VLSI Design CP Solution

Mihail Stamenov

November 2022

Contents

1 Introduction

1.1 Idea

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. As the combinatorial decision and optimization expert, the student is assigned to design the VLSI of the circuits defining their electrical device: given a fixed-width plate and a list of rectangular circuits, decide how to place them on the plate so that the length of the final device is minimized (improving its portability). Consider two variants of the problem. In the first, each circuit must be placed in a fixed orientation with respect to the others. This means that, an n $\times$ m circuit cannot be positioned as an m $\times$ n circuit in the silicon plate. In the second case, the rotation is allowed, which means that an n $\times$ m circuit can be positioned either as it is or as m $\times$ n.

1.2 Format of the Instances

Instance Format An instance of VLSI is a text file consisting of lines of integer values. The first line gives w, which is the width of the silicon plate. The following line gives n, which is the number of necessary circuits to place inside the plate. Then n lines follow, each with xi and yi , representing the horizontal and vertical dimensions of the i-th circuit. For example, a file with the following lines:

9

5

3 3

2 4

2 8

3 9

4 12

describes an instance in which the silicon plate has the width 9, and we need to place 5 circuits, with the dimensions 3 $\times$ 3, 2 $\times$ 4, 2 $\times$ 8, 3 $\times$ 9, and 4 $\times$ 12. Figure 1 shows the graphical representation of the instance.

1.3 Output format

Where to place a circuit i can be described by the position of i in the silicon plate. The solution should indicate the length of the plate l, as well as the position of each i by its ˆxi and ˆyi , which are the coordinates of the left-bottom corner i. This could be done by for instance adding l next to w, and adding ˆxi and ˆyi next to xi and yi in the instance file. To exemplify, the solution of the instance depicted in Figure 1 could look like:

9 12

5

3 3 4 0

2 4 7 0

2 8 7 4

3 9 4 3

4 12 0 0

which says for instance that the left-bottom corner of the 3×3 circuit is at (4, 0). The solutions are represented graphically as in Figure 1. If we consider a solution which allows rotation on each line i corresponding to circuit i a boolean variable is added to indicate if we had to rotate the circuit or not:

9 12

5

3 3 4 0 True

2 4 7 0 True

2 8 7 4 False

3 9 4 3 True

4 12 0 0 False

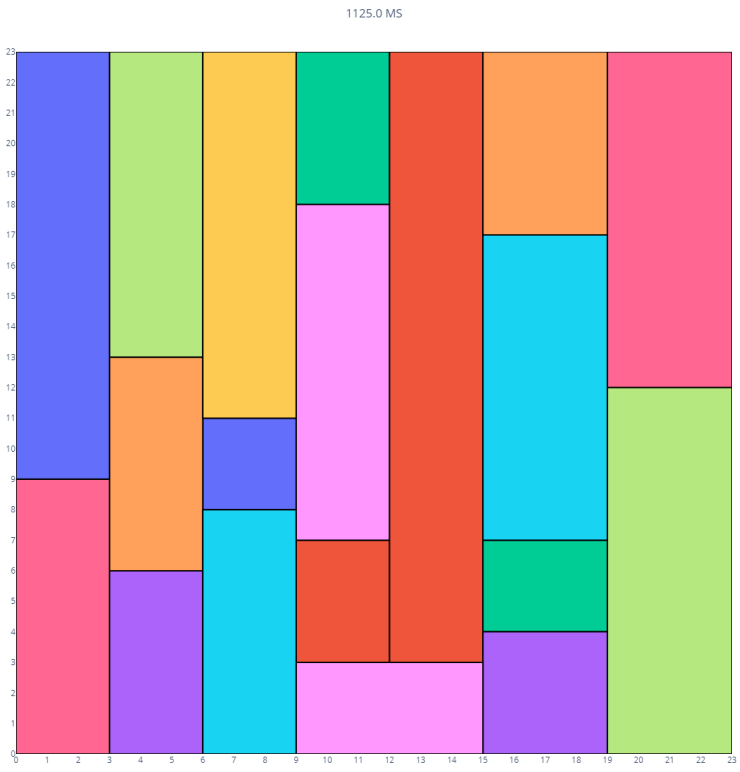

Figure 1:

2 CP

2.1 Variables

2.1.1 Types of Variables in models

The problem contains 2 parts. First we will discuss all variables needed if the circuits are not allowed to rotated. The first kind of variables are containing the information about the bottom left corner of each block. In the models they are represented as 2 arrays named respectively **blocks_x** and **blocks_y**.

The second important kind of variable is **chip_height** which is the height of the chip and **white_space** which indicates how much empty space would we have if we subtract the area of the blocks from the area of the chip which is **chip_width*chip_height**. In both formulations of the problem we have to minimize **chip_height** or **white_space**.

There is 3-rd type variables which are transitional variables and are used to break symmetries. They are an array with 2*n_blocks length and the first n_blocks of the array are the x-s of the left right corner and the last n_blocks are the y-s.

2.1.2 Bounds

The optimal solution has height of the chip less or equal then every other solution height. Since we want to minimize the height of the chip we need to find proper bounds for the variable. The upper bound is named **chip_max_height** and it can be defined like so :

$$min\_blocks\_on\_row = \left\lfloor \frac{chip\_width}{max\_block\_width} \right\rfloor$$

$$chip\_max\_height = max\_block\_height * \begin{cases} \frac{n\_blocks}{min\_blocks\_on\_row} & \text{if } n\_blocks \equiv 0 \pmod{min\_blocks\_on\_row} \\ \frac{n\_blocks}{min\_blocks\_on\_row} + 1 & \text{else.} \end{cases}$$

Let us imagine that all parts have equal width and height of the biggest possible height of all elements and biggest width and find how many we can put on 1 line. Then we calculate the number of elements in column. and we take the result. We are sure that if we have a normal problem and construct the ordering via the logic from previous sentence and then on the place of each big imaginary block we can put a real one. this will be an ordering of our circuits which is not optimal but we know that the optimal solution should have at most this size or less.

Now we will find a lower bound for **chip_height**. We can consider the height of the highest block as a lower bound. On the other hand if there is a perfect placement of the circuits than can fit in a rectangle with sizes **chip_width** and **chip_height**. Then it will be true that:

$$\sum_{i=1}^{n\_blocks} heights[i] * widths[i] \leq chip\_height * chip\_width$$

$$\frac{\sum_{i=1}^{n\_blocks} heights[i] * widths[i]}{chip\_width} \leq chip\_height$$

Since $\frac{\sum_{i=1}^{n\_blocks} heights[i] * widths[i]}{chip\_width} \leq chip\_height$

and $\max(heights) \leq chip\_height$, then we can say that the lower bound would be:

$$\max(\frac{\sum_{i=1}^{n\_blocks} heights[i] * widths[i]}{chip\_width}, \max(heights))$$

2.2 Objective function

The main idea was that the objective function should be the height of the chip. On the other hand if we imagine the chip and all pieces that should be in it then the area of the chip must not be more

then 2 times the sum of the area of each block.

$$height * width \leq 2 * \sum_{i=1}^{n\_blocks} heights[i] * widths[i]$$

$$height * width \leq 2 * \sum_{i=1}^{n\_blocks} heights[i] * widths[i]$$

$$height * width \leq 2 * area\_of\_blocks$$

$$height * width - area\_of\_blocks \leq area\_of\_blocks$$

$$empty_space \leq area\_of\_blocks$$

So if we can reduce the empty_space we will also reduce the height because empty_space is linear function of height because the width and the sizes of blocks are fixed values. In the end I tried to create function which depends both on the white space and the height but still since the height is our goal to minimise it had to be prioritized and that is possible by multiplying it and the number which I used is area_of_blocks since it will always be bigger than the whit_space. We need it to be bigger because if it is not there will be no priority for the height.

2.3 Constraints

2.3.1 Limit constraints

First of all all blocks must be in the limits of the chip and this is verified by taking the greatest x coordinate from the set of coordinates of upper right corner of the circuits. If we are sure that the upper-right corner of circuit is in the chip than the other right corners of the blocks will also be inside the chip. We do the same thing for the height but verifying that the greatest y coordinate of all upper corners is less or equal the chip max height.

2.3.2 Overlapping prevention

We want to avoid rectangles to overlap with one another and we can formalize this with the constraint

$$diffn(blocks\_x, blocks\_y, widths, heights)$$

for the no rotation case.

2.3.3 Schedule problem representation

Our problem can be seen as a bi-dimensional scheduling problem. For the x-axis, the board's height represents the resource bound, the x coordinate blocks_x represents the starting time, the width of a circuit x represents the duration, while the height of a circuit represents the required resource. On the other hand the chip's width can be the resource bound for operations represented with our circuits. Their heights are the time consumption of the operation, their widths are the resource and their y coordinate is the start time of the task.

2.3.4 Symmetry breaking

There are many possible symmetric solutions to this particular problem. We shall avoid them in order to reduce the search space, thus improving the overall performances.

* Symmetry across the x-axis

* Symmetry across the y-axis

* Symmetry around the center of the board (180° rotation)

For the problem with rotation allowed we have one more symmetry:

* Square circuits do not need to rotate.

Since we have to prevent the three symmetries we have noted before we need some kind of ordering that can prevent them. For the symmetry across the x we need only x coordinates so we can order only them. Moreover the same thing can be said for the y access symmetry. The 180° rotation can not be resolved if we are not using both x and y coordinates. For this reason we need an array with all 2*n_blocks which first n_blocks contains the x coordinates and then the y coordinates. For this array of transitional variables we have 3 types of constraints who order a solution and it symmetry alternative in lexicographic order. The first type of constraint prevents x-axis symmetry by creating the alternate solution in which the y are the same but the x now is $chip\_width - blocks\_x_i - widths_i$ and if there is rotation it is $chip\_width - blocks\_x_i - rotation\_widths_i$. For the y-axis it is pretty much the same but in this case we preserve the x coordinates and change the y coordinates in the mirror solution to be $chip\_height - blocks\_y_i - heights_i$ and for rotation case they are changed to $chip\_height - blocks\_y_i - rotation\_heights_i$ this way we prevent rotations and mirroring symmetry by **x** or **y** axis. In the third kind of constraint in the symmetry solution we change both x and y coordinates to $chip\_width - blocks\_x_i - widths_i$ and $chip\_height - blocks\_y_i - heights_i$ respectively. If there is rotation the new coordinates are $chip\_width - blocks\_x_i - rotation\_widths_i$ and $chip\_height - blocks\_y_i - rotation\_heights_i$

2.3.5 Full bottom row

A different kind of constraint was added during the production process because in many cases the solutions tended to leave empty spaces on the first row and the prevention of this problem was to make constraint who demands that the first row must be filled. This turned out to be very helpful and it also was breaking many symmetries. In fact the constraint sums the widths of all circuits which are placed on the bottom row and constraints the sum to be equal to the width of the chip.

2.4 Rotation

2.4.1 Variables and bounds

For the second formulation of the problem where the blocks can be rotated an additional array of variables (**rotated**) is added indicating if circuit i is rotated or not. Since the width and the height of each block can change via rotation 2 new arrays of variables were added to represent each block's dimensions regarding the fact if it is rotated or not.

The bounds for the second formulation are similar with the only difference that in this case we do not know which side will be width and which will be height. Thus, we take the biggest number between all dimensions and use it in our formulas:

$$max\_size = \max(\max(widths), \max(heights))$$

$$min\_blocks\_on\_row = \left\lfloor \frac{chip\_width}{max\_size} \right\rfloor$$

$$chip\_max\_height = max\_size * \begin{cases} \frac{n\_blocks}{min\_blocks\_on\_row} & \text{if } n\_blocks \equiv 0 \pmod{min\_blocks\_on\_row} \\ \frac{n\_blocks}{min\_blocks\_on\_row} + 1 & \text{else.} \end{cases}$$

We also can think of upper bounds for the coordinates which point the bottom left corner of the circuits. In fact since we have bounds for width and height of the chip we can say that the x coordinates will be from 0 to:

$$chip\_width - \min(widths)$$

and for y coordinates the upper bound is:

$$chip\_max\_height - min(heights)$$

The upper bounds in the case of rotation are similar we subtract the minimum size of circuit: $chip\_width - min\_size$ and $chip\_width - min\_size$ where $in\_size = \min(\min(widths), min(heights))$

2.4.2 Constrains

We use similar constraints with the small difference that in this case we do not use the input arrays of widths and heights, but the one that are configured in regards to the array of rotations - $rotation\_widths, rotation\_heights$. So again we have Limit constraints and Overlapping prevention which will look like:

$$diffn(blocks\_x, blocks\_y, rotation\_widths, rotation\_heights)$$

for the case with rotations. Moreover a scheduling problem representation was used For the rotation case it is the same with the exception that we used with the difference that now we apply the constraints on the $rotation\_widths, rotation\_heights$ and not the input heights and widths.

We use the 3 different constraints for symmetry breaking and the transformed version of full bottom row. In this case we were summing the **rotation_widths** of each circuit. Also if the circuit is square we set the corresponding rotation variable to false and this prevents the block to rotate.

2.5 Validation

Since there are many different solvers and search heuristics the 4 most important things to choose are the solver, search heuristics for values, search heuristic for domain and restart. All the times in the conducted tests are calculated in milliseconds.

We are going to make experiments with 2 solvers - Gecode and Chuffed

2.5.1 No Rotation

Firstly, we will consider different solvers and strategies for the formulation of the problem without rotation. For Gecode the different configurations that were tested were:

* input_order indomain_min restart_none
* first_fail indomain_min restart_none
* first_fail indomain_random restart_linear
* dom_w_deg indomain_random restart_linear
* dom_w_deg indomain_random restart_luby
* (sorted by area) indomain_min no_restart

Every module was tested on the first 20 instances and there was time limit of 1 minute to find the best solution. The results were calculated in milliseconds. However they are not satisfying.

instance	input_order indomain_min restart_none	first_fail indomain_min restart_none
1	250.0	250.0
2	250.0	250.0
3	260.0	260.0
4	250.0	300.0
5	300.0	1230.0
6	270.0	430.0
7	10520.0	1310.0
8	1390.0	31100.0
9	31510.0	810.0
10	no solution	no solution
11	no solution	no solution
12	no solution	60030.0
13	no solution	no solution
14	no solution	no solution
15	no solution	no solution
16	no solution	no solution
17	no solution	no solution
18	no solution	no solution
19	no solution	no solution
20	no solution	no solution

instance	first_fail indomain_random restart_linear	dom_w_deg indomain_random restart_linear
1	260.0	260.0
2	260.0	260.0
3	270.0	260.0
4	340.0	340.0
5	400.0	400.0
6	770.0	910.0
7	980.0	560.0
8	2240.0	1470.0
9	2740.0	3190.0
10	5520.0	4550.0
11	60130.0	60140.0
12	60120.0	no solution
13	60120.0	60140.0
14	60030.0	no solution
15	no solution	60020.0
16	no solution	no solution
17	no solution	no solution
18	no solution	no solution
19	no solution	no solution
20	no solution	no solution

instance	dom_w_deg indomain_random restart_luby	(sorted by area) indomain_min no_restart
1	250.0	260.0
2	260.0	260.0
3	270.0	260.0
4	300.0	270.0
5	390.0	330.0
6	530.0	590.0
7	480.0	410.0
8	760.0	1750.0
9	1080.0	540.0
10	870.0	60020.0
11	60030.0	60020.0
12	18500.0	no solution
13	32710.0	60030.0
14	37680.0	60020.0
15	60120.0	no solution
16	60030.0	no solution
17	no solution	60030.0
18	60020.0	no solution
19	no solution	no solution
20	no solution	no solution

The results show that in many cases the model could not find any solution with in a minute.

Since there is no indomain_random and dom_w_deg in the Chuffed solver we will make the same experiments but we will consider this heuristics.

* input_order indomain_min restart_none

* first_fail indomain_min restart_luby

* first_fail indomain_min restart_linear

* (sorted by area) indomain_order restart_none

We conducted the same tests as before and for every model we found out that they solve all 20 instances:

instance	input_order indomain_min restart_none	first_fail indomain_min restart_luby
1	250.0	250.0
2	250.0	250.0
3	260.0	250.0
4	270.0	260.0
5	270.0	260.0
6	280.0	290.0
7	280.0	280.0
8	280.0	290.0
9	280.0	290.0
10	310.0	320.0
11	460.0	630.0
12	610.0	370.0
13	360.0	410.0
14	740.0	720.0
15	660.0	520.0
16	2420.0	450.0
17	2130.0	580.0
18	17100.0	1440.0
19	6220.0	3780.0
20	3180.0	1960.0

instace	first_fail indomain_min restart_linear	(sorted by area) indomain_order restart_none
1	250.0	260.0
2	260.0	250.0
3	260.0	260.0
4	260.0	260.0
5	270.0	270.0
6	280.0	280.0
7	280.0	280.0
8	280.0	290.0
9	290.0	280.0
10	320.0	300.0
11	1100.0	460.0
12	450.0	610.0
13	390.0	360.0
14	730.0	720.0
15	610.0	670.0
16	450.0	2410.0
17	570.0	2130.0
18	1620.0	16890.0
19	2400.0	6250.0
20	2570.0	3160.0

The results are much better but we will need to run the models on the full data set so that we can choose the best one. the results were quite similar but the models first_fail indomain_min restart_luby and (sorted by area) indomain_order restart_none had the best results which can be find in file results.xlsx in folder CP. The average times for solving a problem for each search strategy were 45164, 38540, 55783,5, 45051,75 and the first strategy was not able to find the optimal solution for 5 instances, the second one did not find the optimal soluton of 3 instances, the third strategy did not find the optimal solution for 6 instances and the forth case was not able to find it for 5 instances. Another interesting thing is that the intersection of the 2 sets of instances that did not had optimal solution in 5 minutes is the smallest one compared to the other intersections and the 2 models do not have anything in common. This resulted in a fifth model:

* (sorted by area) indomain_order restart_luby Chuffed

His average time per instance was 20427,5 and was not able to find the optimal solution only for instance 40.

2.5.2 With Rotation

Since the tests from previous section showed that Chuffed solver works better for the problem and the models with restart work better then the others I have considered only models with this specifications

instance	(sorted by area) indomain_order restart_none	(sorted by area) indomain_order restart_luby	(sorted by area) indomain_order restart_linear
1	260	260	250
2	260	260	260
3	260	270	270
4	270	280	280
5	290	290	300
6	310	300	300
7	320	310	330
8	340	360	340
9	340	320	330
10	1070	510	510
11	2880	1540	5420
12	5050	980	6120
13	1160	750	640
14	840	870	1040
15	1260	1050	650
16	268850	16180	840
17	6480	1820	3800
18	108000	17110	16650
19	189170	4790	300230
20	218130	63510	2190

The average times per instance are 40277 5588 17037,5, which means that the second one that uses luby restart is the best one. The average time per instance for all 40 instances is 61024,25 ms and the results of all experiments conducted on all 40 instances can be found in the file results.xlsx in the SAT directory.

3 SAT

The second solution of the problem was encoded via SAT which is given propositional formulas and for this reason the problem's variables had to be represented as a boolean variables and the constraints had to be transformed in formulas.

3.1 Variables

3.1.1 Types of Variables in models

The model uses 2 variables for the coordinates - **blocksX** and **blocksY**. Both variables are in fact arrays with length **n**. Each item in this arrays represents the **x** and **y** coordinates respectively. In other words the **i-th** item from **blocksX** represents the **x** coordinate of the **i-th** circuit and the **j-th** element of **blocksY** is the **y** coordinate of circuit **j**. In fact each element of **blocksX** and **blocksY** is array of boolean variables. The length of each array in **blocksX** is the width of the chip (**w**). The **i-th** array from **blocksX** should contain **width[i]** consecutive truths and **w** - **width[i]** false. The idea of **blocksY** is similar - it contains the footprints of each circuit on the **y** axis. The length of each array in **blocksY** is **maxHeight** which is the upper bound for the height of the chip

The length of the chip is represented with the array of bolleans **l**. The length of l is **maxHeight** − **minHeight** + 1 where **minHeight** represents the minimum height of the chip. Only one of the variables in **l** should be true and if the value in **i-th** cell is true means that the length of the chip is $i + minHeight$ and the indexes of the array start from 0 to **maxHeight** - **minHeight**.

3.1.2 Bounds

We already have mentioned what the bounds of the variables of the model. Moreover we use the same bounds as in the **CP** representation:

$$minBlocksOnRow = \left\lfloor \frac{w}{maxBlockWidth} \right\rfloor$$

$$maxHeight = maxBlockHeight * \begin{cases} \frac{n}{minBlocksOnRow} & \text{if } n \equiv 0 \pmod{minBlocksOnRow} \\ \frac{n}{minBlocksOnRow} + 1 & \text{else.} \end{cases}$$

$$minHeight = \max(\frac{\sum_{i=1}^{n} heights[i] * widths[i]}{w}, \max(heights))$$

3.2 Objective function

The objective function is the height of the chip. Since I could not find a proper formula which to represent the minimization of **chip length** I created a a loop which searches for solution and if one is found a group of new constraints are added restricting the model to use heights which are greater or equal to the one found in the solution. This way if there is a better solution it can be found. The problem is that this way we can search the for solution $\mathbf{maxHeight} - \mathbf{minHeight} + 1$ times.

3.3 Constraints

3.3.1 Limit constraints

This is the first kind of constraints used in the model. In fact this constraint creates every value which can an element of **blocksX** or **blocksY** have and we say that exactly one is possible for the element. For instance if the chip width is 4 and circuit **i** has width 2 then all possible values for **blocksX[i]** are:

$$[1, 1, 0, 0]$$

$$[0, 1, 1, 0]$$

$$[0, 0, 1, 1]$$

For this reasone we add the constraint:

$$Or(And(blocksX[i][0], blocksX[i][1], Not(blocksX[i][2]), Not(blocksX[i][3])),$$

$$And(Not(blocksX[i][0]), blocksX[i][1], blocksX[i][2], Not(blocksX[i][3]))$$

$$And(Not(blocksX[i][0]), Not(blocksX[i][1]), blocksX[i][2], blocksX[i][3]))$$

We have the same constraints for the **blocksY** and with this type of constraints we preserve the circuits inside the chip and we say what are the possible footprints of each circuit on the **x** and **y** axises.

3.3.2 Overlapping prevention

If we know that for circuit **k** the values of **blocksX[k][i] blocksY[k][j]** are true, then this means that block **k** covers the square with coordinates **(i, j)**. We know that each square from the chip can be covered with only one circuit so we add constraint for each field that says that the at most one value

from all circuits can cover it. The constraint for square **(i, j)** will look like:

$$atMostOne(And(blocksX[0][i], blocksY[0][j]), And(blocksX[1][i], blocksY[1][j])..., And(blocksX[n-1][i], blocksY[n-1][j]))$$

3.3.3 Symmetry breaking

The next constraints are preventing the 3 symmetries from the **CP** representation. In fact we create an lexicographic order. First we have to compare the 2 bollean arrays and in fact the one that has 1 (true) earlier is smaller then the other. Thus, a function **boolGreaterEq** was created that returns true if $firstArgument \geq secondArgument$. So the function **lessEq** was created to answer if the first array is less or equal to the second one. The array which has earliest truth is considered smaller and is implemented by stating that the first element of the first array is greater or equal to the first element of the second array and for every **i** in [1, ... , len(array)-1] we have that if

$$firstArray[0] == secondArray[0] \wedge ... \wedge firstArray[i-1] == secondArray[i-1]$$

then $firstArray[i] \geq secondArray[i]$ and thus we have similar function **lexLessEq** which compares array of arrays and is implemented based on the same logic as **lessEq** but using **lessEq** instead of **boolGreaterEq** to compare the items.

And now that we have the function to order an array of arrays we use it to create the 3 constraints. First to create mirror solution by the vertical axes we need to reverse each element of **blocksX** and let us call this transformation **elementRevers**. So we add the constraints

$$lexLessEq(blocksX + blocksY, elementRevers(blocksX) + blocksY)$$

$$lexLessEq(blocksX + blocksY, blocksX + elementRevers(blocksY))$$

$$lexLessEq(blocksX + blocksY, elementRevers(blocksX) + elementRevers(blocksY))$$

to represent the constraint for horizontal, vertical and rotation symmetry.

3.3.4 Length of chip

There are two constraints for the length of the chip. The first one is the one that states that for every possible height the length of the chip will be the height if there is at east one element on that height and there is no elements higher.

The second constraint is that exactly one of all variables in **l** is true since the length of the chip is only one value

3.4 Rotation

3.4.1 Variables and bounds

An additional array of **n** variables was added to the model describing which circuit is rotated and which not.

3.4.2 Constrains

The constraints we used in the no rotation model apply with the exception of the Limit constraints. If the r[i] is true than the circuit is rotated and this means that we need all cases of how the height of the block can be placed on the **x** axis and all cases how the width of the block can be placed on the **y** axis. On the other hand if r[i] is false we need to make the conjuction of the disjunction of all possible ways to place the width on the **x** axis and the disjunction of all possible ways to place the height of the block on the **y** axis. So the 2 implications were added for each block.

3.5 Validation

At the beginning the 4 different encodings of the constraints atMostOne and exactlyOne. The encodings which were considered arethe one studied on lectures:

* Pairwise Encoding
* Sequential Encoding
* Bitwise Encoding
* Heule Encoding

The implementations of the different encodings were taken from the code from lab sessions.

3.5.1 No Rotation

At first the model was tested on first 20 instances using each of the encodings without time limit because the timeout function of the solver seems to not work properly.

instance	Pairwise Encoding	Sequential Encoding	Bitwise Encoding	Heule Encoding
1	32,9	149,6	156,6	154,1
2	40,9	138,1	131,2	145,7
3	70,3	201,5	184	169,4
4	195,5	357,2	334,2	373,4
5	270,8	409	436,9	349,5
6	938,2	1434,5	1431	1111
7	467,4	775,6	816,1	642,6
8	1371,1	2218,5	1551,4	1577,3
9	1040	1645,9	1987	1375,2
10	6386,4	4172,7	7963,1	4282,4
11	158753,6	105558,9	111174,3	51244,7
12	21738	9663,5	19543,2	8651,7
13	8511,9	4163	8714,1	3981
14	38016,9	33275,6	47321,3	24803,8
15	35216,1	27761,6	56383,4	44158,8
16	503367,1	501491,5	671928,7	480615,6
17	182169,1	335209,6	265350,5	268347,7
18	319742,6	402947,5	364834,8	303956
19	421365,9	661928,3	1114260,5	644251,1
20	401815,4	393811,6	810716,8	467006,7
average time	105075,505	124365,685	174260,955	115359,885

The model works very slow. Only the first 15 instance have solution in the time limit and the main reason for this is that the model has too many variables since we create the array of maxHeight elements for each circuit and in most cases the maxHeight is much bigger than the optimal value and it depends on the sizes of the blocks. From the results it seems that the pairwise encoding works best which is strange since it adds the biggest number of constraints. However this is the only encoding that does not add more variables and thus I believe it shows best results. A time limit was added in the while loop that searches for solution but it will stop the program only if more than 300 seconds are elapsed and the solver has returned a result which means that the solver can not be stopped while searching for solution. For instances after 15 the solver could not find even one solution for 300 seconds. Thus, the results that are in the output directory are the first solution the solver found and the times for search are:

instance	Pairwise Encoding
1	36.4
2	46.9
3	71.3
4	208.5
5	268.9
6	923.3
7	486.8
8	1268.5
9	1008.5
10	6331.1
11	154627.6
12	20375.7
13	8415.6
14	36944.0
15	34513.0
16	499947.0
17	178958.4
18	314263.5
19	418645.5
20	400050.0

instance	Pairwise Encoding
21	453274.4
22	369998.9
23	392072.1
24	319312.2
25	563887.6
26	458235.6
27	496107.1
28	318089.6
29	424711.6
30	407186.6
31	475293.0
32	306257.2
33	324654.4
34	389611.5
35	565384.7
36	386209.0
37	330389.9
38	498382.7
39	399246.3
40	-

3.5.2 With Rotation

The results with the rotation model are not good either and it could solve only the first 10 instances and the 13-th one.

instance	Pairwise Encoding
1	32.9
2	111.1
3	167.9
4	365.1
5	915.3
6	1023.6
7	1890.9
8	5151.0
9	5645.4
10	26471.5
11	-
12	-
13	111710.5
14	-
15	-
16	-
17	-
18	-
19	-
20	-

4 LP

The third solution of the problem was encoded via LP which is given linear equalities and inequalities. for this reason the constraints had to be transformed so that they meaning remains but in form of inequalities.

4.1 Variables

4.1.1 Types of Variables in models

The model uses 2 variables for the coordinates - **blocksX** and **blocksY**. Both variables are in fact arrays with length **n**. Each item in this arrays represents the **x** and **y** coordinates of a circuit respectively. In other words the **i-th** item from **blocksX** represents the **x** coordinate of the **i-th** circuit and the **j-th** element of **blocksY** is the **y** coordinate of circuit **j**.

There is a variable that **l** representing the length of the chip.

There are $n*(n-1)/2$ arrays of 4 boolean variables named **b[i][j][k]** where **i** and **j** are the numbers of the 2 circuits and **b[i][j][0]**, **b[i][j][1]**, **b[i][j][2]**, **b[i][j][3]** are used to indicate the position between each 2 circuits.

4.1.2 Bounds

We use the same bounds for as in the **CP** representation:

$$minBlocksOnRow = \left\lfloor \frac{w}{maxBlockWidth} \right\rfloor$$

$$maxHeight = maxBlockHeight * \begin{cases} \frac{n}{minBlocksOnRow} & \text{if } n \equiv 0 \pmod{minBlocksOnRow} \\ \frac{n}{minBlocksOnRow} + 1 & \text{else.} \end{cases}$$

$$minHeight = \max(\frac{\sum_{i=1}^{n} heights[i] * widths[i]}{w}, \max(heights))$$

l can have values from **minHeight** to **maxHeight**. The values of each variable from **blocksX** can be constrained from **0** to **w - widths[i]** and for the variables of **blocksY** from **0** to **maxHeight - heights[i]**. The variables from **b** are boolean so the only values they can have are 0 and 1.

4.2 Objective function

The objective function is to minimise l.

4.3 Constraints

4.3.1 Limit constraints

This is the first kind of constraints used in the model. which says that the sum of each circuit's x-coordinate with the width of the block should be less or equal to the width of the whole chip. We have the same constraint for the y-coordinate and the height of each circuit to be less or equal then the length of the chip. For every i we have:

$$blocksX[i] + widths[i] <= w$$

$$blocksY[i] + heights[i] <= l$$

4.3.2 Overlapping prevention

Every 2 circuits are constraint with 5 constraints. Since we want to prevent the overlapping so we make constraint for each possible position of the 2 blocks. Let us say that that we want to constrain circuit **i** and **j**, and we think of circuit **i** as fixed then there are 4 cases for block **j**. It can be on the left, right, up or down and these are the first four constraints. In fact at most 2 of this constraints can be true and at least 1 should be true. Thus, to each of the first 4 constraint we add big number(either the width of the chip or maxHeight) multiplied by one of **b[i][j][k]** which is used to make the constraint true if it is necessary but the condition is not true. In the and we say that the sum of all 4 b[i][j][k] should be at most 3 and at least 2. The constraint that the sum of all 4 b for circuit **i** and **j** is greater or equal to 2 was dropped since it did not improve the time.

4.3.3 Symmetry breaking

A function which creates lexicographic ordering was implemented but it did not improved the performance. However the function logic was that we have two arrays of coordinates, the first one is the solution and the second is the mirror solution. For each i we have:

$$p[i] <= q[i] + \sum_{j=1}^{i-1} c_j * bigNumber + c[i]$$

$$p[i] == q[i] + c[i]$$

I other words if $p[i] == p[j]$ then $c[j]$ is 0 and if all cj for j¡i are 0 means that all elements before that are equal so the current one will multiply bigNumber with 0 making the constraint important.

4.4 Rotation

4.4.1 Variables and bounds

An additional array **r** of **n** variables was added to the model describing which circuit is rotated and which not.

4.4.2 Constrains

The constraints had to be changed a bit since they depend on the variable of rotatoin. The constraints for the x-axis were change from using $widths[i]$ to using:

$$((1 - r[i]) * widths[i] + r[i] * heights[i])$$

and i the y-axis constraints $heights[i]$ was changed to:

$$(r[i] * widths[i] + (1 - r[i]) * heights[i])$$

Additionally a constraint that sets r[i] to 0 if it is square was added.

4.5 Validation

There were 2 solvers that were considered:

* GUROBI_CMD

* PULP_CBC_CMD

4.5.1 No Rotation

Here are the results from running both solvers on all instances:

instance	GUROBI_CMD	PULP_CBC_CMD
1	33.7 ms	30.4 ms
2	33.9 ms	68.8 ms
3	30.9 ms	99.3 ms
4	34.4 ms	616.5 ms
5	45.9 ms	1076.3 ms
6	47.4 ms	17213.1 ms
7	59.8 ms	6753.2 ms
8	65.3 ms	14545.3 ms
9	46.4 ms	43669.3 ms
10	221.4 ms	36280.9 ms
11	233.4 ms	not solved
12	518.2 ms	51088.8 ms
13	671.8 ms	not solved
14	585.5 ms	not solved
15	871.9 ms	not solved
16	11779.1 ms	not solved
17	1055.9 ms	not solved
18	3421.7 ms	not solved
19	70831.1 ms	not solved
20	6271.8 ms	not solved

instance	GUROBI_CMD	PULP_CBC_CMD
21	156740.0 ms	not solved
22	not solved	not solved
23	3971.5 ms	not solved
24	1355.1 ms	not solved
25	not solved	not solved
26	48864.8 ms	not solved
27	20277.7 ms	not solved
28	17125.6 ms	not solved
29	18478.1 ms	not solved
30	not solved	not solved
31	1328.1 ms	not solved
32	not solved	not solved
33	3223.9 ms	not solved
34	53554.9 ms	not solved
35	not solved	not solved
36	17068.0 ms	not solved
37	not solved	not solved
38	not solved	not solved
39	not solved	not solved
40	not solved	not solved

Clearly the GUROBI_CMD model works better since it solved more instances.

4.5.2 With Rotation

The results with the rotation model are a bit worse and since GUROBI_CMD performed very well it in the previous task it was the only considered solver.

1	31.9 ms	21	300623.7 ms
2	29.9 ms	22	300347.5 ms
3	27.8 ms	23	28571.8 ms
4	33.9 ms	24	7006.6 ms
5	176.6 ms	25	300408.3 ms
6	215.5 ms	26	300311.5 ms
7	128.2 ms	27	18935.4 ms
8	91.3 ms	28	194521.6 ms
9	425.5 ms	29	11752.3 ms
10	774.6 ms	30	301194.9 ms
11	2319.8 ms	31	9329.4 ms
12	3259.4 ms	32	300441.1 ms
13	1549.2 ms	33	42086.4 ms
14	4530.3 ms	34	215805.8 ms
15	2654.4 ms	35	12501.2 ms
16	124068.3 ms	36	30749.6 ms
17	3264.4 ms	37	not solved
18	17392.0 ms	38	not solved
19	not solved	39	not solved
20	257032.1 ms	40	not solved