Combinatorial Decision Making and Optimization

Constraint Programming

Omar G. Younis
omargallal.younis@studio.unibo.it

July 2021

Contents

1 Basic Model

We present here the basic model to solve the problem.

$$\begin{aligned} \min \quad & h \\ \text{s.t.} \quad & x_i + widths_i \leq w \qquad \forall i = 1, 2, ..., n \\ & y_i + heights_i \leq h \qquad \forall i = 1, 2, ..., n \\ & diffn(x, y, widths, heights) \end{aligned} \tag{1}$$

where *x, y, widths* and *heights* are arrays of length n containing, respectively, the x-axis positions, the y-axis positions, the widths and the heights of the rectangles. We denote x_i to indicate the x-axis position for the i-th element; similarly for y_i, $widths_i$ and $height_i$. Moreover, when we say that a rectangle is in the position (x, y) we are referring to the bottom-left corner, as explained in Figure 1. The function *diffn* is a *global constraint* of *MiniZinc* and checks that the rectangle doesn't overlap each other, as explained in the *MiniZinc* documentation[1]. Notice also that *n, w, widths* and *heights* are instance constants; other ones are variable. Naively, in this first model, we impose the following bounds for h:

$$\max \ heights \leq h \leq \sum_{i=1}^{n} heights_i$$

We will improve this bounds in the following section.

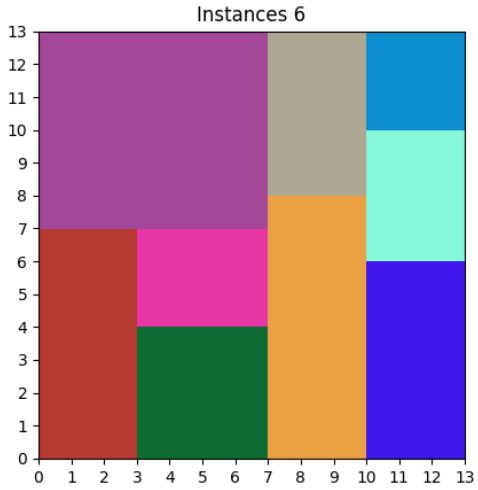


Figure 1: An example of a solution. The (x, y) position of the yellow rectangle is (0, 7).

2 Domain bounds for h

We can improve the efficiency of the search choosing wisely the domain bounds of h. In particular, notice the optimal case in Figure 1 in which we don't have blank spaces. In general:

$$\left\lceil \frac{\sum_{i=1}^{n} widths_i * heights_i}{w} \right\rceil \leq h$$

i.e. the total area of rectangles divided by w and rounded up should be less or equal than h. This lower bounds can be not so strict in some cases, for example when we have a very high rectangle as in Figure 2. In this case, we can take as lower bound the greater height.

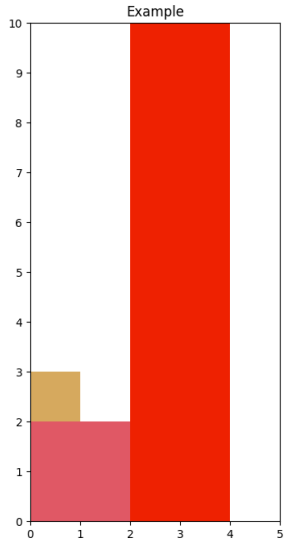


Figure 2: An example of a particular case in which we have a very high rectangle and few small rectangle to cover the blank space.

We can also improve the upper bound. For this one, we divided the rectangles in different subset:

$$W_j \subseteq widths \quad such\ that\ (\sum_{e \in W_j} e) \leq w$$

and we denote with H_j the set of heights of the rectangles associated with W_j and we imposed the following upper bound:

$$h \leq \sum \max H_j$$

We report in Table 1 the solving statistics for the basic model 1 and for the domain bounds model presented in this section.

	Basic Model			**Domain Bounds**		
ins	solveTime	propagations	failures	solveTime	propagations	failures
1	0.000123	93	5	0.000105	36	1
2	0.012001	44	3	0.000108	44	3
3	0.003872	91	4	0.00012	46	1
4	1.21969	3841818	273297	0.007934	13142	1755
5	0.136236	180215	22975	0.015152	18779	2650
6	1.98381	3322537	452291	0.005351	7168	1073
7	0.005843	5755	950	0.000427	337	62
8	0.000228	144	19	0.000232	144	19
9	21.121	25785380	3173640	2.67071	3150778	381721
10	*300*	*281148992*	$3{,}1 \cdot 10^7$	0.127752	123458	21098

Table 1: Comparison of output statistics between the two proposed models. Notice that the 10th instance is not proved by the basic model.

3 Implied constraint

We add to our model two implied contraint as suggested in point 2 of the assignment text. To do that we can rely on the global constraint *cumulative* to improve performance. In particular we have added:

$$\textit{cumulative(x, widths, heights, h)}$$
$$\textit{cumulative(y, heights, widths, w)}$$

i.e. we check respectively that height and width of the available space are not exceeded by the rectangles. We report, in Table 2 a comparison between output statistics of the last two models.

	Domain Bounds			Implied Constraint		
ins	solveTime	propagations	failures	solveTime	propagations	failures
1	0.000105	36	1	0.000127	48	1
2	0.000108	44	3	0.000136	58	3
3	0.00012	46	1	0.000158	79	3
4	0.007934	13142	1755	0.000576	633	47
5	0.015152	18779	2650	0.000335	231	28
6	0.005351	7168	1073	0.000238	147	5
7	0.000427	337	62	0.000638	507	82
8	0.000232	144	19	0.000755	560	87
9	2.67071	3150778	381721	0.000325	198	21
10	0.127752	123458	21098	0.000541	300	32
11	*300*	*189996497*	$2{,}7{\cdot}10^7$	151.932	74494992	$1{,}5{\cdot}10^7$

Table 2: Comparison of output statistics between the domain bounds model and the one with implied constraint. Notice that the 11th instance is not proved by the domain bounds, while the implied constraint managed to solve it.

4 Symmetry breaking

We have founded two possible symmetry breaking constraint. First, the problem is symmetric with respect to x-axis and y-axis. This mean that, starting from a solution, we can find 3 equivalent solution just by flipping axis; see Figure 3 for an example. To avoid to search over these solution, we just impose that the center of the first rectangle should be in the bottom-left quadrant of the available space.

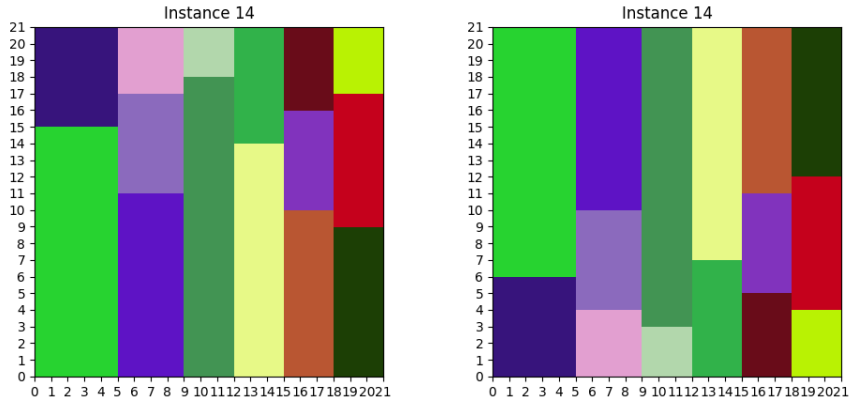


Figure 3: Two equivalent solutions after a flipping of the y-axis

Another symmetry breaking constraint that we have imposed is related to

equals rectangles. In fact, if two rectangles have the same sizes, it is possible to interchange them without change the solution. Formally, we have imposed this constraint:

$\forall i, j \quad with \quad i < j :$

$$(widths_i = widths_j) \wedge (heights_i = heights_j) \rightarrow lex\_less([x_i, y_i], [x_j, y_j])$$

where *lex_less* is the global constraint of MiniZinc. We report in the following table the comparison of output statistics of the last two models.

	Implied Constraint			**Symmetry Breaking**		
ins	solveTime	propagations	failures	solveTime	propagations	failures
11	151.932	74494992	$1{,}5 \cdot 10^7$	*300*	*141211731*	$2{,}5 \cdot 10^7$
12	0.008487	4112	798	0.0152111	9410	1779
13	0.010304	5236	949	7.01119	3715205	704247
14	0.038204	19549	3871	*300*	*149715391*	$2{,}9 \cdot 10^7$
15	0.025739	12632	2662	0.0557981	29036	5779
16	*300*	*129111495*	$2{,}5 \cdot 10^7$	*300*	*120722388*	$2{,}2 \cdot 10^7$
17	2.9583	1356124	266818	0.87102	412857	81990
18	0.73571	301410	59447	0.16312	72740	13145
19	31.1234	9910520	2171106	*300*	*85582870*	$1{,}7 \cdot 10^7$
20	30.2428	9973078	2008609	8.92809	2929757	598816
21	7.09029	2238018	427538	1.72069	567859	110092
22	*300*	*88206987*	$1{,}8 \cdot 10^7$	116.008	31754040	6777154
23	*300*	*179871362*	$3{,}7 \cdot 10^7$	0.0013560	509	29
24	9.48773	4085731	765364	0.0005890	262	9

Table 3: Comparison between output statistics of the model with symmetry breaking constraints and the one without them.

Even if, some instances get an outstanding improvement from symmetry breaking (e.g. instance 23), for others these constraints seems to make matter worse; in particular for instances 11, 14 and 19. The problem is related to the first symmetry breaking constraint which imposed the quadrant position of the *first* rectangle. In fact, in our instances, usually the first rectangle is the smaller one which is the easiest one to place; this goes against the first fail principle. To solve this problem, we have ordered the rectangle by their area, so now we constraint the larger one. Done this, we recompute the output statistics for implied constraint and symmetry breaking models for those instances whose the latter performs worse.

	Ordering+Implied Constraint			**Ordering+Symmetry Breaking**		
ins	solveTime	propagations	failures	solveTime	propagations	failures
11	190.372	90109066	$1.8{\cdot}10^7$	230.788	112541498	$1.8{\cdot}10^7$
14	0.016833	8720	1813	0.0172721	10075	1813
16	229.499	93108098	$1.8{\cdot}10^7$	250.613	98493691	$1.8{\cdot}10^7$
19	202.923	59879261	$1.3{\cdot}10^7$	241.48	75560348	$1.3{\cdot}10^7$

Even if the symmetry breaking model continues to perform worse for these particular instances (but better in general), now the two solve times are comparable.

5 Search Annotations

To guide the solver during the search, we have tried a lot of different search annotations. Each of them have their own pros and cons and perform well for particular instances, but bad for others. For example, one of the most promising search annotations that we have found is this one:

```
seq_search([
    int_search(y, input_order, indomain_min) ::
    int_search(x, input_order, first_fail),
    int_search([h], input_order, indomain_min),
])
```

this means that the solver first place all rectangle starting from the left-bottom and then find a solution with the minimum available value for h. After that, the solver can continue with search imposing that new solutions must be better, i.e have a lesser value for h. One of the problem with this idea is that, for some instances, the solver can be stuck in placing rectangles and never find a solution. At the end, we have opted for this search annotation:

```
int_search([h], input_order, indomain_median)
```

this can be very efficient because, if the median of the h domain is a feasible solution, this throw away half of the search space.

As usual, we report in the following table the output statistics for this last model and the previous one.

	Ordering+Symmetry Breaking			**Search Annotations (Median)**		
ins	solveTime	propagations	failures	solveTime	propagations	failures
11	230.788	112541498	$1.8{\cdot}10^7$	0.0120821	6383	975
12	0.032084	18277	3292	0.033391	18812	3337
13	0.011135	5514	998	0.013926	7119	1123
14	0.0172721	10075	1813	0.033136	16797	2411
15	0.024635	12427	2616	0.028822	13430	2725
16	250.613	98493691	$1.8{\cdot}10^7$	0.0453651	17378	1744
17	2.96767	1348918	265098	0.05808	25511	2896
18	0.735713	299097	58984	18.4366	7893401	795192
19	241.48	75560348	$1.3{\cdot}10^7$	2.86808	820318	93534
20	33.4343	10259269	1989138	34.1024	10571965	2011152
21	8.1903	2492068	443191	0.148044	43289	5434
22	*300*	*83456534*	$1{,}5{\cdot}10^7$	149.351	42251159	4790452
23	0.0008480	371	17	0.003208	1471	89
24	0.0005080	225	4	0.00234101	900	25
25	*300*	*73437917*	$1{,}7{\cdot}10^7$	*300*	*78143731*	$1{,}5{\cdot}10^7$

There are some outstanding improvement that we want to highlight: notices the output statistics differences for instances 11, 16 and 19.

We also have found that using *smallest* as variable selection choice for y seems to be a good choice. In MiniZinc, *smallest* means that the solver select the variable that have the smallest value in its domain. In particular we want to try two different possibilities that we call *Smallest y* and *Median + Smallest y*. The first one is simply:

```
seq_search([
    int_search(y, smallest, indomain_min) ::
    int_search(x, first_fail, indomain_min),
    int_search([h], input_order, indomain_min),
])
```

i.e. we just placing all the rectangle (using smallest principle to y) and then we choose the smallest value of h.
Instead *Median + Smallest y* is:

```
seq_search([
    int_search([h], input_order, indomain_median),
    int_search(y, smallest, indomain_min) ::
    int_search(x, first_fail, indomain_min)
])
```

which is different from the previous one because first we choose the h value to be the median of the domain and then we place all rectangle following the *smallest* principle explained before. We compare the output statistics in the

following table.

	Smallest y			**Median + Smallest y**		
ins	solveTime	propagations	failures	solveTime	propagations	failures
20	0.338669	95296	19198	0.340508	95144	19259
21	22.2027	5980625	1302853	23.2836	5979855	1302877
22	20.936	5105276	1112267	20.7717	5106717	1112340
23	6.5281	2400702	465141	6.83405	2401409	465263
24	0.00902201	3158	456	0.0100581	3721	568
25	22.8739	4858726	1175681	21.8121	4864241	1176135
26	*300*	*163347394*	*1,5·10^7*	*300*	*78032814*	*1,6·10^7*

Even if the results are not so different, it seems that *Smallest y* performs better. Also looking at the last founded solution in instances 26 the latter found a solution with $h = 35$, while the one with the Median found just $h = 37$.

6 Restarting

We have tried to use *luby restart* to avoid been stuck in search. To do so effectively, we should introduce some randomness, so we have changed the search annotation:

int_search([h], input_order, indomain_random) :: restart_luby(500)

So, now, the value of h is chosen randomly and we restart the search every 250 nodes. To understand if this can be effective, we report in the following table a comparison between the two model.

	Search Annotations (Median)			**Luby Restarting (500)**		
ins	solveTime	propagations	failures	solveTime	propagations	failures
20	34.1024	10571965	2011152	*300*	*95499081*	*9336710*
21	0.148044	43289	5434	0.915826	301785	37589
22	149.351	42251159	4790452	*300*	*77067708*	*1,2·10^7*
23	0.003208	1471	89	0.00381801	1546	17
24	0.00234101	900	25	0.00171301	669	0
25	*300*	*78143731*	*1,5·10^7*	*300*	*74763852*	*1,1·10^7*

In this case restarting did not lead to an improvement, maybe due to the fact that using median choice for h is a powerful heuristic and in the previous case we have not used it. So, we have tried to use restarting in the *Smallest y* model, changing it a little bit just to introduce randomness:

seq_search([

```
    int_search(y, smallest, indomain_min) ::
    int_search(x, first_fail, indomain_random),
    int_search([h], input_order, indomain_min),
]) :: restart_luby(500)
```

We compare, in the next table, the results for the above described model and the *Smallest y* one.

	Smallest y			**Median + Smallest y**		
ins	solveTime	propagations	failures	solveTime	propagations	failures
20	0.338669	95296	19198	3.59041	983574	211354
21	22.2027	5980625	1302853	*300*	*80982027*	$1{,}8 \cdot 10^7$
22	20.936	5105276	1112267	*300*	*68476066*	$1{,}5 \cdot 10^7$
23	6.5281	2400702	465141	72.9662	27089543	5587213
24	0.00902201	3158	456	0.00891101	3447	475
25	22.8739	4858726	1175681	*300*	*61949101*	$1{,}5 \cdot 10^7$
26	*300*	*163347394*	$1{,}5 \cdot 10^7$	*300*	*166522622*	$1{,}4 \cdot 10^7$

It is clear that restarting was ineffective. This probably because the introduced randomness doesn't change too much the search strategy and it is not so smart. So, restarting it is just limiting the search instead of avoiding getting stuck.

7 Implied constraint (2)

The results are still not very satisfying so we have tried to found another implied constraint. In particular, we know that a solution can not be a best one if we have left avoidable empty space. To explicit that in an efficient way, we have used the global constraint *member*:

$\forall i = 1, ..., n$
member([0] ++ [y[j] + heights[j] | j in 1..n where j != i], y[i]) $\wedge$
member([0] ++ [x[j] + widths[j] | j in 1..n where j != i], x[i])

The first constraint means that we have to place a rectangle in a row where another rectangle finish, because otherwise we can translate the rectangle downwards. The second constraints is the analogue to the previous one but for the columns. This lead to a further improvement, as shown in Table 6.

	Smallest y			**Implied Constraint (2)**		
ins	solveTime	propagations	failures	solveTime	propagations	failures
20	0.338669	95296	19198	0.160376	134919	5168
21	22.2027	5980625	1302853	3.65382	2789944	146462
22	20.936	5105276	1112267	5.63551	3997210	190249
23	6.5281	2400702	465141	1.91632	2043574	85232
24	0.00902201	3158	456	0.00949001	9937	336
25	22.8739	4858726	1175681	3.64786	1880997	144549
26	*300*	*163347394*	$1{,}5 \cdot 10^7$	85.5815	96433253	2613179

Table 4

8 Results and conclusion

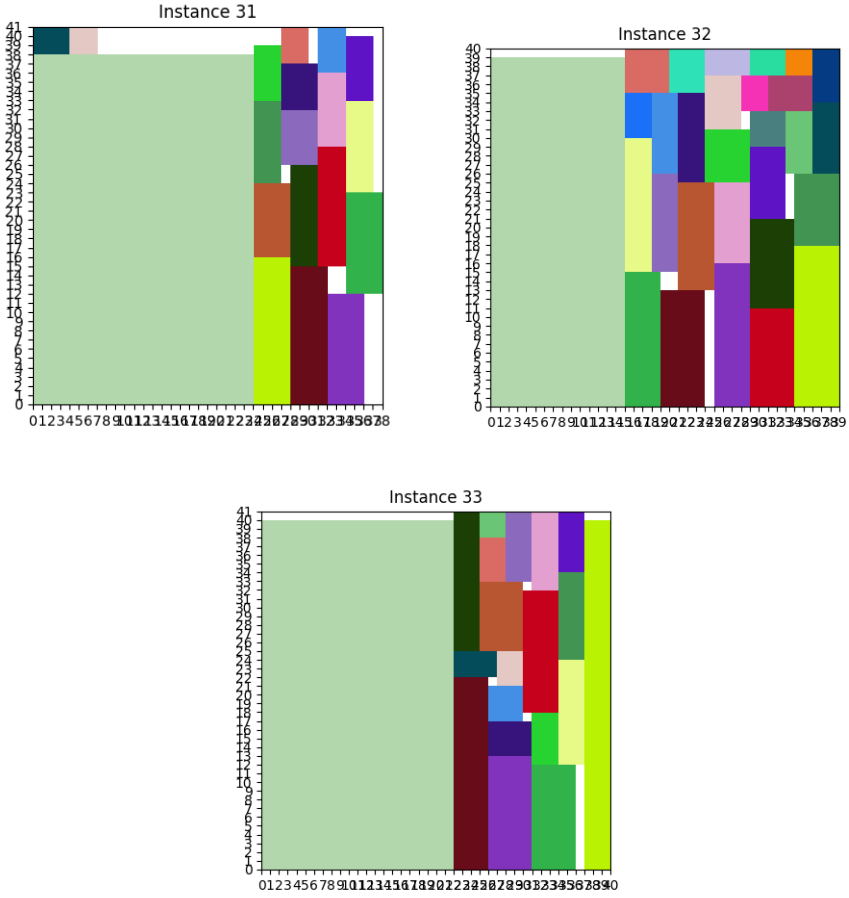
Just to conclude our discussion we want to notice that in a preprocessing step, if w and *widths* have a *maximum common divisor* we can just divide all widths by it and we can apply the same reasoning for heights. For our particular instances this is ineffective. We report in the following table the output statistics of our best model (the last one discussed) using Gecode and Chuffed (up to now we have used always Gecode). It is clear that Chuffed is more efficient in our case. Then we plot the solutions for which the solver was unable to find the optimal one or to prove it.

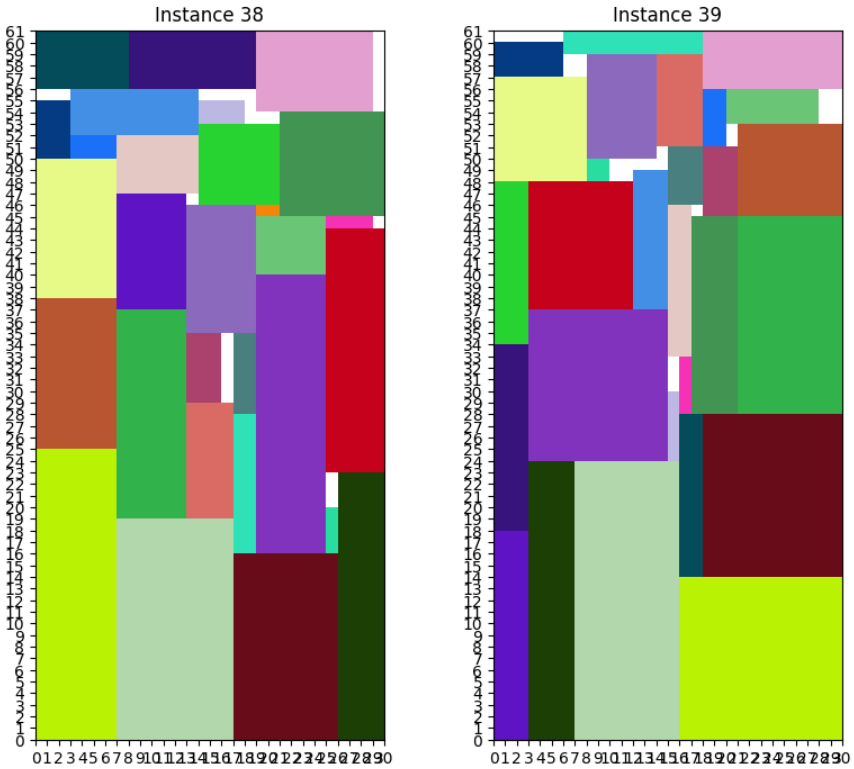
	Best Model - Gecode				
ins	**solveTime**	**propagations**	**nodes**	**failures**	**h**
1	0.004595	64	3	1	8
2	0.000162001	189	10	3	9
3	0.000237001	258	12	3	10
4	0.000374001	514	16	5	11
5	0.000642001	812	41	14	12
6	0.00112801	1583	74	31	13
7	0.00149701	1771	61	23	14
8	0.000818001	1052	50	19	15
9	0.00118401	1593	60	23	16
10	0.0124521	16753	937	458	17
11	33.5326	51525024	1529860	764919	18
12	0.00364601	5091	186	87	19
13	0.00642801	9768	323	149	20
14	0.00223901	2205	95	38	21
15	0.00232601	2962	141	58	22
16	*300*	*349218605*	*28499201*	*14249583*	*24*
17	0.139409	174943	9877	4929	24
18	0.00521501	3983	191	81	25
19	4.01262	3342018	278292	139134	26
20	0.175512	134919	10373	5168	27
21	3.82537	2789944	292951	146462	28
22	5.73697	3997210	380528	190249	29
23	2.05413	2043574	170486	85232	30
24	0.0102501	9937	696	336	31
25	3.62047	1880997	289127	144549	32
26	85.7543	96433253	5226393	2613179	33
27	165.585	167024576	12989716	6494842	34
28	0.0333731	32558	1021	498	35
29	*300*	*283180674*	*18897628*	*9448801*	*37*
30	0.31963	337752	7925	3950	37
31	*300*	*517712711*	*25452697*	*12726333*	*41*
32	*300*	*167629378*	*17564416*	*8782192*	*40*
33	*300*	*469531170*	*21931010*	*10965495*	*42*
34	*300*	*251258155*	*16761723*	*8380849*	*41*
35	*300*	*303988879*	*6718465*	*3359219*	—
36	0.00659001	5878	205	87	40
37	*300*	*304411758*	*7492041*	*3746005*	—
38	*300*	*357064711*	*4184886*	*2092428*	—
39	*300*	*223877637*	*12252494*	*6126226*	*61*
40	*300*	*162047217*	*1355318*	*677622*	—

Table 5

	Best Model - Chuffed				
ins	**solveTime**	**propagations**	**nodes**	**failures**	**h**
1	0.000	207	4	0	8
2	0.000	516	8	0	9
3	0.000	765	9	0	10
4	0.001	2113	18	1	11
5	0.000	3755	29	0	12
6	0.001	5785	38	4	13
7	0.001	5269	37	3	14
8	0.000	5135	28	3	15
9	0.003	28961	107	67	16
10	0.007	51988	226	163	17
11	0.489	4897154	11238	9963	18
12	0.005	50726	82	48	19
13	0.031	407114	736	602	20
14	0.007	86517	109	72	21
15	0.003	32343	88	26	22
16	109.067	387450757	998866	977735	23
17	0.055	632835	922	838	24
18	0.139	2711444	2085	1986	25
19	2.153	37470176	21924	21504	26
20	0.404	7310264	7598	7395	27
21	0.693	9910412	9440	9076	28
22	0.641	7167477	7473	7217	29
23	0.225	1590589	3338	3209	30
24	0.007	62725	144	89	31
25	1.194	25231217	14080	13489	32
26	5.470	30467049	54626	51909	33
27	21.963	74872839	197304	191185	34
28	0.124	2395228	1923	1790	35
29	*300*	*1572485669*	*1877307*	*1817648*	*37*
30	0.404	8927333	3994	3756	37
31	*300*	*974174528*	*3073776*	*3028665*	*41*
32	*300*	*1068961948*	*2221977*	*2165739*	*40*
33	*300*	*1137482402*	*2568439*	*2506318*	*41*
34	2.459	16250909	28423	23325	40
35	46.949	204065706	337175	315297	40
36	0.800	12052971	6611	5912	40
37	*300*	*1752285303*	*1132666*	*988394*	–
38	*300*	*842820404*	*1952943*	*1758235*	*61*
39	*300*	*815441281*	*1859289*	*1741591*	*61*
40	*300*	*2796643213*	*1093478*	*370192*	–

Table 6


Instance 31
Instance 32
Instance 33



9 Appendix - Allowing rotations

If we want to allow rotations we can simply extend our model introducing a binary variable for each rectangle that indicates if that rectangle is rotated or not. Then, we can redefine *widths* and *heights*:

$$widths_i = r_i \cdot sizes_i^{(1)} + (1 - r_i) \cdot sizes_i^{(2)} \quad \forall i = 1, ..., n$$

$$heights_i = r_i \cdot sizes_i^{(2)} + (1 - r_i) \cdot sizes_i^{(1)} \quad \forall i = 1, ..., n$$

with r_i a binary variable indicating if the rectangle i is rotated or not. Alternatively, without introducing new variables, we can model the problem in this way:

$\forall i = 1, ..., n$

$$(width_i = size_i^{(1)} \wedge height_i = size_i^{(2)}) \vee (width_i = size_i^{(2)} \wedge height_i = size_i^{(1)})$$

In both cases everything still works fine, but we now have more degree of freedom. Thus, it emerges another symmetry that we can break. In fact, if the available space is a rectangle, we can find an equivalent solution just by rotating all of 90° degree. To avoid that, in the first modellization we can impose:

$$w = h \rightarrow \sum_{i=1}^{n} r_i \leq \lceil \frac{n}{2} \rceil$$

This reduce the search space by roughly half.
Another source of symmetry is the case of rectangles which are square. In this case, we can just impose:

$$\forall i = 1, ..n \quad sizes_i^{(1)} = sizes_i^{(2)} \rightarrow r_i = 0$$

References

[1] Guido Tack. Peter J. Stuckey, Kim Marriott. The minizinc handbook.