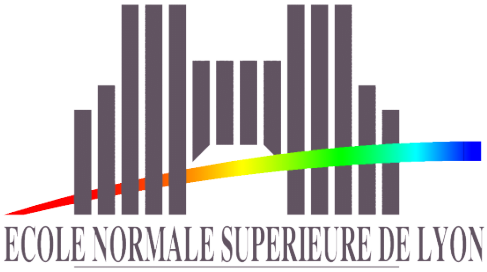
Ecole Normale Superieure de Lyon



August, 28th 2015

Summer internship report

An hybrid approach to solve boolean formulations of routing problems

William Aufort

ENS DE LYON

UPC

"Research is what I'm doing when I don't know what I'm doing".

Wernher von Braun

General introduction

This document is the report of my internship at the UPC (Universitat Politècnica de Catalunya) in Barcelona, Spain. My internship lasted 12 weeks, from 16th of May to 7th of August. It was supervised by Jordi Cortadella.

The physical design of a cell is an important step in integrated circuit design. It can be decomposed into two main tasks: the placement of the transistors and the wire routing. The goal of this second step is to connect the convenient parts of the circuit in order to get the wanted behaviour, using the available metal layers. This task becomes more and more complicated over the years. Indeed, the lithographic gap between the light wavelength and the actual features size is imposing complex design rules that really complicate the automation of the process, and much more difficulties if we want an adaptative or generic algorithm.

Recent methods use boolean formulations of this routing problem and solve it using a SAT solver. The goal of my internship was to move forward in this direction and to study a new kind of algorithms based on an hybrid approach, i.e. using the solver differently.

This report is organized as follows: I first expose the basis about the routing problem and the boolean formulation, in order to lay the foundations of my work and guess the possible improvements. Then, I will detail the main ideas of the hybrid approach. The rest of the report exposes the experiments, new ideas and results we get, following the chronology of my internship.

Acknowledgements

I would like to take the opportunity to thank all the people who have contributed in any ways to this report and to my internship in general:

- ⋆ Of course, my first thanks go to Jordi Cortadella who initiated the project and supervised my work. For his attention, for all the time he awarded to me. His experience was really useful and I appreciated to work with him.
- ⋆ My thanks also go to all the people with who I had several discussions about my work, particularly Professor Jordi Petit.
- ⋆ I also thank all the group with which I gather good moments in Barcelona, mostly during the lunchs: Alberto, Alex, Dany, Eva, Javier, Laureline, Maria Angel, Mayler and Thomas. I hope that I haven't forgotten anyone and that I spelt all the names correctly :).
- ⋆ Then, I would thank all the members of the UPC I had the occasion to meet, especially the technical service, because without them I wouldn't have been able to start my work.
- ⋆ Finally, my last thanks go to all my family and Camille my fiancé, who were always behind me, particularly when I faced difficult times.

Contents

1 The routing problem

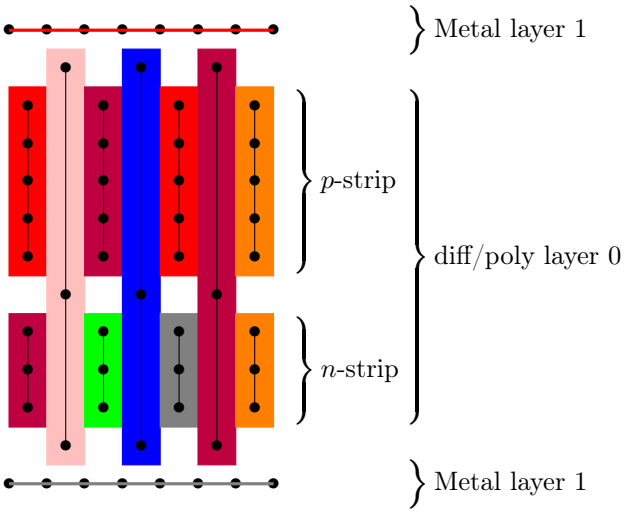
In this first section, I will introduce all the definitions around the routing problem, and also the main work done in [1] which have been the basis of my work.

1.1 Boolean model of the problem

1.1.1 Basis of cell synthesis

A Metal-Oxide-Semiconductor structure (or cell) is created by superposing several layers of conducting and insulating materials. This provides two types of transistors, p and n, which have both a switch behaviour but opposite.

In a cell, the transistors are aligned in two horizontal diffusion strips (p and n), and the polysilicon (used as a gate contact) is laid out with vertical rectangle, both in the first layer of the cell (poly/diff). The Vdd (high potential) and Vss (low potential or ground) signals are aligned in two horizontal wires at the border of the cell, on the second layer (metal 1). Finally, a second metal layer (metal 2) can be also used to route the cell. The figure 1a illustrates in two dimensions all these elements for the AND gate layout. Route this cell consists in connecting the components sharing the same color. Such a route is represented in the figure 1b.



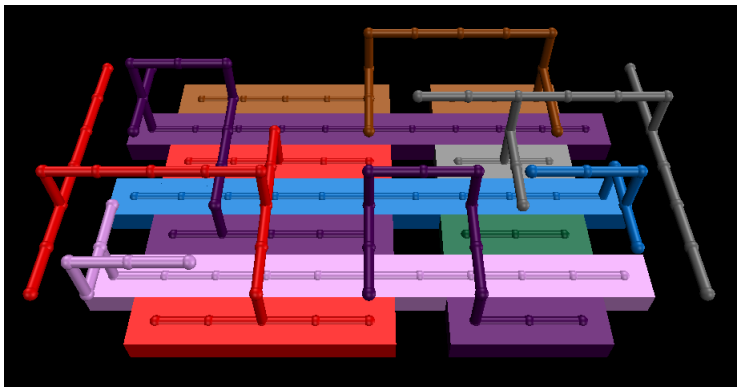


Figure 1: The general shape of a circuit in 2D and 3D

Consequently, an intuitive way to represent the gridded routing problem is to use a graph representation.

1.1.2 Graph model: definitions

The routing region is represented by an undirected graph $G = (V, E)$, where the vertices are the grid points, and the edges are the possible connections between the grid points. Each vertex v has 3D-integer coordinates $(x, y, z) \in \{1, \ldots, W\} \times \{1, \ldots, L\} \times \{1, \ldots, H\}$, where W, L and H denote respectively the width, the length and the height of the grid. In our case, z represents the layer of the layout and we have $H = 3$ for all the cells.

The figure 2 illustrates the grid model and the notations for the edges. The notations $e_{i\bullet,j,k}$, $e_{i,j\bullet,k}$ and $e_{i,j,k\bullet}$ are used to denote the edge started from the vertex (i, j, k), where the symbol $\bullet$ indicates the dimension occupied by the edge.

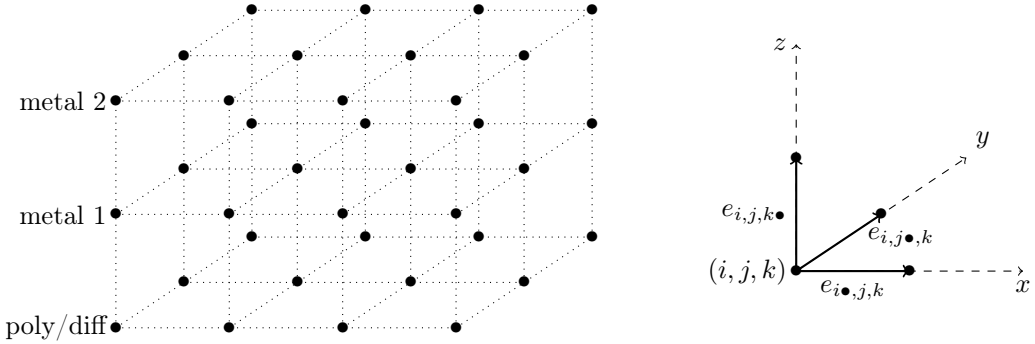


Figure 2: The grid model

From now, for the sake of simplicity, most of the definitions will be illustrated with 2D examples instead of real 3D cells.

Let us introduce the following general vocabulary about the routing problem:

Definition 1.

- A **terminal** is a set of grid points.
- A **net** $n \subset V$ is a set of terminals that must be connected. A **n-terminal** is a terminal associated to the net n. We denote by $|n|$ the number of terminals in n.
- A **subnet** s is a pair of terminals of the same net.

So the routing problem can be defined as follows: given a gridded routing problem, we have to find a set of edges that defines several routes which connect the terminals of each net and satisfy a set of design rules.

The figure 3 illustrates a particular instance of the routing problem with two nets: the red one with three terminals to connect, and the green one with only two terminals. The edges that can be used to route the cell are also represented. The second part of the figure shows a possible solution to our problem.

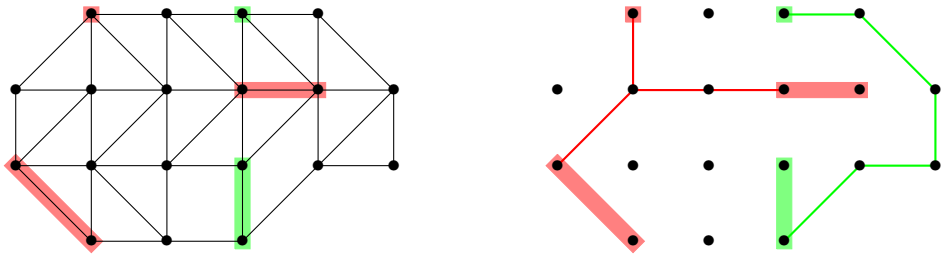


Figure 3: An example of routing problem with two nets (a), with a legal solution (b)

With the definition of a subnet, we can notice that the problem of finding a route for the net n can be reduced to find a route for $|n| - 1$ subnets of n.

1.1.3 The boolean variables

[1] introduces three types of boolean variables, which are denoted with the symbol ρ:

- for each edge e, $\rho(e)$ represents when e is occupied by a wire.
- for each edge e and net n, $\rho(e, n)$ represents when e is occupied by a wire associated to the net n.
- for each edge e, each net n and each subnet $s \in n$, $\rho(e, n, s)$ encodes the subnet associated to the wire. This kind of variables is only used for the routability constraints (see 1.2.2).

1.2 Boolean formulation

The formula we have to satisfy can be written as:

$$\mathcal{F} = \mathcal{C} \wedge \mathcal{R} \wedge \mathcal{DR}$$

where $\mathcal{C}$ (respectively $\mathcal{R}$ and $\mathcal{DR}$) represents the set of consistency constraints (respectively the routability constraints and the design-rules constraints) translated into a boolean formula on the introduced variables.

We will detail now the aim of each of these constraints and how they are built.

1.2.1 Consistency constraints

These constraints just define the relationships between the variables $\rho(e)$, $\rho(e, n)$ and $\rho(e, n, s)$. They traduce the two obvious observations:

Definition 2 (Consistency constraints)**.**

- If an edge is assigned to a net, then it's occupied by a wire:

$$\forall e, \left(\bigvee_{n \in \mathcal{N}} \rho(e, n) \right) \Rightarrow \rho(e)$$

- If an edge is assigned to some subnet of a net, then it must be associated to the same net:

$$\forall e, \forall n, \left(\bigvee_{s \in \text{subnets}(n)} \rho(e, n, s) \right) \Rightarrow \rho(e, n)$$

1.2.2 Routability constraints

[1] simplifies a lot the previous works done about the expression of the routability constraints (for example [4]) in order to get a formulation based on a conjunction of local properties on the vertices.

Let us consider a fixed subnet s of a net n. s contains two regions S and T which have to be connected. We introduce the following expressions:

- $\text{Ext}(S)$ is the set of external edges of S: $\text{Ext}(S) = \{(u, v)/u \in S \wedge v \notin S\}$
- A proposition to force only one external edge assigned to n and s on a set S of grid points:

$$\text{Terminal}(S, n, s) = \sum_{e \in \text{Ext}(S)} \rho(e, n, s) = 1$$

- An expression to denote the number of wired edges assigned to n and s, and connected to some point v of the grid:

$$\text{Nadj}(v, s, n) = \sum_{e \in adv(v)} \rho(e, n, s)$$

- An expression to force a grid point v to have a degree zero or two (with regard to the subnet s of the net n):

$$\text{Degree0or2}(v, n, s) = \text{Nadj}(v, s, n) = 0 \vee \text{Nadj}(v, s, n) = 2$$

We can now express the routability constraints with a boolean formula:

Theorem 1.

The constraints for the routability of s are:

$$\text{Terminal}(S,n,s) \wedge \bigwedge_{v \notin S \cup T} \text{Degree0or2}(v,n,s)$$

Indeed, given a subnet $s = \{S,T\}$ of a net n, Terminal(S,n,s) implies that only one wire can be connected to S. $\bigwedge_{v \notin S \cup T}$ Degree0or2(v,n,s) implies that outside S and T, the edges e such that $\rho(e,n,s)$ is assigned to true form a path which continues until the last edge enters in T or in S. But this second option is impossible because Terminal(S,n,s) is true. What's more, such a path cannot do cycles, because each node in the domain has a degree less than two regarding to this subnet. Consequently we get a correct and unique path between S and T.

One can notice that with this reasoning, a wire e can be involved in more that one subnet of the same net, as illustrated in the figure 4. But clearly we don't want a wire to be involved in several nets, and two adjacent wires to be assigned to different nets. So we have to add some consistency constraints for the routing:

Definition 3 (Consistency of the routing)**.**

- $\forall e, \forall n, \forall n' \neq n, \quad \rho(e,n) \Rightarrow \neg\rho(e,n')$
- $\forall n, \forall e, \forall e'$adjacent to e, $\quad (\rho(e,n) \wedge \rho(e')) \Rightarrow \neg\rho(e',n)$

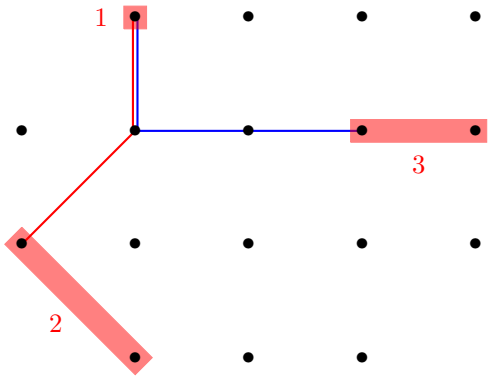


Figure 4: An edge is simultaneoulsy used to route the subnets $\{1,2\}$ and the subnet $\{1,3\}$.

A last remark about the formulas: the formulas Nadj and Terminal contain arithmetical equalities. This type of construction is not allowed in a SAT formula. In practize, [1] uses a method to transform them into Boolean clauses [3].

1.2.3 Design-rules constraints

The design-rule constraints represent all the conditions imposed by the lithographic gap. They can be easily modelled into boolean formulas. We give some examples of rules and formulations:

1. The metal 1 layer of the grid can only be laid out in the horizontal direction:

$$\forall i, \forall j, \quad \neg\rho(e_{i,j_\bullet,m1})$$

2. All horizontal wire segment in the metal 1 layer have a length of at least two units:

$$\forall e_{i_\bullet,j,m1}, \quad \rho(e_{i_\bullet,j,m1}) \Rightarrow (\rho(e_{i-1_\bullet,j,m1}) \vee \rho(e_{i+1_\bullet,j,m1})$$

3. A via cannot have any other via in the eight surronding grid points:

$$\forall i, \forall j, \forall k, \quad \rho(e_{i,j,k\bullet}) \Rightarrow \bigwedge_{\substack{(di,dj)\in\{-1,0,1\} \\ (di,dj)\neq(0,0)}} \neg\rho(e_{i+di,j+dj,k\bullet})$$

In all the experiments made during my internship, I used only one set of design rules. The reader can found these rules in the appendix A.

1.2.4 Optional constraints : subnet windowing

In order to reduce the complexity of the solution, [1] adds additional constraints to restrict the routability of a subnet to a region around the subnet. Indeed, empirically the routes rarely go beyond the bouding box defined by the two terminals of the subnet. Even if with this method it can be possible to get an unsatisfiable formula even if there exists a solution, in practize it works well.

In the formula, this constraints are very easy to express: we just enforce the variables that are outside a given region called Region(n, s) to be assigned to false: $\forall n, \forall s, \forall e$ such that $e \notin \text{Region}(n, s), \neg\rho(e, n, s)$.

1.3 Solve the problem

If the boolean formula introduced previously is satisfiable, a satisfiable assignment induces a valid routing, but we want the one with the best quality. To improve the quality of this solution, we can optimize several parameters, like the total wirelength (mathematically: Card$(\{e/\rho(e) = 1\})$). The approach proposed in [1] combines both integer linear programming and large neighborhood search. The main idea is to iteratively improve the route involving one particular net.

1.4 Implementation details

During my internship, I had access to the C++ code of the implementation of the method described in this section, and also the code of the Minisat solver [2]. It also includes a tool to visualize the circuits which also allows to get the aspect of the circuit at any time of the solving. We can visualize our algorithms step by step, which is pretty useful, particularly for debugging.

2 The hybrid approach: main ideas

2.1 General idea

2.1.1 Pro and cons of the previous method

In the method presented, all the task of finding an admissible solution is left to a SAT solver. It tries to find an assignation of the boolean variables satisfying the formula, but without knowing that it's a particular formula, i.e. corresponding to a particular problem of graphs and paths in a graph.

An easy way to realize that is to look at the solver state (the assignation of the variables) during the solving phase and to interpret it as a current state of the routes. The figure 5 illustrates an intermediate state inside the solver. We can see easily that it is completely not related to a path search logic: some wires are completely disconnected. Also sometimes some routes contains useless wires (mostly at the end of the search).

Another way to illustrate that is to see the solver as a "black box" or a blind person, in which all the actions are done in the SAT world (only variables without meaning), whereas we would like to think in terms of routes.

Consequently, the leading idea of my internship was the following: if we can use the SAT solver but knowing that we are working with a routing problem, can we get a more efficient algorithm?

In practize, there is two motivations of such methods:

- Intuitevely, if we do a kind of guided search using the solver, we could **speed up the search of a particular solution**.
- The solution found could be better, in terms of wirelength minimization than a particular solution found with the solver alone. By the way we must **decrease the following optimization time**.

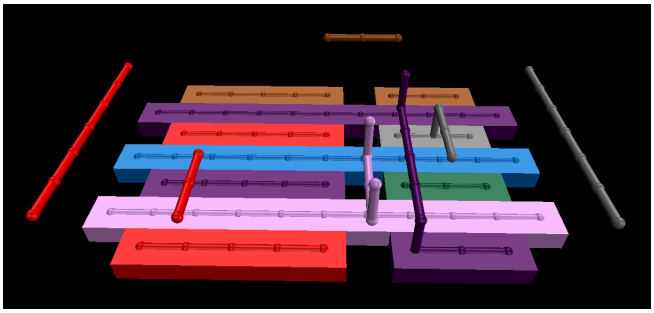


Figure 5: An intermediate state of the solver, with disconnected parts (red, brown and pink)

2.1.2 Interact with the SAT solver

We were interested in an algorithm based on the **interaction** between a SAT solver and a backtracking procedure which works on the principle described in the figure 6.

- The SAT solver is used to check if the partial solution during the backtracking procedure can satisfy all the rules we fixed. In our cell routing problem, the rules inside the solver are mostly the design rules, because the are really difficult to verify "by hand" (more details in the section 2.3.2);
- The interaction between the algorithm and the solver works as follows: the algorithm can select a variable and push it inside the solver, which returns as answer whether the solution still respects the constraints or not;
- The backtracking algorithm never works directly with the variables, but with wires (in our example).

To really make the difference between the solver part and the backtracking algorithm, the figure illustrates the solver as a black box or an oracle in which we don't know anything about the internal behaviour, whereas the backtracker can be customized and controled using different heuristics.

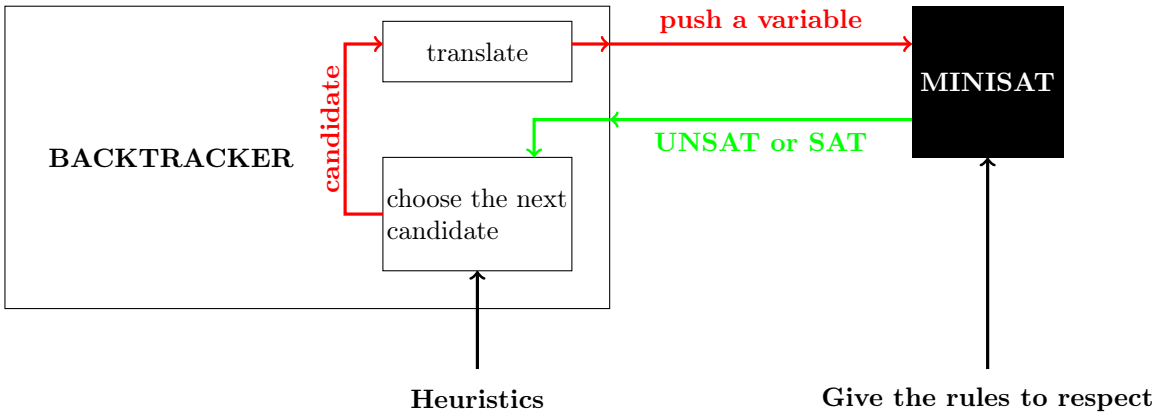


Figure 6: The principe of the hybrid approach

One of my first step during this internship was to implement this kind of algorithm, firstly on a simple problem, and then for the routing problem.

2.2 A simple illustration: the Sudoku problem

During the first days of my internship, I worked on the Sudoku problem in order to understand the principle of the algorithm and implement it in a simple example.

We first model the Sudoku problem with a boolean formula. Let $x_{i,j}^k$ the variable corresponding to the following property of the Sudoku grid: "The cell (i, j) contains the number k". With the set of variables $\left\{x_{i,j}^k, 1 \leq i, j, k \leq 9\right\}$, we can describe all the rules of the Sudoku and the initial grid. There are several ways to do this: force all pair of

elements in a row/column/square to de different, state that in any cell/row/column/square there is exaclty one of the number from 1 to 9,...

An approach only based on a SAT solver is to put all the formula inside a solver and get the solution. An hybrid approach here could be to choose ourself the number we want to assign, transform it into a variable, push it to the solver and see if the rules are respected. The basic strategy I have implemented is to complete the grid top to bottom and left to right. But we can imagine different strategies for example complete the cell where there is the less of possible choices first. If you remember the previous figure 6, we have just changed the heuristic to select the variable, and not the solver formula.

2.3 General algorithm for the routing problem

To adapt the hybrid algorithm described previously to solve any problem, like in the previous example of the Sudoku problem, we have to answer the following questions:

1. What are the candidates on which the backtraking part works on?
2. What are the rules given to the SAT solver?
3. What are the heuristics to choose the next candidate?

2.3.1 The candidates

The general idea of our approach is to use the solver to check a route we are building with our backtracking algorithm. In other words, our candidates are exactly the edges we want to push to route a given subnet. They correspond to the variables $\rho(e, n, s)$ after the translation.

We can also precise now what could be the possible next candidates at a given step of the algorithm: if we choose an edge e and we want to build a path, the next candidates are exactly the neighbouring edges of e if the path is not complete (i.e. we don't have already connected the subnet). Instead, we build another path for another subnet.

At this moment of our work, the shape of the backtracking algorithm for the routing problem is much more precise (see the algorithm 1).

Algorithm 1 Backtrack procedure

```
function BACKTRACK(c)
    Push to the solver the edge e
    if the routing is incorrect then return
    end if
    if the routing is complete then output c
    end if
    candidates ⟵ the next edges reached from e.
    for s ∈ candidates do BACKTRACK(s)
    end for
end function
```

2.3.2 The rules to respect

In the backtracking part, we have a control about the routing constraints since we build the path. Consequently, all the clauses $\text{Terminal}(S, n, s) \wedge \bigwedge_{v \notin S \cup T} \text{Degree0or2}(v, n, s)$ can be removed from the set of clauses, which is quite interesting because they were one of the most difficult to model. But we have to keep the consistency constraints of the routing for two reasons. The trivial one is not to assign several nets to one edge. The second one have been discovered during the implementation and will be discussed in 3.1.

2.3.3 The heuristics?

The detail of the heuristics tried will be described later (see 3.2). But we can already see that there will be different levels of heuristics. By "levels", I mean that the heuristics will not concern a choice on the wires. Indeed, since we want to route a cell, we have to decide in which order we are going to route the different subnets. But for a given

subnet, we also have to choose which pair of terminal we are connecting, because several pairs are possible. Finally, given these two nodes we want to connect, we also have to choose a good heuristic to build a path between them.

At the end, we get three levels of heuristics:

1. The order of the subnets;
2. The order of the try of the terminals inside a subnet;
3. The heuristic to build the path between two given terminals.

An important fact is that these three levels of heuristics are really crutial in the process. To build a path for a specific subnet, we would like a simple or short path in order to not induce congestion which could complicate the routing of the further subnets. What's more, it's not clear that the order of the subnets is not important. For example, one can imagine that if we route the easier subnets first, then the choice for the more difficult one will be highly restricted, and the risk of the unsatisfiable routing will increase a lot. The same reasoning holds for the order of the different terminals of a given subnet.

One can notice that the complexity of such an algorithm is directly proportional to the number of nodes we are going to look in the whole tree of solutions. So the heuristics chosen are the keys to get a good algorithm: if we are doing a clever search in this tree, we will visit less nodes, and so the complexity will be better.

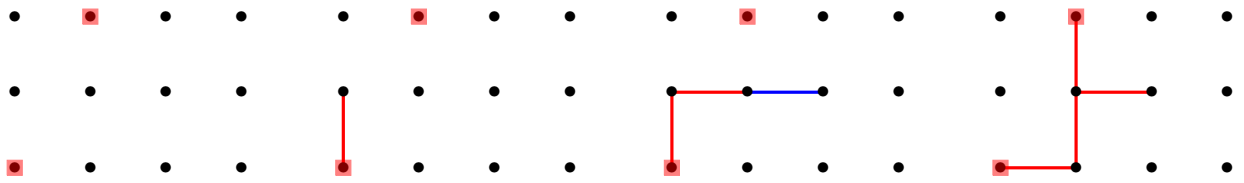
3 First implementation

I started by a naive implementation of the backtracking algorithm in terms of heuristics. The first objective was to have correct basis for the further work, i.e. an algorithm without bugs and for which we have a guarantee of termination, even if we don't see the end of the algorithm. After my first failed attempt when I tried to think about cleverer ways to solve the problem, this implementation had been done quite quickly. But even with a simple implementation, unexpected problems appeared, and I'm going to describe them and the solutions we brought to solve them.

3.1 Why we have to finish the search?

At each step of the algorithm, we ensure that all the rules, and particularly the design rules, are satisfied by the partial routing. But at the end of the search, the routing we get is not necessarily complete. Indeed, remember that some design rules have as consequence the addition of wires, we call them **additive design rules**. For example, in the examples of section 1.2.3, the second rule is an additive design rule (minimal length), whether the first one is not.

In some cases, because of unit propagation inside Minisat, some wires can be added because we don't have the choice to satisfy the corresponding additive design rule, as in figure 7. We suppose for this example and the further that the only additive design-rule constraint we have to respect is the following (in two dimensions): the horizontal segments must have at least length 2. But in most cases, unit propagation is not sufficient and we have to complete the search.

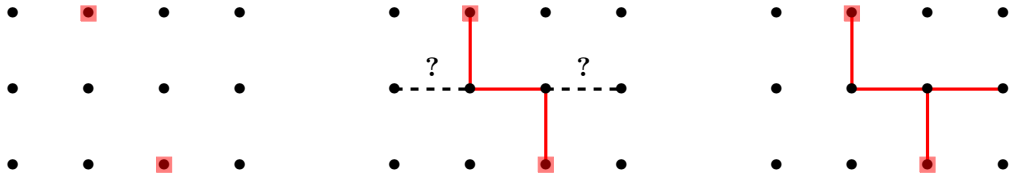


(a) The routing problem (b) First edge (c) Unit propagation of the blue edge (d) A final solution

Figure 7: An example of unit propagation

The figure 8 illustrates this case. In the figure 8b, a path is found, it can respect the design rules but we can choose between two edges to complete it into a route. We need to finish the solve in order to have a real route, as in 8c.

But a surprising thing can happen: this step of finishing the solving phase can fail. In other words, the solver can tell at each moment of the backtracking phase that there is no conflict about the design rules, but these design



(a) The routing problem (b) An incomplete solution (c) A possible final solution

Figure 8: In that case, we need to finish the search

rules are not satisfiable at the end. This phenomenon is caused by conflicts between some design rules with additive ones that cannot be solved with unit propagation. Such conflicts are difficult to identify and much more difficult to reproduce in small examples like the previous ones.

This fact explains also why we have to keep the consistency constraints of the routing in our set of rules to respect. If we let the SAT solver to complete our routing without this constraints, since the additive design rules enroll only variables corresponding to wire, the solver can not deduce the signal corresponding to the added wires.

Finishing the solving phase impose to solve a SAT problem, of course smaller than the previous ones. But since we cannot predict such situations only with our SAT checker, if this kind of situation appears often, it becomes a real mess in terms of complexity.

3.2 Heuristics chosen

The crutial point of the efficiency of our algorithm is the choice of the different heuristics. The intuition is that we want to build the simplest routes successively in order to avoid some congestion phenomenon which must induce a backtrack.

Build a path for two given terminals

A choice which seems good is to try to build for each subnet the shortest route. Consequently, for a given subnet we try to route it according to the **shortest possible path** between the terminals which respects the design rules.

Choice of the terminals inside the subnet

Similarly, to avoid the congestion it could be clever to choose the **closest** pair of terminals first.

Order of the subnets

Intuitively, if we route the simpler subnets first, it would be much more difficult for the longest. Indeed, we will add another difficulty (the current routed part) for these subnets that are already difficult to route because of their size. So an idea for the order of the subnets could be to order them by their **size** in the decreasing order. A representative size could be the minimum length of a path connecting the subnet.

3.3 Tests and first conclusions

After solving these problems, the algorithm worked well on the smallest cell, but even with a little bit more complex cell, we don't see the end of the algorithm: the limits of the naive heuristics had already been reached. The main conclusion is that by ensuring that the algorithm returns a solution if there exists one, the algorithm has to deal with all the tree of possibilities in the worst case, which is a catastrophe in terms of complexity.

4 Some improvements

The main idea of almost all the following improvements we tried is to restrict our search on a part of the tree is order to speed up the computational time. We tried more restrictive heuristics, but also partial routing techniques.

4.1 More restrictives heuristics

The first solution to speed up the search is to restrict it in some part of the tree. A way to do this, in term of paths, is for example to fix a maximal length to the path we want to build for each subnet. Intuitively, we would like to avoid some "snake" path between two terminals, because it must be simplified and it's going to make the rest of the routing more difficult.

One can notice that this strategy is very similar to the windowing strategy described in the section 1.2.4, except that a "snake" route is completely allowed inside the bounding box of the subnet.

But we cannot have a constant the maximal length of all the paths. They have to be related to an intuitive distance between the two regions which compose the subnet: I choose the minimal length between the regions in the graph. More precisely, if we denote by l the minimum distance for a path for a given subnet, the maximal length L for a route of this subnet will be an affine function of l: $L = \alpha.l + \beta$, for well-chosen (α, β).

More precisely, we choose α quite close to 1 not to get huge values for the biggest subnets. Conversely, if β is small, since α is already small, the limit L for the smallest subnets will be too close to the length of a minimal path, which doesn't let a lot of possibilities for the route. The choice we make is $(\alpha, \beta) = (1.2, 2)$.

4.2 Partial routing

4.2.1 General idea

Another idea could be not to route all the cell using our backtracking approach but just a part. Indeed, as we saw with the previous implementation, the problem seems too complex to be solved using intuitive heuristics. What's more, exploring useless branches of the tree, combined with the phenomenon described in sectionÂă3.1 makes the algorithm potentially very slow. While if we tried to finish a partial routing, we can detect unsatisfiable subtrees without trying all the possibilities. The figure 9 summarizes the idea: with the previous algorithm, all the red zone will be explored, even if no solution will be found. In the converse, by calling the solver at the given threshold, only the blue part will be explored.

Note that now in the algorithm, our order on the subnets becomes a selection of some subnets, which allow us to make different choices (cf 4.3).

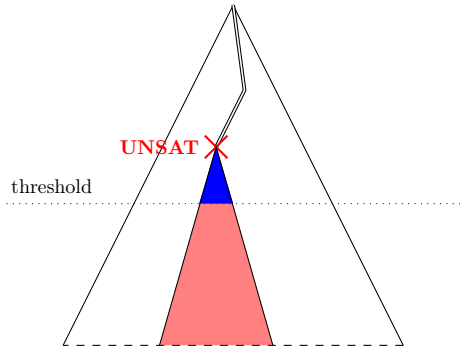


Figure 9: Avoid useless exploration using the threshold.

To implement this partial routing, we fix a given threshold $\lambda \in [0, 1]$ for a cell. If we denote by $\mathcal{S}$ the number of subnets, we route as previously only $\lfloor \lambda.\mathcal{S} \rfloor$ subnets. After this prerouting, we call another SAT solver to finish the work, with the additional constraints which fix the preroute found, and without the routing constraints for the already routed subnets. We can notice that since these constraints are composed only by unit clauses, they will be treated fastly in this new solver.

An additional goal was now to find the appropriate threshold λ. Of course, since the cells have different complexities, we cannot fix a threshold for all the cells.

A quantity we would like to explore as a threshold is the congestion of a cell. We will now give the definition, the ideas behind and we will see if it is a good choice.

4.2.2 Congestion of a cell

In all these definitions, n will denote a net, $s = (S, T)$ a subnet and c a column of the cell (i.e. a vertical element). We suppose that S and T are along a column of the cell, which is the case for almost all the subnets.

Definition 4 (Congestion of a cell)**.**

We define successively:

- the function $\text{cross}(n, s, c) = \begin{cases} 1 & \text{if } c \text{ is between the column of } S \text{ and the one of } T \\ 0 & \text{otherwise} \end{cases}$
- the congestion of the column c :

$$\text{Congestion}(c) = \sum_{(n,s)} \text{cross}(n, s, c)$$

- The congestion of the whole cell is defined by:

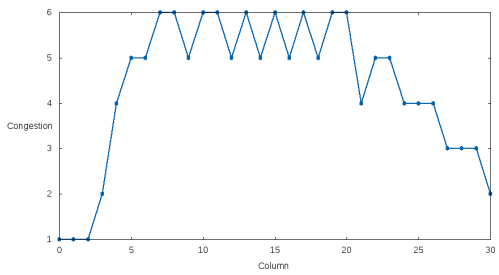
$$\sum_{c} \text{Congestion}(c)$$

- Finally, the normalized congestion of a cell is:

$$\frac{\sum_c \text{Congestion}(c)}{\#\text{columns}}$$

The main idea of the congestion is the following: every route from a column to another one must pass through all the columns between the two at least one time (it could me more, of course). So the quantity $\text{Congestion}(c)$ represents the minimum number of subnets which have to pass through c to be connected. Then, by summing all this quantity we get the congestion of the whole cell.

Intuitively, the congestion mesures the increasing difficulty when routing successively the subnets, and we can imagine that in most cases it is more concentrated in the middle of the cell than on the extremities, as shown in figure 10 for the FA_X1 cell (full 1-bit adder circuit).



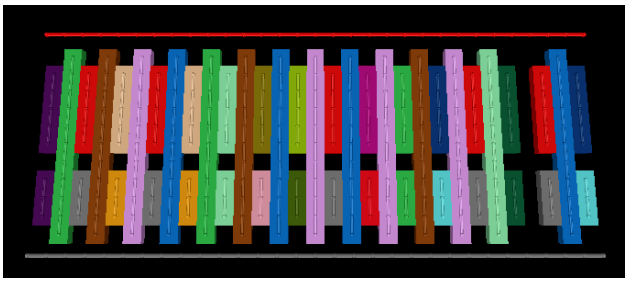


Figure 10: A typical profile of congestion, most concentrated in the center of the cell

Since the congestion is quite proportional to the size of the cell (more particularly to the number of columns), the normalized congestion is introduced in order to compare cells with different sizes. Until now, the word "congestion" will already refer to the normalized one.

4.2.3 A link between threshold and congestion?

I computed for each cell its congestion and the maximal threshold for which the computational time is "reasonable". I use the heuristics used in the last experiments.

The conclusions are quite negative. First, there is no simple link between the congestion and this maximal threshold. Secondly, even if there exists some link between the two, we cannot use the congestion as a threshold, since for a lot of cells the maximal threshold is really small. The results are represented in the graph 11, in which each point represents a cell.

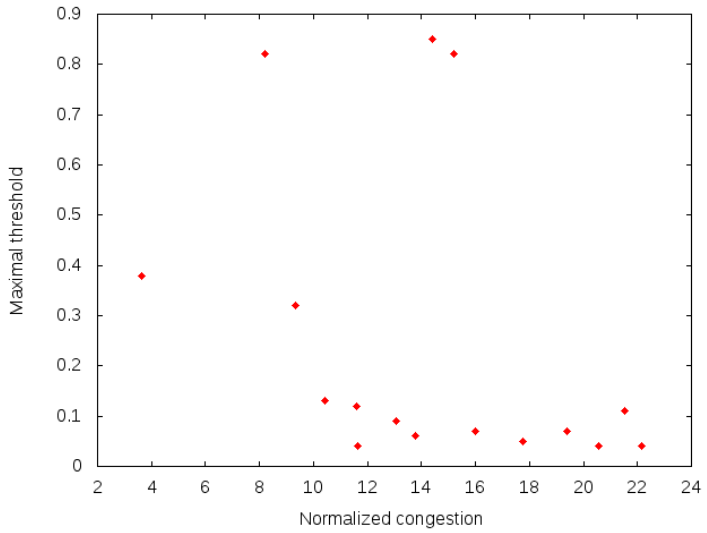


Figure 11: No link between the maximal threshold and the congestion.

4.3 Skipping some nets

Route only a part of the cell allows us to choose which subnets we are going to route, and not only their order. If a subnet seems difficult to route in a given configuration with the current maximal length conditions, we can try to skip it and let the work to the final router.

What's more, since our goal is not to route all the cell anymore, we can adopt another strategy: we can try to simplify as much as possible the problem by routing the easier subnets first. This could reduce the time to find an admissible solution of the same order of complexity. What's more, we must have less risk to explore unsatisfiable leaves by routing the easier first. One the other hand, maybe the problem will not be as much simplified as we would like. Since the issue is not easy to guess, we have to experiment it.

The figure 12 represents the time needed to route a OR-AND-Invert gate (a small size cell) and the wirelenght of the route, depending on the number of subnets we have routed manually. The original router takes less than one second to do the job, whereas after a small number of subnets routed the time goes over two seconds. Nevertheless, the wirelength is better than the original one (even if we route only few subnets) and becomes closer to the optimal one (i.e. found with the router after the optimization phase) with an increasing threshold.

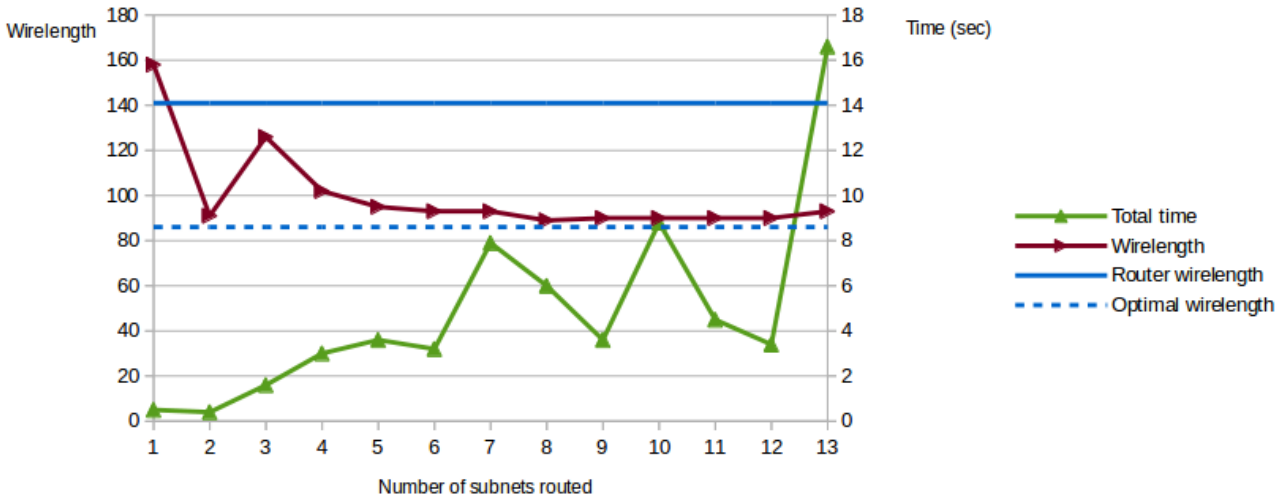


Figure 12: Skipping some nets give contrasted results

As a conclusion, skipping some nets seems to be an interesting idea in terms of quality of the route, but not in terms of computational time. What's more, we are not fixed about the threshold to choose.

5 More deeper in Minisat

After the first fruitless attemps which consisted to interact with the SAT solver, another idea was to guide the solver but more deeper inside. We give more intuition about this work, some basis about the internal behaviour of Minisat which interests us and the improvements tested.

5.1 General idea: selecting variables with more freedom

In the previous approach we used Minisat only as a checker. We had a complete control of the variable chosen and the goal of the solver was only to check that our paths respect a given set of rules.

Consequently, we don't take advantage about advanced features used in Minisat to solve satisfiability problems, like clause learning, backtracking process by conflict analysis, or heuristics to choose the next variable [2]. If we go more deeper inside Minisat, we can interact with it in a less restrictive way. The idea of this new kind of interaction is the following: we only want to indicate to the solver a set of variables we would like it to choose. As [2] says, "Variable ordering is a traditional target for improving SAT-solvers".

More precisely, our idea was that the most important variables in the problem are the subnet ones, because the other ones are only used to consistency or design rules reasons and the consistency constraints implies: $\rho(e, n, s) \Rightarrow \rho(e, n) \Rightarrow \rho(e)$. To emphasize this idea, we had a look on the repartition of the variable chosen by Minisat in the original algorithm. The result was that in all the cases, all the variables are considered identically (see figure 13).

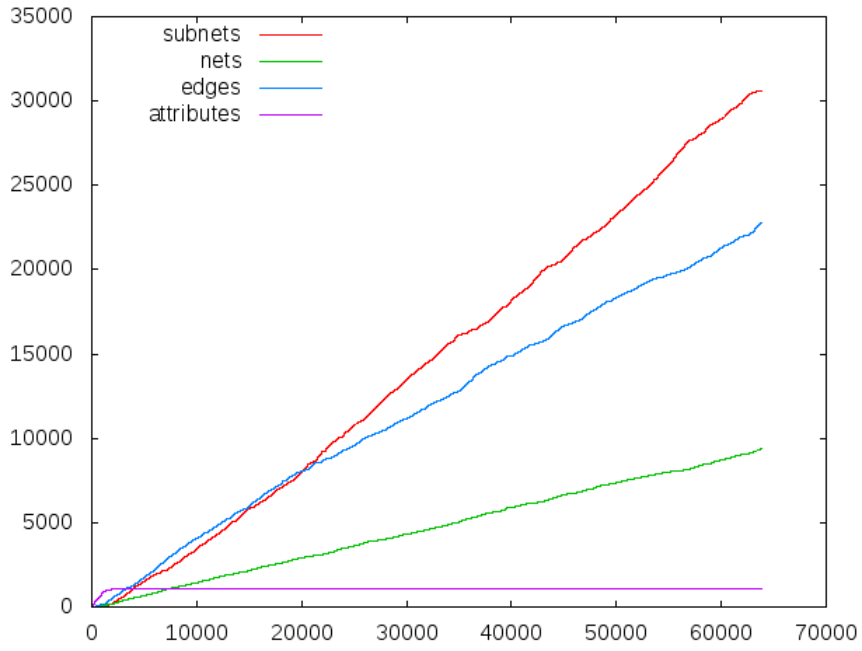


Figure 13: The uniformly chosen repartition of the variable in a Minisat execution

5.2 Variable decision in Minisat

The goal of our new approach is to influence the choice of the variable inside Minisat.

In the original Minisat, a dynamic variable ordering is used based on the activity of the variables [2]. The activity of a variable is related to the conflict analysis: at each time a variable occurs in a conflict clause, its activity is increased (refered as "bumping"). By multiplying the activity of all the variables by a factor less than 1 after each conflict, Minisat gives more advantage on recent conflicts. In this process, no distinction between x and $\bar{x}$, and so the polarity is fixed after, by a default choice (random or fixed) or by choising the latest polarity observed.

Therefore, to modify the choices of Minisat, we can play either on the activity or on the polarity of a variable.

5.3 Improvements proposed and tests

5.3.1 Guide for the subnets choice

As explained before, our guess is that the subnets variable are the most important. To increase their importance inside the solver, we can increase their activity. More precisely, if we want to keep a "path-building" strategy, we can increase the activity of the subnet variables precisely when a subnet variable is chosen, as in the figure 14. If the red edge is selected, we give more importance to the neighbouring (green) ones. In terms of variables, if $\rho(e, n, s)$ is chosen by the solver, then we increase the activity of the variables $\rho(e', n, s)$ for all e' close to e.

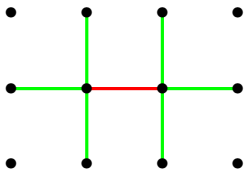


Figure 14: How we increase the activity of the subnets variables

But we have to fix the amount of activity we are going to add. We think about a fixed amount in the same order of magnitude as the default amount fixed in Minisat in the case of a clause learning bumping.

I experimented different values and compare the execution time of the solver only and the total wirelength get for a given set of cells. The results are balanced (see figure 15): neither the execution time nor the wirelength follows an expected behaviour depending on the parameter. But for all the values tested we get the same conclusions. On the one hand, the execution time for all the cells excepted the biggest ones is decreasing. More precisely, for the biggest cells the execution times change a lot with different parameters, whereas they are stable for the other cells. One the other hand, the overall wirelength is increasing. As a remark, in the previous experiment (skipping some subnets in the original idea), we get the opposite conclusions: a decreasing wirelength but increasing execution time.

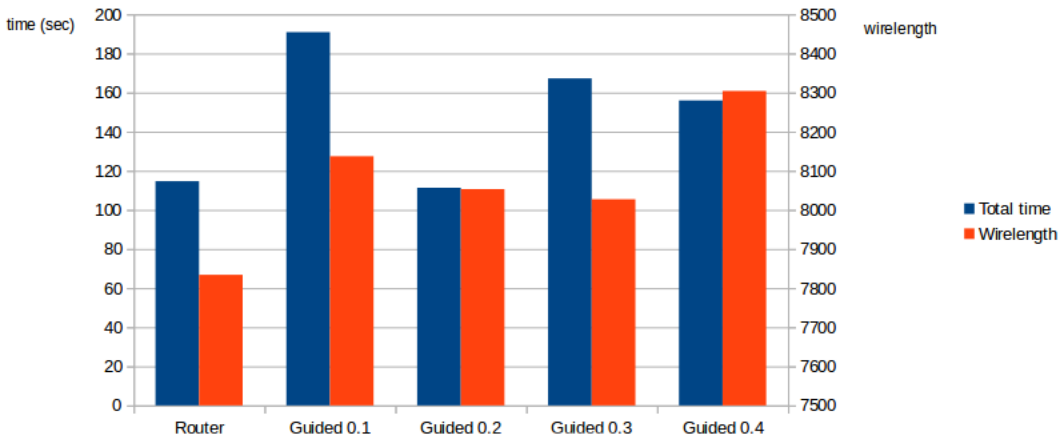


Figure 15: The original router have the best time and total wirelength.

5.3.2 Polarity choice based on the type of variable

As discussed in 5.2, we can influence the choices inside Minisat by modifying the activities of some variables (as done previously) or by modifying the chosen polarity. An idea could be to fix a default polarity depending of the kind of variable (wire, net, subnet).

Intuitively, if we force Minisat to use especially the subnet variables, the other variables will not be used a lot of time. Particularly, if some wire variables are assigned, in most of the cases (excepted for the additive design rules) we can imagine that it will add useless wires in a partially routed solution. By the way, another idea I had was to change the default polarity of the variable corresponding to edges or nets to false.

But in practize, we obtain an increasing total wirelength with this method, which seems really strange. I didn't had time during this internship to enter in the details and find the explanation of this phenomenon.

5.3.3 Less relevant tries

In this last part we will discuss about other minor tries or unexplored ideas we had.

- Initial values: At the beginning we forced Minisat to choose among a given set of variables which correspond to the possible input terminals. This strategy had been implemented in a simple way and we didn't get any improvement.
- Probabilities: When a variable we didn't want to choose is at the top of the priority queue, then we choose it with a given probability, and otherwise we just skip it. This strategy induces some termination problems and so was not experimented.
- Top of the priority queue: Instead of simply increase the activity of the desired variables, we tried to put them directly at the top of the queue. But by this way, the mechanism of clause analysis was completely useless. What's more, the experiments gave bad results.

Conclusion

We started with intuitive ideas (hybrid algorithm, backtracking with checking) to improve the initial algorithm of [1] with a lot of hopes. Unfotunately the results were not so good, because of the complexity of the problem, even if we try less naive heuristics and mechanisms to speed up the search (looking only one some parts of the tree that must contain solutions, doing a partial routing).

After this failure, we tried a more deeper approach in order to let more freedom inside the solver. Even if we didn't have a lot of time to do this, we explored some trails to improve the solver in the particular case of our routing problem, by playing on the activity and the polarity of the variables. A further work will be to focus more on this idea, for example by thinking about a more complex choice of the variable.

Even if we didn't get satisfying results, this internship has been very instructive for me. I had the opportunity to explore an unknown field and to work around a complex problem and try different ideas.

References

[1] Jordi Cortadella, Jordi Petit, Sergio Gómez, and Francesc Moll. A boolean rule-based approach for manufacturability-aware cell routing. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 33(3):409–422, 2014.

[2] Niklas Eén and Niklas Sörensson. An extensible sat-solver [ver 1.2]. `http://minisat.se/`, 2003.

[3] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

[4] Brian Taylor and Larry Pileggi. Exact combinatorial optimization methods for physical design of regular logic bricks. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 344–349, New York, NY, USA, 2007. ACM.

A Set of design rules used

In the experiments I used a set of design rules already defined and used in the program provided to me. I expose here this set of rules:

1. Metal 1 can only be routed in the horizontal direction.
2. Metal 2 can only be routed in the vertical direction.
3. The top row is only for the Vdd signal.
4. The bottom row is only for the Vss signal.
5. Horizontal wired segments in metal 1 must have at least length 2.
6. Vertical wired segments in metal 2 must have at least length 2.
7. A contact (wire between layer 0 and 1) must be followed by a metal 1 wire.
8. A via (wire between the two metal layers) must be followed by a metal 1 wire.
9. A via must be followed by a metal 2 wire.
10. A contact (respectively a via) cannot have any contact (respectively via) in the eight surronding grid points.
11. If two horizontal wired segments have two spaces in between and if any of the endpoints is connected to a via, then no horizontal wires can be present in the five surronding columns of the upper and lower rows of the edges.
12. Any vertical wired edge cannot have another neighbouring wired edge in the six surronding rows of the previous and following columns.
13. The space between two horizontal wire segments in the same row must be at least two units.
14. The space between two vertical wire segments in the same column must be at least two units.
15. Double pattern lithography.

A design rule language had been created before my internship in order to express in a C++ like language all these specifications. The code of the traduced rules can be easily converted into a boolean formula as those of this report. You can find the file here: `http://www.cs.upc.edu/~jpetit/CellRouting/rules.html`. The rule set I used is called DRC13 in this file.