Modelling & solving the multiple couriers planning problem

Caroli Giacomo (giacomo.caroli4@studio.unibo.it)
Galfano Lorenzo (lorenzo.galfano@studio.unibo.it)
Torzi Luca (luca.torzi@studio.unibo.it)

1 Introduction

1.1 Description of the problem

The assignment given consists in solving the Multiple Courier Problem.

1.2 Work management

Every part of the assignment has been thought, designed and realized together. The first step was to analyze the problem and create a proper model to work with, once the model seemed good enough we'd start the implementation part. Lastly, we had to deal with optimizations for each model that didn't come up in the first part.
The project lasted roughly 2 to 3 months, the main difficulties have been:

- Find a suitable model for each technology used in the problem.
- Getting used to work with SAT variables and encodings.

1.3 Common Parameters

For the given problem, the parameters necessary to encode the input variables are:

- m: the number of couriers
- n: the number of items to deliver
- l: an array where l_i is the weight capacity of the courier i
- s: an array where s_i is the weight of the item i
- D: a matrix where each $D_{i,j}$ is the distance between distribution point i and distribution point j

1.4 Preprocessing

The input files provided (.dat) contain only rows of numbers, in a certain known order, representing the parameters listed above. In order to be able to correctly use them in each model, we've defined a function that reads the input file and outputs the five parameters necessary.

Moreover we check if some couriers cannot carry any package and, if it is the case, we remove them from any initial variable. At the end we will add an empty list in the json file to signal that the courier is not carrying any item.

1.5 Decision variables

We only have one decision variable, called *tours*, representing in each row the path that a courier takes to deliver his packages; its second dimension is $n-m+3$, because we have n packages, each courier has to carry at least 1 package (so we subtract m and add $+1$), and we know that it starts and ends its journey at the origin point ($+2$).

It is necessary to point out that for the rest of this paper we will be using a variable called *second_dim* referring to $n - m + 3$, we chose this name simply because it is the "second" dimension of several variables (among them, the most important is *tours* matrix) which will be highly used throughout the various models.

1.6 Objective variable

As the objective variable we chose to use an integer, that stores the value of the maximum distance travelled among all couriers. The goal is to minimize that value in order to obtain a fair division among drivers.

2 CP model

2.1 Decision variables

As described in section 1.5 the only decision variable is $tours$. It is a matrix which has the number of rows equivalent to the number of couriers and the number of columns equivalent to *second_dim*. The domain of this variable goes from 1 to $n + 1$ (where $n + 1$ represents the origin point).

2.2 Objective function

The objective variable is $max\_distance\_of\_tours$ (described in section 1.6).
The objective function minimizes this value by looping through the *tours* matrix, getting the value of the distance travelled by each courier, computing the maximum and backtracking until we find a minimum.

2.3 Constraints

- Each courier must have a total weight of items lesser or equal than the total weight it can carry.
- Each courier must start from the origin point.

$$\forall i \Big(tours_{i,0} = n+1 \Big) \quad i \in 0..m-1 \tag{1}$$

- Given the *tours* matrix, if we have the value of a point i, j where $j > 2$ and $tours_{i,j}$ is equal to the origin point, than every value of that row following must be equal to the origin point, because the courier ended its tour.

$$\forall i \forall_{j=2}^{second\_dim-2} \Big(tours_{i,j} = n+1 \implies tours_{i,j+1} = n+1 \Big), i \in 0..m-1 \tag{2}$$

- Every item that has to be delivered, must be present exactly one time in the *tours* matrix, leaving us with $((second\_dim) * m) - n$ times the origin points in the matrix. This constraint is imposed using the global constraint called *global_cardinality_low_up*.
- In the *tours* matrix, each element in the position $i, 1$ must be different from the origin point, because we have that the number of items is always greater than or equal to the number of couriers, so to obtain a fair division is better to assign to each courier at least an item.

$$\forall i \Big(tours_{i,1} \neq n+1 \Big) \quad i \in 0..m-1 \tag{3}$$

2.3.1 Simmetry breaking constraints

If two couriers can carry the same items, then the solution in which they switch the packages is not considered.
For example:

- courier 1 has to deliver items 1,2, courier load size is 9 and the sum of the weights of the items is 8.
- courier 2 has to deliver items 3,4, courier load size is 10 and the sum of the weights of the items is 7.

The solution in which courier 1 carries items 3,4 and courier 2 carries items 1,2 is ignored, because lexicographic order between the rows of the *tours* matrix (which comply with this constraint) is established.

2.3.2 Implied constraints

Every courier must end its tour at the origin point, this is implied by the third and fourth constraint stated before.

If two couriers have the same load size, we don't consider the solutions in which they switch packages, this constraint is implied by the symmetry breaking one showed above.

2.4 Validation

2.4.1 Experimental design

To run all the models, the Minizinc API on Python was used, testing two different solvers ("Chuffed" and "Gecode"). We then further investigated different approaches by inserting the symmetry breaking described above and using different search strategies.

For all tests in the paper, we operated on the same machine at our disposal:
Lenovo IdeaPad Gaming 3, Intel Core i5 (11300H) 3.1 GHz, 16 GB DDR6.
OS: Pop!_OS 22.04.
Each instance was executed with a time limit of 300 seconds.

2.4.2 Experimental results

ID	gecode firstFail-rand	gecode firstFail-rand + SB	chuffed firstFail	chuffed firstFail + SB	gecode dom_w_deg rand	gecode dom_w_deg rand + restart
1	**14**	**14**	**14**	**14**	**14**	**14**
2	**226**	**226**	**226**	**226**	**226**	226
3	**12**	**12**	**12**	**12**	**12**	**12**
4	220	**220**	220	220	220	220
5	**206**	**206**	**206**	**206**	**206**	**206**
6	**322**	**322**	**322**	**322**	**322**	**322**
7	167	N/A	167	167	183	167
8	186	**186**	**186**	**186**	186	186
9	436	**436**	436	436	436	436
10	244	**244**	244	244	244	244
11	1249	N/A	1274	1410	1831	1524
12	822	N/A	764	785	1358	1290
13	N/A	N/A	1806	1806	N/A	1206
16	286	N/A	286	286	746	286
19	453	N/A	335	472	1367	1475
21	1715	N/A	1588	1872	2321	1721

Table 1: CP experimental results

In the table 1 are represented the experimental results obtained. If a cell has N/A then no solution was found in 300s. The name of the column is the

method we used to run the tests (SB stands for symmetry breaking).
Results not shown give N/A as solution for every method tried.

3 SAT

3.1 Decision variables

In order to meet the encoding requirements of SAT, thus to have only True/False Boolean variables, we added an extra dimension to all input parameters, where the last dimension corresponds to the **binary** encoding of each data item. The *encoding_depth* is calculated by considering the bits required to encode the greatest element possible, for each matrix.

Excluding auxiliary variables, the only decision variable present is:

- tours: $(m) \times (second\_dim) \times (encoding\_depth)$ matrix representing, for each row, the index encoding of delivery points for all couriers.

3.1.1 Auxiliary variables

As stated above, the only decision variable is $tours$, but auxiliary variables have been extensively used in the implementation. We list them below:

- capacities: $(m) \times (encoding\_depth)$ matrix representing the maximum load that can be carried by each courier. The second dimension is representative of binary encoding of values.
- weights: $(m) \times (second\_dim) \times (encoding\_depth)$ matrix representing, for each row, the weight encoding of each item assigned to each courier.
- distances: $(m) \times (second\_dim - 1) \times (encoding\_depth)$ matrix representing the distance traveled by the individual courier, coding each cell in each row as the distance value between two different deliveries.

Summarizing the above, a simplified version of the matrices in a problem with 4 parcels and 3 couriers can be represented as follows: (Please note how the integers are encoded in binary by adding an additional dimension to each of the underlying structures)

$$\begin{bmatrix} 0 & 1 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 4 & 0 & 0 \end{bmatrix}$$
Tours

$$\begin{bmatrix} 0 & 10 & 8 & 0 \\ 0 & 9 & 0 & 0 \\ 0 & 12 & 0 & 0 \end{bmatrix}$$
Weights

$$\begin{bmatrix} 15 & 2 & 0 \\ 7 & 0 & 0 \\ 4 & 0 & 0 \end{bmatrix}$$
Distances

- Elements 1,2,3,4 in Tours are the delivery indexes that will be binary encoded in the third dimension and the 3 rows represent the respective available couriers.

- Elements 10,8,9,12 in Weights represent the weight of the individual item delivered to the respective index. The encoding of each integer, as already illustrated, is in binary format and constructed using a *third dimension* (*encoding_depth*) (not shown here)
- Elements 15,2,7,4 in Distances represent distance measurements between adjacent delivery points. Same considerations as above for integers coding.

3.2 Objective function

In order minimize the greatest distance travelled by any courier, and thus to fulfill the requirement of the problem, we act as follows:

1. Compute a first solution to understand if a solution exists, whatever its maximum distance value is.
2. Iterate, via push() and pop() methods, in search of better solutions using binary search approach, adding new and more strict constraints for each cycle.

The practical implementation is as follows: from the boolean variables in the model, we reconstruct the distance matrix, appropriately translated into decimal format. Once the complete matrix is obtained, it is used to calculate the maximum distance, intended as the maximum sum among the different rows of the matrix.
The largest value is then considered in order to minimize the maximum distance traveled to comply with the initial requirement of the problem itself.

3.3 Constraints

- The binary encoding of each item must appear only once in the $tours$ matrix. To do it we use the *exactly_one_np* function explained during the laboratory sessions: this function performs the *logic and* between *at_least_one_np* and *at_most_one_np*.

$$\forall i \Big(exactly\_one\_np(encodings_i) \Big) \quad i \in 1..n \tag{4}$$

  $encodings_i$ is a list created as follows:

  1. loop through the whole $tours$ matrix along the rows and columns (i,j)
  2. loop for every character in the binary encoding of the item i (i.e. a string composed of 0 and 1, whose length is equal to k)
  3. perform the logic AND between $\neg tours_{i,j,k}$ if the value is 0 or $tours_{i,j,k}$ itself otherwise
  4. add it to the list

We perform this operation such that the logic *and* returns $True$ if the encoding (in the $tours$ matrix) represents the item i. So there must be only one of this *and*s that is $True$ such that one item is present exactly once.

Implementation of the functions used in *exactly_one_np* (x is a generic array with n elements):

- *at_least_one_np*

$$\bigvee x \tag{5}$$

- *at_most_one_np*

$$\bigwedge_{0 \leq i < j < n} \Big(\neg (x_i \wedge x_j) \Big) \tag{6}$$

- In the Tours matrix, for each row, the first element must be zero, which is the origin point representation.

$$\forall i \Big(\bigwedge_{k=0}^{depth\_tours-1} \neg tours_{i,0,k} \Big) \quad i \in 0..m-1 \tag{7}$$

- In the Tours matrix, for each row, if a zero appears in any position different than column zero, all following values must be zero, which is the origin point representation.

$$\forall i \forall_{j=2}^{second\_dim-2} \Big(\Big(\bigwedge_{k=0}^{depth\_tours-1} \neg tours_{i,j,k} \Big) \implies \Big(\bigwedge_{k=0}^{depth\_tours-1} \neg tours_{i,j+1,k} \Big) \Big) \quad i \in 0..m-1 \tag{8}$$

- Each courier must have at least one item, to satisfy the constraint of equal division of work among couriers.

$$\forall i \Big(\bigvee_{k=0}^{depth\_tours-1} tours_{i,1,k} \Big) \quad i \in 0..m-1 \tag{9}$$

- The $tours$ matrix must have a number of zeros equal to $m \times (second\_dim) - n$, to allow proper construction of the $tours$ matrix, in which each item appears only once and all other positions are zero-coded.
  To do it we use the sequential encoding of the *at_least_k* function (because there cannot be more than the number that we pass to it, so we do not use the *exactly_k* function to avoid to add more constraints).
  The *at_least_k* function receives a list containing the *encoding* of the zeros (as explained in the first constraint). Then this function calls the

at_most_k passing the negation of each item in the input list and the k will be $n - k$ (where n is the length of the input).

Below there is the the implementation of the *at_most_k* function. $n * k$ new variables are introduced (named $s_{i,j}$) to indicate that the sum has reached to j by i.

$$
\begin{array}{l}
(\neg x_1 \vee s_{1,1}) \\
(\neg s_{1,j}) \qquad \text{for } 1 < j \leq k \\
(\neg x_i \vee s_{i,1}) \\
(\neg s_{i-1,1} \vee s_{i,1}) \\
(\neg x_i \vee \neg s_{i-1,j-1} \vee s_{i,j}) \quad \Big\} \quad \text{for } 1 < j \leq k \quad \Big\} \text{for } 1 < i < n \\
(\neg s_{i-1,j} \vee s_{i,j}) \\
(\neg x_i \vee \neg s_{i-1,k}) \\
(\neg x_n \vee \neg s_{n-1,k})
\end{array}
$$

- For each row of the Weights matrix, the sum in binary of each element of the row, thus of the weight representation of each item, must be less than the capacity associated with the courier corresponding to the respective row of the Weights matrix. The constraint is realized with the binary adder shown below. The comparison between total package weight and capacity is implemented with the binary subtracter, then using the carry of the binary operation as a boolean flag.

3.3.1 Binary operation

Binary adder
Given two lists of boolean elements a and b having the same size (if one of them is smaller, then it will be padded adding zeros at the beginning), we will produce c, the list of carries (that have a length equal to a and $b + 1$), and d, the sum expressed in binary (that have a length equal to a and b). At the end we will return as result the most significant carry concatenated with d.
N.B. in our encoding the most significant digit is the one with index 0.

To produce c and d we use the constraints below:

- the less significant carry must be false $\neg c_{max\_len-1}$
- encoding of each bit of d

$$
\forall_{i=0}^{|d|-1} \Big(d_i \iff \Big((a_i \wedge \neg b_i \wedge \neg c_{i+1}) \vee (\neg a_i \wedge b_i \wedge \neg c_{i+1}) \vee \\
(\neg a_i \wedge \neg b_i \wedge c_{i+1}) \vee (a_i \wedge b_i \wedge c_{i+1}) \Big) \Big) \tag{10}
$$

- encoding of each carry

$$
\forall_{i=0}^{|d|-1} \Big(c_i \iff \Big((a_i \wedge b_i) \vee (a_i \wedge c_{i+1}) \vee (b_i \wedge c_{i+1}) \Big) \Big) \tag{11}
$$

Binary subtraction
Given two lists of boolean variables (representing the binary encoding of two numbers), we performe the subtraction as follows:

1. pad with zeros the two numbers if they have a different length
2. negate the encoding of each digit of the subtracting, such that we obtain a reversed version of it
3. add one to the new subtracting and consider only the d returned by the binary adder (i.e. we must remove the first element of the list, representing the most significant carry)
4. perform a sum between the minuend and the new subtracting
5. return the most significant carry bit (representing the sign bit of this operation).

This bit is equal to $True$ if the resulting operation leads to a positive number, $False$ otherwise.

3.3.2 Symmetry breaking constraints

Same described in section 2.3.1.

To obtain this result, we used the *binary_subtraction* referred in the section above, to check firstly if both couriers could carry the items of the other courier (named $check1_{i,j}$ and $check2_{j,i}$ below), and then another subtraction between the first two items carried by these couriers to establish lexicographic order (called $lex_{i,j}$ in the formula).

$$\forall_{i=0}^{m-1} \forall_{j=i+1}^{m-1} \Big((check1_{i,j} \wedge check2_{j,i}) \implies lex_{i.j} \Big) \tag{12}$$

3.3.3 Implied constraints

Refer to second implied constraint described in section 2.3.2.

3.4 Validation

3.4.1 Experimental design

The default setting of Z3 has been used. The device used to benchmark this method is described in section 2.4.1.
To implement the timeout and stop the executing process, we used the Python Thread class. The solver runs in a separated thread and it is stopped after 300 seconds; in order to avoid conflicts, the exchange of intermediate results is implemented via a lock system.

3.4.2 Experimental results

ID	SAT	SAT + SB
1	**14**	**14**
2	226	**226**
3	**12**	**12**
4	273	**220**
5	**206**	**206**
6	**322**	**322**
7	177	320
8	205	**186**
9	436	436
10	244	244

Table 2: SAT experimental results.

Results from 11 to 21 are not shown because no solution was found in both cases, i.e. N/A.
We can see that SAT with symmetry breaking performs slightly better than it's counterpart without it (with the only exception of instance n. 7).

4 SMT

4.1 Decision variables

Unlike SAT, SMT doesn't require Boolean variable encodings, hence the definition of such variables was eased.
We decided to use Integer Arithmetic theory, *tours* is an array of IntVectors, defined as follows: $(m) \times (second\_dim)$ matrix representing, for each row, the index of delivery points for all couriers.

4.1.1 Auxiliary variables

We used two auxiliary vaiables called *effectiveWeights* and *effectiveDistances* that are the counterparts of the auxialiary variables called *weights* and *distances* described in section 3.1.1. They are represented using IntVectors (obviously here there is no third dimension).

4.2 Objective function

To minimize the maximum distance travelled among all couriers, binary search is deployed the same way discussed in SAT section 3.2, the only difference is that no binary encoding is used, hence no translation from binary to integer is needed.

4.3 Constraints

- Each position in the *tours* matrix must be between 0 and n. 0 represents the starting point (ending point) while n is one of the possible delivery positions. For each matrix entry, we then impose both $<= n$ and $>= 0$.

$$\forall i \forall j \Big(tours_{i,j} <= n \Big) \quad i \in 0..m-1, j \in 0..second\_dim - 1 \tag{13}$$

$$\forall i \forall j \Big(tours_{i,j} >= 0 \Big) \quad i \in 0..m-1, j \in 0..second\_dim - 1 \tag{14}$$

- For each row of the *tours* matrix, so for each route assigned to each courier, the starting position is imposed to be at the starting point (value = 0)

$$\forall i \Big(tours_{i,0} = 0 \Big) \quad i \in 0..m-1 \tag{15}$$

- Each courier must have assigned at least one package in order to meet the requirement of a fair division of workload among couriers.

$$\forall i \Big(tours_{i,1} > 0 \Big) \quad i \in 0..m-1 \tag{16}$$

- In the *tours* matrix, each courier route must end by returning to the starting (end) point (value = 0). This is done by imposing the last cell of each row to 0.

$$\forall i \Big(tours_{i,second\_dim-1} = 0 \Big) \quad i \in 0..m-1 \tag{17}$$

- To comply with the logic of the problem, it is imposed that each delivery point appears only once in the entire *tours* matrix and thus is visited only once among all couriers. It is excluded from the constraint the location 0, common among all couriers, being the starting (end) point.
  The argument received from the functions *AtMost* and *AtLeast* (built-in in Z3) is a list containing the expressions $tours_{i,j} == k$ (for every i and j) and a 1, indicating that we are doing an *AtMostOne* and *AtLeastOne*.

$$\forall k \Big(AtMostOne(tours_{i,j} = k) \Big) \quad i \in 0..m-1, j \in 0..second\_dim - 1 \tag{18}$$

$$\forall k \Big(AtLeastOne(tours_{i,j} = k) \Big) \quad i \in 0..m-1, j \in 0..second\_dim - 1 \tag{19}$$

- In order to avoid "holes", it is imposed that if a given position index in the *tours* matrix is 0, then all the following must be 0 (end point). This forces couriers to choose a delivery route that does not include going back to the starting point during the route.

$$\forall i \forall_{j=2}^{second\_dim-2} \Big(tours_{i,j} = 0 \implies tours_{i,j+1} = 0 \Big), i \in 0..m-1 \quad (20)$$

- The maximum load assigned to each courier, defined as the sum of the packages (sum of the elements of $effectiveWeight$) it has to load for a route (sequence of delivery indexes), must be less than its maximum capacity.

- In order to contribute to the ultimate goal of the model, and thus to minimize the maximum distance travelled, the solver is also used to insert a support constraint that is used to build the decision variable $effectiveDistances$.

It is understood that within the code, there is extensive use of the solver as a tool for constraining and assigning values in a variety of other operations, not reported here, but whose purpose is to contribute to the points above. Indeed, these are simple form operations useful in simplifying and supporting the management of the code and the relationship between data and solver.

4.3.1 Symmetry breaking constraints

Same described in section 2.3.1.

4.3.2 Implied constraints

Refer to second implied constraint described in section 2.3.2.

4.4 Validation

4.4.1 Experimental design

Same discussed in section 3.4.1.

4.4.2 Experimental results

ID	SMT	SMT w/ SB
1	**14**	**14**
2	**226**	**226**
3	**12**	**12**
4	**220**	**220**
5	**206**	**206**
6	**322**	**322**

7	282	189
8	**186**	**186**
9	**436**	436
10	244	**244**

Table 3: SMT experimental results.

Results from 11 to 21 are not shown because no solution was found in both cases, i.e. N/A.

5 MIP

5.1 Decision Variables

tours: a three-dimensional matrix (i, j, k) that is used to represent and manage the routes of individual couriers.
The encoding of this matrix is similar to the one defined in section 3.1, but instead of using the binary encoding to encode the numbers, we use one-hot encoding.

5.1.1 Auxiliary variables

distance_of_tours: a vector of length m, which stores the distance travelled by each courier to end its tour (that is taken from the tours matrix described before). So, for clarification, the first entry of $distance\_of\_tours$ will have the sum of the distances traveled by the first courier so the total distance operated, the second entry will refer to the second courier, and so on.

5.2 Objective function

The objective function is computed as the maximum of the $distance\_of\_tours$ vector (having length m), using the auxiliary function getMax() [1].
The return value of $getMax()$ function is a new LpInteger variable called y. To compute its value, m new binary variables called d_i are introduced. d_i will be equal to 1 if $distance\_of\_tours_i$ contains the maximum value, 0 otherwise; we then impose that the sum of these variables must be equal to 1 (such that there is only one "max").
Lastly, we impose that $y \geq distance\_of\_tours_i$ and $y \leq distance\_of\_tours_i + (upperbound) * (1 - d_i)$, for every element in $distance\_of\_tours$.
Please note how the auxiliary function called forceAnd() [2] has been introduced to emulate the operation of logical AND and facilitate the construction of the $distance\_of\_tours_i$ matrix.

It's proper to point out some details:

1. Following some pre-processing of the input data, to make the task easier to compute for our model, the origin is not the $(n + 1)^{th}$ row/column of the D matrix but the first one.

2. The upper bound value of our objective function is set as the sum of the first row and the first column, which is the case of a courier having to distribute all the packages and returning to the origin after every single delivery.

5.3 Constraints

- Each item must be delivered exactly one time. To verify this, we sum over the i and j-dimension and if the result is equal to 1 then the condition is met meaning we have each item exactly once.

$$\forall z \Big(\sum_{i=0}^{m-1} \sum_{j=0}^{second\_dim-1} tours_{i,j,z} = 1 \Big) \quad z \in 1..n \tag{21}$$

- Each courier must have a distinct position during the various time steps, in other words: we want for each instant (horizontal matrix index j) the courier to be indicated as being either positioned at starting point (bit flag 1 on starting position $z = 0$) or at another position (bit flag 1 on delivery point $z! = 0$).

$$\forall i \forall j \Big(\sum_{z=0}^{n} tours_{i,j,z} = 1 \Big) \quad i \in 0..m-1; j \in 0..second\_dim - 1 \tag{22}$$

- The first column will be the origin point for each courier, thus having the first value of the depth equal to 1 (following the one-hot encoding).

$$\forall i \Big(tours_{i,0,0} = 1 \Big) \quad i \in 0..m-1 \tag{23}$$

- Each courier must deliver at least one package, meaning that the second column must have at least one value set to 1 in the third dimension.

$$\forall i \Big(\sum_{z=1}^{n} tours_{i,1,z} = 1 \Big) \quad i \in 0..m-1 \tag{24}$$

- In our model the courier returns to the origin only after having completed the tour, which means that we can't have "holes" in our matrix,i.e. after a 0 there must be only other 0s. (if position index $j >= 2$ i.e. at least one package delivered)

$$\forall i \forall j \Big(\sum_{z=1}^{n} tours_{i,j,z} >= \sum_{z=1}^{n} tours_{i,j+1,z} \Big) \quad i \in 0..m-1; j \in 2..second\_dim - 2 \tag{25}$$

- At last, the sum of the weights of the packages that each courier has to deliver must be smaller than the capacity of the courier.

$$\forall i \Big(\sum_{z=1}^{n} \Big(\sum_{j=0}^{second\_dim-1} tours_{i,j,z} \Big) * s_{z-1} <= l_i \Big) \quad i \in 0..m-1 \tag{26}$$

  Where s is the vector containing the weights of each packet.

5.4 Validation

5.4.1 Experimental design

The device used to benchmark this method is described in section 2.4.1.

5.4.2 Experimental results

ID	PULP_CBC_CMD	GLPK_CMD
1	**14**	**14**
2	226	226
3	**12**	**12**
4	220	220
5	**206**	**206**
6	322	322
7	N/A	345
8	186	186
9	436	436
10	289	244

Table 4: MIP experimental results

Results from 11 to 21 are not shown because no solution was found in both cases, i.e. N/A. In the table 4 we could see that the second solver perform a little bit better than the first one.

6 Conclusions

With all four modeling techniques we were able to solve, almost always, instances from 1 to 10. For later instances, we very often arrived at the result N/A indicating that the execution would take longer (over 300s) due to the instrinsic complexity of the instances.
In conclusion, the best results, for the same instance, were obtained on CP, which turned out to be the lightest model of the four, having a more low-level approach, moreover we could tune the search strategy. SMT can be a good second choice and it is possible to use, unlike SAT, integer value and not boolean. The latter feature in fact complicated the work a lot during the modeling of SAT, always having to reason with boolean constraints. For this reason the creation of the model in SAT takes a lot of time: it needs to create the boolean encoding of each variable and constraint. Finally, the MIP model, turned out to be not exactly efficient, probably because of the one-hot encoding choice.

References

[1] Magic code. linear programming set a variable the max between two another variables. https://math.stackexchange.com/q/2446606, 2017-09-26.

[2] D.W. (https://cs.stackexchange.com/users/755/d w) Express boolean logic operations in zero-one integer linear programming (ilp). https://cs.stackexchange.com/q/12118, 2020-04-30.