Modelling and solving the Present Wrapping Problem with Satisfiability Modulo Theory

Cellini Lorenzo - 0000951921
lorenzo.cellini3@studio.unibo.it

November 4, 2020

This work describes a proposed solution to the Present Wrapping Problem in the framework of Satisfiability Modulo Theory (SMT). Two models have been developed: the "basic" one addresses the easier version of the problem, where there is only one copy for each piece and rotation is not allowed; the "general" one addresses the more relaxed version of PWP problem, where rotation is allowed and there could also be repeated rectangles.

1 PROBLEM DESCRIPTION

1.1 The Present Wrapping Problem PWP

The present wrapping problem is a well known constraint optimization problem that can be described as follow: given an enclosing rectangle of area $w * h$ and a list of smaller rectangles i of given dimensions $dimension_{ij}$, where i is the rectangle index and $j \in \{1,2\}$ the $width$ or $heigth$, find a satisfying assignement for the rectangles position such that they cover the entire area. In the literature there exist several different formulation of the same problem, where what changes is the goal: e.g. it could be asked to find the configuration that minimizes the covered area or minimizes only one of the two dimensions. Futhermore the rectangles used to cover the area could take different shapes, like circles or irregular polygons as well.
In this work the problem addressed is the first mentioned: the goal is to find a satisfying assignement for the positions of the given rectangles, in order to exactly cover the entire given enclosing area.

1.2 Data pre- and post-processing

1.2.1 Instances format

The given instances are *.txt* files of the form:

```
9 12
5
```

```
3 3
2 4
2 8
3 9
4 12
```

where the first line represents the width and heigth of the enclosing area, the second line states how many rectangles are at disposal and the following lines are the actual rectangles expressed as the couple $dimension_{i1} * dimension_{i2}$

1.2.2 Instances preprocessing

Instances preprocessing is done by the scripts *basic_model.py* and *general_model.py*, just before solving the instance. The only required input is the instace file directory, provided by passing its path to the parameter *instances_dir*. If this is the only input argument, both the scripts run over all the files in the given directory, otherwise the optional parameter *file_name*, that specifies instance to solve, can be passed.

1.2.3 Scripts output

Both the scripts write the output in the folder "../out/*_model_smt", where "../" means the parent directory with respect to the current directory and "*" stands for "basic" or "general". Each script write the instance solution in the format "<instance_name>-out.txt" and create two subfolders: *graph* and *stats*, where the graphical representation and solving statistics are stored.

2 THE BASIC MODEL

2.1 Modeling

The idea behind this SMT model is the same as the CP one: mapping each rectangle to its bottom-left corner and use the latter as the problem variable. Because the rectangles rotation is not allowed, the only degree of freedom is the rigid translation of the rectangles. So the problem can be translated into: *find a satisfaction assignement for the rectangles bottom-left corners such that the relative rectangles cover the entire area and do not overlap.*
Under this assumption the formulation of the problem constraints is straigthforward:

- for a rectangle i to stay in the enclosing area $w * h$, its bottom left corner coordinates can take values only from $\{0..w - dimension_{i1}, 0..h - dimension_{i2}\}$;
- the non overlap constraints means that when a rectangle i is placed, the set of points in $\{x_i..x_i + dimension_{i1}, y_i..y_i + dimension_{i2}\}$ are removed from the domains of the remaining variables.

REMARK: in all the mathematical formulae I will maintain the notation *(1,2)* for coordinates indexes, but it is worth to highligth that in python array's indexes start from 0, so in the Z3 program we'll see *(0,1)* as arrays' indexes, while variables name will have *(1,2)* as indexes. In other words: $coords_{i1}$ maps to the 0^{th} index of the corresponding array and $coords_{i2}$ maps to the 1^{st} index.

While in CP is useful to add redundant constraints in order to improve constraint propagation and speed the search up, it is not obvious that the same technic brings benefits in the framework of SMT. The same cumulative constraint as the one used in the CP program will be

encoded here, but the decision on wheter or not include it in the final model is postponed to the Conclusion section. The cumulative constraint is:

- the amount of heigth (width) that can be used in any position during packing is up to $h(w)$, so one can think about rectangles as tasks of duration $dimension_{i1}$ ($dimension_{i2}$) and resource consumption $dimension_{i2}$ ($dimension_{i1}$) and the cumulative constraint must hold in any solution.

What discussed for the cumulative constraint also applies to symmetry breaking constraint. I will report here the constraint postponing the decision, on wheter or not include it in the final model, to the Conclusion section.
Symmetry breaking constraint:

- setting a fixed and predetermined position (e.g. the paper origin (0,0)) for the most difficult to place rectangle, the biggest one.

In the following paragraphs a detailed explanation of data, variables and constraints is given.

2.1.1 Data

The following is an excerpt of code from the python Z3 *basic_model.py* program:

```
###################     Data     ###################
print("Declaring problem data...")

# The first line contains the width and height of the enclosing rectangle
first_line = instance_file_read.readline().strip().split(" ")

# w = width
w = int(first_line[0])

# h = height
h = int(first_line[1])

# The second line contains the number of pieces to wrap
second_line = instance_file_read.readline().strip().split(" ")

# n = number of pieces;
n = int(second_line[0])
TILES = range(int(str(n)))

# Extracting the tiles dimensions from the rest of the file
rest_of_lines = instance_file_read.readlines()

# Removing possibly empty lines
rest_of_lines = [line.strip() for line in rest_of_lines if line.strip()]

# dimension is the tensor of sides' lenghts
dimension = []
for line in rest_of_lines:
    rectangle = [int(line.split(" ")[0]), int(line.split(" ")[1])]
    dimension.append(rectangle)
```

As mentioned before, the script preprocesses input instance file on the fly and generates needed data objects (arrays, matrices, constants etc.)
Data used at this stage are the number of rectangles n, width w and heigth h of the enclosing area (paper roll). From these, a $dimension$ tensor is built: it is an *n-array*, where n is the number of rectangles, containing n *2-array* of rectangles' dimensions.

2.1.2 Variables

As stated before, in this basic version of the problem, the only degree of freedom is the rigid translation of rectangles and so the only variables are the coordinates of rectangles' bottom-left corner. For compliance with the CP program, these variables have been called $coords_{i,j}$.

```
###################     Variables     ###################
print("Declaring variables...")

# 1. coords_i_j is the jth coordinate of bottom left corner of ith rectangle
coords = [[Int("coords_%s_%s" % (i + 1, j + 1)) for j in range(2)] for i in TILES]
```

2.1.3 Local constraints

In this and the following paragraph problem constraints are explained.

The first constraint is on the **domain of bottom-left corner**. Indeed, there are a lower and an upper bound on the coordinates that make any rectangle to completely stay in the enclosing area.
This constraint can be written as follow:

```
# Bottom left corner constraint
bl_constraint = [
    And(
    coords[i][0] >= 0,
    coords[i][0] <= w - dimension[i][0],
    coords[i][1] >= 0,
    coords[i][1] <= h - dimension[i][1],
        )
    for i in TILES
]
```

2.1.4 Global constraints

The second constraint is the **non-overlap** constraint, meanig that for any two given rectangles i and j, j can be placed on the rigth OR on the left OR below OR above the rectangle i.
This can be expressed as follows:

```
# Non-overlap constraint
no_overlap_constraint = [
    Or(
        coords[i][0] + dimension[i][0] <= coords[j][0],
        coords[j][0] + dimension[j][0] <= coords[i][0],
        coords[i][1] + dimension[i][1] <= coords[j][1],
        coords[j][1] + dimension[j][1] <= coords[i][1],
    )
    for i in TILES
```

```
    for j in TILES
    if j > i
]
```

At this point the model is already able to find good solutions, but it still miss the cumulative constraint and symmetry breaking constraint. At this stage it is not clear wheter or not these constraints will bring any improvements, but it will be after the discussion of collected runtime statistics.

So, the third constraint is the **cumulative** constraint. This comes from the family of scheduling constraints: indeed rectangles can be treated as tasks, their dimensions are the duration along the respective coordinates, the other dimension represents the resource consumption and the upper bounds are respectively the width and heigth of the paper roll.
Obviously for any packing solution, where the previous two constraints are satisfied, cumulative must also hold. Indeed this is a **redundant** constraint.

In theory, in SMT it is not useful to add redundant constraints, because there is not a constraint propagation phase that cuts variables' domains in order to reduce the search space: on the contrary, adding redundant constraints will increase the number of equations that both the theory solver and the SAT solver must handle, causing the solving phase to slow down. Anyway, cumulative constraint is a project requirement here, so I will include it for now and discuss the best performant model later.

Cumulative constraint can be encoded in the following way:

```
# Implied constraint - "cumulative" in MiniZinc
cumulative_constraint = []

# 1.Cumulative on heigth capacity:
for i in range(w):
    vertical_sum = Sum(
        [
            If(
                And(coords[j][0] <= i, coords[j][0] + dimension[j][0] > i),
                dimension[j][1],
                0,
            )
            for j in TILES
        ]
    )
    cumulative_constraint.append(vertical_sum <= h)

# 2.Cumulative on width capacity:
for i in range(h):
    horizontal_sum = Sum(
        [
            If(
                And(coords[j][1] <= i, coords[j][1] + dimension[j][1] > i),
                dimension[j][0],
                0,
            )
            for j in TILES
```

```
        ]
    )
    cumulative_constraint.append(horizontal_sum <= w)
```

It is encoded in two step: respectively the horizontal and the vertical set of constraints.

2.1.5 Symmetry breaking constraints

As for the CP case, here also there are symmetries (isomorphisms) that can be broken, but, as discussed for cumulative constraint, this shouldn't lead to performance improvement because of how SMT engine works.
As before, I chose to encode the symmetry breaking constraint anyway, and delay the decision about its inclusion in the final model after statistics comparison.

Symmetry breaking constraint:

```
# Symmetry breaking - isomorphisms breaking
surface = [dimension[i][0] * dimension[i][1] for i in TILES]

biggest_rect = np.argmax(surface)

symm_break = [And(coords[biggest_rect][0] == 0, coords[biggest_rect][1] == 0)]
```

As in the CP program, the biggest rectangle is located at runtime and constrained in the (0,0) position.

3 THE GENERAL MODEL

3.1 Modeling

In the general case, what changes is the possibility to have **rectangles with the same dimensions** and **rectangle rotation is now allowed**. This means that the solution space is larger than before because, now, each rectangle can be placed in two different ways and for each instance containing repeated tiles, the solution space contains all the solution given by the permutations of pieces with the same dimensions. The possibility to have repeated rectangles introduces a symmetry, but, again, it is not obvious the need to break it. From the theory perspective, there would not be advantages in breaking symmetries for the SMT engine, but, taking a data-driven approach, I will encode this breaking symmetry constraint and decide about its inclusion in the final model by looking at statistics comparison.
To model this case one can extend the previous basic model: indeed the bottom-left corner coordinates can still be used as variable, but another variable set, $kind_i$, is needed in order to address the rotation possibility. It represents the orientation of the i^{th} rectangle: indeed the problem to place a rectangle and take into account its rotation can be viewed as the problem to assign a value to its bottom-left corner coordinates plus the problem to choose wich, of the two possible values, the horizontal dimension takes (respectively vertical).

3.1.1 Data

In order to address the increased complexity of the problem, some new auxiliary variables are needed.

```
###################     Data     ###################
print("Declaring problem data...")

# The first line contains the width and height of the enclosing rectangle
first_line = instance_file_read.readline().strip().split(" ")

# w = width
w = int(first_line[0])

# h = height
h = int(first_line[1])

# The second line contains the number of pieces to wrap
second_line = instance_file_read.readline().strip().split(" ")

# n = number of pieces;
n = int(second_line[0])
TILES = range(int(str(n)))

# Extracting the tiles dimensions from the rest of the file
rest_of_lines = instance_file_read.readlines()

# Removing possibly empty lines
rest_of_lines = [line.strip() for line in rest_of_lines if line.strip()]

# Building a shape tensor where each element is the couple of
# the two different orientations for each rectangle.

shape = []
count_of_shapes = {}

for line in rest_of_lines:
    dims = []

    rectangle = [int(line.split(" ")[0]), int(line.split(" ")[1])]
    dims.append(rectangle)

    key = "{}x{}".format(rectangle[0], rectangle[1])
    count_of_shapes[key] = count_of_shapes.get(key, 0) + 1

    if rectangle[0] != rectangle[1]:
        dims.append([rectangle[1], rectangle[0]])
        shape.append(dims)
    else:
        dims.append([-99, -99])
        shape.append(dims)

ncopy = list(count_of_shapes.values())
num_distinct_tiles = np.array([ncopy]).size
```

In the previous exrcept of code, constants n, w, h and $TILES$ are the same as in the basic

model.
The *shape* object is a tensor where each element is the couple of the two different orientations for each rectangle. For example, for the instance in the Data pre- and post-processing section, the *shape* tensor would be: `[[[3, 3], [-99, -99]], [[2, 4], [4, 2]], [[2, 8], [8, 2]], [[3, 9], [9, 3]], [[4, 12], [12, 4]]]`
As it can be seen, in case of squares, the dummy array `[-99, -99]` is used: this is needed in order to make *shape* tensor to have the correct number of dimensions. Obviously, squares of side l could be treated as the tensor `[[l, l], [l, l]]` but this would generate symmetric configuration.

The *count_of_shapes* object is a dictionary where the key is the rectangle and the value is the number of times that rectangle appears in the given list. For the previous example, it is:

```
{
    "3x3": 1,
    "2x4": 1,
    "2x8": 1,
    "3x9": 1,
    "4x12": 1
}
```

Finally, *ncopy* and *num_distinct_tiles* store respectively the number of copies for each rectangle (array) and the number of distinct rectangles (constant).

3.1.2 Variables

The only difference with respect the basic case is the new array of variables, called *dimension*. These variables are responsible to select the rectangle rotation among the two possible.

```
####################     Variables     ####################
print("Declaring variables...")

# 1. coords_i_j is the jth coordinate of bottom left corner of ith rectangle
coords = [[Int("coords_%s_%s" % (i + 1, j + 1)) for j in range(2)] for i in TILES]

# 2. dimension_i_j is the length of jth side of ith rectangle.
# The purpose of this variable is to handle the rectangle rotations.
dimension = [
    [Int("dimension_%s_%s" % (i + 1, j + 1)) for j in range(2)] for i in TILES
]
```

3.1.3 Constraints

The **bottom-left corner domain**, the **non overlapping constraint** and the **cumulative** constraints are the same as for the basic model.
The following constraint, instead, is needed in order to take into account rectangles rotation:

```
# Allow the algorithm to select only one rotation among the two possible ones of a rectangl
rotation_constraint = [
    Or(
        And(shape[i][0][0] == dimension[i][0], shape[i][0][1] == dimension[i][1]),
        And(
            shape[i][1][0] == dimension[i][0],
```

```
            shape[i][1][1] == dimension[i][1],
            dimension[i][0] != -99,
            dimension[i][1] != -99,
        ),
    )
    for i in TILES
]
```

It is clear here the role of the variable $shape_{ijk}$, where i is the rectangle index, j is the rotation and k the axis, in selecting the dimension, so implicitly selecting the rotation.

3.1.4 Symmetry breaking constraints

Again, in order to break isomorphisms it suffices to manually set the position of the biggest rectangle but letting the algorithm to choose the orientation. The symmetry breaking constraint become:

```
# Symmetry breaking constraints

# 1. isomorphisms breaking
surface = [
    shape[i][j][0] * shape[i][j][1]
    for i in range(num_distinct_tiles)
    for j in range(1)
]
biggest_rect = np.argmax(np.array(surface))

symm_break = [And(coords[biggest_rect][0] == 0, coords[biggest_rect][1] == 0)]
```

Besides the isomorphism symmetry, this time another class of symmetry arise because of the permutations of rectangles with the same dimensions. This is the kind of symmetry broken by lexicographical ordering. In order to do that the following constraint is added:

```
# 2. Ordering symmetries breaking for rectangles with the same size
order_constraint = []
base = []

for i in range(num_distinct_tiles):
    if i == 0:
        base.append(0)
    else:
        base.append(sum(ncopy[0:i]))

for i in range(num_distinct_tiles):
    for j in range(ncopy[i] - 1):
        order_constraint.append(
            And(
                coords[base[i] + j][0] <= coords[base[i] + j + 1][0],
                Implies(
                    coords[base[i] + j][0] == coords[base[i] + j + 1][0],
                    coords[base[i] + j][1] <= coords[base[i] + j + 1][1],
                ),
```

)
)

This constraint can be understood in this way: for each distinct rectangle loop over its replicas (if there are) and for any pair of equal rectangles, let's call them i and j, $coords_{i1} \leq coords_{j1} \wedge (coords_{i1} = coords_{j1} \rightarrow coords_{i2} \leq coords_{j2})$.

Under the assumption that the given rectangles are ordered ascending, the *base* array contains the starting index of each series of rectangle with the same dimension. For example, given the array $[3x3, 3x3, 3x3, 4x5, 4x5, 6x7]$ the corresponding $base$ array would be $[0, 3, 5]$. The $base$ array lets to apply the lexicogrphical ordering only to rectangles with the same dimensions.

4 SOLVING THE PROBLEM

The following tables show the number of conflicts and running time for basic and general model respectively, for four different combination of active constraints. The goal is to investigate the best combination of constraints and at the same time, check if the symmetry breaking constraint brings some improvement in SMT framework. While both the metrics are reported here, I would rather look only at the number of conflicts, because the running time kpi is too much related to hardware performance.

instance	**cumul + symbre**	**cumul**	**symbre**	**base**
8x8	**1**	4	**1**	**1**
9x9	16	24	5	**1**
10x10	11	13	**7**	10
11x11	59	117	12	**8**
12x12	**67**	111	198	270
13x13	**11**	105	119	156
14x14	**48**	128	257	125
15x15	**66**	108	123	115
16x16	203	**193**	357	633
17x17	339	228	**165**	861
18x18	9206	47917	**968**	7618
19x19	542	**534**	903	7713
20x20	1242	**184**	2115	2567
21x21	1596	**1246**	5900	3789
22x22	647	**278**	7226	628
23x23	21710	66074	30955	**3594**
24x24	**2996**	3070	48408	16506
25x25	**1703**	2235	54422	4778
26x26	**18885**	50705	195303	94394
27x27	**7952**	12554	74549	39275
28x28	26079	13379	40722	**6185**
29x29	198342	261455	**39669**	898219
30x30	2792	34877	31014	**2474**
31x31	2493	8184	**1381**	2577
32x32	81555	158083	**64011**	83746
33x33	18335	**8559**	49177	80720
34x34	19542	**807**	27665	45202
35x35	22388	10558	**5297**	57344
36x36	**8925**	62861	38865	47158
37x37	1456697	**1216635**	1907517	1503147
38x38	**13417**	14987	49744	58708
39x39	**1009095**	1186619	3197486	9399847
40x40	6613	16267	6883	**3670**

Table 4.1: Comparing number of conflicts for basic model on different combination of active constraints

instance	**cumul + symbre**	**cumul**	**symbre**	**base**
8x8	3	4	**2**	**2**
9x9	**2**	6	**2**	**2**
10x10	2	4	**1**	8
11x11	4	11	2	**1**
12x12	4	10	3	**2**
13x13	3	11	3	**2**
14x14	5	17	5	**4**
15x15	6	18	4	**2**
16x16	13	25	**7**	10
17x17	19	35	**6**	15
18x18	635	5501	**34**	218
19x19	226	92	**21**	79
20x20	442	54	37	**23**
21x21	228	168	131	**41**
22x22	170	72	183	**11**
23x23	2747	7986	1177	**64**
24x24	635	413	1750	**335**
25x25	228	416	2359	**92**
26x26	2918	9692	4540	**2657**
27x27	**812**	2953	2845	1410
28x28	2807	3326	2135	**297**
29x29	36950	87995	**2487**	19096
30x30	478	6074	1318	**117**
31x31	288	1743	**65**	111
32x32	9155	92694	**3065**	3590
33x33	2711	**1860**	2598	3107
34x34	2630	**253**	1315	2568
35x35	2779	2964	**300**	2746
36x36	**988**	12128	2425	2613
37x37	1687949	956682	39412	**35863**
38x38	**1454**	2087	2191	2612
39x39	1267819	907482	**57569**	191184
40x40	774	4242	342	**234**

Table 4.2: Comparing running time, in centi-seconds, for basic model on different combination of active constraints

instance	**base**	**ord**	**cumul**	**ord + symbre**	**ord + cumul**	**all**
8x8	18	8	7	**6**	7	8
8x8_symmetry	3	8	4	3	**2**	**2**
9x9	195	147	**5**	34	**5**	41
10x10	46	46	**17**	**17**	**17**	20
10x10_symmetry	412	171	99	133	114	**81**
11x11	975	2220	199	649	**92**	100
12x12	1454	2974	517	1606	230	**176**
13x13	215	1545	489	3493	**165**	246
14x14	4164	17499	1258	9663	747	**79**
15x15	2156	**87**	1486	90	580	260
16x16	1305	32826	1694	2766	1896	**443**
16x16_symmetry	86	584	106	154	145	**48**
17x17	22299	14551	6153	10112	4122	**892**
18x18	1632237	2902071	162324	217584	**30356**	179898
19x19	22258	212961	46188	15294	**3304**	68669
20x20	43677	48964	**1525**	141072	20264	33437
21x21	42965	247161	**1677**	77532	18919	53016
22x22	290478	133075	58093	129802	**3738**	134400
23x23	2681959	810377	**136290**	3020291	482305	1135933
24x24	1085211	200835	303971	218447	286140	**12319**
25x25	2551686	455184	**294542**	2653111	1503066	2811231
26x26	2406351	2449466	-	2623958	**546322**	560058
27x27	2385970	**43177**	-	2414730	282726	806091
28x28	2489856	2450036	-	2484190	**106160**	-
29x29	2312453	2493072	-	2733977	**143251**	-
30x30	121647	2820277	329776	1058976	**7**	-
31x31	434514	2711212	-	95237	**5**	-
32x32	**1973620**	2122234	-	2410299	-	-
33x33	2298444	**280850**	-	740086	-	-
34x34	**33946**	170936	546322	129843	-	-
35x35	216887	1757625	-	**101178**	-	-
36x36	**41274**	255703	282726	2354887	-	-
37x37	**2250975**	2455477	-	2344739	-	-
38x38	2564468	**35193**	106160	2631743	-	-
39x39	**2128746**	2200407	-	2359941	-	-
40x40	381840	**113240**	143251	2573426	-	-

Table 4.3: Comparing number of conflicts for general model on different combination of active constraints

instance	**base**	**ord**	**cumul**	**ord + symbre**	**ord + cumul**	**all**
8x8	**1**	**1**	3	**1**	**1**	2
8x8_symmetry	3	5	**2**	3	**2**	4
9x9	2	2	3	**1**	2	3
10x10	4	4	**3**	**3**	4	**3**
10x10_symmetry	10	**5**	6	**5**	7	7
11x11	10	20	9	**7**	12	11
12x12	26	52	18	17	**9**	12
13x13	**10**	24	23	39	24	18
14x14	57	287	56	107	36	**10**
15x15	35	7	82	**3**	30	19
16x16	**23**	628	86	39	74	26
16x16_symmetry	**3**	10	11	**3**	11	7
17x17	576	369	419	190	256	**65**
18x18	128086	324455	22332	6086	**2470**	23522
19x19	691	12223	4220	384	**327**	5697
20x20	1223	2707	**167**	3533	1445	2521
21x21	1315	19262	**167**	2097	1337	5327
22x22	12081	8558	7539	3945	**374**	17950
23x23	237927	87511	**20465**	230428	117136	570065
24x24	63848	18893	77835	8500	50839	**1157**
25x25	195284	**48043**	65561	172345	787434	2491829
26x26	228229	400593	-	208538	256667	**160484**
27x27	213385	**5172**	-	179457	134863	322289
28x28	267162	344059	-	248057	**34209**	-
29x29	217653	222129	-	282279	**45155**	-
30x30	14276	324082	123994	73321	**8**	-
31x31	53300	435154	-	4527	**3**	-
32x32	**270976**	470070	-	293390	-	-
33x33	192340	**24879**	-	61948	-	-
34x34	**2027**	11645	217872	7911	-	-
35x35	14092	162045	-	**6207**	-	-
36x36	**2913**	19393	72656	250890	-	-
37x37	**236793**	279511	-	270072	-	-
38x38	178903	**2158**	21633	279437	-	-
39x39	**242021**	291638	-	290922	-	-
40x40	24144	**5834**	28641	214839	-	-

Table 4.4: Comparing running time, in centi-seconds, for general model on different combination of active constraints

5 Conclusion

Looking at the tables above, the best basic model seems to be the one with both cumulative and symmetry breaking constraint active, but, taking a look at only instances between 30x30 and 40x40, the most difficult to solve, the verdict is no more so obvious. Indeed each model has 3 "victory" in this interval. The situation becomes a little more clear if one imagines to look at the same table with only the first two columns: this is like redefining the basic model as the basic plus the cumulative constraint and deciding on wheter include symmetry breaking

constraint or not. In this way one can see that the model that includes the symmetry breaking performs better in 6 over 10 cases. So, if the driver is the efficiency in terms of memory, it could be safe to choose the complete model as the production one.
In terms of running time, it is obvious that the faster model is the one with less constraint: in SMT framework each constraint generates a bunch of equations and/or inequalities, so the more the number of constraints the more the number of object to compute and to store in memory.
Maybe, a more effective approach, but beyond the scope of this work, to the symmetry breaking matter could be to intervene at the SAT engine level and prevent it to generate symmetric formulae before feed them to the theory solver.

For what concerns the general model, as it can be seen from tables above, on very hard instances, between 30x30 and 40x40, the best performant model is the baseline where active constraints are only the bottom-left corner, no-overlapping and rotation constraints, thus only these constraints will be included in the final version of the general_model. For instances' entries marked with the "-" symbol I stopped the execution after a computation time greater than one hour.

6 References

- Class lectures
- Z3py tutorial