and `last` is `get_npcells2D(tab)-1` $= 4 - 1 = 3$). After that, we retrieve the required lower and upper

protection for each primary calling `get_pcell2D(tab,i,&row,&col,&lpl,&upl)`. This routine returns the information of primary cell `i`, providing its position (`row`,`col`) in the table, and the required lower and upper protection `lpl` and `upl`. Next we get the lower and upper protection level obtained for primary `i` calling `get_pcelllpl2D(tab,i)` and `get_pcellupl2D(tab,i)`. Finally we print the lower and upper protection levels obtained, showing they are greater than those required. The output of the above piece of code in our example is:

```
primary (1,1):   lower 94.000000 > 9.400000      upper 49.000000 > 14.100000
primary (1,2):   lower 49.000000 > 25.300000     upper 94.000000 > 37.950000
primary (4,3):   lower 48.000000 > 4.800000      upper 85.000000 > 7.200000
primary (4,6):   lower 49.000000 > 4.900000      upper 94.000000 > 7.350000
```

3 Package options

3.1 Conditional compilation

The package has been successfully compiled an tested in both Linux (using gcc 2.95) 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. Note that 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 NF_CSP with MSVC6 in a MS-Windows system. It is also needed for the user's routines that interface with NF_CSP, again only in MS-Windows systems. When `WIN32` is defined, three more symbols are required for compiling the heuristics, PPRN and Dijkstra: `CSP_NF_2D_EXPORTS`, `PPRN_EXPORTS` and `DIJKSTRA_EXPORTS`. They allow 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. These three export symbols **DON'T** have to be defined for compiling the user's application, otherwise it will fail to interface with NF_CSP.
- `CPLEX75`. This symbol is required if one has a CPLEX7.5 license and plans to use it. It is not needed for compiling the user's application. If the symbol is defined, NF_CSP will consider CPLEX7.5 as one of the available solvers. Otherwise, if the symbol was not defined and NF_CSP is asked to use CPLEX7.5, it will return an error.
- `LOGINFO`. If this symbol is defined NF_CSP will display information about the evolution of the protection procedure. It is only intended for debugging purposes. It is not needed for compiling the user's application.

3.2 Heuristics and solvers

NF_CSP implements two heuristic protection methods. The 0-1 flows heuristic is an extension and improvement of that originally described in [5]. The main improvement, from an efficiency point of view, is that in [5] the subproblems were formulated as minimum-cost network flows models (thus, they needed a tool as PPRN or CPLEX7.5), whereas the algorithm implemented

in NF_CSP formulates shortest-paths subproblems. Shortest-paths can be solved—by Dijkstra's algorithm—orders of magnitude faster than the equivalent minimum-cost network flows models.

The second heuristic was inspired in the method described in [7]. The variant implemented in NF_CSP was designed to be faster, and it is currently fairly different from the proposal of [7]. The subproblems formulated are minimum-cost network problems, and are solved through PPRN or CPLEX7.5 (Dijkstra can not be used for this heuristic).

By default the 0-1 flows heuristic and Dijkstra solver will be used for protecting a table. These defaults are set by routine `create_table()`. This option is the fastest of all the combinations considered by NF_CSP. However, it must be noted that solutions provided by the n-flows heuristic are always equal or better than those computed by the 0-1 flows one. On the other hand, for large problems, the n-flows heuristic may be orders of magnitude slower than the 0-1 flows and Dijkstra combination. Choosing one or another option means a tradeoff between quality of solution and efficiency. The same argument can be applied when deciding between the heuristics of NF_CSP or an optimal method for cell suppression.

The heuristic to be used can be selected through

```
put_heuristic2D(tab,heuristic).
```

The second parameter can take values `FLOWS_BIN`, for the 0-1 heuristic, or `FLOWS_N`, for the n-flows one. To retrieve the current heuristic assigned to a table we can call

```
get_heuristic2D(tab).
```

It returns `FLOWS_BIN` or `FLOWS_N`.

It is also possible to change the default solver. If the 0-1 flows heuristic is going to be used, the solver should always be Dijkstra, for efficiency reasons. PPRN or CPLEX7.5 can be chosen for the n-flows heuristic. A particular solver can be set through

```
put_solver2D(tab,solver),
```

where `solver` can be `DIJKSTRA`, `PPRN` or `CPLEX`. Routine

```
get_solver2D(tab)
```

returns the current solver. If, after creating a table, the n-flows heuristic is selected, we must also change the default solver. Otherwise the package will try to use Dijkstra with the n-flows heuristic, returning an error code.

Once the combination heuristic-solver was selected, the protection is performed through

```
csp_heur2D(tab).
```

This routine returns 0 if the table was successfully protected. Otherwise it returns a nonzero code. Section 4 shows the return codes of all the interface routines.

3.3 Type of costs in the 0-1 flows heuristics

The 0-1 flows heuristic solves several shortest-paths problems on a network whose topology is defined by the table. The costs of arcs in this network are dynamically created for each primary

cell by the heuristic. The purpose of these costs is to guide the protection procedure, making unsupressed cells with low weights better candidates for suppression than those with larger weights. Computing these costs can be fairly expensive, and NF_CSP offers two alternatives. The first one, called `FASTER_WORSER`, computes a set of costs efficiently; however these costs are not the best ones, and, usually, provide worser solutions than the second set of costs. This second set is the `SLOWER_BETTER`. As its name shows, the heuristic is slower if these costs are computed, although the solution provided can be better. Several tests showed that, in average, the `FASTER_WORSER` option reduces in a 20% the execution times of the `SLOWER_BETTER` one. The difference in value suppressed is highly dependent on the particular table. However in most tests performed, it was not significant. The default option of NF_CSP is `FASTER_WORSER`. It can be changed with

```
put_cost_type2D(tab,cost_type),
```

where `cost_type` can be `FASTER_WORSER` or `SLOWER_BETTER`. The current type of cost is returned by function

```
get_cost_type2D(tab).
```

3.4 Cell status and zero cells

By default, cells with zero values are considered `PERMANENT`. A cell is permanent if it can not be removed by the heuristics. The other possible status of a cell are `PRIMARY`, `SECONDARY` and `NONREMOVED`. When creating a table, by default all the cells are `NONREMOVED` (candidates for suppression). The cells that the user define as primaries (through `put_pcell2D()`), are marked as `PRIMARY`. The heuristics will remove additional cells, marking them as `SECONDARY`. The user can retrieve the current status of cell (`row`,`col`) calling

```
get_cellstatus2D(tab,row,col),
```

which returns one of the four possible status. It is also possible to set the status of cell (`row`,`col`) through

```
put_cellstatus2D(tab,row,col,status).
```

This routine is intended for setting some cells as permanent. It should not be used for other status, because it could make the heuristics fail.

The only cells set as permanent by default by the heuristics are those with zero values. The user is allowed to change this default behaviour (therefore zero cells will be candidates for suppression, unless explicitly stated in the code with `put_cellstatus2D()`) through

```
put_cells_0_permanent2D(tab,FALSE).
```

The default behaviour can be recovered calling the above routine with `TRUE` as second argument. The current situation for zero cells can be retrieved using

```
get_cells_0_permanent2D(tab).
```

It returns `TRUE` if zero cells will be set permanent; otherwise returns `FALSE`.

3.5 Cell weights

Cell weights are used in the objective function to be minimized by the heuristics. There are four possible types of weights: `VALUE` (weights are equal to cell values), `LOG` (weights are the base-10 logarithm of the cell values), `BIN` (weights are binary, 0 or 1) and `CUSTOM` (the user will set the weights). The default type of weights is `VALUE`. This default can be changed with

```
put_typeweights2D(tab,typeweight),
```

where the second argument can be `VALUE`, `LOG`, `BIN` or `CUSTOM`. The current type is returned by

```
get_typeweights2D(tab).
```

If the type of weights is set to `CUSTOM` then the user will have to call routine

```
put_cellweight2D(tab,row,col,weight)
```

for each cell (`row`,`col`) to initialize its weight with `weight`. If some cell is omitted, its weight will be undefined. This can be a cause of trouble, and should be avoided.

The current weight of a cell is returned by

```
get_cellweight2D(tab,row,col).
```

After the heuristic procedure, we can retrieve the overall weight and overall value suppressed, both for primary and secondary cells, through respectively

```
get_weightsuppressed2D(tab)
```

and

```
get_valuesuppressed2D(tab).
```

3.6 Merit order for primary cells

The heuristics of NF_CSP are iterative processes that sequentially protect each primary cell. The order primary cells are selected (named merit order in NF_CSP) may modify the final solution. The user can choose between three merit orders: `NORMAL`, `ASCENDENT` and `DESCENDENT`. `NORMAL` is the order defined by the user when setting the primary cells through `put_pcell2D()`. If the `ASCENDENT` order is selected, cells will be protected according to theirs cell values sorted in ascendent order (i.e., the first cell protected will be that with the lowest cell value, and so on). The order is the opposite if `DESCENDENT` is chosen. The default merit order is `NORMAL`. This default can be modified with

```
put_typemorder2D(tab,merit),
```

where the second parameter can be any of the three merit orders. The current merit order is returned by

```
get_typemorder2D(tab).
```

To get information about the primary cells, NF_CSP offers two sets of functions, one for "unsorted" and other for "sorted" primary cells. All of them have as one of theirs parameters the position `i` of the primary cell. The unsorted-version functions provide the information for the `i`-th primary cell entered by the user. The sorted-version functions return the information for the `i`-th primary cell according to the current merit order. Clearly, if the current merit order is `NORMAL` both the sorted and unsorted-version functions return the same information. The sorted-version functions are

```
get_pcell2D(tab,i,&row,&col,&lpl,&upl),
        get_pcelllpl2D(tab,i),
        get_pcellupl2D(tab,i).
```

The unsorted-version ones are

```
get_sorted_pcell2D(tab,i,&row,&col,&lpl,&upl),
        get_sorted_pcelllpl2D(tab,i),
        get_sorted_pcellupl2D(tab,i).
```

The first function of each set provides the position (`row`,`col`) and required lower (`lpl`) and upper (`upl`) protection of primary `i`. The last two functions of each set return respectively the obtained lower and upper protection for primary `i`, either after calling the heuristics, or after performing the auditing. After the auditing, they return the real protection. After the heuristics, they provide a lower bound of the real protection.

3.7 Lower bounding procedure

The heuristics provide an approximate solution to the cell suppression problem. To know how far that solution is from the optimal one, we should get some lower bound to the optimal objective function (i.e., minimum value or minimum weight suppressed). NF_CSP includes a procedure for computing such lower bound. It is computed inside the heuristics before starting the protection procedure. The lower bounding procedure is an improvement (i.e., provides better lower bounds) of that originally suggested in [7]. Computing the lower bound means solving a linear programming problem, and a CPLEX7.5 license is needed for that. Moreover, for large tables, it can be an expensive computation, even more costly than the protection procedure. For these reasons, by default the lower bound is not computed. If we want to compute it, we can call

```
put_comp_lowbound2D(tab,TRUE).
```

Setting the second parameter of the above function to `FALSE` we can deactivate the computation of the lower bound. To know if the lower bounding procedure is activated or not, NF_CSP provides

```
get_comp_lowbound2D(tab).
```

If returns `TRUE` if activated, otherwise it returns `FALSE`. The lower bound obtained is returned by function

```
get_comp_lowbound2D(tab).
```

If the lower bounding procedure was not activated, the above function returns 0.

The solution provided by the lower bounding procedure can also be used as a starting point for the heuristics. By default, that solution is not considered, and the heuristics start from scratch. This behaviour can be modified with

```
put_use_lowbound2D(tab,uselb).
```

The lower bound solution will either be used or not in the heuristics if the second parameter of the above function is respectively `TRUE` or `FALSE`. To know the current situation about the usage of the lower bound solution, we can call

```
get_use_lowbound2D(tab).
```

It returns `TRUE` if the lower bound solution is going to be used, otherwise it returns `FALSE`. Clearly, if the lower bound solution is asked to be used, and the lower bound procedure was not activated, the heuristics will return an error code.

3.8 Auditing

NF_CSP implements an auditing phase for computing the lower and upper bounds than an external attacker could derive for the primary cells after the publication of the table. For two dimensional tables, these values can be computed by solving two minimum-cost network flows problems. This is done in NF_CSP either by CPLEX7.5 or PPRN. NF_CSP first attempts to use CPLEX7.5; if no license is available then it switches to PPRN. The auditing phase of NF_CSP was just developed for testing purposes, and it is not the most efficient procedure. It may be significantly improved by computing the two bounds through specialized maximum-flows algorithms.

The function that performs the auditing is

```
csp_auditing2D(tab,r,s).
```

Arguments `r` and `s` mean the range of primary cells considered (from `r` to `s`). If one just wants to audit the primary cell `i`, we'll set `r` and `s` to `i`.

After the auditing is performed, the lower and upper protection levels for primary `i` can be retrieved respectively by functions

```
get_pcelllpl2D(tab,i),
get_pcellupl2D(tab,i),
```

and

```
get_sorted_pcelllpl2D(tab,i),
get_sorted_pcellupl2D(tab,i)
```

(see Subsection 3.6 for an explanation of the differences between the plain and sorted variants). If x is the cell value of primary `i`, and l and u are the values returned respectively by the `get_pcelllpl2D(tab,i)` and `get_pcellupl2D(tab,i)` functions, the attacker knows that the real value of this primary is in the range $[x - l, x + u]$.

4 Interface routines

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

4.1 Creating and removing tables

- **Function:** `int create_table(TABLE2D **tab, int m, int n, int p)`

  Purpose: It creates and initializes a table of `m` rows, `n` columns and `p` primaries.

  Returns: 0 if the table was successfully created; otherwise (e.g., if not enough memory for allocating the table) it returns -1.

  Input arguments: `m` is the number of rows; `n` is the number of columns; `p` is the number of primary cells.

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

  Input/Output arguments: None

  Example:

```
TABLE2D *tab;
int ret;
ret= create_table(&tab,10,15,20); // creates a 10×15 table with 20 primaries
```

- **Function:** `void delete_table(TABLE2D **tab)`

  Purpose: Deletes a table, freeing its memory space.

  Returns: Nothing.

  Input arguments: None

  Output arguments: None

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

  Example:

```
TABLE2D *tab;
...
delete_table(&tab);
```

4.2 Entering table information

- **Function:** `void put_cellvalue2D(TABLE2D *tab, int row, int column, double value)`

  Purpose: It fills cell (`row`,`column`) with `value` without checking that `row` and `column` are within bounds.

  Returns: Nothing.

  Input arguments: `row` is the cell row; `column` is the cell column; `value` is the cell value.

  Output arguments: None.

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

  Example:

```
TABLE2D *tab;
...
put_cellvalue2D(tab,1,1,10.0); // cell (1,1) is 10.0
```

- **Function:** `void put_cellweight2D(TABLE2D *tab, int row, int column, double weight)`

  Purpose: It sets the weight of cell (`row`,`column`) without checking that `row` and `column` are within bounds.

  Returns: Nothing.

  Input arguments: `row` is the cell row; `column` is the cell column; `weight` is the cell weight.

  Output arguments: None.

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

  Example:

```
TABLE2D *tab;
...
put_cellweight2D(tab,1,1,2.5); // weight of cell (1,1) is 2.5
```

- **Function:** `void put_cellstatus2D(TABLE2D *tab, int row, int column, STATUS_CELL status)`

  Purpose: It sets the status of cell (`row`,`column`) without checking that `row` and `column` are within bounds. See Subsection 3.4 for a detailed explanation of the available status.

  Returns: Nothing.

  Input arguments: `row` is the cell row; `column` is the cell column; `status` is the cell status, which can be `PRIMARY`, `SECONDARY`, `PERMANENT` or `NONREMOVED`. The user should only call this routine to set `PERMANENT` cells.

  Output arguments: None.

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

  Example:

```
TABLE2D *tab;
...
put_cellstatus2D(tab,1,1,PERMANENT); // cell (1,1) is set PERMANENT
```

- **Function:** `void put_cells_0_permanent2D(TABLE2D *tab, char setperm)`

  Purpose: It sets if cells with a zero value must be automatically set permanent by the heuristics. If not changed through this function, the default is to consider zero cells permanent. See Subsection 3.4 for a detailed explanation.

  Returns: Nothing.

  Input arguments: `setperm` is a boolean char; if `TRUE`, zero cells will be set permanent, otherwise they will preserve their current status.

  Output arguments: None.

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

  Example:

```
TABLE2D *tab;
...
put_cells_0_permanent2D(tab,FALSE);
```

- **Function:** `void put_pcell2D(TABLE2D *tab, int pcell, int pi, int pj, double plpl, double pupl)`

  Purpose: It sets the information for the primary cell number `pcell`, without checking that `pcell` is within bounds.

  Returns: Nothing

  Input arguments: `pcell` is the primary cell number to be considered; `pi` and `pj` are the row and column position of this primary cell; `plpl` and `pupl` are the lower and upper protection levels required for this primary.

  Output arguments: None.

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

  Example:

```
TABLE2D *tab;
...
put_pcell2D(tab,3,7,9,10.5,20.1);
// cell (7,9) is the third primary, with required lower protection
// level 10.5 and required upper protection level 20.1
```

- **Function:** `void put_comp_lowbound2D(TABLE2D *tab, char comp_lb)`

  Purpose: It sets if the lower bound must or not be computed for table `tab`. See Subsection 3.7 for details.

  Returns: Nothing.

  Input arguments: `comp_lb` is a boolean char; if `TRUE`, the lower bound will be computed, otherwise it will not.

  Output arguments: None.

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

  Example:

```
TABLE2D *tab;
...
put_comp_lowbound2D(tab,TRUE);
```

- **Function:** `void put_use_lowbound2D(TABLE2D *tab, char use_lb)`

  Purpose: It sets if the solution obtained by the lower bounding procedure must or not be used in the heuristics for table `tab`. See Subsection 3.7 for details.

  Returns: Nothing.

  Input arguments: `use_lb` is a boolean char; if `TRUE`, the solution provided by the lower bounding procedure will be used in the heuristics, otherwise it will not.

  Output arguments: None.

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

  Example:

```
TABLE2D *tab;
...
put_use_lowbound2D(tab,TRUE);
```

- **Function:** `void put_typeweights2D(TABLE2D *tab, TYPE_WEIGHTS tweights)`

  Purpose: It sets the type of cell weights to be used during the protection heuristics. If not changed through this function, the default type is `VALUE`.

  Returns: Nothing.

  Input arguments: `tweights` is the type of weights. It can be `VALUE`, `LOG`, `BIN` or `CUSTOM`. See Subsection 3.5 for an explanation of each type of weights.

  Output arguments: None.

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

  Example:

```
TABLE2D *tab;
...
put_typeweights2D(tab,CUSTOM);
```

- **Function:** `void put_typemorder2D(TABLE2D *tab, TYPE_MERIT_ORDER morder)`

  Purpose: It sets the type of merit order for the primary cells (i.e., the order they will be processed by the heuristics). If not changed through this function, the default order is `NORMAL`.

  Returns: Nothing.

  Input arguments: `morder` is the type of merit order. It can be `NORMAL`, `ASCENDENT` or `DESCENDENT`. See Subsection 3.6 for an explanation of each type of merit order.

  Output arguments: None.

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

  Example:

```
TABLE2D *tab;
...
put_typemorder2D(tab,DESCENDENT);
```

- **Function:** `void put_heuristic2D(TABLE2D *tab, HEURISTIC heuristic)`

  Purpose: It sets which of the two available heuristics will be used for protecting the table. If not changed through this function, the default heuristic is the 0-1 flows. See Subsection 3.2 for more details.

  Returns: Nothing.

  Input arguments: `heuristic` is the heuristic to be used. It can take values `FLOWS_BIN` (0-1 flows heuristic) or `FLOWS_N` (n-flows heuristic).

  Output arguments: None.

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

  Example:

```
TABLE2D *tab;
...
put_heuristic2D(tab,FLOWS_N);
```

- **Function:** `void put_solver2D(TABLE2D *tab, SOLVER solver)`

  Purpose: It sets which of the three available solvers will be used for the subproblems generated by the heuristics. If not changed through this function, the default solver is Dijkstra. See Subsection 3.2 for more details.

  Returns: Nothing.

  Input arguments: `solver` is the solver to be used. It can take values `DIJKSTRA`, `CPLEX` or `PPRN`.

  Output arguments: None.

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

  Example:

```
TABLE2D *tab;
...
put_solver2D(tab,PPRN);
```

- **Function:** `void put_cost_type2D(TABLE2D *tab, COST_TYPE cost_type)`

  Purpose: It sets the type of arc costs for the subproblems of the 0-1 flows heuristic. If not changed through this function, the default type is `FASTER_WORSER`.

  Returns: Nothing.

  Input arguments: `cost_type` is the type of costs. It can be `FASTER_WORSER` or `SLOWER_BETTER`. See Subsection 3.3 for an explanation of each type of costs.

  Output arguments: None.

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

  Example:

```
TABLE2D *tab;
...
put_cost_type2D(tab,SLOWER_BETTER);
```

4.3 Retrieving table information

- **Function:** `int get_nrows2D(TABLE2D *tab)`

  Purpose: It provides the number of rows of table `tab`.

  Returns: The number of rows.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D * tab;
...
int nrows= get_nrows2D(tab);
```

- **Function:** `int get_ncolumns2D(TABLE2D *tab)`

  Purpose: It provides the number of columns of table `tab`.

  Returns: The number of columns.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D * tab;
...
int ncols= get_ncolumns2D(tab);
```

- **Function:** `int get_npcells2D(TABLE2D *tab)`

  Purpose: It provides the number of primary cells of table `tab`.

  Returns: The number of primary cells.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D * tab;
...
int p= get_npcells2D(tab);
```

- **Function:** `int get_nseccells2D(TABLE2D *tab)`

  Purpose: It provides the number of secondary cells of table `tab`.

  Returns: The number of secondary cells.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D * tab;
...
int s= get_nseccells2D(tab);
```

- **Function:** `double get_cellvalue2D(TABLE2D *tab, int row, int column)`

  Purpose: It provides the value of cell (`row`,`column`) without checking that `row` and `column` are within bounds.

  Returns: The value of cell (`row`,`column`).

  Input arguments: `tab` is the table; `row` is the cell row; `column` is the cell column.

Output arguments: None.

Input/Output arguments: None.

Example:

```
TABLE2D *tab;
...
double v= get_cellvalue2D(tab,3,4); // value of cell (3,4)
```

- **Function:** `double get_cellweight2D(TABLE2D *tab, int row, int column)`

  Purpose: It provides the weight of cell (`row`,`column`) without checking that `row` and `column` are within bounds.

  Returns: The weight of cell (`row`,`column`).

  Input arguments: `tab` is the table; `row` is the cell row; `column` is the cell column.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
...
double v= get_cellweight2D(tab,3,4); // weight of cell (3,4)
```

- **Function:** `STATUS_CELL get_cellstatus2D(TABLE2D *tab, int row, int column)`

  Purpose: It provides the status of cell (`row`,`column`) without checking that `row` and `column` are within bounds.

  Returns: The status of cell (`row`,`column`) (which can be `PRIMARY`, `SECONDARY`, `PERMANENT` or `NONREMOVED`). See Subsection 3.4 for an explanation of each status.

  Input arguments: `tab` is the table; `row` is the cell row; `column` is the cell column.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
...
STATUS_CELL status= get_cellstatus2D(tab,3,4); // status of cell (3,4)
```

- **Function:**

```
char is_cell_primary2D(TABLE2D *tab, int row, int column)
char is_cell_secondary2D(TABLE2D *tab, int row, int column)
char is_cell_permanent2D(TABLE2D *tab, int row, int column)
char is_cell_nonremoved2D(TABLE2D *tab, int row, int column)
```

  Purpose: Auxiliary functions to know if cell (`row`,`column`) has a particular status.

  Returns: `TRUE` if the cell has the particular status; otherwise, `FALSE`.

Input arguments: `tab` is the table; `row` is the cell row; `column` is the cell column.

Output arguments: None.

Input/Output arguments: None.

Example:

```
TABLE2D *tab;
...
if (is_cell_secondary2D(tab,3,4)) {
...  // treatment for cell (3,4) if secondary
}
```

- **Function:** `char get_cells_0_permanent2D(TABLE2D *tab)`

  Purpose: To know if cells with a zero value must be automatically set permanent by the heuristics. See Subsection 3.4 for a detailed explanation.

  Returns: `TRUE` if zero cells will be set permanent; otherwise, `FALSE`.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
...
if (get_cells_0_permanent2D(tab)) {
...
}
```

- **Function:**
  `void get_pcell2D(TABLE2D *tab, int pcell, int * pi, int* pj, double *lpl, double *upl)`
  `void get_sorted_pcell2D(TABLE2D *tab, int pcell, int * pi, int* pj, double *lpl, double *upl)`

  Purpose: These functions provide the location and required protection levels of the primary cell number `pcell`. The first function considers primary cells are in the order provided by the user. The second function considers the current merit order. See Subsection 3.6 for details.

  Returns: Nothing.

  Input arguments: `tab` is the table; `pcell` is the primary cell number.

  Output arguments: `pi` and `pj` are respectively the row and column of the primary cell; `lpl` and `upl` are respectively the required lower and upper protection levels.

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
int pi,pj;
double lpl,upl;
```

```
...
// location and protection required for the third primary cell
// entered by the user
get_pcell2D(tab,3,&pi,&pj,&lpl,&upl);
...
// location and protection required for the third primary cell
// according to the merit order
get_sorted_pcell2D(tab,3,&pi,&pj,&lpl,&upl);
```

- **Function:**
  `double get_pcelllpl2D(TABLE2D *tab, int pcell)`
  `double get_sorted_pcelllpl2D(TABLE2D *tab, int pcell)`

  Purpose: These functions provide the lower protection level currently obtained by the primary cell number `pcell`. The first function considers primary cells are in the order provided by the user. The second function considers the current merit order. See Subsection 3.6 for details.

  Returns: Current lower protection level of the primary cell.

  Input arguments: `tab` is the table; `pcell` is the primary cell number.

  Output arguments: None

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
...
// current lower protection obtained for the third primary cell
// entered by the user
double lpl= get_pcelllpl2D(tab,3);
...
// current lower protection obtained for the third primary cell
// according to the merit order
double lpl= get_sorted_pcelllpl2D(tab,3);
```

- **Function:**
  `double get_pcellupl2D(TABLE2D *tab, int pcell)`
  `double get_sorted_pcellupl2D(TABLE2D *tab, int pcell)`

  Purpose: These functions provide the upper protection level currently obtained by the primary cell number `pcell`. The first function considers primary cells are in the order provided by the user. The second function considers the current merit order. See Subsection 3.6 for details.

  Returns: Current upper protection level of the primary cell.

  Input arguments: `tab` is the table; `pcell` is the primary cell number.

  Output arguments: None

  Input/Output arguments: None.

Example:

```
TABLE2D *tab;
...
// current upper protection obtained for the third primary cell
// entered by the user
double upl= get_pcellupl2D(tab,3);
...
// current upper protection obtained for the third primary cell
// according to the merit order
double upl= get_sorted_pcellupl2D(tab,3);
```

- **Function:** `char get_comp_lowbound2D(TABLE2D *tab)`

  Purpose: To know if the lower bound will or not be computed by the heuristics. See Subsection 3.7 for details.

  Returns: `TRUE` if the lower bound has to be computed; otherwise, `FALSE`.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
...
if (get_comp_lowbound2D(tab)) {
...
}
```

- **Function:** `char get_use_lowbound2D(TABLE2D *tab)`

  Purpose: To know if the solution computed by the lower bounding procedure will or not be used by the heuristics. See Subsection 3.7 for details.

  Returns: `TRUE` if the lower bound solution will be used; otherwise, `FALSE`.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
...
if (get_use_lowbound2D(tab)) {
...
}
```

- **Function:** `double get_lowerbound2D(TABLE2D *tab)`

Purpose: It provides the lower bound computed by the lower bounding procedure. See Subsection 3.7 for details.

Returns: The lower bound.

Input arguments: `tab` is the table.

Output arguments: None.

Input/Output arguments: None.

Example:

```
TABLE2D *tab;
...
double lb= get_lowerbound2D(tab);
```

- **Function:** `TYPE_WEIGHTS get_typeweights2D(TABLE2D *tab)`

  Purpose: It provides the type of cell weights to be used during the protection heuristics.

  Returns: `VALUE`, `LOG`, `BIN` or `CUSTOM`. See Subsection 3.5 for an explanation of each type of weights.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
...
TYPE_WEIGHTS tweights= get_typeweights2D(tab);
```

- **Function:** `TYPE_MERIT_ORDER get_typemorder2D(TABLE2D *tab)`

  Purpose: It provides the type of merit order for the primary cells (i.e., the order they will be processed by the heuristics).

  Returns: `NORMAL`, `ASCENDENT` or `DESCENDENT`. See Subsection 3.6 for an explanation of each type of merit order.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
...
TYPE_MERIT_ORDER tmorder= get_typemorder2D(tab);
```

- **Function:** `HEURISTIC get_heuristic2D(TABLE2D *tab)`

  Purpose: It provides which of the two available heuristics will be used for protecting the table.

Returns: `FLOWS_BIN` (0-1 flows heuristic) or `FLOWS_N` (n-flows heuristic). See Subsection 3.2 for details on each heuristic.

Input arguments: `tab` is the table.

Output arguments: None.

Input/Output arguments: None.

Example:

```
TABLE2D *tab;
...
HEURISTIC heur= get_heuristic2D(tab);
```

- **Function:** `SOLVER get_solver2D(TABLE2D *tab)`

  Purpose: It provides which of the three available solvers will be used for the subproblems generated by the heuristics.

  Returns: `DIJKSTRA`, `CPLEX` or `PPRN`. See Subsection 3.2 for details on each solver.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
...
SOLVER solver= get_solver2D(tab);
```

- **Function:** `COST_TYPE get_cost_type2D(TABLE2D *tab)`

  Purpose: It provides the type of arc costs for the subproblems of the 0-1 flows heuristic.

  Returns: `FASTER_WORSER` or `SLOWER_BETTER`. See Subsection 3.3 for an explanation of each type of costs.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
...
COST_TYPE tcost= get_cost_type2D(tab);
```

- **Function:** `double get_valuesuppressed2D(TABLE2D *tab)`

  Purpose: It provides the overall cell value suppressed by the heuristic for the protection of the table, both for primaries and secondaries.

  Returns: The cell value suppressed (primaries and secondaries).

  Input arguments: `tab` is the table.

Output arguments: None.

Input/Output arguments: None.

Example:

```
TABLE2D *tab;
...
double valsup= get_valuesuppressed2D(tab)
```

- **Function:** `double get_weightsuppressed2D(TABLE2D *tab)`

  Purpose: It provides the overall cell weight suppressed by the heuristic for the protection of the table, both for primaries and secondaries.

  Returns: The cell weight suppressed (primaries and secondaries).

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
...
double weightsup= get_weightsuppressed2D(tab)
```

- **Function:** `int get_numnfproblems2D(TABLE2D *tab)`

  Purpose: It provides the number of network flows subproblems (either shortest-paths or minimum-cost network flows) solved by the heuristic for the protection of the table.

  Returns: The number of network flows subproblems.

  Input arguments: `tab` is the table.

  Output arguments: None.

  Input/Output arguments: None.

  Example:

```
TABLE2D *tab;
...
int nf= get_numnfproblemes2D(tab);
```

4.4 Executing heuristics and other procedures

- **Function:** `int csp_heur2D(TABLE2D *tab);`

  Purpose: It protects a table. The type of heuristic, type of solver, lower bounding information, etc., must have been previously set by the user; otherwise, the default ones will be used.

Returns:

0 if the table was successfully protected;
-1 if not enough memory for protecting the table;
-2 if some network flows subproblem was detected as infeasible;
-3 if some error was found defining a network flows subproblem;
-4 if some error happened initializing the network flows subproblem;
-5 if some network flows subproblem produced some error during its solution;
-7 if there are problems with the CPLEX license;
-8 if some error happened creating the CPLEX network object;
-9 if some error was found in the lower bounding procedure;
-50 if the combination heuristic/solver set by the user is not appropriate;
-51 if some error happened when ordering the primary cells by merit order;
-53 if the package was compiled without CPLEX75 support and CPLEX is the solver to be used;
-54 if the lower bound wants to be computed and the package was compiled without CPLEX75 support;
-55 if the lower bound solution wants to be used without being computed.

Return codes -2 to -9 are mainly associated to the solvers. Return codes -50 to -55 are mainly due to bad user settings.

Input arguments: None.

Output arguments: None.

Input/Output arguments: On input, `tab` is the table to be protected. On output, `tab` is the protected table.

Example:

```
TABLE2D *tab;
...
int retstat= csp_heur2D(tab);
```

- **Function:** `int csp_auditing2D(TABLE2D *tab, int ini, int fin)`

Purpose: It performs the auditing (i.e., lower and upper protection levels obtained) for the primary cells in the range `ini`...`fin`, once the table was successfully protected. This function considers the order provided by the user for primary cells (not the merit order). This auditing was designed just for testing purposes, and it does not implement the most efficient algorithm. By default it attempts to use CPLEX; if a license is not available it switches to PPRN.

Returns:

0 if the primaries were successfully audited;
-1 if not enough memory for auditing the primaries;
-2 if `ini` or `ifi` are out of bounds;
-5 if some error was found solving the minimum-cost network flows subproblem for the lower or upper protection level of some primary.

Input arguments: `ini` is the first primary to be protected; `fin` is the last primary to be protected.

Output arguments: None.

Input/Output arguments: On input, `tab` is a protected table. On output, `tab` is a protected and audited (only for cells in range `ini`...`fin`) table.

Example:

```
TABLE2D *tab;
...
//auditing for all the primaries
int retstat= csp_auditing2D(tab,0,get_npcells2D(tab)-1);
```

References

[1] Ahuja, R.K, Magnanti, T.L., Orlin, J.B., *Network Flows*, Prentice Hall (1993).

[2] Castro, J., PPRN 1.0, User's Guide, Technical report DR 94/06 Dept. of Statistics and Operations Research, Universitat Politècnica de Catalunya, Barcelona, Spain, 1994.

[3] Castro, J., Network flows heuristics for complementary cell suppression: an empirical evaluation and extensions, in *LNCS 2316, Inference Control in Statistical Databases*, J. Domingo-Ferrer (Ed), (2002) 59–73.

[4] Castro, J., Nabona, N. An implementation of linear and nonlinear multicommodity network flows. *European Journal of Operational Research* 92, (1996) 37–53.

[5] Cox, L.H., Network models for complementary cell suppression. *J. Am. Stat. Assoc.* 90, (1995) 1453–1462.

[6] ILOG CPLEX, *ILOG CPLEX 7.5 Reference Manual Library*, ILOG, (2000).

[7] Kelly, J.P., Golden, B.L, Assad, A.A., Cell Suppression: disclosure protection for sensitive tabular data, *Networks* 22, (1992) 28–55.

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	Network Flows heuristics for Tau-Argus (2D tables)
Package Owner	Dept. of Statistics and Operations Research Universitat Politècnica de Catalunya Barcelona
Contact Person	Jordi Castro, `jcastro@eio.upc.es`
Starting Date	January 2001
Ending Date	December 2002
Programming Environment	Linux and gcc, ported to Windows and MS-Visual C++

B Quantitative information

Total number of files	13 files
Total number of lines	3879 lines
Total size	111574 bytes

C List of files (alphabetical order)

	File Name	**Location**	**Lines**	**Bytes**
1	c_qsort_comp.c	Tau-Argus-UPC\csp_nf\src\2Dtables\	18	317
2	c_qsort_comp.h	Tau-Argus-UPC\csp_nf\src\2Dtables\	16	306
3	csp_auditing2D.cpp	Tau-Argus-UPC\csp_nf\src\2Dtables\	199	5403
4	csp_heur2D.cpp	Tau-Argus-UPC\csp_nf\src\2Dtables\	188	5319
5	csp_heur2D.h	Tau-Argus-UPC\csp_nf\src\2Dtables\	180	6105
6	csp_heur_flows_01.cpp	Tau-Argus-UPC\csp_nf\src\2Dtables\	1036	32167
7	csp_heur_flows_n.cpp	Tau-Argus-UPC\csp_nf\src\2Dtables\	526	16204
8	csp_lower_bound.cpp	Tau-Argus-UPC\csp_nf\src\2Dtables\	439	10424
9	csp_solve_cplex.cpp	Tau-Argus-UPC\csp_nf\src\2Dtables\	81	2172
10	csp_solve_dijkstra.cpp	Tau-Argus-UPC\csp_nf\src\2Dtables\	143	3204
11	csp_solve_pprn.cpp	Tau-Argus-UPC\csp_nf\src\2Dtables\	344	9905
12	csp_table2D.cpp	Tau-Argus-UPC\csp_nf\src\2Dtables\	221	5279
13	csp_table2D.h	Tau-Argus-UPC\csp_nf\src\2Dtables\	488	14769
14	**TOTAL**		3879	111574

D List of routines (alphabetical order)

1. check_protection_levels_for_lbp2D(TABLE2D *tab)
2. compute_weights2D(TABLE2D *tab)
3. comp_ascendent(const void *a, const void *b)
4. comp_descendent(const void *a, const void *b)
5. create_table(TABLE2D **ptab, int m, int n, int p)
6. csp_add_arc_cycle(CYCLE *cycle, int row, int col, SENSE sense)
7. csp_allocateNF(int nnu, int nar, NFPro **pnf)
8. csp_arc2col(int arc, int ntab)
9. csp_arc2row(int arc, int ntab)
10. csp_auditing2D(TABLE2D *tab, int ini, int fin)
11. csp_check_add_cycle(CYCLE *cycle, int ncells)
12. csp_create_cycle(CYCLE **cycle, int ini_max_card)
13. csp_delete_cycle(CYCLE **cycle)
14. csp_empty_cycle(CYCLE *cycle)
15. csp_freeNF(NFPro **nf)
16. csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot)
17. csp_GetCycleNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j, CYCLE *cycle, double *pos_min_cycle, double *neg_min_cycle)
18. csp_get_arc_cycle(CYCLE *cycle, int i, int *row, int *col, SENSE *sense)
19. csp_get_card_cycle(CYCLE *cycle)
20. csp_heur2D(TABLE2D *tab)
21. csp_heur2D_Cox(TABLE2D *tab)
22. csp_heur2D_KGA(TABLE2D *tab)
23. csp_ij2Xp(int ntab, int i, int j)
24. csp_IniDijkstra(TABLE2D *tab, NFPro *Ncp, DSP *Dsp)
25. csp_iniKGA_NF(TABLE2D *tab, NFPro *Ncp)
26. csp_iniNcp(TABLE2D *tab, NFPro *Ncp)
27. csp_lower_bound(TABLE2D *tab)
28. csp_resetNF(TABLE2D *tab, NFPro *Ncp, CYCLE *cycles_target, int target_i, int target_j)

29. csp_reset_KGA_problem_target(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, TYPE_PROT type_prot)
30. csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp)
31. csp_set_costsNF_v2(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp)
32. csp_set_KGA_problem_target(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, double prot_req, TYPE_PROT type_prot)
33. csp_set_targetNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j)
34. csp_SolveNF_CPLEX75(NFPro *Ncp, CPXENVptr env, CPXNETptr net)
35. csp_SolveNF_Dijkstra(DSP *Dsp)
36. csp_SolveNF_PPRN_0lb(NFPro *Ncp)
37. csp_SolveNF_PPRN_reduced(NFPro *Ncp)
38. delete_table(TABLE2D **ptab)
39. get_cells_0_permanent2D(TABLE2D *tab)
40. get_cellstatus2D(TABLE2D *tab, int row, int column)
41. get_cellvalue2D(TABLE2D *tab, int row, int column)
42. get_cellweight2D(TABLE2D *tab, int row, int column)
43. get_comp_lowbound2D(TABLE2D *tab)
44. get_index_of_primary2D(TABLE2D *tab, int row, int col)
45. get_lowerbound2D(TABLE2D *tab)
46. get_ncolumns2D(TABLE2D *tab)
47. get_npcells2D(TABLE2D *tab)
48. get_nrows2D(TABLE2D *tab)
49. get_nseccells2D(TABLE2D *tab)
50. get_numnfproblems2D(TABLE2D *tab)
51. get_pcell2D(TABLE2D *tab, int pcell, int * pi, int* pj, double *lpl, double *upl)
52. get_pcelllpl2D(TABLE2D *tab, int pcell)
53. get_pcellupl2D(TABLE2D *tab, int pcell)
54. get_protlev_le_cellvalue2D(TABLE2D *tab)
55. get_sorted_pcell2D(TABLE2D *tab, int pcell, int * pi, int* pj, double *lpl, double *upl)
56. get_sorted_pcelllpl2D(TABLE2D *tab, int pcell)

57. get_sorted_pcellupl2D(TABLE2D *tab, int pcell)
58. get_use_lowbound2D(TABLE2D *tab)
59. get_valuesuppressed2D(TABLE2D *tab)
60. get_weightsuppressed2D(TABLE2D *tab)
61. is_cell_nonremoved2D(TABLE2D *tab, int row, int column)
62. is_cell_permanent2D(TABLE2D *tab, int row, int column)
63. is_cell_primary2D(TABLE2D *tab, int row, int column)
64. is_cell_secondary2D(TABLE2D *tab, int row, int column)
65. objcostNF(NFPro *nf)
66. put_cellstatus2D(TABLE2D *tab, int row, int column, STATUS_CELL status)
67. put_cells_0_permanent2D(TABLE2D *tab, char setperm)
68. put_cellvalue2D(TABLE2D *tab, int row, int column, double value)
69. put_cellweight2D(TABLE2D *tab, int row, int column, double weight)
70. put_comp_lowbound2D(TABLE2D *tab, char comp_lb)
71. put_cost_type2D(TABLE2D *tab, COST_TYPE cost_type)
72. put_heuristic2D(TABLE2D *tab, HEURISTIC heuristic)
73. put_index_of_primary2D(TABLE2D *tab, int row, int col, int pcell)
74. put_lowerbound2D(TABLE2D *tab, double lb)
75. put_numnfproblems2D(TABLE2D *tab, int nnf)
76. put_pcell2D(TABLE2D *tab, int pcell, int pi, int pj, double plpl, double pupl)
77. put_pcelllpl2D(TABLE2D *tab, int pcell, double lpl)
78. put_pcellupl2D(TABLE2D *tab, int pcell, double upl)
79. put_protlev_le_cellvalue2D(TABLE2D *tab, char boolean)
80. put_solver2D(TABLE2D *tab, SOLVER solver)
81. put_sorted_pcelllpl2D(TABLE2D *tab, int pcell, double lpl)
82. put_sorted_pcellupl2D(TABLE2D *tab, int pcell, double upl)
83. put_typemorder2D(TABLE2D *tab, TYPE_MERIT_ORDER morder)
84. put_typeweights2D(TABLE2D *tab, TYPE_WEIGHTS tweights)
85. put_use_lowbound2D(TABLE2D *tab, char use_lb)
86. put_valuesuppressed2D(TABLE2D *tab, double v)

87. put_weightsuppressed2D(TABLE2D *tab, double v)

88. set_0_value_cells_permanent2D(TABLE2D *tab)

89. show_infoarcs(NFPro *nf)

90. sort_primarycells2D(TABLE2D *tab)

E Routines description (alphabetical order)

Routine Name	check_protection_levels_for_lbp2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.cpp
Routine Size	22 lines
Routine Comment	
	Check if some (upper) protection level is higher than cell value; this is required to know if the improved lower bounding procedure can or not be used
Parent Routines	
	• csp_heur2D(TABLE2D *tab)
Child Routines	
	• get_cellvalue2d(table2d *tab, int row, int column) • get_npcells2d(table2d *tab) • get_pcell2d(table2d *tab, int pcell, int * pi, int* pj, double *lpl, double *upl) • put_protlev_le_cellvalue2d(table2d *tab, char boolean)

Routine Name	compute_weights2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.cpp
Routine Size	42 lines
Routine Comment	
	Compute weights according to type_weights; cell values are considered in absolute value to avoid problems with logarithms and negative costs in network flows problems
Parent Routines	
	• csp_heur2D(TABLE2D *tab)
Child Routines	
	• get_cellvalue2d(table2d *tab, int row, int column) • get_ncolumns2d(table2d *tab) • get_nrows2d(table2d *tab) • put_cellweight2d(table2d *tab, int row, int column, double weight)

Routine Name	comp_ascendent(const void *a, const void *b)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\c_qsort_comp.c
Routine Size	5 lines
Routine Comment	
Parent Routines	
Child Routines	

Routine Name	comp_descendent(const void *a, const void *b)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\c_qsort_comp.c
Routine Size	8 lines
Routine Comment	
Parent Routines	
Child Routines	

Routine Name	create_table(TABLE2D **ptab, int m, int n, int p)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.cpp
Routine Size	79 lines
Routine Comment	
	Creates and initializes 2D table of m rows, n columns and p primaries returns 0 if everything goes fine return -1 if not enough memory
Parent Routines	
Child Routines	
	• delete_table(table2d **ptab) • sort_primarycells2d(table2d *tab)

Routine Name	csp_add_arc_cycle(CYCLE *cycle, int row, int col, SENSE sense)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.h
Routine Size	10 lines
Routine Comment	
	Add arc to cycle without checking bounds; a prior call to csp_check_add_cycle() is assumed
Parent Routines	
	• csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_GetCycleNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j, CYCLE *cycle, double *pos_min_cycle, double *neg_min_cycle) • csp_heur2D_Cox(TABLE2D *tab)
Child Routines	
	• csp_check_add_cycle(cycle *cycle, int ncells)

Routine Name	csp_allocateNF(int nnu, int nar, NFPro **pnf)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.cpp
Routine Size	32 lines
Routine Comment	
	Allocates network flow problem Input/Output parameters: nnu: number of nodes nar: number of arcs NFPro **pnf : NF structure Returns: 0: if everything was fine -1: if not enough memory
Parent Routines	
	• csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	
	• csp_freenf(nfpro **nf)

Routine Name	csp_arc2col(int arc, int ntab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.h
Routine Size	9 lines
Routine Comment	
	Number of column-cell, given the arc in network flow problem ntab : number of cols. of the table arc : arc number (0..2*ntab*mtab-1)
Parent Routines	
	• csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_GetCycleNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j, CYCLE *cycle, double *pos_min_cycle, double *neg_min_cycle) • csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp)
Child Routines	

Routine Name	csp_arc2row(int arc, int ntab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.h
Routine Size	8 lines
Routine Comment	
	Number of row-cell, given the arc in network flow problem ntab : number of cols. of the table arc : arc number (0..2*ntab*mtab-1)
Parent Routines	
	• csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_GetCycleNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j, CYCLE *cycle, double *pos_min_cycle, double *neg_min_cycle) • csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp)
Child Routines	

Routine Name	csp_auditing2D(TABLE2D *tab, int ini, int fin)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_auditing2D.cpp
Routine Size	191 lines
Routine Comment	
	Auditing for 2D CSP. Input parameters: ini: initial primary cell to audit (according to non-sorted order) fin: final primary cell to audit (according to non-sorted order) Input/output parameters: tab: table to be audited, previously protected by a call to csp_heur2D Returns: 0: if everything was fine -1: if not enough memory -2: if ini or fin out of bounds -3: problems with cplex license (not applicable) -4: error creating cplex network object (not applicable) -5: error solving problem
Parent Routines	
Child Routines	
	• csp_allocatenf(int nnu, int nar, nfpro **pnf) • csp_freenf(nfpro **nf) • csp_solvenf_cplex75(nfpro *ncp, cpxenvptr env, cpxnetptr net) • csp_solvenf_pprn_0lb(nfpro *ncp) • get_cellvalue2d(table2d *tab, int row, int column) • get_ncolumns2d(table2d *tab) • get_npcells2d(table2d *tab) • get_nrows2d(table2d *tab) • get_pcell2d(table2d *tab, int pcell, int * pi, int* pj, double *lpl, double *upl) • get_pcelllpl2d(table2d *tab, int pcell) • get_pcellupl2d(table2d *tab, int pcell) • put_pcelllpl2d(table2d *tab, int pcell, double lpl) • put_pcellupl2d(table2d *tab, int pcell, double upl)

Routine Name	csp_check_add_cycle(CYCLE *cycle, int ncells)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.h
Routine Size	32 lines
Routine Comment	
	Check if ncells can be added to cycle without exceeding its current storage; if not, it reallocates the cycle. returns 0 if everything goes fine returns -1 if not enough memory for reallocation
Parent Routines	
	• csp_add_arc_cycle(CYCLE *cycle, int row, int col, SENSE sense) • csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_GetCycleNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j, CYCLE *cycle, double *pos_min_cycle, double *neg_min_cycle) • csp_heur2D_Cox(TABLE2D *tab)
Child Routines	

Routine Name	csp_create_cycle(CYCLE **cycle, int ini_max_card)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.cpp
Routine Size	24 lines
Routine Comment	
	Create cycle with initially ini_max_card positions Input/Output parameters: CYCLE: ** to cycle structure ini_max_card: initial storage Returns: 0: if everything was fine -1: if not enough memory
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	
	• csp_delete_cycle(cycle **cycle)

Routine Name	csp_delete_cycle(CYCLE **cycle)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.cpp
Routine Size	23 lines
Routine Comment	
	Deletes a cycle Input/Output parameters: CYCLE cycle: ** to the cycle
Parent Routines	
	• csp_create_cycle(CYCLE **cycle, int ini_max_card) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	csp_empty_cycle(CYCLE *cycle)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.h
Routine Size	7 lines
Routine Comment	
	Empties cycle
Parent Routines	
	• csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_GetCycleNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j, CYCLE *cycle, double *pos_min_cycle, double *neg_min_cycle) • csp_heur2D_Cox(TABLE2D *tab)
Child Routines	

Routine Name	csp_freeNF(NFPro **nf)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.cpp
Routine Size	21 lines
Routine Comment	
	Frees the allocated memory for the network flow problem Input/Output parameters: NFPro nf : ** to the data structure of problem
Parent Routines	
	• csp_allocateNF(int nnu, int nar, NFPro **pnf) • csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur_flows_n.cpp
Routine Size	59 lines
Routine Comment	
	Finds the cycle in the current optimal solution of NF, and makes a first update of protections of primary cells in cycle. Input parameters: TABLE2D *tab : table NFPro *Ncp : NF problem int target_i: row number of target cell. int target_j: col number of target cell. TYPE_PROT type_prot: type of problem being solved (LOWER or UPPER) Output parameters: CYCLE *cycle: cells in cycle. double *min_cycle : minimum a_ij of cells in cycle (including target) Returns: 0: if everything was fine -1: if not enough memory
Parent Routines	
	• csp_heur2D_KGA(TABLE2D *tab)
Child Routines	
	• csp_add_arc_cycle(cycle *cycle, int row, int col, sense sense) • csp_arc2col(int arc, int ntab) • csp_arc2row(int arc, int ntab) • csp_check_add_cycle(cycle *cycle, int ncells) • csp_empty_cycle(cycle *cycle) • get_cellvalue2d(table2d *tab, int row, int column) • get_index_of_primary2d(table2d *tab, int row, int col) • get_ncolumns2d(table2d *tab) • get_pcelllpl2d(table2d *tab, int pcell) • get_pcellupl2d(table2d *tab, int pcell) • is_cell_primary2d(table2d *tab, int row, int column) • put_pcelllpl2d(table2d *tab, int pcell, double lpl) • put_pcellupl2d(table2d *tab, int pcell, double upl)

Routine Name	csp_GetCycleNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j, CYCLE *cycle, double *pos_min_cycle, double *neg_min_cycle)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur_flows_01.cpp
Routine Size	98 lines
Routine Comment	
	Finds the cycle in the current optimal solution of NF Input parameters: TABLE2D *tab : table NFPro *Ncp : NF problem DSP *Dsp : shortest path problem int target_i: row number of target cell. int target_j: col number of target cell. Output parameters: CYCLE *cycle: cells in cycle. double *pos_min_cycle : minimum a_ij of sense POS in the cycle (including target) double *neg_min_cycle : minimum a_ij of sense NEG in the cycle (including target) Returns: 0: if everything was fine -1: if not enough memory
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab)
Child Routines	
	• csp_add_arc_cycle(cycle *cycle, int row, int col, sense sense) • csp_arc2col(int arc, int ntab) • csp_arc2row(int arc, int ntab) • csp_check_add_cycle(cycle *cycle, int ncells) • csp_empty_cycle(cycle *cycle) • csp_ij2xp(int ntab, int i, int j) • get_cellvalue2d(table2d *tab, int row, int column) • get_ncolumns2d(table2d *tab)

Routine Name	csp_get_arc_cycle(CYCLE *cycle, int i, int *row, int *col, SENSE *sense)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.h
Routine Size	9 lines
Routine Comment	
	Get arc of cycle without checking bounds
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_resetNF(TABLE2D *tab, NFPro *Ncp, CYCLE *cycles_target, int target_i, int target_j) • csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • csp_set_costsNF_v2(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp)
Child Routines	

Routine Name	csp_get_card_cycle(CYCLE *cycle)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.h
Routine Size	6 lines
Routine Comment	
	Get cardinality of cycle
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_resetNF(TABLE2D *tab, NFPro *Ncp, CYCLE *cycles_target, int target_i, int target_j) • csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • csp_set_costsNF_v2(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp)
Child Routines	

Routine Name	csp_heur2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.cpp
Routine Size	53 lines
Routine Comment	
	Heuristics for 2D CSP Input/output parameters: tab: table to be protected Returns: -50: the combination heuristic/solver is not approppriate -51: error sorting primary cells by merit order -53: code compiled without Cplex library; you can not use Cplex -54: code compiled without Cplex library; lower bounding procedure not available -55: lower bound solution can not be used if not computed other: error code returned by the heuristic (0 if everything goes fine)
Parent Routines	
Child Routines	
	• check_protection_levels_for_lbp2d(table2d *tab) • compute_weights2d(table2d *tab) • csp_heur2d_cox(table2d *tab) • csp_heur2d_kga(table2d *tab) • get_cells_0_permanent2d(table2d *tab) • get_comp_lowbound2d(table2d *tab) • get_use_lowbound2d(table2d *tab) • set_0_value_cells_permanent2d(table2d *tab) • sort_primarycells2d(table2d *tab)

Routine Name	csp_heur2D_Cox(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur_flows_01.cpp
Routine Size	398 lines
Routine Comment	
	Cox-based heuristics for 2D CSP Input/output parameter: tab: table to be protected (input); protected table (output) Returns: 0: if everything was fine -1: if not enough memory -2: if problem detected as infeasible (no solution was found) -3: error defining NF problem -4: error initializing NF problem -5: error solving NF problem -7: problems with cplex license -8: error creating cplex network object -9: error in lower bounding procedure
Parent Routines	
	• csp_heur2D(TABLE2D *tab)
Child Routines	
	• csp_add_arc_cycle(cycle *cycle, int row, int col, sense sense) • csp_allocatenf(int nnu, int nar, nfpro **pnf) • csp_check_add_cycle(cycle *cycle, int ncells) • csp_create_cycle(cycle **cycle, int ini_max_card) • csp_delete_cycle(cycle **cycle) • csp_empty_cycle(cycle *cycle) • csp_freenf(nfpro **nf) • csp_getcyclenf(table2d *tab, nfpro *ncp, dsp *dsp, int target_i, int target_j, cycle *cycle, double *pos_min_cycle, double *neg_min_cycle) • csp_get_arc_cycle(cycle *cycle, int i, int *row, int *col, sense *sense) • csp_get_card_cycle(cycle *cycle) • csp_ij2xp(int ntab, int i, int j) • csp_inidijkstra(table2d *tab, nfpro *ncp, dsp *dsp) • csp_inincp(table2d *tab, nfpro *ncp) • csp_lower_bound(table2d *tab) • csp_resetnf(table2d *tab, nfpro *ncp, cycle *cycles_target, int target_i, int target_j) • csp_set_costsnf_v1(table2d *tab, int target_i, int target_j, double prot_req, cycle *cycles_target, nfpro *ncp , dsp *dsp) • csp_set_costsnf_v2(table2d *tab, int target_i, int target_j, double prot_req, cycle *cycles_target, nfpro *ncp , dsp *dsp) • csp_set_targetnf(table2d *tab, nfpro *ncp, dsp *dsp, int target_i, int target_j) • csp_solvenf_cplex75(nfpro *ncp, cpxenvptr env, cpxnetptr net)

	• csp_solvenf_dijkstra(dsp *dsp) • csp_solvenf_pprn_reduced(nfpro *ncp) • get_cellvalue2d(table2d *tab, int row, int column) • get_cellweight2d(table2d *tab, int row, int column) • get_comp_lowbound2d(table2d *tab) • get_index_of_primary2d(table2d *tab, int row, int col) • get_lowerbound2d(table2d *tab) • get_ncolumns2d(table2d *tab) • get_npcells2d(table2d *tab) • get_nrows2d(table2d *tab) • get_numnfproblems2d(table2d *tab) • get_pcelllpl2d(table2d *tab, int pcell) • get_pcellupl2d(table2d *tab, int pcell) • get_sorted_pcell2d(table2d *tab, int pcell, int * pi, int* pj, double *lpl, double *upl) • get_sorted_pcelllpl2d(table2d *tab, int pcell) • get_sorted_pcellupl2d(table2d *tab, int pcell) • get_valuesuppressed2d(table2d *tab) • get_weightsuppressed2d(table2d *tab) • is_cell_nonremoved2d(table2d *tab, int row, int column) • is_cell_primary2d(table2d *tab, int row, int column) • objcostnf(nfpro *nf) • put_cellstatus2d(table2d *tab, int row, int column, status_cell status) • put_numnfproblems2d(table2d *tab, int nnf) • put_pcelllpl2d(table2d *tab, int pcell, double lpl) • put_pcellupl2d(table2d *tab, int pcell, double upl) • put_sorted_pcelllpl2d(table2d *tab, int pcell, double lpl) • put_sorted_pcellupl2d(table2d *tab, int pcell, double upl) • put_valuesuppressed2d(table2d *tab, double v) • put_weightsuppressed2d(table2d *tab, double v)

Routine Name	csp_heur2D_KGA(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur_flows_n.cpp
Routine Size	288 lines
Routine Comment	
	KGA-based heuristics for 2D CSP Input/output parameter: tab: table to be protected (input); protected table (output) Returns: 0: if everything was fine -1: if not enough memory -2: if problem detected as infeasible (no solution was found) -3: error defining NF problem -4: error initializing NF problem -5: error solving NF problem -7: problems with cplex license -8: error creating cplex network object -9: error in lower bounding procedure
Parent Routines	
	• csp_heur2D(TABLE2D *tab)
Child Routines	
	• csp_allocatenf(int nnu, int nar, nfpro **pnf) • csp_create_cycle(cycle **cycle, int ini_max_card) • csp_delete_cycle(cycle **cycle) • csp_freenf(nfpro **nf) • csp_getcyclekga_nf(table2d *tab, nfpro *ncp, int target_i, int target_j, cycle *cycle, double *min_cycle, type_prot type_prot) • csp_get_arc_cycle(cycle *cycle, int i, int *row, int *col, sense *sense) • csp_get_card_cycle(cycle *cycle) • csp_ij2xp(int ntab, int i, int j) • csp_inikga_nf(table2d *tab, nfpro *ncp) • csp_lower_bound(table2d *tab) • csp_reset_kga_problem_target(table2d *tab, nfpro *ncp, int target_i, int target_j, type_prot type_prot) • csp_set_kga_problem_target(table2d *tab, nfpro *ncp, int target_i, int target_j, double prot_req, type_prot type_prot) • csp_solvenf_cplex75(nfpro *ncp, cpxenvptr env, cpxnetptr net) • csp_solvenf_pprn_reduced(nfpro *ncp) • get_cellvalue2d(table2d *tab, int row, int column) • get_cellweight2d(table2d *tab, int row, int column) • get_comp_lowbound2d(table2d *tab)

	• get_index_of_primary2d(table2d *tab, int row, int col) • get_lowerbound2d(table2d *tab) • get_ncolumns2d(table2d *tab) • get_npcells2d(table2d *tab) • get_nrows2d(table2d *tab) • get_numnfproblems2d(table2d *tab) • get_pcelllpl2d(table2d *tab, int pcell) • get_pcellupl2d(table2d *tab, int pcell) • get_sorted_pcell2d(table2d *tab, int pcell, int * pi, int* pj, double *lpl, double *upl) • get_sorted_pcelllpl2d(table2d *tab, int pcell) • get_sorted_pcellupl2d(table2d *tab, int pcell) • get_valuesuppressed2d(table2d *tab) • get_weightsuppressed2d(table2d *tab) • is_cell_nonremoved2d(table2d *tab, int row, int column) • is_cell_primary2d(table2d *tab, int row, int column) • objcostnf(nfpro *nf) • put_cellstatus2d(table2d *tab, int row, int column, status_cell status) • put_numnfproblems2d(table2d *tab, int nnf) • put_pcelllpl2d(table2d *tab, int pcell, double lpl) • put_pcellupl2d(table2d *tab, int pcell, double upl) • put_valuesuppressed2d(table2d *tab, double v) • put_weightsuppressed2d(table2d *tab, double v)

Routine Name	csp_ij2Xp(int ntab, int i, int j)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.h
Routine Size	9 lines
Routine Comment	
	Number of arc with POS orientation (increases flow) of cell (i,j) ntab : number of cols. of the table i : row number of the cell (0..mtab-1) j : column number the cell (0..ntab-1)
Parent Routines	
	• csp_GetCycleNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j, CYCLE *cycle, double *pos_min_cycle, double *neg_min_cycle) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_resetNF(TABLE2D *tab, NFPro *Ncp, CYCLE *cycles_target, int target_i, int target_j) • csp_reset_KGA_problem_target(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, TYPE_PROT type_prot) • csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • csp_set_costsNF_v2(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • csp_set_KGA_problem_target(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, double prot_req, TYPE_PROT type_prot) • csp_set_targetNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j)
Child Routines	

Routine Name	csp_IniDijkstra(TABLE2D *tab, NFPro *Ncp, DSP *Dsp)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_solve_dijkstra.cpp
Routine Size	110 lines
Routine Comment	
	Loads Dsp-¿mnk, mnl, car of permanent cells and computes the adjacencies for the Dijkstra algorithm Input parameters: TABLE2D *tab : 2D table NFPro *Ncp : NF problem DSP *Dsp : Dijkstra structure
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab)
Child Routines	
	• get_ncolumns2d(table2d *tab) • get_nrows2d(table2d *tab)

Routine Name	csp_iniKGA_NF(TABLE2D *tab, NFPro *Ncp)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur_flows_n.cpp
Routine Size	100 lines
Routine Comment	
	Defines the topology and injections of the network flow problem Input parameters: TABLE2D *tab : Data structure of the CSP problem Output parameters: NFPro *Ncp : current NF problem Returns: 0: if everything was fine -2: if internal error (currently deactivated check)
Parent Routines	
	• csp_heur2D_KGA(TABLE2D *tab)
Child Routines	
	• get_cellvalue2d(table2d *tab, int row, int column) • get_cellweight2d(table2d *tab, int row, int column) • get_ncolumns2d(table2d *tab) • get_nrows2d(table2d *tab) • is_cell_permanent2d(table2d *tab, int row, int column) • is_cell_primary2d(table2d *tab, int row, int column)

Routine Name	csp_iniNcp(TABLE2D *tab, NFPro *Ncp)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur_flows_01.cpp
Routine Size	83 lines
Routine Comment	
	Defines the topology and injections of the network flow problem Input parameters: TABLE2D *tab : Data structure of the CSP problem Output parameters: NFPro *Ncp : current NF problem Returns: 0: if everything was fine -2: if internal error (currently deactivated check)
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab)
Child Routines	
	• get_ncolumns2d(table2d *tab) • get_nrows2d(table2d *tab) • is_cell_permanent2d(table2d *tab, int row, int column)

Routine Name	csp_lower_bound(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_lower_bound.cpp
Routine Size	431 lines
Routine Comment	
	Lower bounding procedure Input/Output parameter: tab: original 2D table problem to be solved Returns: 0: if everything was fine -1: if not enough memory -2: error managing cplex LP object (returned by cplex7.5) -3: error solving or recovering the solution of cplex network object (returned by cplex7.5) -4: problems with cplex license (returned by cplex7.5) -7: internal error; should never happen
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_lower_bound(TABLE2D *tab)
Child Routines	
	• csp_lower_bound(table2d *tab) • get_cellvalue2d(table2d *tab, int row, int column) • get_cellweight2d(table2d *tab, int row, int column) • get_ncolumns2d(table2d *tab) • get_npcells2d(table2d *tab) • get_nrows2d(table2d *tab) • get_pcell2d(table2d *tab, int pcell, int * pi, int* pj, double *lpl, double *upl) • get_protlev_le_cellvalue2d(table2d *tab) • get_use_lowbound2d(table2d *tab) • get_valuesuppressed2d(table2d *tab) • get_weightsuppressed2d(table2d *tab) • put_cellstatus2d(table2d *tab, int row, int column, status_cell status) • put_lowerbound2d(table2d *tab, double lb) • put_valuesuppressed2d(table2d *tab, double v) • put_weightsuppressed2d(table2d *tab, double v)

Routine Name	csp_resetNF(TABLE2D *tab, NFPro *Ncp, CYCLE *cycles_target, int target_i, int target_j)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur_flows_01.cpp
Routine Size	39 lines
Routine Comment	
	Reset initial capacities of NF problem Input parameters: TABLE2D *tab : Data structure of the CSP problem CYCLE *cycles_target : cells needed up to the current iter. to protect the target cell int target_i : row number of the target cell int target_j : column number of the target cell Input/Output parameters: NFPro *Ncp: NF problem
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab)
Child Routines	
	• csp_get_arc_cycle(cycle *cycle, int i, int *row, int *col, sense *sense) • csp_get_card_cycle(cycle *cycle) • csp_ij2xp(int ntab, int i, int j) • get_ncolumns2d(table2d *tab)

Routine Name	csp_reset_KGA_problem_target(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, TYPE_PROT type_prot)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur_flows_n.cpp
Routine Size	33 lines
Routine Comment	
	Reset capacities of target in NF problem Input parameters: TABLE2D *tab : Data structure of the CSP problem int target_i : row number of the target cell int target_j : column number of the target cell TYPE_PROT type_prot: LOWER or UPPER problem Input/Output parameters: NFPro *Ncp: NF problem
Parent Routines	
	• csp_heur2D_KGA(TABLE2D *tab)
Child Routines	
	• csp_ij2xp(int ntab, int i, int j) • get_cellvalue2d(table2d *tab, int row, int column) • get_ncolumns2d(table2d *tab)

Routine Name	csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur_flows_01.cpp
Routine Size	220 lines
Routine Comment	
	Set costs of NF problem, using the stratification originally suggested by Cox. It is an accurate stratification, but expensive to compute Input parameters: TABLE2D *tab : Data structure of the CSP problem int target_i : row number of the target cell int target_j : column number of the target cell double prot_req : Protection required in current N.F. prob. CYCLE *cycles_target : cycles_target with cells needed up to the current iter. to protect the target cell. Output parameters: NFPro *Ncp : current NF problem DSP *Dsp : current shortest path problem Returns: 0: if everything was fine -1: if not enough memory
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab)
Child Routines	
	• csp_arc2col(int arc, int ntab) • csp_arc2row(int arc, int ntab) • csp_get_arc_cycle(cycle *cycle, int i, int *row, int *col, sense *sense) • csp_get_card_cycle(cycle *cycle) • csp_ij2xp(int ntab, int i, int j) • get_cellvalue2d(table2d *tab, int row, int column) • get_cellweight2d(table2d *tab, int row, int column) • get_ncolumns2d(table2d *tab) • get_npcells2d(table2d *tab) • get_nrows2d(table2d *tab) • get_nseccells2d(table2d *tab) • is_cell_permanent2d(table2d *tab, int row, int column)

Routine Name	csp_set_costsNF_v2(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur_flows_01.cpp
Routine Size	150 lines
Routine Comment	
	Set costs of NF problem, using an alternative stratification where a value M ¿ maxsumS2,sumS4 is used. We consider M= 4*a_m+1,n+1, which always satisfies the above property. This cost stratification is not not so accurate as that of csp_set_costsNF_v1, but cheaper to compute. We also need a value McardSi > card(Si) for all i=1..4. This McardSi values are computed in the declaration of these variables. Input parameters: TABLE2D *tab : Data structure of the CSP problem int target_i : row number of the target cell int target_j : column number of the target cell double prot_req : Protection required in current N.F. prob. CYCLE *cycles_target : cycles_target with cells needed up to the current iter. to protect the target cell. Output parameters: NFPro *Ncp : current NF problem DSP *Dsp : current shortest path problem Returns: 0: if everything was fine -1: if not enough memory
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab)
Child Routines	
	• csp_get_arc_cycle(cycle *cycle, int i, int *row, int *col, sense *sense) • csp_get_card_cycle(cycle *cycle) • csp_ij2xp(int ntab, int i, int j) • get_cellvalue2d(table2d *tab, int row, int column) • get_cellweight2d(table2d *tab, int row, int column) • get_ncolumns2d(table2d *tab) • get_npcells2d(table2d *tab) • get_nrows2d(table2d *tab) • get_nseccells2d(table2d *tab) • is_cell_permanent2d(table2d *tab, int row, int column)

Routine Name	csp_set_KGA_problem_target(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, double prot_req, TYPE_PROT type_prot)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur_flows_n.cpp
Routine Size	32 lines
Routine Comment	
	Set values for the target cell in the network flow problem Input parameters: TABLE2D *tab : data structure of the CSP problem int target_i : row number of the target cell int target_j : column number of the target cell double prot_req : protection required TYPE_PROT type_prot: LOWER or UPPER problem Output parameters: NFPro *Ncp : data structure of NF problem to be solved
Parent Routines	
	• csp_heur2D_KGA(TABLE2D *tab)
Child Routines	
	• csp_ij2xp(int ntab, int i, int j) • get_ncolumns2d(table2d *tab)

Routine Name	csp_set_targetNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur_flows_01.cpp
Routine Size	34 lines
Routine Comment	
	Set target values of Ncp-¿max_flx and Ncp-¿min_flx Input parameters: TABLE2D *tab : data structure of the CSP problem int target_i : row number of the target cell int target_j : column number of the target cell Output parameters: NFPro *Ncp : data structure of the current (N',c') problem DSP *Dsp : Dijkstra structure
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab)
Child Routines	
	• csp_ij2xp(int ntab, int i, int j) • get_ncolumns2d(table2d *tab)

Routine Name	csp_SolveNF_CPLEX75(NFPro *Ncp, CPXENVptr env, CPXNETptr net)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_solve_cplex.cpp
Routine Size	75 lines
Routine Comment	
	Solves the network flow problem (N',c') Input parameters: NFPro Ncp: NF problem env: cplex environment net: cplex network problem Returns: 0: if everything was fine -1: if not enough memory -2: error managing cplex network object (returned by cplex7.5) -3: error solving or recovering the solution of cplex network object (returned by cplex7.5) -4: infeasible problem -5: unbounded problem
Parent Routines	
	• csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	csp_SolveNF_Dijkstra(DSP *Dsp)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_solve_dijkstra.cpp
Routine Size	27 lines
Routine Comment	
	Solves the network flow (shorthest path) problem Input parameters: DSP *Dsp : Dijkstra structure, already updated Returns: 0: if everything was fine -5: if error in solving shortest path problem
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab)
Child Routines	

Routine Name	csp_SolveNF_PPRN_0lb(NFPro *Ncp)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_solve_pprn.cpp
Routine Size	141 lines
Routine Comment	
	Solves the network flow problem through a change of variables with zero lower bounds. Input parameters: NFPro Ncp : NF problem Returns: 0: if everything was fine -1: if not enough memory -2: if infeasible problem -3: if return status in pprn2 other than optimal or infeasible or unbounded -4: if unbounded problem
Parent Routines	
	• csp_auditing2D(TABLE2D *tab, int ini, int fin)
Child Routines	

Routine Name	csp_SolveNF_PPRN_reduced(NFPro *Ncp)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_solve_pprn.cpp
Routine Size	192 lines
Routine Comment	
	Solves the network flow problem using a reduced network: constant arcs (e.g., lower bound equal to upper bound) are removed There is no check of the connectivity of the reduced network. If someday there happens to be a problem, routine csp_SolveNF_PPRN_0lb can instead be used (which preserves the network topology since no arcs are removed). Input parameters: NFPro Ncp : NF problem Returns: 0: if everything was fine -1: if not enough memory -2: if infeasible problem -3: if return status in pprn2 other than optimal or infeasible or unbounded -4: if unbounded problem
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	delete_table(TABLE2D **ptab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.cpp
Routine Size	12 lines
Routine Comment	
	Deletes a non-empty table
Parent Routines	
	• create_table(TABLE2D **ptab, int m, int n, int p)
Child Routines	

Routine Name	get_cells_0_permanent2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get if 0 value cells must be set permanent
Parent Routines	
	• csp_heur2D(TABLE2D *tab)
Child Routines	

Routine Name	get_cellstatus2D(TABLE2D *tab, int row, int column)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get cell status without checking bounds
Parent Routines	
	• csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_heur2D_Cox(TABLE2D *tab) • csp_lower_bound(table2d *tab)
Child Routines	

Routine Name	get_cellvalue2D(TABLE2D *tab, int row, int column)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get cell value without checking bounds
Parent Routines	
	• check_protection_levels_for_lbp2D(TABLE2D *tab) • compute_weights2D(TABLE2D *tab) • csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_GetCycleNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j, CYCLE *cycle, double *pos_min_cycle, double *neg_min_cycle) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_iniKGA_NF(TABLE2D *tab, NFPro *Ncp) • csp_lower_bound(TABLE2D *tab) • csp_reset_KGA_problem_target(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, TYPE_PROT type_prot) • csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • csp_set_costsNF_v2(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • set_0_value_cells_permanent2D(TABLE2D *tab) • sort_primarycells2D(TABLE2D *tab)
Child Routines	

Routine Name	get_cellweight2D(TABLE2D *tab, int row, int column)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get cell weight without checking bounds
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_iniKGA_NF(TABLE2D *tab, NFPro *Ncp) • csp_lower_bound(TABLE2D *tab) • csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • csp_set_costsNF_v2(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp)
Child Routines	

Routine Name	get_comp_lowbound2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get if lower bound must be computed
Parent Routines	
	• csp_heur2D(TABLE2D *tab) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	get_index_of_primary2D(TABLE2D *tab, int row, int col)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get index of primary cell in array of primaries, without checking status
Parent Routines	
	• csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	get_lowerbound2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get lower bound
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	get_ncolumns2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get number of columns
Parent Routines	
	• compute_weights2D(TABLE2D *tab) • csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_GetCycleNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j, CYCLE *cycle, double *pos_min_cycle, double *neg_min_cycle) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_IniDijkstra(TABLE2D *tab, NFPro *Ncp, DSP *Dsp) • csp_iniKGA_NF(TABLE2D *tab, NFPro *Ncp) • csp_iniNcp(TABLE2D *tab, NFPro *Ncp) • csp_lower_bound(TABLE2D *tab) • csp_resetNF(TABLE2D *tab, NFPro *Ncp, CYCLE *cycles_target, int target_i, int target_j) • csp_reset_KGA_problem_target(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, TYPE_PROT type_prot) • csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • csp_set_costsNF_v2(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • csp_set_KGA_problem_target(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, double prot_req, TYPE_PROT type_prot) • csp_set_targetNF(TABLE2D *tab, NFPro *Ncp, DSP *Dsp, int target_i, int target_j) • set_0_value_cells_permanent2D(TABLE2D *tab)
Child Routines	

Routine Name	get_npcells2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get number of primary cells
Parent Routines	
	• check_protection_levels_for_lbp2D(TABLE2D *tab) • csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_lower_bound(TABLE2D *tab) • csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • csp_set_costsNF_v2(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • sort_primarycells2D(TABLE2D *tab)
Child Routines	

Routine Name	get_nrows2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get number of rows
Parent Routines	
	• compute_weights2D(TABLE2D *tab) • csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_IniDijkstra(TABLE2D *tab, NFPro *Ncp, DSP *Dsp) • csp_iniKGA_NF(TABLE2D *tab, NFPro *Ncp) • csp_iniNcp(TABLE2D *tab, NFPro *Ncp) • csp_lower_bound(TABLE2D *tab) • csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • csp_set_costsNF_v2(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • set_0_value_cells_permanent2D(TABLE2D *tab)
Child Routines	

Routine Name	get_nseccells2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get number of sencondary cells
Parent Routines	
	• csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • csp_set_costsNF_v2(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp)
Child Routines	

Routine Name	get_numnfproblems2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get number of nf problems
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	get_pcell2D(TABLE2D *tab, int pcell, int * pi, int* pj, double *lpl, double *upl)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	10 lines
Routine Comment	
	Returns basic info of primary cell pcell without checking bounds
Parent Routines	
	• check_protection_levels_for_lbp2D(TABLE2D *tab) • csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_lower_bound(TABLE2D *tab) • sort_primarycells2D(TABLE2D *tab)
Child Routines	

Routine Name	get_pcelllpl2D(TABLE2D *tab, int pcell)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	7 lines
Routine Comment	
	Returns current provided lower protection level of primary cell pcell without checking bounds
Parent Routines	
	• csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	get_pcellupl2D(TABLE2D *tab, int pcell)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	7 lines
Routine Comment	
	Returns current provided upper protection level of primary cell pcell without checking bounds
Parent Routines	
	• csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	get_protlev_le_cellvalue2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get if protection levels are less or equal than cell values
Parent Routines	
	• csp_lower_bound(TABLE2D *tab)
Child Routines	

Routine Name	get_sorted_pcell2D(TABLE2D *tab, int pcell, int * pi, int* pj, double *lpl, double *upl)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	12 lines
Routine Comment	
	Returns basic info of sorted primary cell pcell without checking bounds according to current merit order sort
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	get_sorted_pcelllpl2D(TABLE2D *tab, int pcell)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	7 lines
Routine Comment	
	Returns current provided lower protection level of primary cell pcell without checking bounds according to current merit order sort
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	get_sorted_pcellupl2D(TABLE2D *tab, int pcell)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	7 lines
Routine Comment	
	Returns current provided upper protection level of primary cell pcell without checking bounds according to current merit order sort
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	get_use_lowbound2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get if lower bound solution must be used
Parent Routines	
	• csp_heur2D(TABLE2D *tab) • csp_lower_bound(TABLE2D *tab)
Child Routines	

Routine Name	get_valuesuppressed2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get total value of suppressed cells
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_lower_bound(TABLE2D *tab)
Child Routines	

Routine Name	get_weightsuppressed2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Get total weight of suppressed cells
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_lower_bound(TABLE2D *tab)
Child Routines	

Routine Name	is_cell_nonremoved2D(TABLE2D *tab, int row, int column)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Returns true if cell is NONREMOVED; otherwise, false
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	is_cell_permanent2D(TABLE2D *tab, int row, int column)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	4 lines
Routine Comment	
	Returns true if cell is PERMANENT; otherwise, false
Parent Routines	
	• csp_iniKGA_NF(TABLE2D *tab, NFPro *Ncp) • csp_iniNcp(TABLE2D *tab, NFPro *Ncp) • csp_set_costsNF_v1(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp) • csp_set_costsNF_v2(TABLE2D *tab, int target_i, int target_j, double prot_req, CYCLE *cycles_target, NFPro *Ncp , DSP *Dsp)
Child Routines	

Routine Name	is_cell_primary2D(TABLE2D *tab, int row, int column)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	4 lines
Routine Comment	
	Returns true if cell is PRIMARY; otherwise, false
Parent Routines	
	• csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_iniKGA_NF(TABLE2D *tab, NFPro *Ncp)
Child Routines	

Routine Name	is_cell_secondary2D(TABLE2D *tab, int row, int column)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	4 lines
Routine Comment	
	Returns true if cell is SECONDARY; otherwise, false
Parent Routines	
Child Routines	

Routine Name	objcostNF(NFPro *nf)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.cpp
Routine Size	10 lines
Routine Comment	
	Print objective function cost of flows in nf
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	put_cellstatus2D(TABLE2D *tab, int row, int column, STATUS_CELL status)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	9 lines
Routine Comment	
	Put cell status without checking bounds and updates number of secondary cells
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_lower_bound(TABLE2D *tab) • put_pcell2D(TABLE2D *tab, int pcell, int pi, int pj, double plpl, double pupl) • set_0_value_cells_permanent2D(TABLE2D *tab)
Child Routines	

Routine Name	put_cells_0_permanent2D(TABLE2D *tab, char setperm)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Put if 0 value cells must be set permanent
Parent Routines	
Child Routines	

Routine Name	put_cellvalue2D(TABLE2D *tab, int row, int column, double value)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	7 lines
Routine Comment	
	Put cell value without checking bounds
Parent Routines	
Child Routines	

Routine Name	put_cellweight2D(TABLE2D *tab, int row, int column, double weight)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	13 lines
Routine Comment	
	Put cell weight without checking bounds
Parent Routines	
	• compute_weights2D(TABLE2D *tab)
Child Routines	

Routine Name	put_comp_lowbound2D(TABLE2D *tab, char comp_lb)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Put if lower bound must be computed
Parent Routines	
Child Routines	

Routine Name	put_cost_type2D(TABLE2D *tab, COST_TYPE cost_type)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Put type of cost stratification (only useful for 0-1 flows heuristic)
Parent Routines	
Child Routines	

Routine Name	put_heuristic2D(TABLE2D *tab, HEURISTIC heuristic)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	12 lines
Routine Comment	
	Put heuristic
Parent Routines	
Child Routines	

Routine Name	put_index_of_primary2D(TABLE2D *tab, int row, int col, int pcell)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Put index of primary cell in array of primaries, without checking status
Parent Routines	
	• put_pcell2D(TABLE2D *tab, int pcell, int pi, int pj, double plpl, double pupl)
Child Routines	

Routine Name	put_lowerbound2D(TABLE2D *tab, double lb)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Put lower bound
Parent Routines	
	• csp_lower_bound(TABLE2D *tab)
Child Routines	

Routine Name	put_numnfproblems2D(TABLE2D *tab, int nnf)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Put number of nf problems
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	put_pcell2D(TABLE2D *tab, int pcell, int pi, int pj, double plpl, double pupl)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	13 lines
Routine Comment	
	Put basic info of primary cell pcell without checking bounds status of cell is also updated
Parent Routines	
Child Routines	
	• put_cellstatus2d(table2d *tab, int row, int column, status_cell status) • put_index_of_primary2d(table2d *tab, int row, int col, int pcell)

Routine Name	put_pcelllpl2D(TABLE2D *tab, int pcell, double lpl)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	7 lines
Routine Comment	
	Put current provided lower protection level of primary cell pcell without checking bounds
Parent Routines	
	• csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	put_pcellupl2D(TABLE2D *tab, int pcell, double upl)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	7 lines
Routine Comment	
	Put current provided upper protection level of primary cell pcell without checking bounds
Parent Routines	
	• csp_auditing2D(TABLE2D *tab, int ini, int fin) • csp_GetCycleKGA_NF(TABLE2D *tab, NFPro *Ncp, int target_i, int target_j, CYCLE *cycle, double *min_cycle, TYPE_PROT type_prot) • csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab)
Child Routines	

Routine Name	put_protlev_le_cellvalue2D(TABLE2D *tab, char boolean)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Put if protection levels are less or equal than cell values
Parent Routines	
	• check_protection_levels_for_lbp2D(TABLE2D *tab)
Child Routines	

Routine Name	put_solver2D(TABLE2D *tab, SOLVER solver)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	12 lines
Routine Comment	
	Put solver
Parent Routines	
Child Routines	

Routine Name	put_sorted_pcelllpl2D(TABLE2D *tab, int pcell, double lpl)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	7 lines
Routine Comment	
	Put current provided lower protection level of primary cell pcell without checking bounds according to current merit order sort
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab)
Child Routines	

Routine Name	put_sorted_pcellupl2D(TABLE2D *tab, int pcell, double upl)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	7 lines
Routine Comment	
	Put current provided upper protection level of primary cell pcell without checking bounds according to current merit order sort
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab)
Child Routines	

Routine Name	put_typemorder2D(TABLE2D *tab, TYPE_MERIT_ORDER morder)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	12 lines
Routine Comment	
	Put type of merit order
Parent Routines	
Child Routines	

Routine Name	put_typeweights2D(TABLE2D *tab, TYPE_WEIGHTS tweights)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	12 lines
Routine Comment	
	Put type of weights
Parent Routines	
Child Routines	

Routine Name	put_use_lowbound2D(TABLE2D *tab, char use_lb)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	12 lines
Routine Comment	
	Put if lower bound solution must be used
Parent Routines	
Child Routines	

Routine Name	put_valuesuppressed2D(TABLE2D *tab, double v)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	6 lines
Routine Comment	
	Put total value of suppressed cells
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_lower_bound(TABLE2D *tab)
Child Routines	

Routine Name	put_weightsuppressed2D(TABLE2D *tab, double v)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.h
Routine Size	10 lines
Routine Comment	
	Put total weight of suppressed cells
Parent Routines	
	• csp_heur2D_Cox(TABLE2D *tab) • csp_heur2D_KGA(TABLE2D *tab) • csp_lower_bound(TABLE2D *tab)
Child Routines	

Routine Name	set_0_value_cells_permanent2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.cpp
Routine Size	14 lines
Routine Comment	
	Mark cells with 0 value as permanent
Parent Routines	
	• csp_heur2D(TABLE2D *tab)
Child Routines	
	• get_cellvalue2d(table2d *tab, int row, int column) • get_ncolumns2d(table2d *tab) • get_nrows2d(table2d *tab) • put_cellstatus2d(table2d *tab, int row, int column, status_cell status)

Routine Name	show_infoarcs(NFPro *nf)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_heur2D.cpp
Routine Size	13 lines
Routine Comment	
	Show some information about arcs of a NF problem
Parent Routines	
Child Routines	

Routine Name	sort_primarycells2D(TABLE2D *tab)
Routine Location	Tau-Argus-UPC\csp_nf\src\2Dtables\csp_table2D.cpp
Routine Size	42 lines
Routine Comment	
Parent Routines	
	• create_table(TABLE2D **ptab, int m, int n, int p) • csp_heur2D(TABLE2D *tab)
Child Routines	
	• get_cellvalue2d(table2d *tab, int row, int column) • get_npcells2d(table2d *tab) • get_pcell2d(table2d *tab, int pcell, int * pi, int* pj, double *lpl, double *upl)