Real-Life Employee Timetabling Problem: Instance Example

Ignacio Castiñeiras[1], Fernando Sáenz-Pérez[2] *
ncasti@fdi.ucm.es, fernan@sip.ucm.es
1 Dept. Sistemas Informáticos y Computación, 2 Dept. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain

Technical Report SIC-02/13. March 2013

Abstract

This technical report focuses on a concrete instance of the real-life employee timetabling problem presented in the paper "Applying CP(FD), CLP(FD) and CFLP(FD) to a Real-Life Employee Timetabling Problem". It provides additional material that would help the reader of the paper in both understanding the algorithm proposed as well as the code of the models provided online.

1. Motivation

The paper "Applying CP(FD), CLP(FD) and CFLP(FD) to a Real-Life Employee Timetabling Problem" presents a case study based on a real-life Employee Timetabling Problem, formulating it and then making a comparison of its modeling and solution with three different programming paradigms embedding Finite Domain constraint solving: Constraint Programming, Constraint Logic Programming and Constraint Functional Logic Programming. Although the paper is self-contained, a section instantiating the formulated algorithm to a concrete instance would be helpful for a better understanding of the proper algorithm as well as of the code of the available CP(FD), CLP(FD) and CFLP(FD) models provided online. It is the requirement that this paper fulfils.

The structure of the technical report is as follows: Section 2 presents a description of the concrete instance being used. Next, Section 3 parameterizes the solving approach to this concrete instance. Then, Section 4 parameterizes the running algorithm to the instance, in particular showing the variables, constraints posted to the solver as well as the additional data structures used to compute the solution. Finally, Section 5 presents some conclusions.

2. Instance Description

A department is filling its employee timetabling for the next week, which starts on Monday. Whereas each working day contains three shifts to be accomplished (of twenty, twenty two and twenty four hours, resp), each weekend day contains two shifts to be accomplished (both of twenty four hours). The department employs thirteen workers, twelve of them regular workers divided into three teams of four workers and an extra worker which belongs to no team. There is also a set of expected absences of concrete workers for concrete days that have to be taken into account when scheduling the timetable.

Only one team works each day, and each team is selected to work each three days. If a team is selected to work on a day, then only the regular workers of this team (as well as the extra worker) are able to work at the department on that day. The extra worker can be selected to work at most one out of each three consecutive days. Given the set of different shift types provided by working and weekend days, we also measure its distribution among the different regular workers of each team (by constraining the maximum deviation between the shifts assigned to the different workers.)

As each regular worker is expected to work the same number of hours, optimization arises in the problem because the department must pay regular workers for each extra hour they work, and any hour that the extra worker works is paid as two extra hours of a regular worker. Optimal schedule minimizes the extra hour payment.

* This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, UCM-BSCH-GR35/10-A-910502, and S2009TIC-1465

3. Solving Approach

Let us name p_tt the algorithm for solving the proposed instance. It receives the input parameters nd, nt, ntw, er, e f, ws and T. Special input parameters P, W and SS are provided to configure the FD constraint solver. P sets the propagation mode to incremental or batch. W and SS set the search strategy to: Order the variables by workers or days, and label the variables by first unbound or first fail, resp. The result computed by p_tt is the pair (timetabling, eh). The former is an nd *x* w assignment (where each position (i, j) represents the shift assigned on day j to worker i.) The latter represents the total amount of extra hours of such assignment. Table 1 summarizes the notation of the problem.

Timetabling is an nd *x* w matrix, in our example a seven *x* thirteen one. However, Fig. 1 shows that, due to the dependencies between the teams (just one team works each day and each team works each three days) most of the positions of timetabling are going to be zero. For example, if t_1 is selected for working on days one, four and so on, then the variables of the other teams for those days are going to be zero, as well as the variables of team t_1 for days two, three, five, six and so on.

In this context, we can better represent Timetabling as Table, an (ntw+1) *x* nd matrix (in our example a five *x* seven matrix), where for each day, the first ntw rows represent the regular workers of the selected team for that day. Fig. 2 shows the mapping from Timetabling to Table, where the first four rows represent to $\{w_1, w_2, w_3, w_4\}$ on days one, four, etc, to $\{w_9, w_{10}, w_{11}, w_{12}\}$ on days 2, 5, etc, and to $\{w_5, w_6, w_7, w_8\}$ on days 3, 6, etc.

The assignment shown in Fig. 2 is a feasible assignment of teams to days, i.e., an assignment satisfying that each day there are enough workers of the selected team to accomplish the scheduled shifts and that ew is never selected to work more than one out of each three consecutive days. In general, there are three possible situations arising when assigning working teams to concrete days.

- In our example, on the second day, dc2 ≡ ws1 (which request [20, 22, 24] as its m = 3 shifts). For that day, t3 has four regular workers {w9, w10, w11, w12} available (a = 4). As a ≥ m, t3 can be assigned to dc2, and both ew and one of {w9, w10, w11, w12} do not have to work (assigned to a 0).
- On the first day, dc1 ≡ ws1 (which request [20, 22, 24] as its m = 3 shifts). For that day, t1 has two regular workers {w3, w4} available (a = 2). As a = m - 1, t1 can be assigned to dc2, but ew has to be selected to work.
- On the first day, dc1 ≡ ws1 (which request [20, 22, 24] as its m = 3 shifts). For that day, t2 has one regular workers {w8} available (a = 1). As a < m - 1, t1 can not be assigned to dc2.

Variable Name	Variable Description	Example
Nd	Number of days	nd = 7
Dc	Day classification	dc = [1,1,1,1,1,2,2]
Ws	Different kind of working days	ws = [[20,22,24], [24,24]]
ws_1	*working day of kind 1 has three shifts (m = 3)*	ws_1 *= [20,22,24]*
ws_2	*working day of kind 2 has three shifts (m = 2)*	ws_1 *= [24,24]*
W	Number of workers	w = 13
Nt	Number of teams	nt = 3
Ntw	Number of team workers	ntw = 4
$w_{(i-1)*ntw+1}$, …, w_{i*ntw}	*Workers of team* t_i	t_1*= {*w_1*, *w_2*, *w_3*, *w_4*}, *t_2*= {*w_5*, *w_6*, *w_7*, *w_8*}, *t_3*= {*w_9*, *w_{10}*, *w_{11}*, *w_{12}*}*
Ew	*Extra Worker*	ew = w_{13}
Er	Period of consecutive days on which ew can work at most one of them	er = 3
Abs	Absences of (regular workers, days)	abs=[(1,1),(2,1),(5,1),(6,1),(7,1),(10,1),(11,1), (12,1),(5,6),(6,6),(7,6),(10,6),(11,6),(12,6)]
S	Different kind of shifts	s = [0,20,22,24]
$T_{ti, sj}$	Measure of the distribution of shifts of kind S_j to the regular workers of team t_i	$T_{t1\ s0}$ = 1
cv_1, …, cv_{ntw}	*Number of shifts of kind s = 0 the regular workers of* t_1 *= {*w_1*, *w_2*, *w_3*, *w_4*} are assigned to*	cv_1*=2;* cv_2*=1;* cv_3*=1;* cv_4*=1; (*$T_{t1\ s0}$ *is equal to the maximum of the differences of cv)*
T	Maximum value any $T_{ti, sj}$ can take	T = 1
Ef	Factor for the working hours of ew	ef = 2
P	Incremental or batch propagation	P = true (incremental)
W	Order of the variable set (by days or by workers)	W= calByWorkers
SS	Label order (input order or first fail)	SS = fstUnbound (input order)

Table 1. List of variables used in the problem

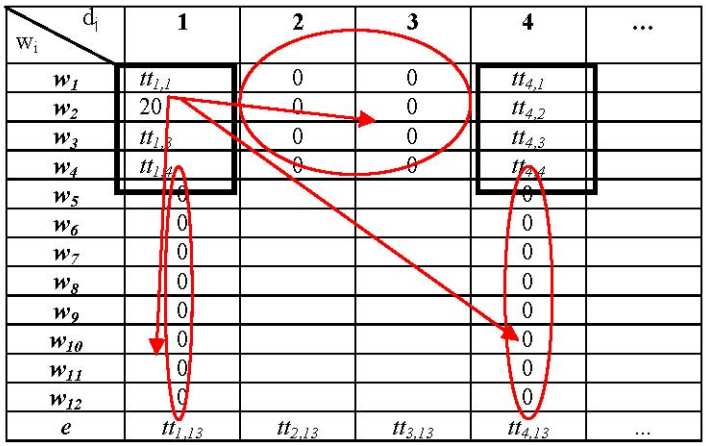

Fig. 1. Team dependencies in timetabling

There are nt teams, implying up to nt! possible assignments of teams to days: {tda_{n1}, …, $tda_{nt!}$}. As the optimal schedule can be in any of them, all must be explored. In our example, only two of the six tda are feasible, as due to the absences provided by abs, only team t_1 can work on day one. The two feasible assignments are then assigning t_2 (resp. t_3) to the second day and t_3 (resp. t_2) to the third day.

The main idea of the algorithm is to solve the problem by decomposing it into independent subproblems (exponentially easier to be solved), and then solve them sequentially. The idea of splitting the problem relies on the fact that only one team works each day, and the different teams rotate (each team works exactly each three days). Thus, the algorithm relies on a four stage process: (i) team assign, (ii) tt split, (iii) tt solve and (iv) tt map. First stage team_assign just concerns with finding any feasible assignment tda_i. Starting from a feasible bijection tda_i, the stages tt_split, tt_solve and tt_map are executed, generating nt subproblems, solving them and mapping the solution to (timetabling, eh). In our example, we can split the generated Table associated to each feasible tda_i into 3 independent subproblems: tt_1 (with columns 1, 4 and so on), tt_2 (with columns 2, 5 and so on) and tt_3 (with columns 3, 6 and so on).

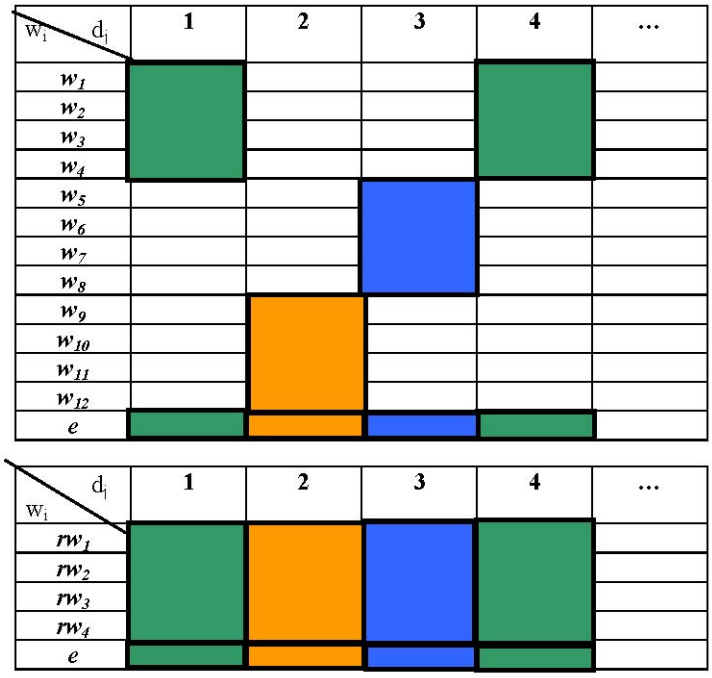

Fig. 2. Mapping from Timetable to Table

4. Algorithm Description

We run the algorithm by calling to: p_tt nd nt ntw er ef ws abs dc T P W SS = (timetabling, eh)
In our example, we run:

```
p_tt 7 3 4 3 2 [[20,22,24],[24,24]] [(1,1), (2,1), (5,1), (6,1), (7,1), (10,1), (11,1), (12,1), (5,6), (6,6), (7,6),
                (10,6), (11,6), (12,6)] [1,1,1,1,1,2,2] 1 true calByWorkers  fstUnbound = (timetabling, eh)
```

We obtain the following solution:

```
timetabling = [ [ 0, 0, 22, 24, 0, 0, 0, 0, 0, 0, 0, 0, 20 ],
                [ 0, 0, 0, 0, 0, 20, 22, 24, 0, 0, 0, 0, 0 ],
                [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 22, 24, 0 ],
              [ 0, 20, 24, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
              [ 0, 0, 0, 0, 24, 22, 20, 0, 0, 0, 0, 0, 0 ],
              [ 0, 0, 0, 0, 0, 0, 0, 0, 24, 0, 0, 0, 24 ],
              [ 24, 24, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ]
eh = 133
```

In the next sections we describe (one by one) the four stages of the algorithm. For each one we provide the main call to the stage (and the obtained output), and we reuse the steps provided in the paper (showed in italics) to further instantiate them to our instance example.

4.1. team_assign

We can run the stage by calling to: team_assign nd nt ntw er ws abs dc == (d, e, oabs)
In our example, we run:

```
team_assignment 7 3 4 3 [ [20, 22, 24], [24, 24] ] [ (1,1), (2,1), (5,1), (6,1), (7,1), (10,1), (11,1), (12,1), (5,6),
                                (6,6), (7,6), (10,6), (11,6), (12,6)] [1, 1, 1, 1, 1, 2, 2] == (d, e, oabs)
```

We obtain two feasible solutions:
Solution 1:

```
d = [1, 2, 3, 1, 2, 3, 1]; e = [1, 0, 0, 0, 0, 1, 0]; oabs = [ [1,2,5,6,7,10,11,12], [], [], [], [], [5,6,7,10,11,12], [] ]
```

Solution 2:

```
d = [1, 3, 2, 1, 3, 2, 1]; e = [1, 0, 0, 0, 0, 1, 0]; oabs = [ [1,2,5,6,7,10,11,12], [], [], [], [], [5,6,7,10,11,12], [] ]
```

4.1.1. Sequence of steps (as they are provided in the paper):

(1) Create Table, an nd x (ntw+1) matrix of variables. In our example:

```
Table = [ [V1, V2, V3, V4, V5], [V6, V7, V8, V9, V10], [V11, V12, V13, V14, V15], [V16, V17, V18, V19, V20],
[V21, V22, V23, V24, V25], [V26, V27, V28, V29, V30], [V31, V32, V33, V34, V35] ]
```

(2) Fit abs to oabs (a list of lists representing the absences ordered per day.) Create atd (a list of lists representing the absences per day and team) and etd (a list of lists representing the request of ew per day and team). In our example:

```
oabs = [ [1, 2, 5, 6, 7, 10, 11, 12], [], [], [], [], [5, 6, 7, 10, 11, 12], [] ]
atd = [ [2, 3, 3], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 3, 3], [0, 0, 0] ]
etd = [ [1, 1, 1], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 1, 1], [0, 0, 0] ]
```

(3) Create the arrays of nd FD variables d ($d_i \equiv$ team assigned to dc_i), a ($a_i \equiv$ amount of regular worker absences on dc_i) and e ($e_i \equiv$ is ew requested on dc_i). In our example:

```
d = [D1, D2, D3, D4, D5, D6, D7]
a = [A1, A2, A3, A4, A5, A6, A7]
e = [E1, E2, E3, E4, E5, E6, E7]
```

(4) Initialize d to {1, ..., nt}, constrained by $d_i == d_{i+nt}$, and by an all different constraint over $\{d_1, ..., d_{nt}\}$. In our example:

```
D1 == D4; D4 == D7; D2 == D5; D3 == D6
domain [D1, D2, D3, D4, D5, D6, D7] 1 3
all_different [D1, D2, D3, D4, D5, D6, D7]
```

(5) Initialize a to {0, …, ntw}. Each m_i (amount of shifts to be covered on dc_i) is explored to prune a_i upper bound. Link d, a and atd. In our example:

```
domain [A1, A2, A3, A4, A5, A6, A7] 0 4
A1 <= 2; A2 <= 2; A3 <= 2; A4 <= 2; A5 <= 2; A6 <= 3; A7 <= 3;
(D1 == 1) => (A1 == 2); (D1 == 2) => (A1 == 3); (D1 == 3) => (A1 == 3);
(D2 == 1) => (A2 == 0); (D2 == 2) => (A2 == 0); (D2 == 3) => (A2 == 0);
(D3 == 1) => (A3 == 0); (D3 == 2) => (A3 == 0); (D3 == 3) => (A3 == 0);
(D4 == 1) => (A4 == 0); (D4 == 2) => (A4 == 0); (D4 == 3) => (A4 == 0);
(D5 == 1) => (A5 == 0); (D5 == 2) => (A5 == 0); (D5 == 3) => (A5 == 0);
(D6 == 1) => (A6 == 0); (D6 == 2) => (A6 == 3); (D6 == 3) => (A6 == 3);
(D7 == 1) => (A7 == 0); (D7 == 2) => (A7 == 0); (D7 == 3) => (A7 == 0);
```

(6) Initialize e to 0, 1. The sum of each {ei; …, ei+er} is constrained to be <= 1. Link d, e and etd. In our example:

```
domain [E1, E2, E3, E4, E5, E6, E7] 0 1
sum [E1,E2,E3] <= 1, sum [E2,E3,E4] <= 1, sum [E3,E4,E5] <= 1, sum [E4,E5,E6] <= 1,
                                                                    sum [E5,E6,E7] <=1,
(D1 == 1) => (E1 == 1); (D1 == 2) => (E1 == 1); (D1 == 3) => (E1 == 1);
(D2 == 1) => (E2 == 0); (D2 == 2) => (E2 == 0); (D2 == 3) => (E2 == 0);
(D3 == 1) => (E3 == 0); (D3 == 2) => (E3 == 0); (D3 == 3) => (E3 == 0);
(D4 == 1) => (E4 == 0); (D4 == 2) => (E4 == 0); (D4 == 3) => (E4 == 0);
(D5 == 1) => (E5 == 0); (D5 == 2) => (E5 == 0); (D5 == 3) => (E5 == 0);
(D6 == 1) => (E6 == 0); (D6 == 2) => (E6 == 1); (D6 == 3) => (E6 == 1);
(D7 == 1) => (E7 == 0); (D7 == 2) => (E7 == 0); (D7 == 3) => (E7 == 0);
```

(7) Label d to obtain the feasible team assignments tda. In our example:

```
labeling [] [D1, D2, D3, D4, D5, D6, D7]
```

4.2. tt_split

We can run the stage by calling to: tt_split nd nt ntw ws oabs dc d e == (tt, tt_list, dc_list, h)
In our example (considering just the first solution of team_assign) we run:

```
tt_split 7 3 4 [[20,22,24],[24,24]]  [[1,2,5,6,7,10,11,12],[],[],[],[],[5,6,7,10,11,12],[]]  [1,2,3,1,2,3,1]
                                                                          [1,0,0,0,0,1,0]  == (tt, tt_list, dc_list, h)
```

We obtain the following solution:

```
Table = [[0,0,A,B,C],[D,E,F,G,0],[H,I,J,K,0],[L,M,N,O,0],[P,Q,R,S,0],[T,0,0,0,U],[V,W,X,Y,0]]
tt = [[[0,0,A,B,C],[L,M,N,O,0],[V,W,X,Y,0]],[[D,E,F,G,0],[P,Q,R,S,0]],[[H,I,J,K,0],[T,0,0,0,U]]]
dc_list = [[1,1,2],[1,1],[1,2]]
h = 35
```

4.2.1. Sequence of steps (as they are provided in the paper):

(1) Receive Table, d, e and oabs from team assign. In our example:

```
Table = [[V1, V2, V3, V4, V5], [V6, V7, V8, V9, V10], [V11, V12, V13, V14, V15], [V16, V17, V18, V19,
            V20], [V21, V22, V23, V24, V25], [V26, V27, V28, V29, V30], [V31, V32, V33, V34, V35] ];
d = [1, 2, 3, 1, 2, 3, 1]; e = [1, 0, 0, 0, 0, 1, 0]; oabs = [ [1,2,5,6,7,10,11,12], [], [], [], [], [5,6,7,10,11,12], [] ]
```

(2) Compute TotZ (a list of lists representing the selected worker absences per day), using di and $oabs_i$ to know which selected regular workers are absent on dc_i, and e_i to know if ew is selected for dc_i. In our example:

```
TotZ = [ [1, 2], [5], [5], [5], [5], [2, 3, 4], [5] ]
```

(3) For each dc_i traverse $Table_i$, binding to zero each variable indexed by $TotZ_i$. In our example:

```
Table = [[0,0,A,B,C],[D,E,F,G,0],[H,I,J,K,0],[L,M,N,O,0],[P,Q,R,S,0],[T,0,0,0,U],[V,W,X,Y,0]]
```

(4) Split Table by teams: {tt_1, …, tt_{nt}}, each of them an {(nd / nt) x (ntw + 1)} matrix. In our example:
tt = [[[0,0,A,B,C],[L,M,N,O,0],[V,W,X,Y,0]], [[D,E,F,G,0],[P,Q,R,S,0]], [[H,I,J,K,0],[T,0,0,0,U]]]
tt1 = [[0, 0, A, B, C], [L, M, N, O, 0], [V, W, X, Y, 0]]
tt2 = [[D, E, F, G, 0], [P, Q, R, S, 0]]
tt3 = [[H, I, J, K, 0], [T, 0, 0, 0, U]]

We also split dc to obtain the day classification associated to each tt_i:
dc_list = [[1,1,2],[1,1],[1,2]]
dc_list1 = [1, 1, 2]
dc_list2 = [1, 1]
dc_list3 = [1, 2]

And we compute the number of hours that each regular worker is expected to work. In our example, according to ws, each day of dc ≡ ws_1 (resp. ws_2) implies sixty six (resp. forty eight) working hours. So, for dc = [1, 1, 1, 1, 1, 2, 2] the timetable contains a total of four hundred and twenty six working hours. As there are twelve regular workers, each one is expected to work thirty five hours.
h = 35

4.3. tt_solve

We can run the stage by calling to: tt_solve ntw ws ef T h W SS tt dc_list == eh
In our example (considering just the solving of tt_1, from now on just tt) we run:
tt_solve 4 [[20,22,24],[24,24]] 2 1 35 fstUnbound calByWorkers [[0,0,A,B,C],[L,M,N,O,0],[V,W,X,Y,0]]
[1,1,2] == eh
We obtain the following solution:
tt = [[0, 0, 22, 24, 20], [0, 20, 24, 22, 0], [24, 24, 0, 0, 0]]
eh = 71

4.3.1. Sequence of steps (as they are provided in the paper):

(1) Transpose tt to obtain trans_tt, a matrix {(ntw + 1) x (nd/nt)} ordered by workers instead of by days. In our example:
trans_tt = [[0, L, V], [0, M, W], [A, N, X], [B, O, Y], [C, 0, 0]]

(2) For each day $dc_i = ws_j$ of tt: Parse ws_j to obtain the different working slots, say values v (and their cardinalities, say c). Initialize tt_i with domain v, and post a global constraint ensuring their distribution with (v, c). In our example:
belongs [0, 0, A, B, C] [0, 20, 22, 24]
distribute [2, 1, 1, 1] [0, 20, 22, 24] [0, 0, A, B, C]

belongs [L, M, N, O, 0] [0, 20, 22, 24]
distribute [2, 1, 1, 1] [0, 20, 22, 24] [L, M, N, O, 0]

belongs [V, W, X, Y, 0] [0, 24]
distribute [3, 2] [0, 24] [V, W, X, Y, 0]

(3) Compute ls (a list of the different kind of shifts s_x to be scheduled in tt). In our example:
ls = [0, 20, 22, 24]

(4) Compute ln (a mate list for ls, where each ln_i contains the number of shifts of type ls_i to be scheduled in tt). In our example:
ln = [7, 2, 2, 4]

(5) For each ls_i: Generate new (ntw+1) variables cv, each cv_k assigned to the amount of shifts of type ls_i the worker trans tt_k is assigned to, and the sum of cv constrained to be ln_i. In our example:

count 0 [0, L, V] == cv0_1; count 0 [0, M, W] == cv0_2; count 0 [A, N, X] == cv0_3; count 0 [B, O, Y] == cv0_4;
count 0 [C, 0, 0] == cv0_5;
sum [cv0_1, cv0_2, cv0_3, cv0_4, cv0_5] == 7

count 20 [0, L, V] == cv20_1; count 20 [0, M, W] == cv20_2; count 20 [A, N, X] == cv20_3;
count 20 [B, O, Y] == cv20_4; count 20 [C, 0, 0] == cv20_5;
sum [cv20_1, cv20_2, cv20_3, cv20_4, cv20_5] == 2

count 22 [0, L, V] == cv22_1; count 22 [0, M, W] == cv22_2; count 22 [A, N, X] == cv22_3;
count 22 [B, O, Y] == cv22_4; count 22 [C, 0, 0] == cv22_5;
sum [cv22_1, cv22_2, cv22_3, cv22_4, cv22_5] == 2

count 24 [0, L, V] == cv24_1; count 24 [0, M, W] == cv24_2; count 24 [A, N, X] == cv24_3;
count 24 [B, O, Y] == cv24_4; count 24 [C, 0, 0] == cv24_5;
sum [cv24_1, cv24_2, cv24_3, cv24_4, cv24_5] == 4

(6) Tight the distribution with T by constraining the differences of cv to be in the domain {-T, ..., T}. In our example:
aux0_1 == cv0_1 - cv0_2; aux0_2 == cv0_1 - cv0_3; aux0_3 == cv0_1 - cv0_4; aux0_4 == cv0_2 - cv0_3;
aux0_5 == cv0_2 - cv0_4; aux0_6 == cv0_3 - cv0_4;
domain [aux0_1, aux0_2, aux0_3, aux0_4, aux0_5, aux0_6] (-1) 1

aux20_1 == cv20_1 – cv20_2; aux20_2 == cv20_1 – cv20_3; aux20_3 == cv20_1 – cv20_4; aux20_4 == cv20_2 – cv20_3;
aux20_5 == cv20_2 – cv20_4; aux20_6 == cv20_3 – cv20_4;
domain [aux20_1, aux20_2, aux20_3, aux20_4, aux20_5, aux20_6] (-1) 1

aux22_1 == cv22_1 – cv22_2; aux22_2 == cv22_1 – cv22_3; aux22_3 == cv22_1 – cv22_4; aux22_4 == cv22_2 – cv22_3;
aux22_5 == cv22_2 – cv22_4; aux22_6 == cv22_3 – cv22_4;
domain [aux22_1, aux22_2, aux22_3, aux22_4, aux22_5, aux22_6] (-1) 1

aux24_1 == cv24_1 – cv24_2; aux24_2 == cv24_1 – cv24_3; aux24_3 == cv24_1 – cv24_4; aux24_4 == cv24_2 – cv24_3;
aux24_5 == cv24_2 – cv24_4; aux24_6 == cv24_3 – cv24_4;
domain [aux24_1, aux24_2, aux24_3, aux24_4, aux24_5, aux24_6] (-1) 1

(7) Compute the extra hours of tt via the sum of the hours of the regular workers (and a comparison with standard hours to be accomplished) plus the sum of the hours of ew x ef. In our example:
sum [0, L, V] == HoW1; HoW1 – 35 == EWL1;
(EWL1 >= 0) => (eH1 == EWL1); (EWL1 < 0) => (eH1 == 0);

sum [0, M, W] == HoW2; HoW2 – 35 == EWL2;
(EWL2 >= 0) => (eH2 == EWL2); (EWL2 < 0) => (eH2 == 0);

sum [A, N, X] == HoW3; HoW3 – 35 == EWL3;
(EWL3 >= 0) => (eH3 == EWL3); (EWL3 < 0) => (eH3 == 0);

sum [B, O, Y] == HoW4; HoW4 – 35 == EWL4;
(EWL4 >= 0) => (eH4 == EWL4); (EWL4 < 0) => (eH4 == 0);
sum [EWL1, EWL2, EWL3, EWL4] == Tot_EWL;
sum [C, 0, 0] == EWL5

eh = Tot_EWL + (2 * EWL5)

(8) Label tt to find the assignment that minimizes the extra hours. In our example:
labeling [toMinimize eh] [0, L, V, 0, M, W, A, N, X, B, O, Y, C, 0, 0]

4.4. tt_map

We can run the stage by calling to: tt_map nt ntw d tt == Timetabling
In our example we run:

```
tt_map 3 4 [1,2,3,1,2,3,1] [ [ [0, 0, 22, 24, 20], [0, 20, 24, 22, 0], [24, 24, 0, 0, 0] ], [ [0,20,22,24,0],
                                                    [24,22,20,0,0] ], [ [0,20,22,24,0], [24,0,0,0,24] ] ] == Timetabling
```

We obtain the following solution:

```
Timetabling = [ [ 0, 0, 22, 24, 0, 0, 0, 0, 0, 0, 0, 0, 20 ],
                [ 0, 0, 0, 0, 0, 20, 22, 24, 0, 0, 0, 0, 0 ],
                [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 22, 24, 0 ],
              [ 0, 20, 24, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
              [ 0, 0, 0, 0, 24, 22, 20, 0, 0, 0, 0, 0, 0 ],
              [ 0, 0, 0, 0, 0, 0, 0, 0, 24, 0, 0, 0, 24 ],
              [ 24, 24, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ]
```

5. Conclusions

In this technical report we have seen a concrete instance of the real-life employee timetabling problem presented in the paper “Applying CP(FD), CLP(FD) and CFLP(FD) to a Real-Life Employee Timetabling Problem”. We have presented a description of the concrete instance being used. Next, we have parameterized the solving approach to this concrete instance. Finally, we have parameterized the running algorithm to the instance, in particular showing the variables, constraints posted to the solver as well as the additional data structures used to compute the solution.