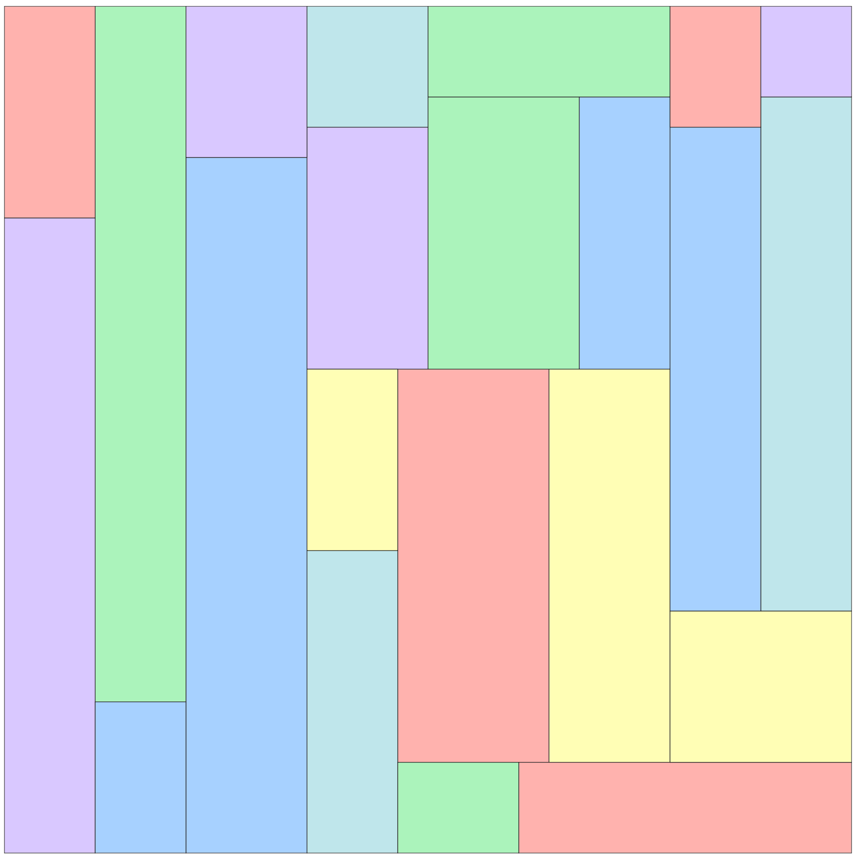
VLSI/Strip packing problem

Project report

Combinatorial Decision-Making and Optimization

Michele Calvanese michele.calvanese@studio.unibo.it
Vincenzo Collura vincenzo.collura2@studio.unibo.it
Davide Femia davide.femia@studio.unibo.it
Samuele Marino samuele.marino@studio.unibo.it



University of Bologna
Italy
11 July 2022

Contents

1 Introduction

VLSI (Very Large Scale Integration) refers to the trend of integrating circuits into silicon chips. A typical example is the smartphone. The modern trend of shrinking transistor sizes, allowing engineers to fit more and more transistors into the same area of silicon, has pushed the integration of more and more functions of cellphone circuitry into a single silicon die (i.e. plate). This enabled the modern cellphone to mature into a powerful tool that shrank from the size of a large brick-sized unit to a device small enough to comfortably carry in a pocket or purse, with a video camera, touchscreen, and other advanced features.

VLSI is equivalent to the 2D Strip packing problem (2SP): a set of rectangles and a strip of fixed width are given, and the goal is to find a non-overlapping disposition of the rectangles such that the height of the disposition is minimized. This optimization problem is strongly NP-hard. The corresponding decision problem where the strip height is also fixed is NP-complete.

In this report we describe Combinatorial Optimization approaches to the strip packing problem. We employ the following four techniques:

- Constraint Programming (CP);
- Boolean SATisfiability (SAT);
- Satisfiability Modulo Theory (SMT);
- Linear Programming (LP).

For each technique, we analyze the problem both where the rectangle's orientations are fixed, and where the rectangles are allowed to be rotated by 90 degrees.

2 Preliminaries

2.1 Instances

An instance of the strip packing problem consists of the following variables:

- the strip width W;
- n rectangles $r_i = (w_i, h_i)$, $i \in \{1, \dots, n\}$, where w_i and h_i are the width and the height of r_i respectively.

The goal is to find the coordinates (x_i, y_i) of the bottom left corners of the rectangles, such that they do not overlap and such that the height H of the strip is minimized. We assume integer values for the dimensions of the rectangles and the strip.

We have 40 different instances, sorted roughly in increasing order of difficulty. How they and the outputs are formatted is explained in section 1.2 of the project assignment [1].

2.2 Common preprocessing steps

For all the techniques we employed, we also tried to sort the rectangles in non-increasing order before giving them as input to the model. The core idea is that the larger rectangles are placed first, reducing the possibilities of placing the remaining rectangles and (hopefully) also the computation time.

2.3 Evaluation metrics

To evaluate the performance of each model we look at the following metrics (shown here in decreasing priority):

- Number of solved instances;
- Mean relative runtime compared to a chosen baseline model. This is defined as the geometric mean of the ratios between the computation times of the model of interest and the baseline. It is only calculated using the instances that were solved by both models.

We also show the average runtime calculated on the instances that were solved by each model. Since all the instances are independent of each other and because small relative variations on the computation time for the larger instances skew this metric a lot, this is not really a good metric to quantify the performance of the model, and is shown for illustrative purposes.

3 CP

Constraint Programming is a paradigm for solving combinatorial problems by stating, in a declarative fashion, a set of constraints that must hold on the feasible solutions for a given set of decision variables. In this section, we describe a CP approach for the VLSI problem.

3.1 Model description

3.1.1 Parameters and Decision variables

In order to model the problem, the following parameters were defined:

- **w**: the width of the plate;
- **n**: the number of items to be placed in the plate;
- **minH**, **maxH**: the lower and the upper bounds for the plate height;
- **bc**, **sbc**: two variables representing respectively the index of the biggest item in terms of the occupied area, and the second biggest item;
- **dimX**, **dimY**: two arrays containing respectively the horizontal and vertical dimensions of each item.

Then, we defined the model's decision variables:

- **l**: the plate height;
- **x**, **y**: two arrays that represent the coordinates of each item's origin point (the bottom left corner of the item) along the x and y axes.
  Note that the domains of these two set of variables differ because the highest feasible x-coordinate cannot be greater than `w - min(dimX)`, and the highest feasible y-coordinate cannot be greater than `maxH - min(dimY)`.

We want to minimize plate height, so we simply define the objective function as `solve minimize l`.

3.1.2 Constraints

Main problem constraints

The problem definition begins with the specification of a constraint which imposes a no-overlap relationship between items. To do this, we used the library of global constraints which comes with MiniZinc. The main reason behind this choice is that a search process for a model which takes advantage of global constraints is faster because many solvers implement special, efficient inference (propagation) algorithms for this special type of constraints.

Then we give bounds to the origins of each circuit enforcing the blocks to stay within the plate. Here are the main constraints' specifications:

```
%main problem constraints
constraint diffn(x, y, dimX, dimY) :: domain;
constraint forall (i in CIRCUITS) (x[i] + dimX[i] <= w);
constraint forall (i in CIRCUITS) (y[i] + dimY[i] <= l);
```

The annotation `domain` simply tells the solver to mantain GAC *(Generalized Arc Consistency)* instead of BC *(Bounds Consistency)*. That's because propagation with GAC can remove a larger number of inconsistent values from the variables' domains of the variables, thus reducing the search space even more.

Implied constraints

Implied constraints are logical consequences of the initial specification of the problem. Though adding an implied constraint to the specification of a constraint satisfaction problem does not change the set of solutions, it can reduce the amount of search the solver has to do by increasing propagation.

Considering the problem at hand, we can make the following consideration: if we draw a horizontal line and sum the horizontal sides of the traversed circuits, the sum can be at most **w**. A similar property holds if we draw a vertical line. Basically we can state that:

$$\begin{aligned} \sum_{i \mid \, y_i \leq y \leq y_i + \mathbf{dimY_i}} x_i \quad &\leq \mathbf{w} \qquad \forall y \\ \sum_{i \mid \, x_i \leq x \leq x_i + \mathbf{dimX_i}} y_i \quad &\leq \mathbf{l} \qquad \forall x. \end{aligned} \tag{3.1}$$

These 2 constraints can be implemented through the global constraint `cumulative(s,d,r,b)`, which states:

$$\sum_{i \mid \, s_i \leq u \leq s_i + d_i} r_i \quad \leq b \qquad \forall u. \tag{3.2}$$

We therefore impose:

```
constraint cumulative(x, dimX, dimY, l) :: domain;
constraint cumulative(y, dimY, dimX, w) :: domain;
```

We then consider that, if 2 items cannot be placed one of the side of the other, they must be on top of each other (we don't know which is in top of which). A similar property holds for 2 items that cannot be placed on top of each other. To impose these 2 constraint we write:

```
constraint forall(i,j in CIRCUITS where j>i)(
if dimX[i]+dimX[j]>w then
    y[i]<y[j] -> y[i]+dimY[i]<=y[j] /\ y[j]<y[i]-> y[j]+dimY[j]<=y[i]
endif);

constraint forall(i,j in CIRCUITS where j>i)(
dimY[i]+dimY[j]>l ->
    (x[i]<x[j] -> x[i]+dimX[i]<=x[j] /\ x[j]<x[i]-> x[j]+dimX[j] <=x[i])
);
```

Symmetry breaking constraints

This problem has many symmetries. Here, we describe the ones we used.

Rotation and reflection. Given a perfectly packed board of circuits, we can identify 6 main symmetries: rotation of the plate by 90°, 180° and 270°, reflection over the x- and y-axis, and both rotation and reflection simultaneously. To break these symmetries, an ordering is imposed between the biggest item and the second biggest item. By forcing the biggest item to be to the bottom left of the second biggest one, 3 symmetric solution are eliminated (i.e. when the second biggest item is to the top left, top right or bottom right of the biggest one). We can also reduce the domain for the biggest item by stating

$$x_{\mathbf{bc}} \leq \left\lfloor \frac{\mathbf{w} - \mathbf{dimX_{bc}}}{2} \right\rfloor \quad \text{and} \quad y_{\mathbf{bc}} \leq \left\lfloor \frac{\mathbf{l} - \mathbf{dimY_{bc}}}{2} \right\rfloor. \tag{3.3}$$

Row-item and column-item. Whenever two items i, j of equal height are besides each other, they can be freely swapped. The same applies for items of equal width on top of each other. To break this type of symmetry, we impose an ordering between the two items on the varying coordinate whenever we have this kind of situation.

3-items symmetry. This symmetry is the 3-item equivalent of the previous one. It involves 2 adjacent circuits i, j of equal height, which are both adjacent to a bigger circuit k for which we have that the width of k is the sum of the widths of i, j. In this case, we can freely swap the "composite" circuit i, j with circuit k. A similar reasoning applies to the horizontal case where i, j share the same width, and the height of k is the sum of the heights of i, j. To break these symmetries, we impose an ordering on the coordinates of i and k.

Items with same dimensions. items with the same dimensions can be freely swapped. Therefore, we impose that the origin point of one of them is bottom left with respect to the other one.

Summary. Here there is a summary with the code for all the symmetry breaking constraints:

```
% break symmetries for the biggest item
constraint symmetry_breaking_constraint(
x[bc]<=(w-dimX[bc]) div 2 /\ y[bc] <= (l-dimY[bc]) div 2);

% biggest item bottom left to second biggest item
constraint x[bc] <= x[sbc] /\ y[bc] <= y[sbc];

% row-item and column-item symmetry
constraint symmetry_breaking_constraint(
    forall (i,j in CIRCUITS where i < j) ((x[i] == x[j] /\ dimX[i] == dimX[j])
        -> y[i] <= y[j] - dimY[i]));
constraint symmetry_breaking_constraint(
    forall (i,j in CIRCUITS where i < j) ((y[i] == y[j] /\ dimY[i] == dimY[j])
        -> x[i] <= x[j] - dimX[i]));

% three items symmetry
constraint symmetry_breaking_constraint(
    forall (i,j,k in CIRCUITS where i > j /\ j > k)
        ((x[i] == x[j] /\ dimX[i] == dimX[j] /\ y[i] == y[k] /\ dimY[i] + dimY[j
            ] == dimY[k]) -> x[k] <= x[i] ));
constraint symmetry_breaking_constraint(
    forall (i,j,k in CIRCUITS where i > j /\ j > k)
        ((y[i] == y[j] /\ dimY[i] == dimY[j] /\ x[i] == x[k] /\ dimX[i] + dimX[j
            ] == dimX[k]) -> y[k] <= y[i] ));

% items of the same dimensions
constraint symmetry_breaking_constraint(
    forall(i,j in CIRCUITS where j>i)
        (if dimX[i]==dimX[j] /\ dimY[i]==dimY[j] then
            x[i]<=x[j] /\ y[i]<=y[j] endif));
```

3.1.3 Rotation

The general case of the problem at hand allows the rotations of the circuits, meaning that a circuit of size $\mathbf{dimX}_k \times \mathbf{dimY}_k$ can be placed on the plate as a $\mathbf{dimY}_k \times \mathbf{dimX}_k$ circuit. To allow our model to rotate circuits, we introduce additional variables and constraints. In particular, we add an array of boolean variables `rot` of size n, to keep track of the rotations of the individual circuits; moreover, we add two arrays `rdimX` and `rdimY` of decision variables. Finally, we change all the previous constraints defined on `dimX` and `dimY` to now use `rdimX` and `rdimY`. This is because now also the item dimensions are a variable, which depends on whether each circuit has been rotated or not.

To reduce to the domains of the new variables **rdimX** and **rdimY** to a minimum, we add these constraints:

```
constraint forall(i in CIRCUITS)(rdimX[i] = dimX[i] \/ rdimX[i] = dimY[i]);
constraint forall(i in CIRCUITS)(rdimY[i] = dimX[i] \/ rdimY[i] = dimY[i]);
```

We then introduce a new symmetry breaking constraint, where we impose that square circuits must not to be rotated (as their size is not affected by rotations):

```
constraint forall(i in CIRCUITS)(if dimX[i]==dimY[i] then rot[i]=false endif);
```

Finally, we add the domain reductions constraints for the **rot** variable, where we impose that we cannot rotate items whose height/width exceed the width/height of the plate:

```
constraint forall(i in CIRCUITS)(if dimY[i]>w then rot[i]=false endif);
constraint forall(i in CIRCUITS)(dimX[i]>l -> rot[i]=false);
```

3.2 Search strategies

One of the key features of CP is the possibility of interacting with the search process by specifying different search strategies, such as heuristics for variable and value ordering, and restart strategies. In MiniZinc, by default there is no such specification, so the search process is handled entirely by the underlying solver (which can be chosen as well), but indeed we can specify how the search should be carried out. Note that the search strategy is not really part of the model, but for sure it is essential in tackling more difficult problems.

In MiniZinc, we can specify extra search information to the constraint solver using annotations. In particular, we consider a search annotation for specifying how to carry out the search on the **x**, **y** integer variables of our model, i.e. the `int_search(<variables>, <varchoice>, <constrainchoice>)` annotation, where `<variables>` is a one dimensional array of var int, `<varchoice>` is a variable choice annotation (i.e., a *variable ordering heuristic*, VOH) and `<constrainchoice>` is a value ordering strategy. We will consider the following three VOHs:

- `input_order`: chooses the variables in the given order, which for our model means to choose the biggest circuit not yet positioned (we order the input items by area in non increasing order, like explained in sectin 2.2.)

- `dom_w_deg`: chooses the variable x that minimizes $dom(x)/w(x)$, where $dom(x)$ is the (current) domain size of x and $w(x)$ is the weighted degree of x, computed as the sum of the weights of all constraints involving x (initially set to 1 and incremented each time the constraint fails).

- `impact`: chooses the variable with the highest impact so far. The intuitive idea behind impact-based heuristics is to estimate, for each variable, how much each possible assignment reduces the search tree size (the bigger the reduction, the "harder" the assignment; an assignment which fails has an impact of 1); thus, the variable with the highest impact is the "most difficult" one to assign and it is chosen, based on the first-fail principle.

As value ordering strategy, we use `indomain_min`, which simply assigns to the variable its smallest domain value.

Moreover, any kind of depth-first search used for solving optimization problems suffers from the problem that wrong decisions made at the top of the search tree can take an exponential amount of search to undo. One common way to alleviate this issue is to restart the search from the top, thus giving the solver the possibility of making different decisions. In MiniZinc, we can use restart annotations to control how frequently a restart occurs. We consider the following restart strategies:

- `restart_none`: no restart strategy is applied.
- `restart_luby(scale)`: the i-th restart occurs after $L[i] \cdot$ `scale` nodes in the search tree, where $L[i]$ is the i-th number of the Luby sequence. We set `scale` $= 150$.
- `restart_geometric(base,scale)`: the i-th restart occurs after `scale` $\cdot$ `base`^{i} nodes. We set `base` $= 2$ and `scale` $= 50$.

Indeed, note that restart strategies only make sense when applied to non-deterministic search strategies - in our case, only `dom_w_deg` and `impact` - as it does not make any sense to restart the search if the solver always explores the search tree in the same way.

3.3 Methods

We tried to use various MiniZinc solvers: Gecode [2], OR-Tools [3] and Chuffed [4]. Preliminary tests showed that Chuffed performed best, so we used it for our CP computations. We ran all the possible 9 combinations of the 3 VOH and the 3 restart strategies explained in the previous section. To evaluate our results, we used the same metrics described in 2.3. All computations were done on a laptop with an Intel Core i7-8565U CPU and 8 GB of RAM. We set a timeout of 300 s.

3.4 Results

We moved on investigating the different search strategies discussed in section 3.2, as well as the rotation-enabled model discussed in section 3.1.3. A summary of the results is provided in table 3.1, whereas full plots providing runtimes for the individual instances of the baseline models chosen in Table 3.1 for no rotation and rotation are in figure 3.1.

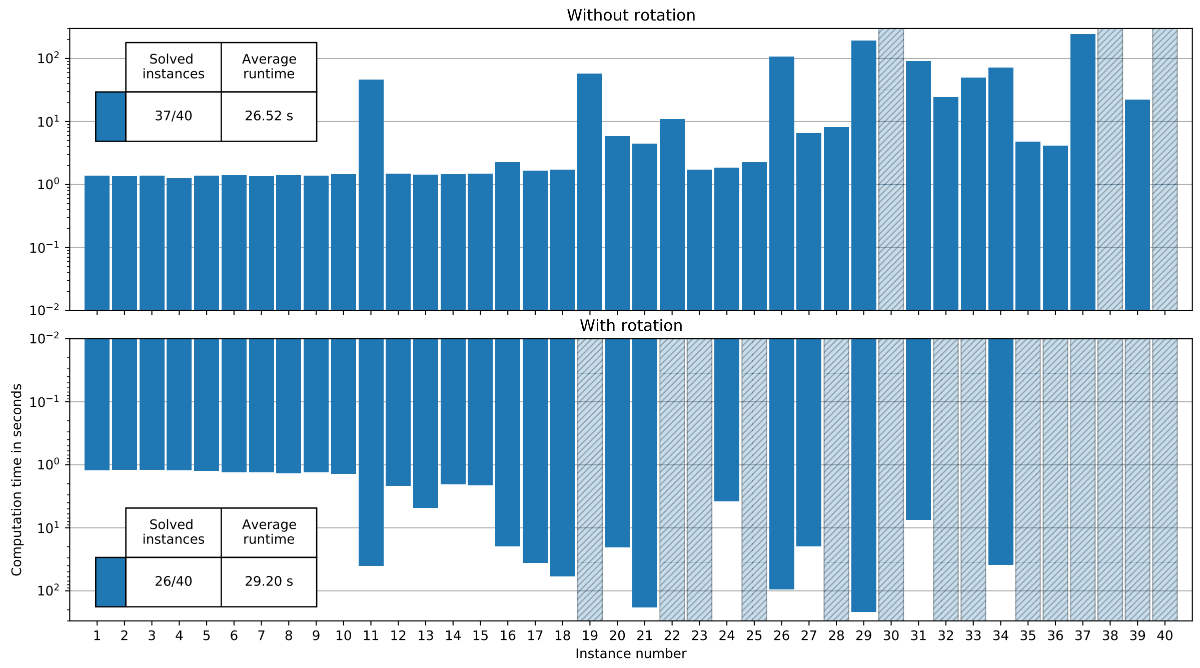


Figure 3.1: runtimes for the individual instances of the no rotation model with input_order as VOH and *luby*(150) restart, and the rotation model with input_order as VOH and without restart

Table 3.1: For the case without rotation we choose input_order as VOH and restart `luby(50)` as baseline for the mean relative runtime. For the rotation instead we choose `input_order` as VOH and no restart as baseline for the mean relative runtime. The baseline chosen for the 2 cases are clearly the best in our case.

Rotations	Search heuristics	Restart strategy	Solved instances	Mean relative runtime	Average runtime
No	`input_order`	`None`	30	1.12	16.95
		`luby(150)`	37	1.00	26.52
		`geometric(2,50)`	37	1.01	26.42
	`dom_w_deg`	`None`	36	1.59	18.11
		`luby(150)`	30	1.18	18.63
		`geometric(2,50)`	30	1.08	20.30
	`impact`	`None`	30	1.12	16.79
		`luby(150)`	30	1.43	17.61
		`geometric(2,50)`	30	0.99	16.73
Yes	`input_order`	`None`	26	1.00	29.20
		`luby(150)`	12	4.79	38.92
		`geometric(2,50)`	13	5.95	53.00
	`dom_w_deg`	`None`	25	0.98	21.93
		`luby(150)`	18	2.62	33.60
		`geometric(2,50)`	18	1.80	32.48
	`impact`	`None`	26	1.20	28.78
		`luby(150)`	18	1.86	33.40
		`geometric(2,50)`	17	1.83	20.62

3.5 Conclusions

We want to highlight the fact that the `input_order` heuristic combined with an ordering of the items by non-increasing area during a preprocessing step, which is indeed a deterministic search strategy, works better when enabling restart. Although in principle this does not make any sense (restart should work only with search strategies that use randomization), we think this is due to the solver employed (Chuffed).

4 SAT

In SAT (*Boolean SATisfiability problem*), the goal is to find whether, given a set of propositional formulas, there exists an interpretation which satisfies said set of formulas.

For this, there are many SAT-solvers, which are specialized in solving SAT problems (usually the problem has to be written in CNF (*Conjunctive Normal Form*)). The goal is to find an efficient SAT encoding in CNF of the strip-packing problem, which then can be passed to a SAT solver. In our case, we used the `z3` theorem prover in its Python library (version `4.8.17.0`) [5].

It is worth noting that in SAT it is impossible to have a target function to minimize (which in this case would be the height of the strip). This means that, to find the optimal height, we have to define a lower bound and an upper bound for the strip height, and use bisection search to find if a given strip height is SAT. Given a strip width W and a set of N rectangles with widths w_i and heights h_i, the lower and upper bounds for the strip height are defined as follows:

$$\begin{aligned} H_{\text{lb}} &= \left\lceil \frac{1}{W} \sum w_i \cdot h_i \right\rceil \qquad (4.1) \\ H_{\text{ub}} &= \frac{1}{2} \sum h_i. \end{aligned}$$

The optimal height is the minimum height for which the problem is satisfiable. To solve this problem in SAT, we mainly implemented the concepts explained in the 2003 paper by *T.Soh et.al.* [6]. The methods described in this paper give the most efficient SAT-encoding for the 2D strip-packing problem known to date.

In the following, we will therefore assume the width W and the height H of the strip to be given constants.

4.1 Theory: order encoding

To encode the values of coordinates of the rectangles we used order encoding. It works as follows: let x be an integer variable of domain $\{0, \ldots, n\}$, and $c \in \{0, \ldots, n-1\}$ an integer value. The order encoding literal px_c describes the truth value of the inequality $x \leq c$. The value of x can therefore be encoded in the set of literals $\{px_0, , \ldots, px_{n-1}\}$ of size n. An example of order encoding can be found in table 4.1. The number of literals of the order encoding of an integer variable is therefore at least equal to the cardinality of said variable's domain minus 1.

Table 4.1: For instance, let's say that x is an integer of domain $\{0, 1, 2, 3\}$. Its order encoding will be the set of literals $\{px_0, px_1, px_2\}$ of length 3. The correspondence between the value of x and the truth values of its order encoding can be seen in the table below.

value of x	order encoding
0	$\{T, T, T\}$
1	$\{F, T, T\}$
2	$\{F, F, T\}$
3	$\{F, F, F\}$

4.1.1 Ordering constraints

Since in an interpretation of a SAT problem with order encoding we never have a *False* after a *True*, we need the following axioms:

$$px_c \Rightarrow px_{c+1}, \quad c \in \{0, \ldots, n-2\}. \qquad (4.2)$$

In other words, if for instance we have an interpretation where $x = 1$, $x \leq 0$ does not hold (px_0 is *False*), but $x \leq 1$ does (px_1 is *True*). This is also true for all comparisons $x \leq c$ where c is greater or equal to 1 (all other px_c are *True*). It therefore never happens that px_c is *True* and px_{c+1} is *False*.

To obtain a CNF, the implication in (4.2) can be rewritten as a disjunction:

$$\neg px_c \vee px_{c+1}, \quad c \in \{\, 0, \, \ldots, \, n-2 \,\}. \tag{4.3}$$

Any SAT problem in order encoding needs these axiom clauses. In the following, we will refer to them as *ordering constraints*.

4.1.2 Comparing integers

The real advantage of order encoding becomes evident when comparing integers. Given two integer variables $x_1, x_2 \in \{\, 0, \, \ldots, \, n \,\}$ and an integer constant a, order encoding makes it very efficient to transform the comparison

$$x_1 + a \leq x_2 \tag{4.4}$$

in a CNF on the literals $px_{1,c}$ and $px_{2,c}$, with $c \in \{\, 0, \, \ldots, \, n-1 \,\}$. This is achieved by writing the comparison as a CNF of primitive comparisons $x_i \leq c$ for each value of c. This then directly corresponds to the literal $px_{i,c}$.

This is explained in more detail here: at first, we need to reduce the domains of x_1 and x_2 to make sure that $x_1 \leq n - a$ and $x_2 \geq a$. We need to add the following primitive comparisons as constraints:

$$\begin{aligned} x_1 &\leq k, \quad k \in \{\, n-a, \, \ldots, \, n \,\} \\ x_2 &\geq k, \quad k \in \{\, 0, \, \ldots, \, a \,\}. \end{aligned} \tag{4.5}$$

For x_2, we need to transform the $\geq$ comparison in $\leq$. We therefore obtain:

$$\neg(x_2 \leq k), \quad k \in \{\, 0, \, \ldots, \, a-1 \,\}. \tag{4.6}$$

Finally, we have to add the implications for all the possible intermediate values:

$$x_2 \leq a + k \;\Rightarrow\; x_1 \leq k, \quad k \in \{\, 0, \, \ldots, \, n-a-1 \,\}. \tag{4.7}$$

The final conjunction of primitive comparisons therefore looks like this:

$$\begin{aligned} & x_1 \leq k, && k \in \{\, n-a, \, \ldots, \, n \,\} \\ & \neg(x_2 \leq k), && k \in \{\, 0, \, \ldots, \, a-1 \,\}. \\ & x_2 \leq a + k \;\Rightarrow\; x_1 \leq k, && k \in \{\, 0, \, \ldots, \, n-a-1 \,\}. \end{aligned} \tag{4.8}$$

Since each comparison $x_i \leq c$ corresponds to the literal $px_{i,c}$, and the final SAT encoding of equation (4.4) is now straightforward:

$$\begin{aligned} & px_{1,k}, && k \in \{\, n-a, \, \ldots, \, n-1 \,\} \\ & \neg px_{2,k}, && k \in \{\, 0, \, \ldots, \, a-1 \,\} \\ & px_{2,a+k} \;\Rightarrow\; px_{1,k}, && k \in \{\, 0, \, \ldots, \, n-a-1 \,\}. \end{aligned} \tag{4.9}$$

For a true CNF, the final implication can be easily transformed into a disjunction.

4.2 Model description

In our problem, given a strip of width W and height H, and rectangles r_i, $i \in \{\, 1, \, \ldots, \, n \,\}$ with widths w_i and heights h_i, we want to find whether the strip packing problem is SAT, and if it is, we want to find the coordinates (x_i, y_i) of each rectangle.

4.2.1 Variables

At first, we need a SAT encoding for the coordinates of each rectangle. As we explained before, we use order encoding. For each rectangle r_i, $i \in \{1, \ldots, n\}$, we need a set of variables for the x and y coordinates:

$$\begin{aligned} &px_{i,e}\,, && e \in \{0, \ldots, W-1\}; \\ &py_{i,f}\,, && f \in \{0, \ldots, H-1\}. \end{aligned} \tag{4.10}$$

In addition, for each pair of rectangles r_i and r_j $(i < j)$, we also have the variables:

$$lr_{i,j}\,, \quad lr_{j,i}\,, \quad ud_{i,j}\,, \quad ud_{j,i}\,, \tag{4.11}$$

which encode their positions relative to each other. These variables are defined as follows: $lr_{i,j}$ is *True*, if rectangle i is placed to the left of rectangle j. $ud_{i,j}$ is *True*, if rectangle i is placed under of rectangle j. This is illustrated in figure 4.1.

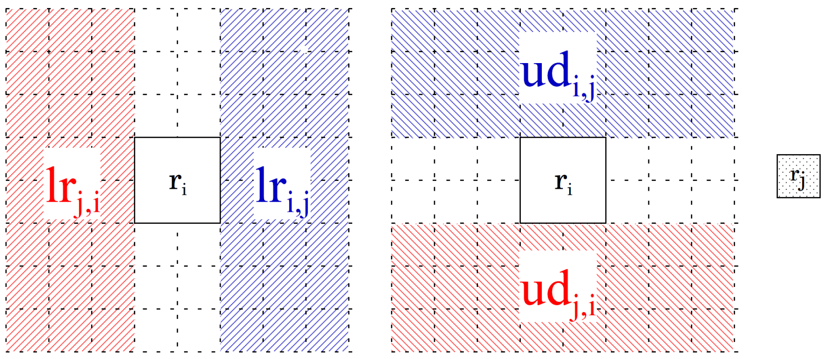


Figure 4.1: In the left image, for the rectangle r_i, the variable $lr_{i,j}$ is *True* if rectangle r_j is in the blue area, and the variable $lr_{j,i}$ is *True* if rectangle r_j is in the red area. The same goes for $ud_{i,j}$ and $ud_{j,i}$ in the right image. This picture is taken from the author's talk about the paper from *T.Soh et.al.* [7].

4.2.2 Constraints

We first need *ordering constraints* for $px_{i,c}$ and $py_{i,c}$, as explained in equation (4.3):

$$\begin{aligned} &\neg px_{i,e} \lor px_{i,e+1}\,, && e \in \{0, \ldots, W-1\}; \\ &\neg py_{i,f} \lor px_{i,f+1}\,, && f \in \{0, \ldots, H-1\} \end{aligned} \tag{4.12}$$

for each rectangle i. We also need to reduce the domain of the possible positions. This is necessary, because a rectangle i (of width w_i) must be positioned such that $x_i \leq W - w_i$. If we do not impose this constraint, the rectangle's right border would end up further right than the rightmost edge of the strip. The same holds true for the y coordinate. We therefore add the *domain-reducing constraints*:

$$\begin{aligned} &px_{i,e}\,, && e \in \{W-w_i, \ldots, W-1\} \\ &py_{i,f}\,, && f \in \{H-h_i, \ldots, H-1\} \end{aligned} \tag{4.13}$$

for each rectangle i.

Finally, we need the *non-overlapping constraints*. Using the relative positioning variables defined before, we can formulate the following a 4-literal clause:

$$lr_{i,j} \lor lr_{j,i} \lor ud_{i,j} \lor ud_{j,i}\,, \tag{4.14}$$

with $i < j$. The meaning of this clause is that rectangle j must not be positioned inside rectangle i. However, this clause alone is not sufficient, because it does not take the width and height of rectangle j into account. We therefore need additional clauses. For instance, if we position rectangle i left of rectangle j, we have to make sure that there is no overlap, i.e. that $x_i + w_i \leq x_j$. The same holds for the y coordinate, and when we invert the roles of rectangles i and j. This can be summarized as follows:

$$\begin{aligned} lr_{i,j} &\Rightarrow x_i + w_i \leq x_j \\ lr_{j,i} &\Rightarrow x_j + w_j \leq x_i \\ ud_{i,j} &\Rightarrow y_i + h_i \leq y_j \\ ud_{j,i} &\Rightarrow y_j + h_j \leq y_i \end{aligned} \tag{4.15}$$

for each pair of rectangles (i, j) with $i < j$. As mentioned before, we need to encode these inequalities using order encoding (see equation (4.9)). The non-overlapping constraints therefore become the following sets of clauses (as an example, only the encoding of the first clause is shown, the other ones are analogous):

$$\begin{aligned} lr_{i,j} &\Longrightarrow px_{i,e}\,, && e \in \{\, W - w_i,\, \ldots,\, W - 1 \,\}; \\ lr_{i,j} &\Longrightarrow \neg px_{j,e}\,, && e \in \{\, 0,\, \ldots,\, w_i - 1 \,\}; \\ lr_{i,j} &\Longrightarrow (px_{j,e+w_i} \Rightarrow px_i)\,, && e \in \{\, 0,\, \ldots,\, W - w_i - 1 \,\}. \end{aligned} \tag{4.16}$$

Due to the domain-reducing constraints defined in (4.13), the first set of clauses is redundant and can be removed. To obtain a true CNF, the implications can be rewritten as disjunctions:

$$\begin{aligned} &\neg lr_{i,j} \vee \neg px_{j,e}\,, && e \in \{\, 0,\, \ldots,\, w_i - 1 \,\}; \\ &\neg lr_{i,j} \vee \neg px_{j,e+w_i} \vee px_i\,, && e \in \{\, 0,\, \ldots,\, W - w_i - 1 \,\}. \end{aligned} \tag{4.17}$$

Summary

Here is a summary of the complete set of clauses in CNF.

- Ordering constraints, $i \in \{\, 1,\, \ldots,\, n \,\}$, from (4.12):

$$\begin{aligned} &\neg px_{i,e} \vee px_{i,e+1}\,, && e \in \{\, 0,\, \ldots,\, W - 1 \,\} \\ &\neg py_{i,f} \vee px_{i,f+1}\,, && f \in \{\, 0,\, \ldots,\, H - 1 \,\} \end{aligned}$$

- Domain-reducing constraints, from (4.13):

$$\begin{aligned} &px_{i,e}\,, && e \in \{\, W - w_i,\, \ldots,\, W - 1 \,\} \\ &py_{i,f}\,, && f \in \{\, H - h_i,\, \ldots,\, H - 1 \,\} \end{aligned}$$

- Non-overlapping constraints, $i, j \in \{\, 1,\, \ldots,\, n \,\}$, $i < j$, from (4.14) and (4.17):

$$lr_{i,j} \vee lr_{j,i} \vee ud_{i,j} \vee ud_{j,i}$$

$$\begin{aligned} &\neg lr_{i,j} \vee \neg px_{j,e+w_i} \vee px_i\,, && e \in \{\, 0,\, \ldots,\, W - w_i - 1 \,\} \\ &\neg lr_{j,i} \vee \neg px_{i,e+w_j} \vee px_j\,, && e \in \{\, 0,\, \ldots,\, W - w_j - 1 \,\} \\ &\neg ud_{i,j} \vee \neg py_{j,f+h_i} \vee px_i\,, && f \in \{\, 0,\, \ldots,\, H - h_i - 1 \,\} \\ &\neg ud_{j,i} \vee \neg py_{i,f+h_j} \vee px_j\,, && f \in \{\, 0,\, \ldots,\, H - h_j - 1 \,\} \end{aligned}$$

$$\begin{aligned} &\neg lr_{i,j} \vee \neg px_{j,e}\,, && e \in \{\, 0,\, \ldots,\, w_i - 1 \,\} \\ &\neg lr_{j,i} \vee \neg px_{i,e}\,, && e \in \{\, 0,\, \ldots,\, w_j - 1 \,\} \\ &\neg ud_{i,j} \vee \neg py_{j,f}\,, && f \in \{\, 0,\, \ldots,\, h_i - 1 \,\} \\ &\neg ud_{j,i} \vee \neg py_{i,f}\,, && f \in \{\, 0,\, \ldots,\, h_j - 1 \,\} \end{aligned}$$

4.2.3 Symmetry breaking

We also implemented symmetry breaking. It does not make any sense to add more constraints because, in contrast to CP, there is no propagation in SAT, but only backjumping and learned clauses. Adding more constraints would just reduce the number of possible solutions, and would make finding a solution slower. We therefore need to reduce the number of clauses.

We implemented 3 of the 4 symmetries described in *T.Soh et.al.* [6]:

- LR: Reducing the possibilities for placing large rectangles. For each rectangle r_i and r_j, if $w_i + w_j > W$, we can not pack these rectangles in the horizontal direction. It is therefore always the case that $ud_{i,j}$ or $ud_{j,i}$ is true, therefore we can remove the non-overlapping constraints containing $lr_{i,j}$ and $lr_{j,i}$:

$$\cancel{lr_{i,j}} \vee \cancel{lr_{j,i}} \vee ud_{i,j} \vee ud_{j,i}$$

$$\begin{array}{ll}
\cancel{\neg lr_{i,j} \vee \neg px_{j,e+w_i} \vee px_i}\,, & \cancel{e \in \{0,\ldots,W-w_i-1\}} \\
\cancel{\neg lr_{j,i} \vee \neg px_{i,e+w_j} \vee px_j}\,, & \cancel{e \in \{0,\ldots,W-w_j-1\}} \\
\neg ud_{i,j} \vee \neg py_{j,f+h_i} \vee px_i\,, & f \in \{0,\ldots,H-h_i-1\} \\
\neg ud_{j,i} \vee \neg py_{i,f+h_j} \vee px_j\,, & f \in \{0,\ldots,H-h_j-1\} \\
 & \\
\cancel{\neg lr_{i,j} \vee \neg px_{j,e}}\,, & \cancel{e \in \{0,\ldots,w_i-1\}} \\
\cancel{\neg lr_{j,i} \vee \neg px_{i,e}}\,, & \cancel{e \in \{0,\ldots,w_j-1\}} \\
\neg ud_{i,j} \vee \neg py_{j,f}\,, & f \in \{0,\ldots,h_i-1\} \\
\neg ud_{j,i} \vee \neg py_{i,f}\,, & f \in \{0,\ldots,h_j-1\}.
\end{array}$$

  The same can be done in the vertical direction.

- SR: Breaking symmetries for same-sized rectangles. For each rectangle r_i and r_j with $(w_i, h_i) = (w_j, h_j)$, we can fix the positional relation of these rectangles (i must be on the bottom left of j). We therefore modify the non-overlapping constraints as follows:

$$\begin{array}{l}
lr_{i,j} \vee \cancel{lr_{j,i}} \vee ud_{i,j} \vee ud_{j,i} \\
ud_{i,j} \Rightarrow lr_{j,i}
\end{array}$$

$$\begin{array}{ll}
\neg lr_{i,j} \vee \neg px_{j,e+w_i} \vee px_i\,, & e \in \{0,\ldots,W-w_i-1\} \\
\cancel{\neg lr_{j,i} \vee \neg px_{i,e+w_j} \vee px_j}\,, & \cancel{e \in \{0,\ldots,W-w_j-1\}} \\
\neg ud_{i,j} \vee \neg py_{j,f+h_i} \vee px_i\,, & f \in \{0,\ldots,H-h_i-1\} \\
\neg ud_{j,i} \vee \neg py_{i,f+h_j} \vee px_j\,, & f \in \{0,\ldots,H-h_j-1\} \\
 & \\
\neg lr_{i,j} \vee \neg px_{j,e}\,, & e \in \{0,\ldots,w_i-1\} \\
\cancel{\neg lr_{j,i} \vee \neg px_{i,e}}\,, & \cancel{e \in \{0,\ldots,w_j-1\}} \\
\neg ud_{i,j} \vee \neg py_{j,f}\,, & f \in \{0,\ldots,h_i-1\} \\
\neg ud_{j,i} \vee \neg py_{i,f}\,, & f \in \{0,\ldots,h_j-1\}.
\end{array}$$

- LS: Reducing the domain for the largest rectangle. We can place the largest rectangle r_m on the bottom left part of the grid. Therefore, we have that $x_m \leq \left\lfloor \frac{W-w_m}{2} \right\rfloor$, and $y_m \leq \left\lfloor \frac{H-h_m}{2} \right\rfloor$. The domain reducing constraints for r_m need to be modified accordingly:

$$\begin{array}{ll}
px_{m,e}\,, & e \in \left\{ \left\lfloor \frac{W-w_m}{2} \right\rfloor, \ldots, W-1 \right\} \\
py_{m,f}\,, & f \in \left\{ \left\lfloor \frac{H-h_m}{2} \right\rfloor, \ldots, H-1 \right\}.
\end{array} \tag{4.18}$$

We also need to update the non-overlapping constraints. Each rectangle r_i that respects the condition $w_i > \left\lfloor \frac{W - w_m}{2} \right\rfloor$ (i.e, a rectangle that is wider than half the width of the strip), cannot be placed left of r_m. We need to modify the non-overlapping constraints accordingly:

$$\cancel{lr_{i,m}} \vee lr_{m,i} \vee ud_{i,m} \vee ud_{m,i}$$

$$\begin{aligned}
&\cancel{\neg lr_{i,m} \vee \neg px_{m,e+w_i} \vee px_i\,,} && \cancel{e \in \{0, \ldots, W - w_i - 1\}} \\
&\neg lr_{m,i} \vee \neg px_{i,e+w_m} \vee px_m\,, && e \in \{0, \ldots, W - w_m - 1\} \\
&\neg ud_{i,m} \vee \neg py_{m,f+h_i} \vee px_i\,, && f \in \{0, \ldots, H - h_i - 1\} \\
&\neg ud_{m,i} \vee \neg py_{i,f+h_m} \vee px_m\,, && f \in \{0, \ldots, H - h_m - 1\}
\end{aligned}$$

$$\begin{aligned}
&\cancel{\neg lr_{i,m} \vee \neg px_{m,e}}\,, && \cancel{e \in \{0, \ldots, w_i - 1\}} \\
&\neg lr_{m,i} \vee \neg px_{i,e}\,, && e \in \{0, \ldots, w_m - 1\} \\
&\neg ud_{i,m} \vee \neg py_{m,f}\,, && f \in \{0, \ldots, h_i - 1\} \\
&\neg ud_{m,i} \vee \neg py_{i,f}\,, && f \in \{0, \ldots, h_m - 1\}.
\end{aligned}$$

The same can be done in the vertical direction.

4.2.4 Accounting for rotation

Up until now, we assumed the orientation of each rectangle to be fixed. To also allow 90-degree rotation, we introduce a new variable

$$R_i \tag{4.19}$$

for each rectangle i. R_i is *True* if rectangle i rotated, and *False* otherwise. If a rectangle is rotated, its height and width swap roles. This means that it is pretty easy to update the constraints using rotation by adding implications from R_i and $\neg R_i$. Only the domain-reducing constraints and the non-overlapping constraints need to be modified. The ordering constraints are unaffected.

The domain-reducing constraints are modified as follows:

$$\begin{aligned}
\neg R_i &\Rightarrow px_{i,e}\,, && e \in \{W - w_i, \ldots, W - 1\} \\
\neg R_i &\Rightarrow py_{i,f}\,, && f \in \{H - h_i, \ldots, H - 1\}
\end{aligned} \tag{4.20}$$

$$\begin{aligned}
R_i &\Rightarrow px_{i,e}\,, && e \in \{W - h_i, \ldots, W - 1\} \\
R_i &\Rightarrow py_{i,f}\,, && f \in \{H - w_i, \ldots, H - 1\}.
\end{aligned} \tag{4.21}$$

As can be seen, when rectangle i is rotated, w_i and h_i just swap roles. The non-overlapping constraints are also modified similarly. As an example, we only show the constraints involving $lr_{i,j}$:

$$\begin{aligned}
\neg R_i &\implies \neg lr_{i,j} \vee \neg px_{j,e}\,, && e \in \{0, \ldots, w_i - 1\}; \\
\neg R_i &\implies \neg lr_{i,j} \vee \neg px_{j,e+w_i} \vee px_i\,, && e \in \{0, \ldots, W - w_i - 1\}
\end{aligned} \tag{4.22}$$

$$\begin{aligned}
R_i &\implies \neg lr_{i,j} \vee \neg px_{j,e}\,, && e \in \{0, \ldots, h_i - 1\}; \\
R_i &\implies \neg lr_{i,j} \vee \neg px_{j,e+h_i} \vee px_i\,, && e \in \{0, \ldots, W - h_i - 1\}.
\end{aligned} \tag{4.23}$$

For the other variables $lr_{j,i}$, $ud_{i,j}$ and $ud_{j,i}$ this is done analogously.

Symmetry breaking can also be applied in this case, in the same way described before. With rotation there is a further symmetry: for squares, it does not matter if they are rotated or not. Therefore, we could get rid of their corresponding rotation literals, and reduce the number of clauses. This was not done due to time constraints from our part, but it is a possible improvement to be made in the future.

4.3 Methods

We compared the baseline model explained in section 4.2.2 with the model with symmetry breaking from 4.2.3. In addition, we also tried to preprocess the input data by sorting the rectangles by area in non-increasing order before passing them to the model, as explained in section 2.2.

We thus had the following three configurations, which were the same for the models with and without rotation:

- Baseline model, no preprocessing;
- Model with symmetry breaking, no preprocessing;
- Model with symmetry breaking, preprocessing of the input data sorting the rectangles by area in non-increasing order.

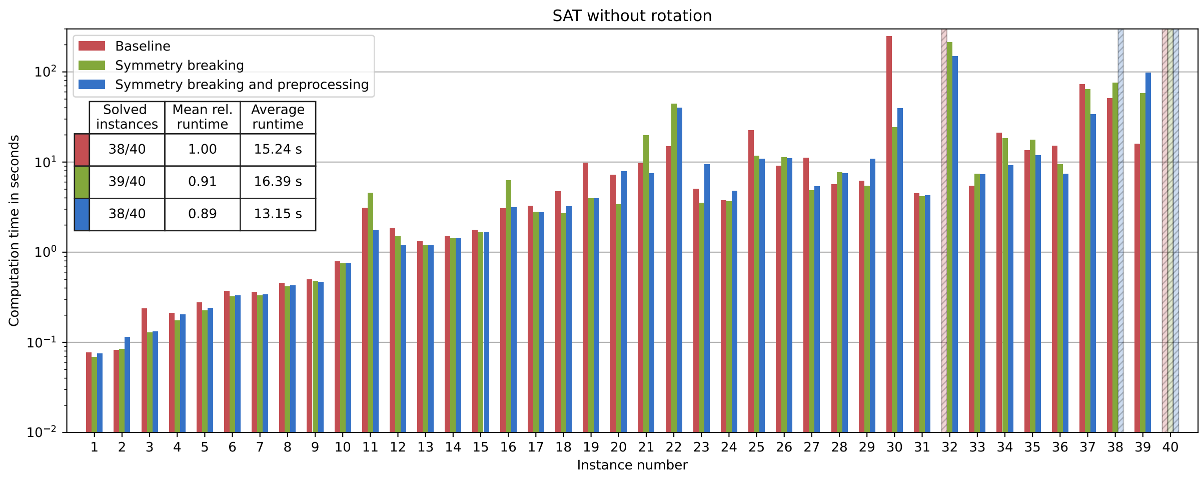
The evaluation metrics are explained in section 2.3. All the computations were done on a Dell XPS 15 9570, equipped with an Intel Core i7 8750H CPU. We set a timeout of 300 s.

4.4 Results

4.4.1 No rotation

No configuration was able so solve instance 40. The baseline configuration was not able to solve instance 32, and the configuration with symmetry breaking and data preprocessing was not able to solve instance 38. The configuration with symmetry breaking and no preprocessing performed best and was able to solve all the other instances.

The mean relative runtime is 0.91 for the configuration with symmetry breaking and no preprocessing, and 0.89 for the one with symmetry breaking and preprocessing. All the results are summarized in figure 4.2.



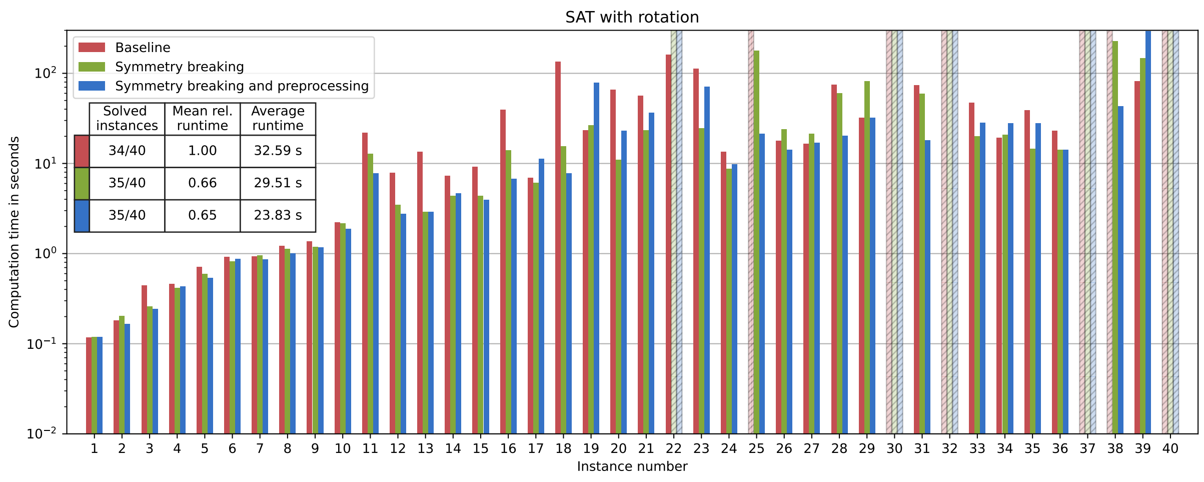
	Solved instances	Mean rel. runtime	Average runtime
🟥	38/40	1.00	15.24 s
🟩	39/40	0.91	16.39 s
🟦	38/40	0.89	13.15 s

Figure 4.2: Results for SAT without rotation. The plot shows the computation time for each instance in each configuration. The shaded instances were not solved before the timeout. The metrics for each configuration are shown in the table.

4.4.2 Rotation

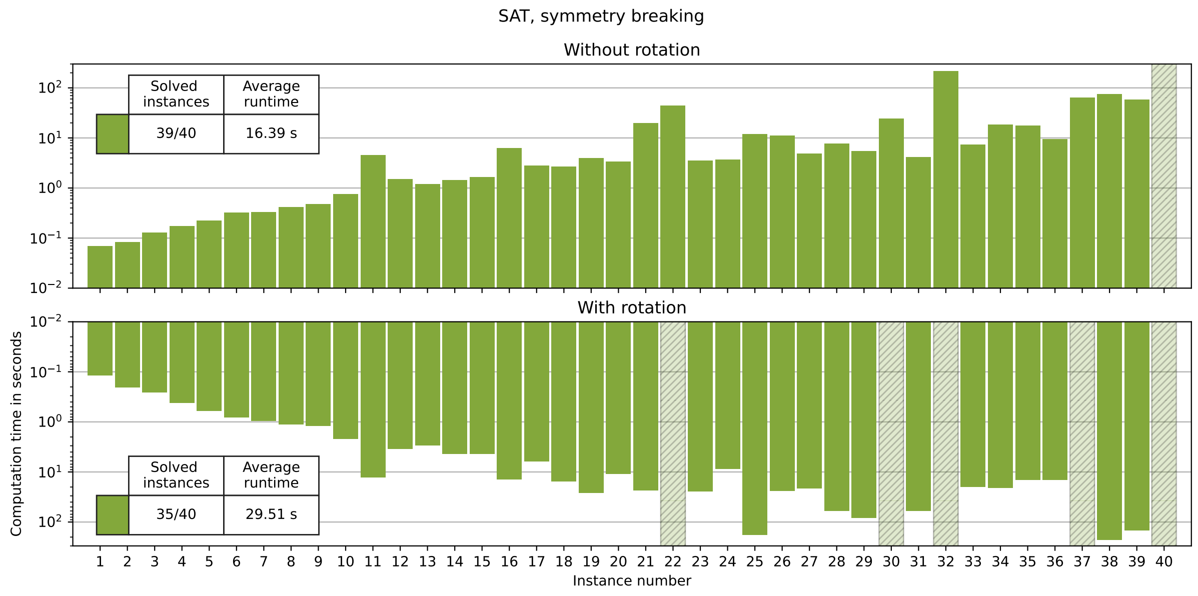
No configuration was able so solve instances 30, 32, 37 and 40. The baseline configuration solved 34 instances in total. The model with symmetry breaking solved a total of 35 instances, both with and without preprocessing. Interestingly, instance 22 was only solved by the baseline configuration (in 161 s). The configurations employing symmetry breaking were not able to solve it. In contrast, they solved instances 25 in 178 s and 21 s, and instance 38 in 226 s and 43 s. The times are respectively without and with preprocessing.

The mean relative runtime is 0.66 for the configuration with symmetry breaking and no preprocessing, and 0.65 for the one with symmetry breaking and preprocessing. All the results are summarized in figure 4.3. In addition, in figure 4.4 there is a comparison between the results with and without rotation. The configuration used there is the one with symmetry breaking and no preprocessing.



	Solved instances	Mean rel. runtime	Average runtime
🟥	34/40	1.00	32.59 s
🟩	35/40	0.66	29.51 s
🟦	35/40	0.65	23.83 s

Figure 4.3: Results for SAT with rotation. The plot shows the computation time for each instance in each configuration. The shaded instances were not solved before the timeout. The metrics for each configuration are shown in the table.



Without rotation:

	Solved instances	Average runtime
🟩	39/40	16.39 s

With rotation:

	Solved instances	Average runtime
🟩	35/40	29.51 s

Figure 4.4: Comparison between the SAT results without rotation and with rotation allowed. The configuration used here is the one with symmetry breaking and no preprocessing.

4.5 Conclusions

From these experiments, we can conclude that the encoding we chose is well-suited for solving the strip-packing problem. Symmetry breaking offers a significant performance increase compared to the baseline model. In the case where rotation is not allowed, it reduced the computation time by about 10 % on average. With rotation, this figure rises to about 35 %. This significant performance increase enabled the model to solve more instances before the timeout. These results are not unexpected, because the very purpose of symmetry breaking is to reduce the total number clauses to solve.

Sorting the input rectangles by area in non-increasing order reduces the computation time by about a further 2 %, both where rotation is not allowed and where it is. However, in the first case, instance 38 was not solved before the timeout. The sample we have is too small to draw a sensible conclusion regarding this type of preprocessing. Tests with more instances may be required.

4.6 Possible improvements

The encoding of the problem where rotation is allowed could be further improved as follows:

- Put the implications from R_i only for $e, f \in \{\, \min(w_i, h_i),\ \ldots,\ \max(w_i, h_i) \,\}$ in the domain-reducing constraints, and use the normal constraints for $e \in \{\, \max(w_i, h_i),\ \ldots,\ W \,\}$ and $f \in \{\, \max(w_i, h_i),\ \ldots,\ H \,\}$.
- Add symmetry breaking for squares by not introducing the variable R_i, and modifying the corresponding clauses.

Regarding testing, the computations with the various model configurations were done only once. To have more statistically significant results they should be carried out multiple times. In addition, using a laptop is also not ideal, because its CPU performance is influenced by the ambient temperature, for how long it has been computing and other environmental factors. A well-cooled desktop computer would provide more consistent results.

5 SMT

In SMT (*Satisfiability Modulo Theory*) the goal is to determine the satisfiability of first-order logic formulas through procedures over a background theory.

Like in SAT, the goal is to find an efficient SMT encoding of the strip-packing problem, which can then be solved by an SMT-solver. There are many SMT-solver which are specialized in solving SMT problems. A possibility we had was to use the same `z3` Python library [5] we used for SAT in chapter 4. However, this limits us to just use the `z3` solver. For SMT, there is SMT-LIB [8], which a standard library to encode SMT problems. This is useful, because it allows us to use any solver. For our problem, we used the `z3` [9] and `cvc5` [10] solvers.

In principle, in SMT it is possible to have a target function to minimize. However, this is not supported by SMT-LIB. So we used the same approach used in SAT (Offline OMT(`LRA`)), namely to fix the height H of the strip, and using bisection search to find the optimum. The lower and upper bounds for H are defined in the same way as in equation (4.1).

In the following, we will therefore assume the width W and the height H of the strip to be given constants.

5.1 Model description

Since in SMT-LIB all the clauses have to be written explicitly, we wrote Python code that generates an SMT-LIB file that can be passed to an external solver. We used the LIA (*Linar Integer Arithmetic*) logic. In the following, we show the variables and constraints that we used.

5.1.1 Variables

At first, we need an SMT encoding for the coordinates of each rectangle. For each rectangle r_i, $i \in \{1, \ldots, n\}$, of width w_i and height h_i, we introduce the integer variables x_i and y_i for the x and y coordinates:

$$\begin{aligned} x_i &\in \{0, \ldots, W-1\}, \\ y_i &\in \{0, \ldots, H-1\}. \end{aligned} \tag{5.1}$$

5.1.2 Constraints

At first, we need to define the *domain-constraints* of each x_i and y_i:

$$\begin{aligned} x_i &\geq 0, & y_i &\geq 0, \\ x_i &\leq W - w_i, & y_i &\leq H - h_i. \end{aligned} \tag{5.2}$$

The only other constraints needed are the *non-overlapping constraints*. For each pair of rectangles r_i and r_j $(i < j)$ we add a disjunction containing the following inequalities:

$$x_i + w_i \leq x_j \quad \vee \quad x_j + w_j \leq x_i \quad \vee \quad y_i + h_i \leq y_j \quad \vee \quad y_j + h_j \leq y_i. \tag{5.3}$$

5.1.3 Symmetry breaking

Like in SAT, we can employ symmetry breaking. In this case, we do not reduce the number of clauses, but the number of conditions in some of them. The symmetries we broke are the same we used in SAT. Here is a brief reminder, with the corresponding SMT constraints:

- LR: Reducing the possibilities for placing large rectangles. For each rectangle r_i and r_j, if $w_i + w_j > W$, we can not pack these rectangles in the horizontal direction. We can therefore remove the inequalities regarding for the horizontal direction in the non-overlapping constraints:

  $$\cancel{x_i + w_i \leq x_j} \quad \vee \quad \cancel{x_j + w_j \leq x_i} \quad \vee \quad y_i + h_i \leq y_j \quad \vee \quad y_j + h_j \leq y_i \,.$$

  The same can be done in the vertical direction.

- SR: Breaking symmetries for same-sized rectangles. For each rectangle r_i and r_j with $(w_i, h_i) = (w_j, h_j)$, we can fix the positional relation of these rectangles (i must be on the left of j). We therefore modify the non-overlapping constraints as follows:

  $$x_i + w_i \leq x_j \quad \vee \quad \cancel{x_j + w_j \leq x_i} \quad \vee \quad y_i + h_i \leq y_j \quad \vee \quad y_j + h_j \leq y_i \,.$$

- LS: Reducing the domain for the largest rectangle. We can place the largest rectangle r_m on the bottom left part of the grid. The domain reducing constraints for r_m need to be modified accordingly:

  $$x_m \leq \left\lfloor \frac{W - w_m}{2} \right\rfloor , \qquad\qquad y_m \leq \left\lfloor \frac{H - h_m}{2} \right\rfloor .$$

  We also need to update the non-overlapping constraints. Each rectangle r_i that respects the condition $w_i > \lfloor \frac{W - w_m}{2} \rfloor$ (i.e, a rectangle that is wider than half the width of the strip), cannot be placed left of r_m. We need to modify the non-overlapping constraints accordingly:

  $$\cancel{x_i + w_i \leq x_m} \quad \vee \quad x_m + w_m \leq x_i \quad \vee \quad y_i + h_i \leq y_m \quad \vee \quad y_m + h_m \leq y_i \,.$$

  The same can be done in the vertical direction.

5.1.4 Accounting for rotation

Up until now, we assumed the orientation of each rectangle to be fixed. To also allow 90-degree rotation, we use a slightly different approach compared to SAT. For each rectangle r_i, we introduce the new variables:

$$R_i \,, \quad W_i \,, \quad H_i \,. \tag{5.4}$$

Like in SAT, R_i is a boolean variable that indicates whether r_i is rotated or not. It is *True* in the first case, and *False* otherwise. W_i and H_i are integer variables that stand for the width and the height of r_i respectively. If a rectangle is rotated, its height and width swap roles. Therefore, we need to add the following clauses, which we call *rotation-constraints*:

$$\begin{aligned} \neg R &\implies W_i = w_i \,, & \neg R &\implies H_i = h_i \,, \\ R_i &\implies W_i = h_i \,, & R_i &\implies H_i = w_i \,. \end{aligned} \tag{5.5}$$

They assign the values to W_i and H_i based on whether r_i is rotated or not. The domain-constraints also need to be modified as follows:

$$\begin{aligned} \neg R_i &\implies x_i \leq W - w_i \,, & \neg R_i &\implies y_i \leq H - h_i \,, \\ R_i &\implies x_i \leq W - h_i \,, & R_i &\implies y_i \leq H - w_i \,. \end{aligned} \tag{5.6}$$

In the non-overlapping constraints, we just substitute the constants w_i and h_i with the variables W_i and H_i respectively:

$$x_i + W_i \leq x_j \quad \vee \quad x_j + W_j \leq x_i \quad \vee \quad y_i + H_i \leq y_j \quad \vee \quad y_j + H_j \leq y_i \,, \tag{5.7}$$

for each pair of rectangles r_i and r_j ($i < j$).

Symmetry breaking can also be applied in the same way described before. Like in SAT, we could introduce the rotation symmetry for squares. This was not done, but it is a possible improvement to be tried in the future.

5.2 Methods

We encoded our instances in SMT-LIB, and used an external solver to compute them. As mentioned before, we tried `z3` and `cvc5`. Preliminary tests showed that the former performs much better than the latter. Regarding the logic, we tried several ones. Preliminary tests showed the *LIA* (Linear Integer Arithmetic) and *QF_IDL* (Quantifier-free difference logic over integers) to perform best, with former performing slightly better. Therefore, the results presented below were computed using `z3` with the *LIA* logic.

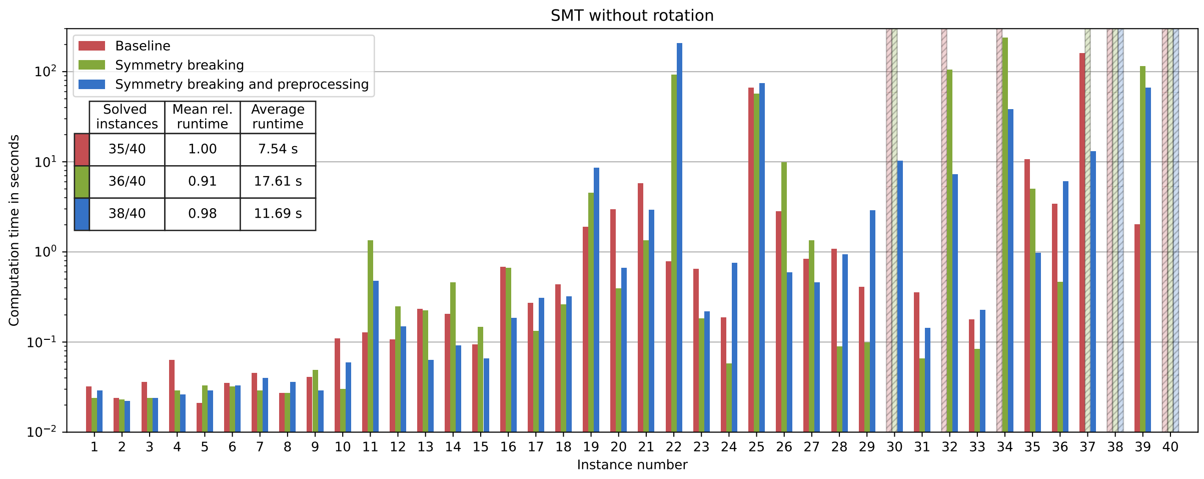
Regarding the model, the preprocessing and the computation hardware, we did the same things as we did for SAT. See section 4.3 for all the details. The evaluation metrics are explained in section 2.3.

5.3 Results

5.3.1 No rotation

No configuration was able so solve instances 38 and 40. The baseline configuration solved 35 instances in total. The model with symmetry breaking solved a total of 36 instances without preprocessing, and 38 instances with preprocessing.

The mean relative runtime is 0.91 for the configuration with symmetry breaking and no preprocessing, and 0.98 for the one with symmetry breaking and preprocessing. All the results are summarized in figure 5.1.



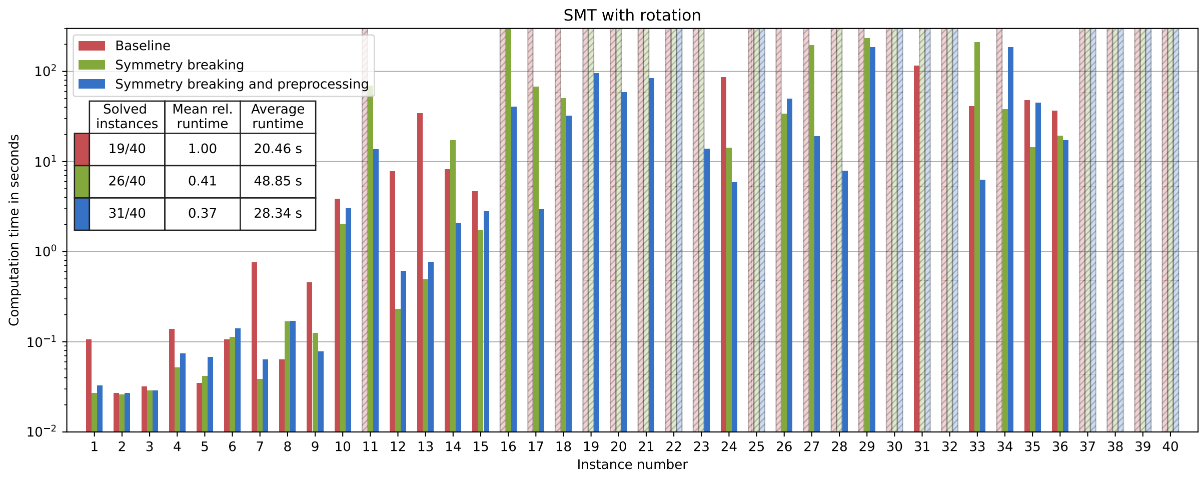
	Solved instances	Mean rel. runtime	Average runtime
🟥	35/40	1.00	7.54 s
🟩	36/40	0.91	17.61 s
🟦	38/40	0.98	11.69 s

Figure 5.1: Results for SMT without rotation. The plot shows the computation time for each instance in each configuration. The shaded instances were not solved before the timeout. The metrics for each configuration are shown in the table.

5.3.2 Rotation

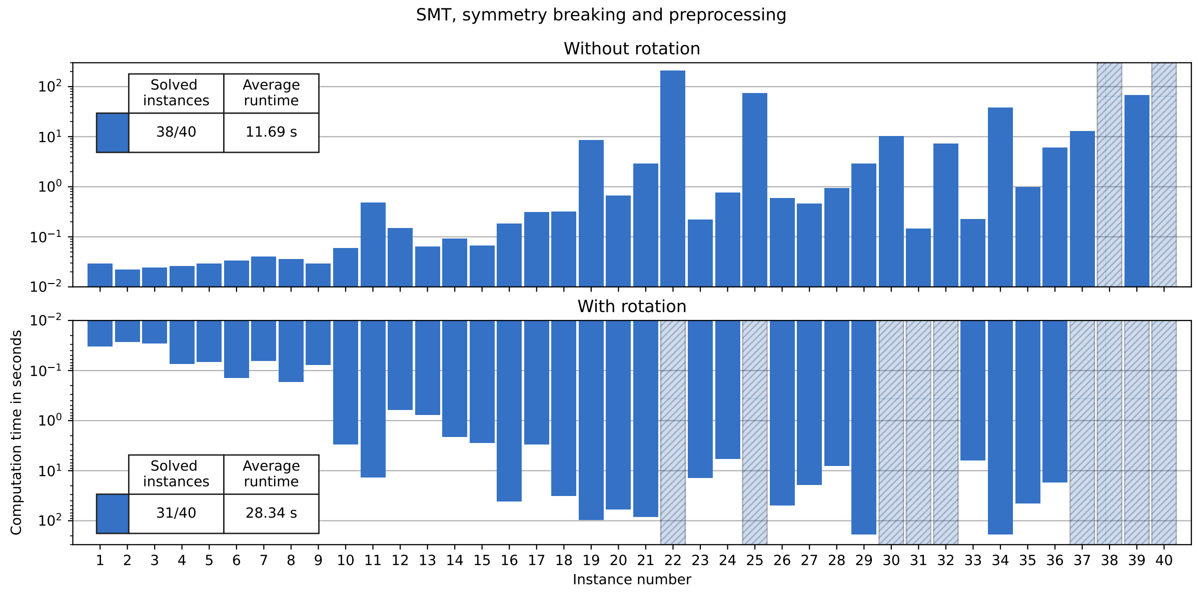
No configuration was able so solve instances 22, 25, 30, 32, and 37 to 40. The baseline configuration solved 19 instances in total. The model with symmetry breaking solved a total of 26 instances without preprocessing, and 31 instances with preprocessing. Interestingly, instance 31 was only solved by the baseline configuration (in 115 s).

The mean relative runtime is 0.41 for the configuration with symmetry breaking and no preprocessing, and 0.37 for the one with symmetry breaking and preprocessing. All the results are summarized in figure 5.2. In addition, in figure 5.3 there is a comparison between the results with and without rotation allowed. The configuration used there is the one with symmetry breaking and preprocessing.



	Solved instances	Mean rel. runtime	Average runtime
🟥	19/40	1.00	20.46 s
🟩	26/40	0.41	48.85 s
🟦	31/40	0.37	28.34 s

Figure 5.2: Results for SMT with rotation. The plot shows the computation time for each instance in each configuration. The shaded instances were not solved before the timeout. The metrics for each configuration are shown in the table.



Without rotation:

	Solved instances	Average runtime
🟦	38/40	11.69 s

With rotation:

	Solved instances	Average runtime
🟦	31/40	28.34 s

Figure 5.3: Comparison between the SMT results without rotation and with rotation allowed. The configuration used here is the one with symmetry breaking and preprocessing.

5.4 Conclusions

From these experiments, we can conclude that the encoding and preprocessing are critical when trying to solve the strip-packing problem using SMT. Symmetry breaking offers a significant performance increase compared to the baseline model. In the case where rotation is not allowed, it reduced the computation time by about 10 % on average. With rotation, this figure drops to only 2 %, but it makes the model able to solve 3 more instances.

The benefits of symmetry breaking and preprocessing become even more evident in the case where rotation is allowed: here, the model with symmetry breaking is able to solve 7 more instances compared to the baseline model. Preprocessing adds 5 further instances. The computation time improves by a significant margin too: without preprocessing, it drops by 59 %, and with preprocessing it drops by 63 % compared to baseline.

These results are in line with the expectation, because the purpose of symmetry breaking, similar to SAT, is to reduce the total number of constraints to satisfy. In contrast to SAT, the performance benefit of preprocessing is evident here. This is because the constraints are written in input order in the SMT-LIB file, so the solver tries to fix the positions of the larger rectangles first.

5.5 Possible improvements

In addition to the adding of symmetry breaking for squares in the case where rotation is allowed, solvers other than `z3` and `cvc5` could be tried. In addition, like in all the other solving approaches, the computations should be carried out multiple times on a desktop computer.

6 LP

LP (*Linear programming*) is a method to achieve the best outcome in a mathematical model whose requirements are represented by linear relationships. Linear programming is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints.

Our goal is to find an efficient LP encoding of the strip packing problem (2SP), which then can be passed to an LP solver. In order to solve the task we used the adaptation of the *Positions and Covering* (P&C) proposed by [11]. The adaptation of the P&C methodology for the 2SP is schematized in figure 6.1.

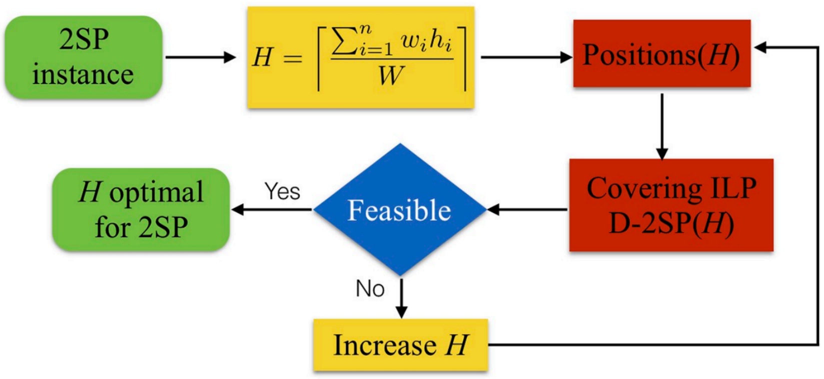


Figure 6.1: Methodology for solving the problem in LP. Calculating *Positions*(H) (first red rectangle) is explained in section 6.1.2 about preprocessing. The part about Covering ILP D-2SP(H) is explained in section 6.1.3. This picture is taken from [11].

In our case the task of calculating H and consequently Positions(H) will be carried out by the Python interpreter, while to actually solve the problem, given H and the positions correlated to H, will be our LP program written in MiniZinc and performed with the `Gurobi` (version 9.5.2) [12] and `CPLEX` (version 22.1.0.0) [13] solvers.

6.1 Model description

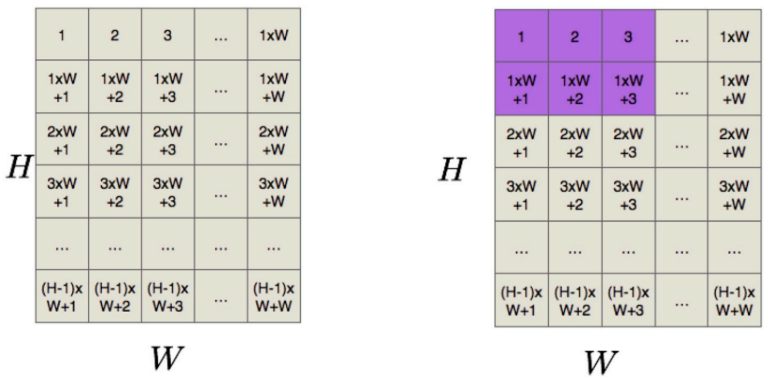
6.1.1 Estimating the initial strip height

Given an instance the first step is to compute the height H assuming that a perfect packing exists. Like in SAT, it is computed by dividing the total area of the items by the width W of the strip:

$$H = \left\lceil \frac{\sum_{i=1}^{n} w_i h_i}{W} \right\rceil . \tag{6.1}$$

6.1.2 Preprocessing

Now given H, we need to delineate a grid inside the strip, where each square has a particular identification. We arbitrarily choose the enumeration that starts at the top left corner square and ends at the bottom right square, as in figure 6.2a. Then for each item, if its top left corner overlaps with a tile and its width and height dimensions do not exceed the size of the strip, a valid position is created as in figure 6.2b.



(a) **(b)**

Figure 6.2: Figure (a) shows the grid with the enumerated tiles. Figure (b) shows a valid position for an item with dimensions 2×3 with its top-left corner at tile 1. Notice that the position which starts in the tile $1 \times W$ would not be a valid position for this item. This picture is taken from [11].

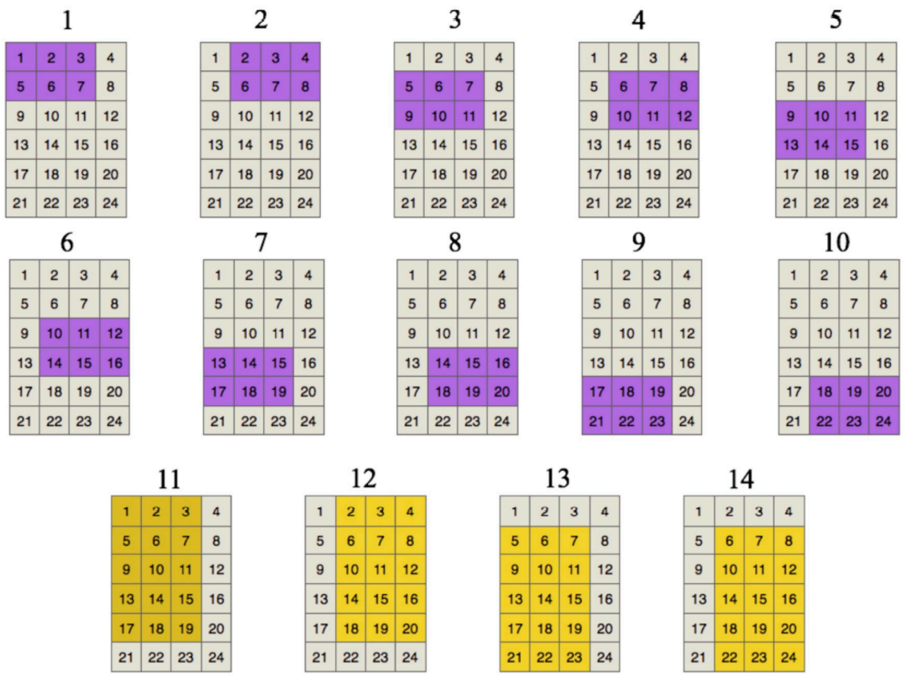


Figure 6.3: The set from 1 to 10 (purple) corresponds to the valid postitions for the 2×3, and the set from 11 to 14 (yellow) to the ones of the 5×3 item in the 6×4 strip. The resulting correspondence matrix can be seen in table 6.1. This picture is taken from [11].

Given more items we can show the set of valid positions: for example, in figure 6.3 we have all positions for an item of dimensions 2×3 and an item of dimensions 5×3 in a 6×4 strip. This gives us a total of 14 valid positions.

The resulting set of valid positions may be viewed as a correspondence matrix $\mathbf{C} = c_{jp}$, where the rows represent the set of valid positions and the columns are the tiles in the strip. This matrix is only composed of 1's and 0's, where $c_{jp} = 1$ if valid position j covers tile p, and $c_{jp} = 0$ otherwise. The correspondence matrix for the tiles in the previous example can be seen in table 6.1.

Table 6.1: This table shows the result of the correspondence matrix calculated from the strip and objects in figure 6.3. The rows correspond to the indices of the valid positions and the columns to the indices of the strip tiles.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	24
1	1	1	1	0	1	1	1	0	0	0	0	0	0	0	...	0
2	0	1	1	1	0	1	1	1	0	0	0	0	0	0	...	0
3	0	0	0	0	1	1	1	0	1	1	1	0	0	0	...	0
4	0	0	0	0	0	1	1	1	0	1	1	1	0	0	...	0
5	0	0	0	0	0	0	0	0	1	1	1	0	1	1	...	0
6	0	0	0	0	0	0	0	0	0	1	1	1	0	1	...	0
7	0	0	0	0	0	0	0	0	0	0	0	0	1	1	...	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	1	...	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	1
11	1	1	1	0	1	1	1	0	1	1	1	0	1	1	...	0
12	0	1	1	1	0	1	1	1	0	1	1	1	0	1	...	0
13	0	0	0	0	1	1	1	0	1	1	1	0	1	1	...	0
14	0	0	0	0	0	1	1	1	0	1	1	1	0	1	...	1

6.1.3 Constraints

Now a linear programming formulation for the decision version of the 2SP, is solved to optimality. After the preprocessing, we need to define the constraints. First, we need to define these important sets:

- I: set of all items;
- J: set of valid positions for all the items in the strip;
- $V(i) \subseteq J$: subset of valid positions associated to an item i.
- P: set of the tiles of the strip.

The decision variables for the LP model are:

$$x_{ij} = \begin{cases} 1 & \text{if item } i \text{ is placed in valid position } j, \\ 0 & \text{otherwise,} \end{cases} \tag{6.2}$$

where $i \in I$ and $j \in V(i)$. The formulation to solve the decision problem is then as follows:

$$\min z = 0 \tag{6.3}$$

s.t.

$$\sum_{i \in I} \sum_{j \in V(i)} c_{jp} \, x_{ij} \leq 1, \qquad p \in P_i \tag{6.4}$$

$$\sum_{j \in V(i)} x_{ij} = 1, \qquad i \in I \tag{6.5}$$

$$\sum_{j \in J \setminus V(i)} x_{ij} = 0, \qquad i \in I \tag{6.6}$$

$$\sum_{i \in I} \sum_{j \in V(i)} \sum_{p \in P} c_{jp} \, x_{ij} \leq WH, \tag{6.7}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in I, \; j \in J, \tag{6.8}$$

where (6.3) establishes that any feasible solution for the 2SP is desirable (`solve minimize 0` in MiniZinc). Constraints (6.4) avoid overlapping, by assigning at most one item at each tile of the strip. Constraints (6.5) guarantee that all the items will be packed exactly once into the strip. Constraints (6.6) guarantee that all the items will be packed into the strip only in a valid position. Finally, constraints (6.7) impose that the maximum capacity of the strip must not be exceeded.

A consideration should be made about constraint (6.5). This constraint could be acted upon as an optimization step. In the case where multiple items with the same dimensions are present in the input instance, these items could be treated as a single item i that has to be placed k_i times onto the strip (where k_i is the number of occurrences of item i). This is achieved by modifying 1 to k_i. We did not pursue this optimization, because in our case the instances do not contain duplicate items, and we would not be able to quantify the effect of this preprocessing step on the final outcome. However, this could be investigated upon in a further development of the project by using suitable problem instances.

The MiniZinc modelling for these constraints is the following:

```
constraint forall(p in TILES)(
                    sum([C[j,p]*x[i,j]|i in CIRCUITS, j in V[i-1]+1..V[i]])<=1);

constraint forall(i in CIRCUITS)(sum([x[i,j]|j in V[i-1]+1..V[i]])=1);

constraint forall(i in CIRCUITS)(
                    sum([x[i,j]|j in POSITIONS diff V[i-1]+1..V[i]])=0);

constraint
    sum([C[j,p]*x[i,j]|i in CIRCUITS, j in V[i-1]+1..V[i], p in TILES])<=W*H;
```

In the case where the covering model is feasible, H is the optimal height of the strip. The resulting configuration is thus taken as valid. The output is then post-processed, such that from the valid position assigned to an item we extract the coordinates of its bottom left corner.

In the case where the covering model is not feasible, the value of H is increased by one, and the procedure restart from the preprocessing step. A new set of valid positions (and its correspondence matrix) is generated with the updated H, and the covering model is solved again. The procedure ends when P&C finds a feasible solution. The resulting height is then the optimal one.

6.1.4 Accounting for rotation

The core idea is to add a rotated copy for each item in the input instance. We therefore need to extend the input instance, before all the computations. The new procedure can be seen in figure 6.4. Associations between each object and its rotated copy need to be made. Finally, the item packing constraints need to be updated, in particular (6.5) and (6.6).

The instance extension step works as follows: we create rotated copies of the items that respect all the following criteria:

$$w_i < H\,, \tag{6.9}$$
$$h_i < W\,, \tag{6.10}$$
$$h_i \neq w_i\,, \tag{6.11}$$
$$(h_i, w_i) \neq (w_j, h_j) \qquad \forall j \neq i\,, \tag{6.12}$$

with $i, j \in I$. We have a new set I_{ext}, which contains I and the added items that respect the above conditions. The items that do not respect the first three criteria are added to a set that we call `notAssoc`. All the other items are added to `assoc`, which is composed of the triple (i, j, t), with $i \in I$, $j \in I_{\text{ext}}$ and $t \in \{1, 2\}$. The items i that respect all the above conditions are added

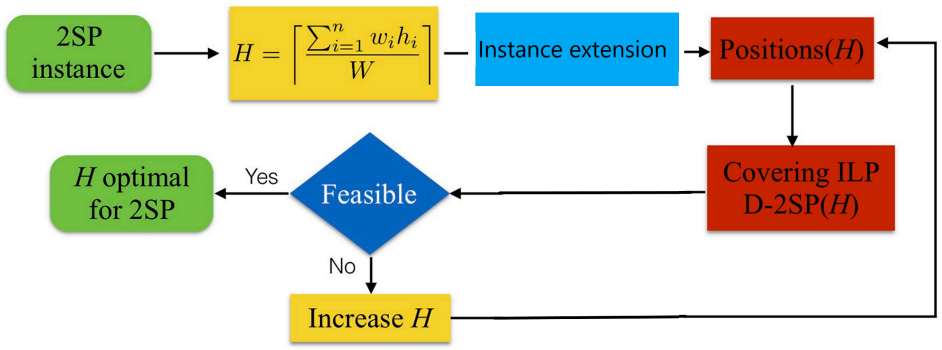


Figure 6.4: Methodology for solving the problem in LP with rotation. In the instance extension step the `assoc` and `notAssoc` sets are created. This picture was modified from [11].

to `assoc` with its rotated copy $j \in I_{\text{ext}} \setminus I$ as items of type 1. Therefore, we add the following triple to `assoc`:

$$(i, j, 1) \tag{6.13}$$

The items $i \in I$ that only do not respect the last condition, which means that there exists an item $j \in I$ that is a rotated copy of i, are added to `assoc` as items of type 2. Therefore, we add the following triple to `assoc`:

$$(i, j, 2) \tag{6.14}$$

Finally, the constraints (6.5) and (6.6) need to be updated:

$$\sum_{j \in V(i)} x_{ij} = 1, \qquad i \in \texttt{notAssoc}, \tag{6.15}$$

$$\sum_{j \in J \setminus V(i)} x_{ij} = 0, \qquad i \in \texttt{notAssoc}, \tag{6.16}$$

$$\sum_{k \in V(i)} x_{ik} + \sum_{k \in V(j)} x_{jk} = t, \qquad (i, j, t) \in \texttt{assoc}, \tag{6.17}$$

$$\sum_{k \in J \setminus V(i)} x_{ik} + \sum_{k \in J \setminus V(j)} x_{jk} = 0, \qquad (i, j, t) \in \texttt{assoc}. \tag{6.18}$$

Note that the set of valid positions J is recalculated on the new set I_{ext}.

Here is the formalization of the new constraints in MiniZinc:

```
constraint forall(a in ASSOC)(
            sum([x[assoc[a,1],j]  | j in V[assoc[a,1]-1]+1..V[assoc[a,1]]  ]) +
            sum([x[assoc[a,2],j]  | j in V[assoc[a,2]-1]+1..V[assoc[a,2]]  ])
            = assoc[a,3]);

constraint forall(a in ASSOC)(
    sum([x[assoc[a,1],j]  | j in POSITIONS diff V[assoc[a,1]-1]+1..V[assoc[a,1]]
      ]) +
    sum([x[assoc[a,2],j]  | j in POSITIONS diff V[assoc[a,2]-1]+1..V[assoc[a,2]]
      ])
    = 0);

constraint forall(i in notAssoc)(sum([x[i,j]|j in V[i-1]+1..V[i]])=1);

constraint forall(i in notAssoc)
            (sum([x[i,j]|j in POSITIONS diff V[i-1]+1..V[i]]) = 0);
```

To better illustrate how the instance extension works, here is an example. Given the instance in table 6.2 with 5 items and $W = 5$, we have:

$$\texttt{assoc} = [[2,3,2],[5,6,1]] \quad \text{and} \quad \texttt{notAssoc} = [1,4]. \tag{6.19}$$

Table 6.2: Example instance. The width is $W = 5$. I is the set of indices from 1 to 5. The last row is the extension of I. Items 2 and 3 form an association of type 2, while 5 and 6 form an association of type 1. Item 1 is not added to I_{ext}, because its height is larger than W, and item 4 is not added because it is a square.

index	width	height
1	5	7
2	4	5
3	5	4
4	3	3
5	4	3
6	3	4

The optimization of H works as before.

6.2 Methods

We tried to use three MiniZinc solvers: `COIN-BC` (bundled in MiniZinc), `Gurobi` [12] and `CPLEX` [13]. Preliminary tests showed that `COIN-BC` performed much worse than the other two, so we did not use it for our CP computations. We also tried to sort the input items by area, and in the rotation case we also sorted I_{ext} adjusting `assoc` and `notAssoc` accordngly. We ran all the 4 possible combinations:

- `CPLEX` without sorting;
- `CPLEX` with sorting;
- `Gurobi`, without sorting;
- `Gurobi`, with sorting.

To evaluate our results, we used the same metrics described in 2.3. All computations were done on a laptop with an Intel Core i7-8565U CPU and 8 GB of RAM. We set a timeout of 300 s.

6.3 Results

6.3.1 No rotation

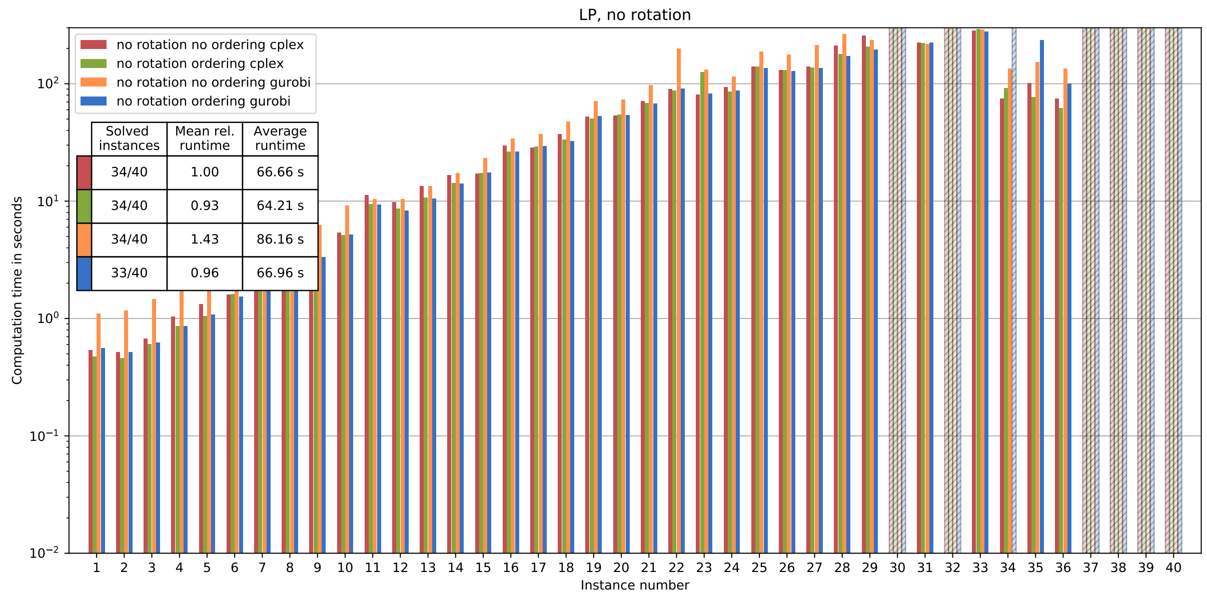
As can be seen in figure 6.5, no configuration was able so solve instances 30, 32, 37, 38, 39 and 40. We can immediately see that the best configuration is the one employing `CPLEX` an ordering. It solved 34 instances with a mean relative runtime of 0.93 and an average runtime of 64.21 s.

The worst is the one employing `Gurobi` and ordering. It was not able to solve instance 34, which the others did. The computation however is similar to the best configuration.

The two medium solutions are the two with no ordering, but the one using `CPLEX` than the one employing `Gurobi`.

6.3.2 Rotation

As can be seen in figure 6.6, no configuration was able so solve instances 28, 29, 30, 32, 33, 37, 38, 39 and 40. We can immediately see that the best configuaration is the one employing `Gurobi` and ordering. It solved 30 instances with a mean runtime of 0.83 and an average runtime of 70.50 s.

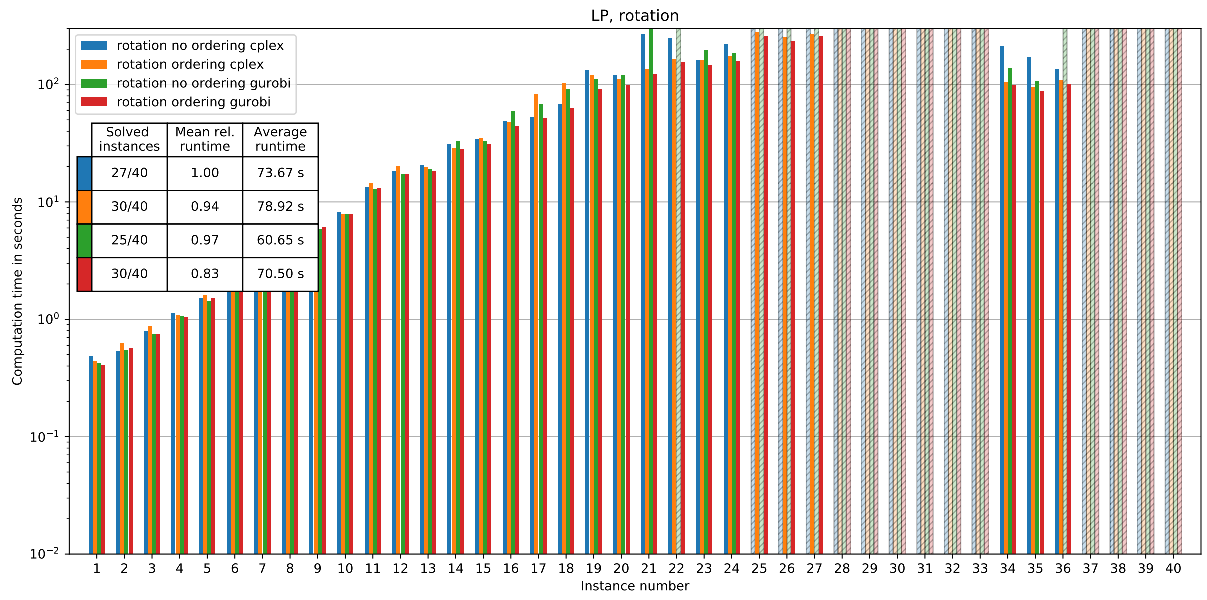


	Solved instances	Mean rel. runtime	Average runtime
🟥	34/40	1.00	66.66 s
🟩	34/40	0.93	64.21 s
🟧	34/40	1.43	86.16 s
🟦	33/40	0.96	66.96 s

Figure 6.5: Results for LP without rotation. The plot shows the computation time for each instance in each configuration. The shaded instances were not solved before the timeout. The metrics for each configuration are shown in the table.

The worst is the one employing `Gurobi` and with no ordering. It solved 5 instances less (22, 25, 26, 27, 36).

The two medium solutions are the ones employing `CPLEX`. The one with ordering solves 30 instances, like the best one, but with mean relative runtime of 0.94 instead of 0.83. The one with no ordering solves 27 instances.



	Solved instances	Mean rel. runtime	Average runtime
🟦	27/40	1.00	73.67 s
🟧	30/40	0.94	78.92 s
🟩	25/40	0.97	60.65 s
🟥	30/40	0.83	70.50 s

Figure 6.6: Results for LP with rotation. The plot shows the computation time for each instance in each configuration. The shaded instances were not solved before the timeout. The metrics for each configuration are shown in the table.

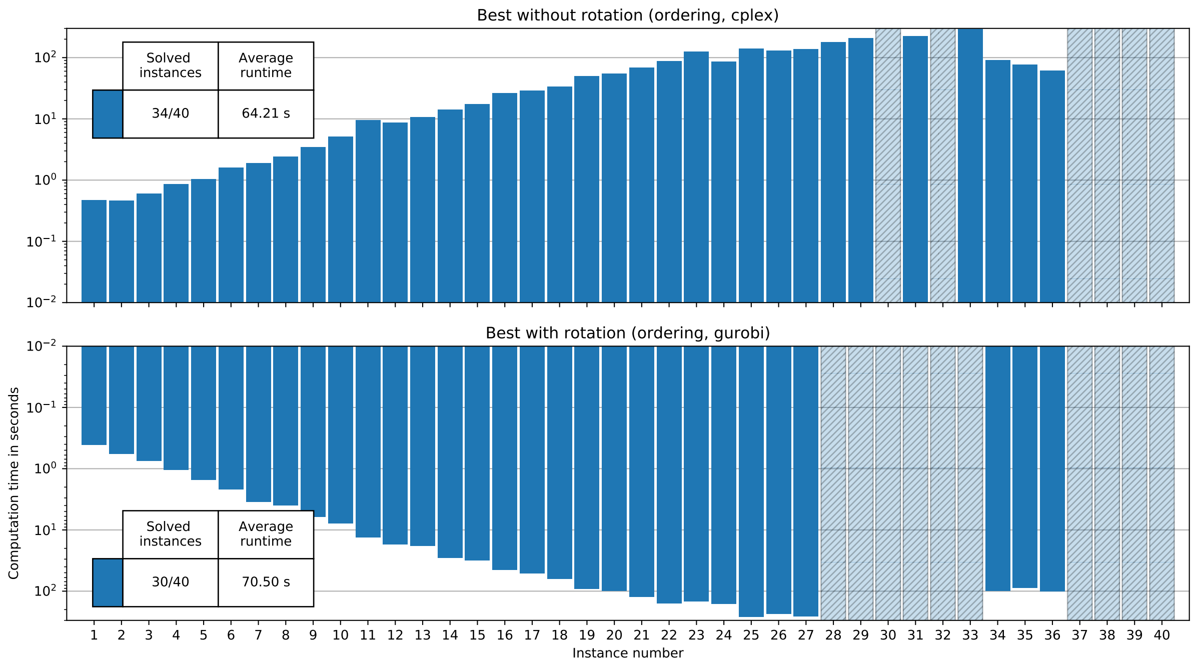


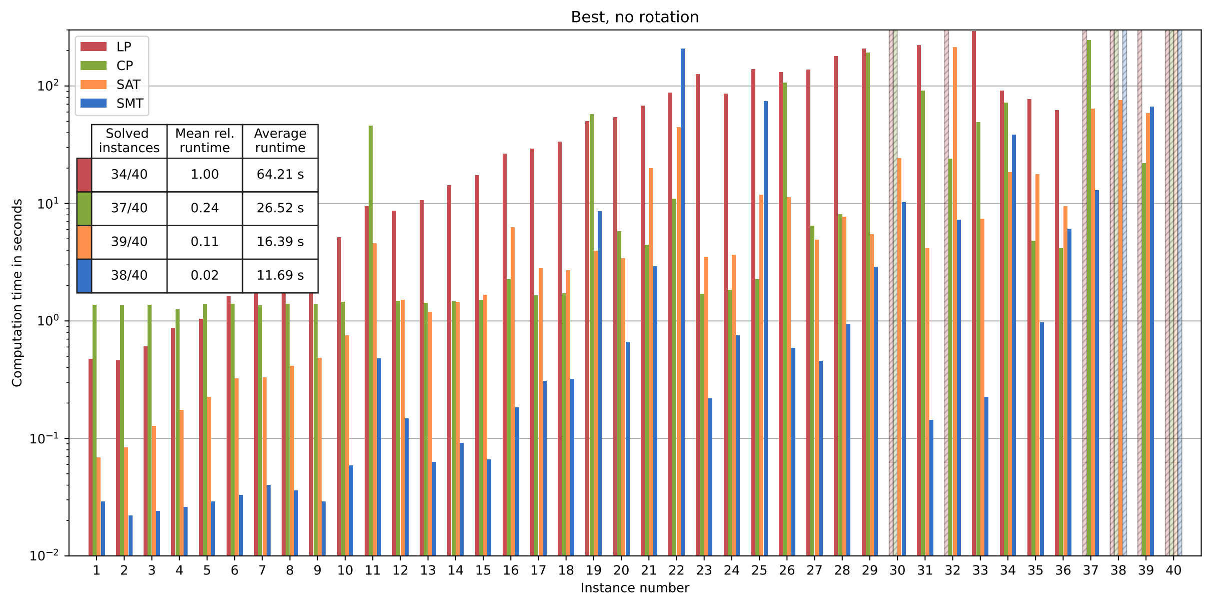
Figure 6.7: Comparison between the better configurations of no rotation version and rotation version.

6.4 Conclusions

From these experiments, we can conclude that for the version without rotation, the `CPLEX` solver works better than `Gurobi`. In this case, ordering increases the performance by a little in the all instance. For the version with rotation the `Gurobi` solver works better than `CPLEX`. Also in this case ordering in general increase a little bit the performance of the solver in the all instance. However, the performances of all configurations are quite similar.

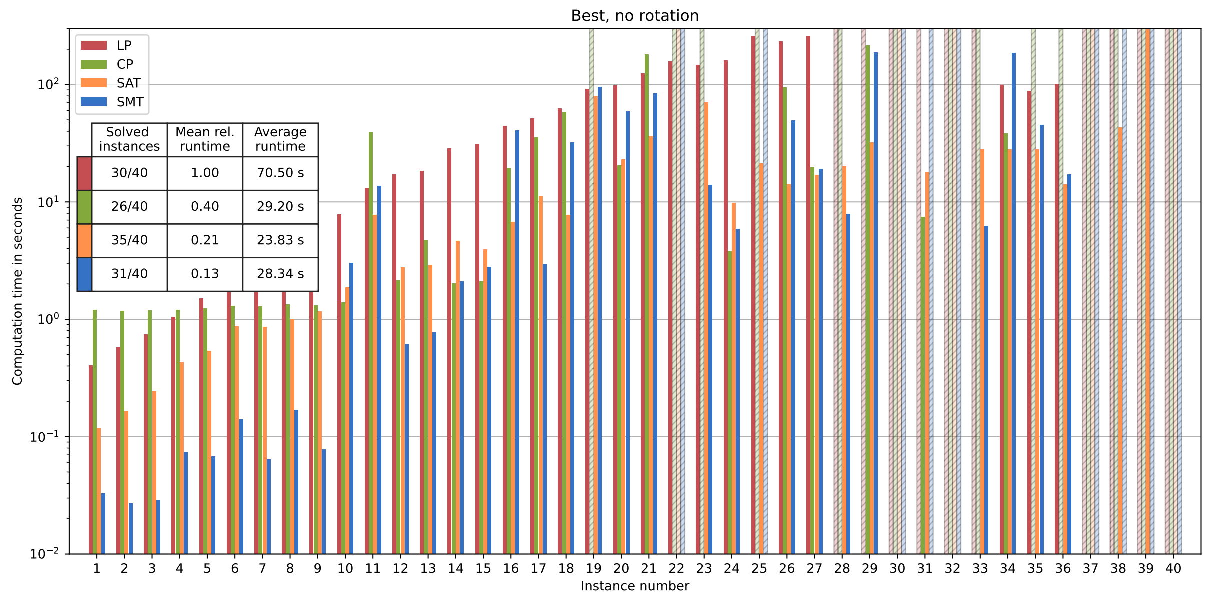
7 Final comparison

As a general overview, here we show a comparison of the best configurations for each technique.



	Solved instances	Mean rel. runtime	Average runtime
🟥	34/40	1.00	64.21 s
🟩	37/40	0.24	26.52 s
🟧	39/40	0.11	16.39 s
🟦	38/40	0.02	11.69 s

Figure 7.1: Results for all the techniques without rotation. The plot shows the computation time for each instance in each configuration. The shaded instances were not solved before the timeout. The metrics for each configuration are shown in the table.



	Solved instances	Mean rel. runtime	Average runtime
🟥	30/40	1.00	70.50 s
🟩	26/40	0.40	29.20 s
🟧	35/40	0.21	23.83 s
🟦	31/40	0.13	28.34 s

Figure 7.2: Results for all the techniques with rotation. The plot shows the computation time for each instance in each configuration. The shaded instances were not solved before the timeout. The metrics for each configuration are shown in the table.

Bibliography

[1] Zeynep Kiziltan and Roberto Amadini. Project Work, Combinatorial Decision Making and Optimization, Topic: Modelling & solving a combinatorial optimization problem, 2022.

[2] Gecode solver. https://www.gecode.org/.

[3] Laurent Perron and Vincent Furnon. Or-tools.

[4] Chuffed solver. https://github.com/chuffed/chuffed.

[5] z3-solver PyPi. https://pypi.org/project/z3-solver/.

[6] Takehide Soh, Katsumi Inoue, Naoyuki Tamura, Mutsunori Banbara, and Hidetomo Nabeshima. A sat-based method for solving the two-dimensional strip packing problem. *Fundam. Inform.*, 102:467–487, 01 2010.

[7] Takehide Soh, Katsumi Inoue, Naoyuki Tamura, Mutsunori Banbara, and Hidetomo Nabeshima. A sat-based method for solving the two-dimensional strip packing problem. https://sourceforge.net/p/potassco/mailman/attachment/1323943585.4102.431.camel@white.sevalidation.com/1/, 2008. The 15th RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion Udine, 12-13 December 2008.

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). https://smtlib.cs.uiowa.edu/, 2016.

[9] z3-solver. https://github.com/Z3Prover/z3.

[10] cvc5-solver. https://github.com/cvc5/cvc5.

[11] Nestor M. Cid-Garcia and Yasmin A. Rios-Solis. Exact solutions for the 2d-strip packing problem using the positions-and-covering methodology. *PLOS ONE*, 16(1):1–20, 01 2021.

[12] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[13] IBM ILOG Cplex. V12. 1: User's manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.