Final Project of Combinatorial Decision Making and Optimization, Module 1

SAT Model

Federico Cichetti - federico.cichetti@studio.unibo.it

March 8, 2022

Contents

Introduction

This report describes the choices behind the implementation of different solutions for the Very Large Scale Integration (VLSI) problem, presented as a project work for the course of Combinatorial Decision Making and Optimization, Module 1.

This document in particular focuses on how the problem can be modeled using *propositional logic* and solved efficiently using a *SAT solver*. Some of the concepts used in this solution have been presented in the report describing the *CP* model and are therefore not repeated here. Furthermore, our main source for the ideas presented in this solution is [1], which provides an impressive SAT model and discusses many techniques for improving search efficiency.

1 Model

Our SAT model is quite complex due to the difficulty of expressing high level concepts with the limited language of propositional logic, but still it's the most efficient and powerful out of all the models we implemented for the project.

1.1 Order Encoding

The core idea we adopted from [1] is that of starting from a limited but sufficient amount of constraints expressed in the language of constraint programming and *translating them* into propositional logic using *order encoding*.

Order encoding [2] is a simple but effective way to *translate CSP constraints into boolean variables*. In practice, a constraint of the form $x \leq c$ with x an integer variable and c an integer value, is encoded into a set of boolean variables $p_{x,c}$ for all meaningful c.

Example 1. We have variable $x_1 \in \{0, 1, 2, 3\}$. We can express $x_1 = 1$ using the following set of boolean variables:

$$\neg px_{1,0}, \quad px_{1,1}, \quad px_{1,2}, \quad px_{1,3}$$

From this set of constraints, we can retrieve the correct assignment $x_1 = 1$ by ordering the variables and picking the value of the first true assignment (value i for which $px_{c,i-1}$ is false and for all $k \geq i$, $px_{c,k}$ is true).

Example 2. We have variables $x_1, x_2 \in \{0, 1, 2, 3\}$. The constraint $x_1 + 1 \leq x_2$ can be decomposed into this set of *primitive comparisons*:

$$\neg(x_2 \leq 0), \quad (x_1 \leq 0) \vee \neg(x_2 \leq 1)^1, \quad (x_1 \leq 1) \vee \neg(x_2 \leq 2), \quad (x_1 \leq 2)$$

This set of constraints is expressed using the following set of boolean variables:

$$\neg px_{2,0}, \quad px_{1,0} \vee \neg px_{2,1}, \quad px_{1,1} \vee \neg px_{2,2}, \quad px_{1,2}$$

1.1.1 Ordering Integrity

Order encoding requires that an internal *ordering integrity* is maintained: for instance if $x_2 \leq 0$, the constraints $x_2 \leq 1, x_2 \leq 2, \ldots$ are always true. Therefore, $px_{2,i}$ must be true $\forall i > 0$. The order integrity constraint is usually maintained with a constraint like the following:

$$\neg px_{2,0} \vee px_{2,1}, \quad \neg px_{2,1} \vee px_{2,2}, \ldots$$

expressing the impossibility to have $px_{c,i}$ true and $px_{c,i+1}$ false.

[1] Personally, an intuitive way to understand the *or-not* comparisons is to think of them as *if-then* comparisons on conditions that are not respected: for instance, *if* $x_2 \leq 1$ *then* $x_1 \leq 0$. Also, *if* $x_1 > 0$, *then* $x_2 > 1$. The logic expression also adds to the meaning the fact that both the expressions can be true at the same time, and that if one is true we don't really care about the other's truth value as far as the constraint's satisfiability is concerned.

1.2 Variables

Given any instance of the problem, the values we have access to are:

- W: The width of the plate, which is fixed.
- n: The number of circuits to be placed.
- The measures of the circuits, that we save into two arrays: cw for widths and ch for heights. The set of circuits is C.
- The initial solution, obtained with the algorithm we have described in the CP report. In SAT there are no warm start mechanisms, so we use the initial solution just to have an *higher bound* on the height of the circuit: $maxh$.
- Like for the CP model, we can also provide a lower bound to the height by summing all circuits' areas and dividing by W. This lower bound is referred to as $lowh$ and is formally defined as $lowh = \lfloor \frac{\sum_i ch_i cw_i}{W} \rfloor$.

All of these are integer variables and are therefore only used for indexing our arrays of boolean variables. The boolean variables we instantiate for the problem are the following:

- $px_{c,e}$ for all circuits $c \in \{0, \dots, n-1\}$, for all possible x-axis positions $e \in \{0, \dots, W-1\}$. Each one of these variables indicates whether circuit c is placed at an x-position $\leq e$.
- $py_{c,f}$ for all circuits $c \in \{0, \dots, n-1\}$, for all possible y-axis positions $f \in \{0, \dots, maxh-1\}$. Each one of these variables indicates whether circuit c is placed at an y-position $\leq f$.

1.3 Constraints

The SAT encoding shares with the CP model the principles behind the main constraints: the two rules of interest in the positioning of the blocks are:

- The circuits should be placed *entirely* within the W and $maxh$ boundaries of the board.
- The circuits should not overlap.

The problem is that we are working within the *order encoding* framework, which, in general, makes posing constraints related to the positioning of the circuits quite complex because, as we discussed in 1, the only way to know the exact position of a circuit is to order variables and select the first true one, which is something that can't be done dynamically while the solver is running. The main constraints have to be re-designed keeping this shift from *absolute positioning* coordinates to *relative positioning* coordinates into account.

1.3.1 Keeping Circuits in Bounds

The idea is that circuit c with dimensions $cw_c \times ch_c$ cannot be placed closer than cw_c to the W limit, nor closer than ch_c to the $maxh$ limit. We can express this constraint by asking that *at least one* of the variables between $px_{c,0}$ and $px_{c,W-cw_c}$ and between $py_{c,0}$ and $py_{c,maxh-ch_c}$ is assigned true. This in turn means that all circuits respect $px_c \leq W - cw_c$ and $py_c \leq maxh - ch_c$, which is the objective of the first requirement.

$$\bigwedge_{c \in C} \bigvee_{0 \leq e \leq W - cw_c} (px_{c,e}) \tag{1}$$

$$\bigwedge_{c \in C} \bigvee_{0 \leq f \leq maxh - ch_c} (py_{c,f}) \tag{2}$$

Being positional encodings, the arrays px and py also respect the ordering integrity constraints we mentioned in Section 1.1.1:

$$\bigwedge_{c \in C} \bigwedge_{0 \leq e < W - cw_c} (\neg px_{c,e} \vee px_{c,e+1}) \tag{3}$$

$$\bigwedge_{c \in C} \bigwedge_{0 \leq f < maxh - ch_c} (\neg py_{c,f} \vee py_{c,f+1}) \tag{4}$$

1.3.2 Bidimensional No-Overlap

The second requirement is a two-dimensional no-overlap constraint, which, in the notation we used for the CP report, can be decomposed into:

$$\forall ci, cj \in C, ci < cj,$$
$$(xpos_{ci} + ws_{ci} \leq xpos_{cj}) \vee (xpos_{cj} + ws_{cj} \leq xpos_{ci}) \vee (ypos_{ci} + hs_{ci} \leq ypos_{cj}) \vee (ypos_{cj} + hs_{cj} \leq ypos_{ci})$$

Unfortunately, we don't have *global constraints* like in CP, so we aim for a more direct translation of this decomposition into our framework.

In [1], the authors propose to add two arrays of variables to better express this complex constraint:

- $lr_{i,j}$ for all circuits $i, j \in \{0, \ldots, n-1\}$, indicating whether circuit c_i is placed to the left of circuit c_j.
- $ud_{i,j}$ for all circuits $i, j \in \{0, \ldots, n-1\}$, indicating whether circuit c_i is placed below circuit c_j.

The meaning of these variables can be visualized in Figure 1

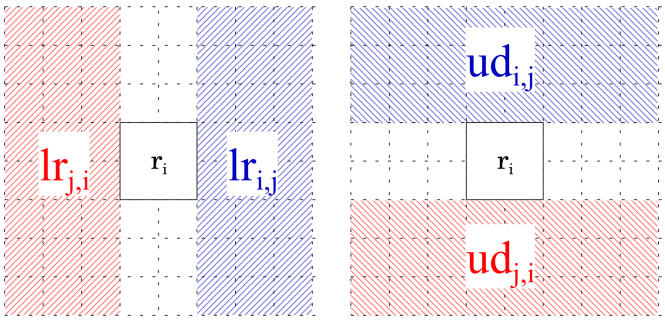


Figure 1: The meaning of variables $lr_{i,j}, lr_{j,i}, ud_{i,j}, ud_{j,i}$ between two circuits r_i and r_j. Source.

The no-overlap constraint can then be expressed by first defining what the relations lr and ud *imply* for the positioning of the circuits and then requiring that for each pair of circuits at least one between the possible relations $lr_{i,j}, lr_{j,i}, ud_{i,j}, ud_{j,i}$ is true.

The relations definition is quite tricky. The main idea is that given that a circuit ci is on the left or above circuit cj, if ci is placed *after* position e, then cj cannot be placed before position $e + cw_{ci}$, or position $e + ch_{ci}$ on the vertical case. A similar concept is valid for all placements of all pairs of circuits, so we can add the following constraints to the solver:

$$\bigwedge_{ci,cj \in C, ci < cj} \bigwedge_{0 \leq e < W - cw_{ci}} (\neg lr_{ci,cj} \vee px_{ci,e} \vee \neg px_{cj,e+cw_{ci}}) \tag{5}$$

$$\bigwedge_{ci,cj \in C, ci < cj} \bigwedge_{0 \leq e < W - cw_{cj}} (\neg lr_{cj,ci} \vee px_{cj,e} \vee \neg px_{ci,e+cw_{cj}}) \tag{6}$$

$$\bigwedge_{ci,cj \in C, ci < cj} \bigwedge_{0 \leq f < maxh - ch_{ci}} (\neg ud_{ci,cj} \vee py_{ci,f} \vee \neg py_{cj,f+ch_{ci}}) \tag{7}$$

$$\bigwedge_{ci,cj \in C, ci<cj} \bigwedge_{0 \leq f < maxh - ch_{cj}} (\neg ud_{cj,ci} \vee py_{cj,f} \vee \neg py_{ci,f+ch_{cj}}) \tag{8}$$

To this set of constraints, we add:

$$\bigwedge_{ci,cj \in C, ci<cj} (lr_{ci,cj} \vee lr_{cj,ci} \vee ud_{ci,cj} \vee ud_{cj,ci}) \tag{9}$$

which ensures that all pairs of circuits have at least one of the previously defined relations active. This large set of constraints overall ensures that circuits do not overlap and satisfy the second requirement for producing a solution.

1.3.3 Implied Constraints

We also add a constraint that proved successful in our CP model, that is the requirement of *exactly one* circuit to be placed in position (0,0). We have seen in class that the constraint *exactly one* can be decomposed into *at least one*, which is an $\vee$ of all boolean variables involved in the constraint, and *at most one*, which poses $\neg(b_1 \wedge b_2)$ constraints between all pairs of boolean variables b_1 and b_2. Therefore, we can add a constraint like the following to the problem:

$$exactly\_one_{c \in C}(px_{c,0} \wedge py_{c,0}) \tag{10}$$

Furthermore, we can add the following constraints: if circuit ci is placed on the left of or below circuit cj, cj cannot be placed in the first cw_{ci} or ch_{ci} positions, because there wouldn't be enough space to fit both circuits. Formally:

$$\bigwedge_{ci,cj \in C, ci<cj} \bigwedge_{0 \leq e < cw_{ci}} (\neg lr_{ci,cj} \vee \neg px_{cj,e}) \tag{11}$$

$$\bigwedge_{ci,cj \in C, ci<cj} \bigwedge_{0 \leq e < cw_{cj}} (\neg lr_{cj,ci} \vee \neg px_{ci,e}) \tag{12}$$

$$\bigwedge_{ci,cj \in C, ci<cj} \bigwedge_{0 \leq f < ch_{ci}} (\neg ud_{ci,cj} \vee \neg px_{cj,f}) \tag{13}$$

$$\bigwedge_{ci,cj \in C, ci<cj} \bigwedge_{0 \leq f < ch_{cj}} (\neg ud_{cj,ci} \vee \neg px_{ci,f}) \tag{14}$$

1.3.4 Symmetry Breaking Constraints

As already highlighted in the CP report, the model contains 3 very obvious kinds of symmetries that we need to eliminate: vertical, horizontal and diagonal flippings.

Within the order encoding framework, it becomes quite difficult to express symmetry breaking constraints imposing a lexicographic ordering of the circuits like we did for CP, because we only know position relations, not exact coordinates for the circuits.

In [1], the authors propose to exploit the positional relations in favour of breaking symmetry, by *fixing* the relations between a *single pair* of circuits. Randomly sampling $ci, cj \in C, ci < cj$, we impose:

$$lr_{cj,ci} = \bot \wedge ud_{cj,ci} = \bot \tag{15}$$

Doing so, we restrict the possibilities of the solver to place these circuits in another configuration and we implicitly break diagonal symmetry since such a flipping would invert at least one of the relations.

This reduction does not guarantee to break both horizontal and vertical flipping symmetries at the same time, but it ensures that at least one of them is broken. This is due to the fact that the constraint allows a configuration where one of $lr_{ci,cj}$ or $ud_{ci,cj}$ is true while the other is false (i.e. when cj is shorter in height than

ci and is placed within the same rows as ci, so it's entirely at its right). In this case, a vertical flipping does not change any of the relations, so it's technically valid, while an horizontal flipping would break $lr_{cj,ci} = \bot$ and is therefore forbidden.

1.3.5 Other Reductions

We also implemented other types of search space reductions proposed by the paper:

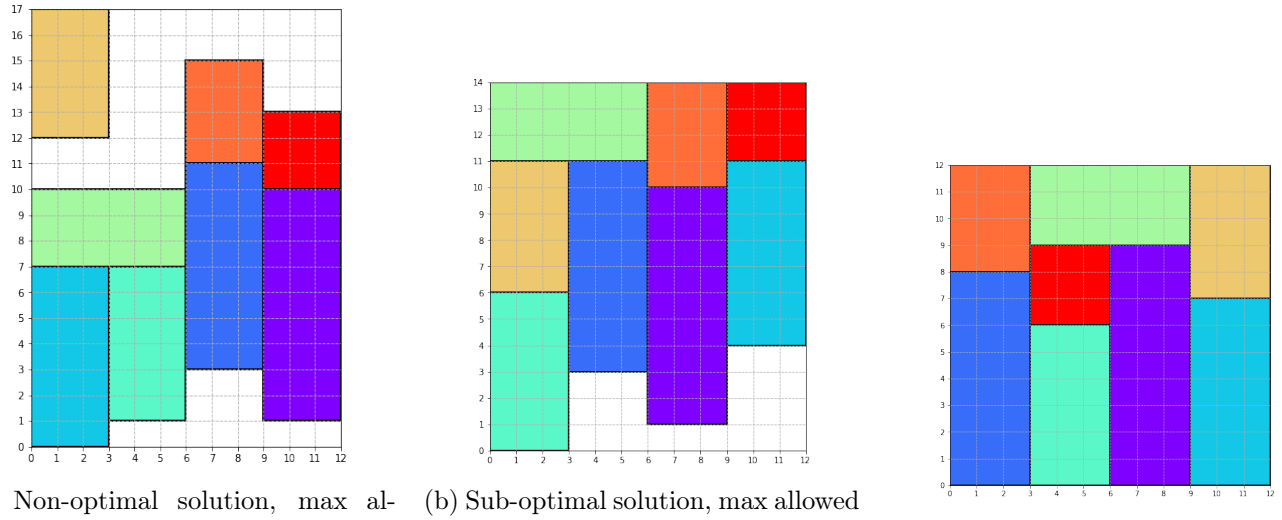
1. **Same Rectangle Reduction**: If two rectangles have the same dimension, we can decide the positional relation between them so that the solver will only be able to try a subset of the possible placements and will never swap the two circuits. We set $lr_{cj,ci} = \bot$ and then require that either ci is at the left or below cj.

$$\bigwedge_{ci,cj \in C, ci<cj, ch_{ci}=ch_{cj}, cw_{ci}=cw_{cj}} (\neg lr_{cj,ci} \wedge (lr_{ci,cj} \vee \neg ud_{cj,ci})) \tag{16}$$

2. **Large Rectangle Reduction**: For each pair of rectangles $ci, cj \in C, ci < cj$, if the sum of their widths is greater than W ($cw_{ci} + cw_{cj} > W$) we cannot pack them horizontally, meaning that they can never be on the left or on the right of one another, but only above or below. Therefore, we can set $lr_{ci,cj} = \bot$, $lr_{cj,ci} = \bot$ and remove all constraints which contain $\neg lr_{ci,cj}$ or $\neg lr_{cj,ci}$ (i.e. half of the constraints in Sections 1.3.2 and 1.3.3) because they will always be satisfied. The same can be said on the vertical dimensions: $(ch_{ci} + ch_{cj} > maxh) \implies (ud_{ci,cj} = \bot \wedge ud_{cj,ci} = \bot)$ and the removal of all constraints containing these variables.

2 Search

A model with the constraints we presented above is only able to provide a solution placing circuits in a $W \times maxh$ board, like the one presented in Figure 2a. Of course, we would like to *minimize* the height, so obtain a solution like the one in Figure 2c. SAT solving has no mechanisms to deal with *optimization problems* directly, therefore we adopt the indirect, but powerful method presented in [1], implemented using the same order encoding framework as all other variables.



(a) Non-optimal solution, max allowed height set to $maxh$.

(b) Sub-optimal solution, max allowed height set to 14.

(c) Optimal solution of the instance.

Figure 2: Solutions of the same instance with different cumulative heights.

First of all, we add yet another set of boolean variables to the model: ph_o for $o \in \{lowh, \ldots, maxh\}$ indicating whether all circuits can be packed with no overlaps within a total height that is $\leq o$. Given that

we have an initial solution at height $maxh$, ph_{maxh} is always true. Since the height boundary is expressed in order encoding, we have the usual ordering integrity constraint:

$$\bigwedge_{o \in \{lowh,\ldots,maxh-1\}} (\neg ph_o \vee ph_{o+1}) \tag{17}$$

By definition, this set of variables should affect the placement of the circuits in the board so that if ph_o is true for some o, it means that all circuits $c \in C$ must have been placed before $o - ch_c$: no circuit must extend above o. We translate this requirement into the following constraint:

$$\bigwedge_{o \in \{lowh,\ldots,maxh-1\}} \bigwedge_{c \in C} (\neg ph_o \vee py_{c,o-ch_c}) \tag{18}$$

The direct consequence of adding these variables to the problem is that we have the power to *decide* the maximum height to be reached by the circuits. Indeed, all it takes to keep circuits within the height limit o is to add $ph_o = \top$ as a constraint into the problem: if the problem is satisfiable at that height level, thanks to our constraints, the solver will assign y-coordinates that respect that boundary.

2.1 From Satisfiability to Optimality

Even though we have a powerful mechanism to set the maximum height boundary of the board, our model is missing an automated way to obtain the *optimal solution*. In most of the provided instances, the optimal h is equal to $lowh$ because there is always at least one way to place circuits so that there are no empty spaces in between. The problem is that we cannot make this assumption for the general case, so we need an algorithm to *test solvability* at many different boundaries and *choose* the best one as our optimal h.

In [1], the *bisection method* is proposed as an efficient way to obtain the best solution. The idea is to try and solve the problem using as height boundary $o = l + \lfloor (m - l)/2 \rfloor$ (so, setting $ph_o = \top$), where l and m are initially set to $lowh$ and $maxh$. If the problem is SAT, m is reduced to o, while if it is not SAT, l is increased to o. Then, we restore the previous solver state (retract $ph_o = \top$ and all learnt clauses), we update o with the new boundaries and repeat the process until $l = m$. When this happens, if the problem is not SAT with $ph_l = \top$, we declare the problem unsolvable, otherwise l is the best possible height boundary for the problem and we can extract an optimal solution from the rest of the variables.

Solving multiple SAT problems can of course be quite time-consuming and does not scale well to large instances, but for most of the provided instances the solver is already quite fast and the sequence of problems is solved in just a few seconds (see Table 1). Furthermore, Z3 efficiently implements an *incremental solving* mechanism which allows us to *save* (`push`) the status of a solver and *restore* (`pop`) it later. We heavily employ this mechanism for adding and retracting the constraints relative to ph variables.

3 Rotation Extension

We were able to extend the described model to also support circuit rotations. Semantically, the constraints of the extended model have the same meaning of those in the original model, but because of the need to deal with many more special cases we almost *doubled* the number of total constraints. On the other hand, rather than introducing variables for the current widths and heights of the circuits as in CP, we only add a *single* variable for each circuit $c \in C$: rot_c.

For all circuits, we assume that the *normal* situation of a circuit is the one described previously, with cw_c being its width and ch_c its height: this is the situation described by variable $rot_c = \bot$. If $rot_c = \top$, we instead consider the circuit to be rotated and thus its width becomes ch_c and its height cw_c.

We add two completely new constraints:

- A circuit cannot be rotated if its normal width is greater than the maximum height, or if its normal height is greater than the maximum width. Formally:

$$\bigwedge_{c \in C, (ch_c > W) \vee (cw_c > maxh)} (\neg rot_c) \tag{19}$$

- It's pointless to rotate a circuit whose height and width are the same:

$$\bigwedge_{c \in C, ch_c = cw_c} (\neg rot_c) \tag{20}$$

Then, we need to add cases to all other constraints. The general pattern is that for each of the previously expressed constraints, if there is a reference to cw_c or ch_c for some circuit $c \in C$, we need to add two constraints to the model that allows rotation:

$$\neg rot_c \implies \text{old constraint}$$

$$rot_c \implies \text{old constraint, but all references to } cw_c \text{ and } ch_c \text{ are swapped}$$

For instance, the domain constraints 1 and 2 become:

$$\bigwedge_{c \in C} \neg rot_c \implies \bigvee_{0 \leq e \leq W - cw_c} (px_{c,e}) \tag{21}$$

$$\bigwedge_{c \in C} rot_c \implies \bigvee_{0 \leq e \leq W - \boldsymbol{ch_c}} (px_{c,e}) \tag{22}$$

$$\bigwedge_{c \in C} \neg rot_c \implies \bigvee_{0 \leq f \leq maxh - ch_c} (py_{c,f}) \tag{23}$$

$$\bigwedge_{c \in C} rot_c \implies \bigvee_{0 \leq f \leq maxh - \boldsymbol{cw_c}} (py_{c,f}) \tag{24}$$

Constraints 3 and 4 undergo a similar transformation, while 9, 10, 15, 16 and 17 are kept with no additions because they do not reference heights nor widths. The transformations for constraints 5 and 7 are:

$$\bigwedge_{ci,cj \in C, ci < cj} \neg rot_{ci} \implies \bigwedge_{0 \leq e < W - cw_{ci}} (\neg lr_{ci,cj} \vee px_{ci,e} \vee \neg px_{cj,e+cw_{ci}}) \tag{25}$$

$$\bigwedge_{ci,cj \in C, ci < cj} rot_{ci} \implies \bigwedge_{0 \leq e < W - \boldsymbol{ch_{ci}}} (\neg lr_{ci,cj} \vee px_{ci,e} \vee \neg px_{cj,e+\boldsymbol{ch_{ci}}}) \tag{26}$$

$$\bigwedge_{ci,cj \in C, ci < cj} \neg rot_{ci} \implies \bigwedge_{0 \leq f < maxh - ch_{ci}} (\neg ud_{ci,cj} \vee py_{ci,f} \vee \neg py_{cj,f+ch_{ci}}) \tag{27}$$

$$\bigwedge_{ci,cj \in C, ci < cj} rot_{ci} \implies \bigwedge_{0 \leq f < maxh - \boldsymbol{cw_{ci}}} (\neg ud_{ci,cj} \vee py_{ci,f} \vee \neg py_{cj,f+\boldsymbol{cw_{ci}}}) \tag{28}$$

Constraint 18 is also duplicated:

$$\bigwedge_{o \in \{lowh, \ldots, maxh-1\}} \bigwedge_{c \in C} (\neg rot_c \implies (\neg ph_o \vee py_{c,o-ch_c})) \tag{29}$$

$$\bigwedge_{o \in \{lowh, \ldots, maxh-1\}} \bigwedge_{c \in C} (rot_c \implies (\neg ph_o \vee py_{c,o-\boldsymbol{cw_c}})) \tag{30}$$

We don't show the transformations for all other constraints for brevity, but the key concept is the same of these highlighted examples.

The last noticeable difference between the two models is that we need remove the *Large Rectangle Reduction*, because we are not able to infer the rotation and thus the current widths of circuits at compile-time.

4 Implementation

The model has been implemented in Python using the Z3 Python API for expressing the variables and constraints. Similarly to the CP program, we have built a *launcher* which contains the code for loading the instances specified by the user, for instancing the solver, the variables and constraints and for saving the solution as well as optionally displaying it graphically. The algorithms that provide the initial solution and sort the circuits by their size have been re-used from the CP program implementation.

Differently from CP, the output of the solving process is a mapping $var \to v \in \{\top, \bot\}$ from each of the boolean variables instanced in the model to the value that has been assigned to it by the SAT solver. To obtain a clear solution, we apply a *post-processing step* where we take the set of *true* variables and transform it into the sequence of horizontal and vertical coordinates by taking the first variable in order for each circuit, as explained in Example 1.

During the solution, we keep track of the elapsed time with different timers: in particular, we noticed that the actual solving of the problem (`solver.check()` call) is of course the heaviest part of the program, but the post-processing step also requires some time due to the amount of variables, string filtering and such. Since the post-processing step is fundamental for the coherence of the solution, we decided to add these two timers together to express the final time seen in Table 1.

5 Results

We managed to solve 35 of the provided instances within the 5 minutes limit (depending on randomness of search and hardware), so 4 instances more with respect to the CP model and 2 with respect to SMT. With the rotation extension, we were able to solve 21 instances, an improvement of 1 instance with respect to CP.

Table 1 shows the results and solution times we obtained for each of the 40 instances using our models. In general, we observe that in almost all situations these model is the best performing one with a great margin with respect to the other methods.

References

[1] Takehide Soh et al. "A SAT-based Method for Solving the Two-dimensional Strip Packing Problem". In: *Fundam. Inform.* 102 (Jan. 2010), pp. 467–487. DOI: 10.3233/FI-2010-314.

[2] Naoyuki Tamura et al. "Compiling finite linear CSP into SAT". In: vol. 14. Sept. 2006. ISBN: 978-3-540-46267-5. DOI: 10.1007/11889205_42.

Instance	Solving Time	Solved	Solving Time (rot., s)	Solved (rot.)
1	0.001	True	0.000998	True
2	0.001005	True	0.003	True
3	0.001001	True	0.003	True
4	0.005	True	0.006	True
5	0.006003	True	0.019991	True
6	0.010005	True	0.094997	True
7	0.008997	True	0.171993	True
8	0.011999	True	0.02699	True
9	0.025002	True	0.177991	True
10	0.015002	True	1.527996	True
11	0.526994	True	15.465997	True
12	0.147004	True	10.307425	True
13	0.036	True	7.526007	True
14	0.200004	True	13.441997	True
15	0.044002	True	26.935004	True
16	2.577	True	>300	False
17	0.077002	True	60.443352	True
18	0.400993	True	>300	False
19	2.365996	True	>300	False
20	1.112994	True	>300	False
21	12.019999	True	>300	False
22	83.508985	True	>300	False
23	6.980997	True	218.606984	True
24	1.282004	True	90.249996	True
25	30.123996	True	>300	False
26	8.323	True	>300	False
27	2.245996	True	>300	False
28	12.357997	True	>300	False
29	3.068996	True	>300	False
30	>300	False	>300	False
31	0.726995	True	4.689826	True
32	>300	False	>300	False
33	4.249995	True	101.827429	True
34	8.593995	True	>300	False
35	22.312999	True	>300	False
36	3.814996	True	64.042751	True
37	>300	False	>300	False
38	>300	False	>300	False
39	51.760747	True	>300	False
40	>300	False	>300	False

Table 1: Solving times and optimality status achieved by our models on the 40 instances of the problem. Times don't include the overhead of the Python launcher nor the time for finding an initial solution, but in most cases they are negligible.