ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CDMO Module - 01

(Combinatorial Decision Making and Optimization)

Project Report (SAT)

Group Members

Davide Mercanti (davide.mercanti@studio.unibo.it)

Faisal Ramzan (Faisal.ramzan@studio.unibo.it)

Index

Project Work - SAT

In this case the model consists of a Python file (*models.py*) in which boolean variables and constraints on those variables are expressed as classes of the z3 library. The file also contains some extra code useful to interface the z3 library with the other components of this software (the one that displays solutions, for example).

Variables, problem constraints and objective function (1)

In the following formulas we will use:

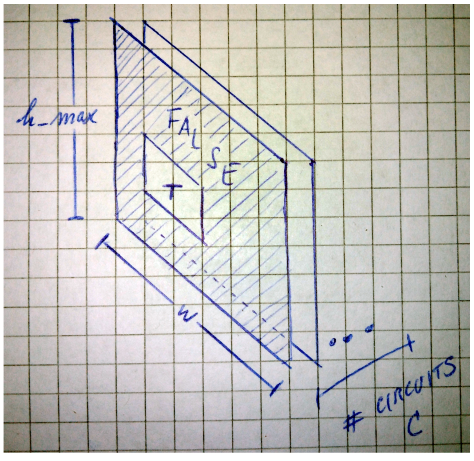
$xDims$	$dimensions\,list$
$yDims$	$y-dimensions\,list$
N	$n.of\,circuits$
W	$width$
G_c	$c^{th}\,component\,of\,G(c \in C)$
G_{cxy}	$x, y\,position\,of\,G_c$
C	$set\,of\,circuits\,\{1 \ldots N\}$

Given that we are dealing with satisfiability, the model considered the height H as fixed as optimal, thus is computed as

$$H = \left\lceil \frac{\sum_{i=1}^{N} (xDims_i * yDims_i)}{W} \right\rceil$$

(and then H could be incremented if the model fails to find a solution).
The problem variables are W*H*N booleans. For each circuit "c" a W*H grid G_c is built up by the model. In these grids, every point (variable) is true iff. it corresponds to the border or the inner part of c.

Initially, the problem constraints were expressed as follows (for typographical reasons "and" is expressed as a product, "or" as a coproduct). For each G_c, it is imposed that for every viable position to place *c* x_{cand}, y_{cand}, if the circuit is placed in that position, than the sorrounding variables are False, as are False the variables in the same position belonging to the other layers $G_{c'}, \forall c' \neq c; c' \in C$(to enforce non-overlapping).



$$\prod_{c\in C} \coprod_{\substack{0\le x_{cand}\le W-xDims_c \\ 0\le y_{cand}\le H-yDims_c}} \left(\prod_{\substack{x_{cand}\le x\le x_{cand}+xDims_c \\ y_{cand}\le y\le y_{cand}+yDims_c}} G_{cxy} \right) \wedge$$

$$\wedge \prod_{\substack{0\le x<x_{cand} \\ y_{cand}\le y\le y_{cand}+yDims_c}} \neg G_{cxy} \wedge \prod_{\substack{0\le x<W \\ 0\le y<y_{cand}}} \neg G_{cxy} \wedge \prod_{\substack{x_{cand}+xDims_c\le x<W \\ y_{cand}\le y<y_{cand}+yDims_c}} \neg G_{cxy} \wedge$$

$$\wedge \prod_{\substack{0\le x<W \\ y_{cand}+yDims_c\le y<H}} \neg G_{cxy} \wedge \left(\prod_{\substack{c'\in C, c'\neq c \\ x_{cand}\le x\le x_{cand}+xDims_c \\ y_{cand}\le y\le y_{cand}+yDims_c}} \neg G_{c'xy} \right)$$

The build process of this model has a quite large complexity (even if the solver is quite fast, once the model is ready), since all possible positions in each 2-D grid are considered, and for each one of them a big conjunction has to be imposed.

A better model can be constructed by switching to a 1-D perspective: in each row (column) there has to be either one portion of True variables of the length of the block x-dimension (y-dimension) or no True variable at all. Furthermore, to ensure non-overlapping, for each x, y position there has to be at most one layer of the grid that contains a True variable (or "exactly one" layer, if we assume that no hole may exist in the chip).
With these constraints is still possible that a layer of G is empty (unless we used the "exactly one" formulation) and also that contains more than one block (all of them of the same size) on different rows and columns. This would anyway leave one or more empty (entirely False) layers. To avoid these problems, the easiest way is imposing that no empty layer exists.
This intuitions can be formalized as follows.

1. no overlap:

$$\prod_{\substack{0\le x<W;0\le y<H \\ (c,c')\in C^2; c\neq c'}} \neg \left(G_{cxy} \wedge G_{c'xy} \right)$$

2. no empty layer:

$$\prod_{c \in C} \coprod_{\substack{0 \le x < W \\ 0 \le y < H}} G_{cxy}$$

3. shape of every row:

$$\prod_{\substack{c \in C \\ 0 \le x < W}} \left[\coprod_{0 \le y < H - yDims_c} \left(\prod_{\substack{0 \le y' < y \\ y + yDims_c \le y' < H}} \neg G_{cxy'} \ \wedge \prod_{y \le y' < y + yDims_c} G_{cxy'} \right) \vee \prod_{0 \le y < H} \neg G_{cxy} \right]$$

4. the shape of every column is similar.

Efficiency and parallelization

So far, the model only involves boolean variables and FOL constraints.
Since $(N * W * H) \in O(k^3) with\, k \stackrel{def}{=} min(W, H)$ variables must be created and constrained, the computational bottleneck is not the execution of the solver, but rather the creation of the model itself (that must be allocated in memory).
This is evident comparing the total time with the one required by *check()* (the time to discover satisfiability):

Solution found for instance 9 in **12.1** s; check time: **0.193** s - H=16
Solution found for instance 10 in **17.0** s; check time: **0.681** s - H=17
Solution found for instance 11 in **30.4** s; check time: **1.755** s - H=18
Solution found for instance 12 in **28.2** s; check time: **1.393** s - H=19
Solution found for instance 13 in **32.2** s; check time: **1.618** s - H=20
Solution found for instance 14 in **40.1** s; check time: **1.621** s - H=21
Solution found for instance 15 in **51.0** s; check time: **2.261** s - H=22
Solution found for instance 16 in **82.4** s; check time: **14.499** s - H=23
Solution found for instance 17 in **74.8** s; check time: **3.337** s - H=24

As shown, for this set of inputs, the time required by the solver usually does not exceed few seconds.
In the attempt of overcoming this problem, some easy modifications have been tried:

- As commented above, the model only deals with booleans, thus It can be solved by a "pure" SAT solver. Here we have forced z3 to use a solver optimized for a quantifier-free, finite-domain logic, by instantiating the solver with: *SolverFor("QF_FD").*
- A proper use of the *simplify()* function on the formulas to be asserted in the model usually reduces a bit the total time.

- An explicit set or list comprehension instead of an equivalent formulation with the *itertools* functions in some cases is faster; for example:
  `{(a,b) for a in range(0,W) for b in range(0,MAX_H)}`
  has proven to be faster than:
  `itertools.product(range(0,W), range(0,MAX_H))`
  Note that this detail is implementation-dependent.

Despite all this, the total time was still too high to justify the use of a SAT model. As for the case of CP, to gain better results we have relied on parallelization. This time, more than the solver, what had to be parallelized was the creation of the model.

Since this model exhibits an evident symmetry between the constraints imposed over the x and y axes, a natural choice was to separate from the main flow the computation of these two sets of constraints, thus obtaining three concurrent processes:

- the one responsible for creating the constraint set 1 and 2 (main process);
- the one responsible for creating the constraint set 3 (constraints on the rows);
- the one responsible for creating the constraint set 4 (constraints on the columns).

Sadly, the implementation of z3Py relies on C pointers to internally represent formulae, so that all the objects representing them are not serializable and therefore not easy to be passed or shared among processes. To avoid the serialization problem while keeping the code simple, we have decided to convert each formula computed by each process into its string representation give by *sexpr()*. Then, is the main process to gather these representations and to create the model using *from_string()*.

This allows for a very good speedup (usually between 1/3 t and 1/2 t), but It has the drawback of requiring a big amount of memory, that becomes too big on hard instances.
A hybrid approach (that avoid the string representation for constraint sets 1 and 2) has given the best tradeoff; for comparison these are the obtained timings (take them with caution, given that they also depend from the machine load at test time):

Solution found for instance 9 in **8.3** s; check time: 0.202 s - H=16
Solution found for instance 10 in **12.5** s; check time: 0.309 s - H=17
Solution found for instance 11 in **23.2** s; check time: 2.208 s - H=18
Solution found for instance 12 in **22.0** s; check time: 1.775 s - H=19
Solution found for instance 13 in **25.1** s; check time: 2.025 s - H=20
Solution found for instance 14 in **31.3** s; check time: 2.410 s - H=21
Solution found for instance 15 in **39.6** s; check time: 3.327 s - H=22
Solution found for instance 16 in **56.6** s; check time: 5.286 s - H=23
Solution found for instance 17 in **59.8** s; check time: 6.175 s - H=24

Implied Constraints (2)

Sticking to a strictly boolean representation of the model, the only way to express the inequality "the sum of the horizontal circuit dimensions in every row is at most *W*" is by a conjunction of disjunctions such as:

$$\prod_{\substack{c \in C \\ 0 \le y < H}} \coprod_{0 \le i < W} \coprod_{\substack{0 \le X < W \\ P \in \rho(G_{cXy})}} \left(\prod_{x \in P_{0,i}} x \wedge \prod_{x \in P_{i+1,W-1}} \neg x \right)$$

where G_{cXy} denotes the set $\{G_{cxy}\, s.t.\, x \in X\}$; ρ the set of permutations of the element of the argument and $P_{l,u}$ the elements of the permutation in a position between l and u.

But this would certainly add further complexity to the build process.
Furthermore, our model already uses a similar constraint to impose the shape of each layer and than this implied constraint would not significantly speedup the solver.

Another approach is the one of switching to pseudo-boolean constraints [1], such as the two we can express with **PbLe**:

$$\prod_{0 \le y < H} \left(\sum_{\substack{c \in C \\ 0 \le x < W}} G_{cxy} \right) \le W$$

$$\prod_{0 \le x < W} \left(\sum_{\substack{c \in C \\ 0 \le y < H}} G_{cxy} \right) \le H$$

We have determined empirically that the introduction of these constraints cannot reduce the time needed by the solver, and also increases the time for model building. As in the case of CP, the constraint involving H may be useful to bound H itself, in the case where H has to be determined by the model. This is not the case, since here we assume (iteratively) that H is fixed.

1 https://theory.stanford.edu/~nikolaj/programmingz3.html#sec-pseudo-boolean-constraints