Cell Routing using SAT

Sergio Rodríguez Guasch

November 27, 2018

Contents

1 Introduction

In this project we study the Cell Routing Problem and how it can be solved with SAT. We consider a simplified version of the problem and, after formally formulating it, we show a SAT encoding, and implement a program that is capable to solve instances of this problem. Our solver is also capable to optimize solutions.

2 The Cell Routing Problem

A simplified version of the cell routing problem can be formulated as follows: given a graph $G = (V, E)$, and a set of mutually disjoint subsets of V $S_1, ..., S_k$, find a subset of the edges such that, for all $S_i \in S$ it connects all the vertices of S_i between them, keeps them disconnected from any vertex that belongs to any other subset S_j. Additional goals as reducing the amount of used edges and other constraints can be included. From now on, the subsets $S_1, .., S_k$ may be also referred as *nets*.
There are many variants of this problem where additional constraints appear, and other kind of spaces and representations are considered. For simplicity, this project will focus on grid graphs, and all edges with have unit length. Although all figures will be in 2D, our implementation can deal with n-dimensional grids. In this project we study and implement a SAT encoding for this problem. The implemented encoding is based on the one that is described in [1].

3 Making the problem easier: subnets

The first problem one may find is that it is not trivial at all to find an efficient encoding to impose that some set of vertices of a graph is connected by some subset of edges. A possible solution may consist of a set of auxiliary variables $R_{u,v} \quad \forall (u, v) \in V \times V$ that are true if and only if u can reach v, and by adding unit clauses for all the needed $R_{u,v}$. This encoding is obviously bad, as $O(|V|^2)$ variables and clauses would be needed. However, this problem becomes easier if we have the guarantee that we will only deal with subsets of size 2, as we could simply impose these two endpoints to have degree exactly one, and the rest of vertices to have degree zero or two. For this purpose we can take a set $S \subset V$, consider various pairs $(u, v) : u \in S, v \in S$ and impose that u and v are connected. We can see that if we impose this constraint for all $(u, v) \in S \times S$ we will, in practical terms, implement the same constraint as before. We must note that the property *vertices u and v are reachable* is transitive. This observation makes us conclude that we can implement the original constraint with only $n-1$ *sub-constraints*. In this project we have opted for considering the coordinates of the vertices of our grid graph as coordinates and to compute a Euclidean Minimum Spanning Tree on all the nets, and impose a *sub constraint* for each edge of the tree. This kind of spanning tree was chosen because, according to [1], it minimizes the resulting bounding boxes between pairs of points. An example

of this kind of tree with its corresponding routing can be found in figure 1. We must note that this simplification may prevent the problem to be solvable or have an optimal solution that is worse than the real one.

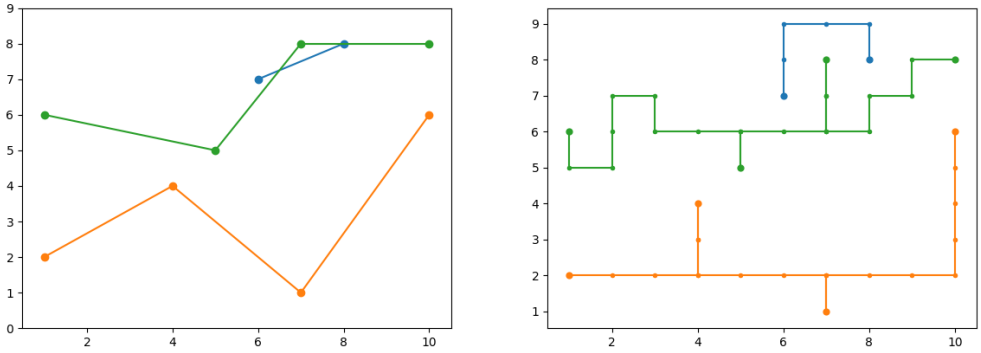

Figure 1: To the left, the computed Euclidean Minimum Spanning Tree for a given instance, to the right, a possible routing generated from this tree.

4 Basic SAT encoding

We will use the following notation and abbreviations:

- V as the set of vertices
- E as the set of edges
- N as the set of nets
- $S(n)$ as the set of subnets of some net n
- $|S_{max}|$ as the size of the biggest set of subnets $S(n)$
- $edg(s)$ as the set of edges of some vertex s such that they contain s
- $adj(v)$ the set of edges consisting of v and some u from the adjacent vertices of u

This sat encoding uses a variable hierarchy of this form:

- X_e = Edge e is used by some net
- $X_{e,n}$ = Edge e is used by net n
- $X_{e,n,s}$ = Edge e is used by subnet s of net n

The first constraint to impose is derived from the meaning of the variables

$$X_{e,n,s} \implies X_{e,n} \implies X_e \quad \forall e \in E, n \in N \quad \forall s \in S(n)$$

The other direction of the implications can be ignored, as they will not affect the correctness of our solutions, and can make the resulting formula too big.
Each edge can be used by at most one net. We must note that this does not prevent to have two subnets using the same edge

$$\sum_{n \in N} (X_{e,n}) \leq 1 \quad \forall e \in E$$

In order to guarantee that a sequence of edges will be generated for each subnet, we need to force the endpoints of the subnet to have exactly one set edge

$$\sum_{e \in edg(u)} X_{e,n,s} = 1 \quad \forall n \in N \forall s \in S(n) \forall u \in endpoints(s)$$

Non-endpoint vertices are forced to have either zero or two neighbors at the same time

$$\sum_{e \in adj(v)} X_{e,n,s} = 0 \vee \sum_{e \in adj(v)} X_{e,n,s} = 2 \quad \forall n \in N \forall s \in S(n) \forall v \notin endpoints(s)$$

These two constraints will force subnets (and, therefore, nets) to be connected. However, they not prevent random, unnecessary cycles to appear, as we can see in figure 2. It is not necessary to explicitly prevent them to appear because we can always delete them manually and wire length optimization will implicitly make them disappear, too.
A vertex cannot have more than one net passing through it. This can be reformulated as given an edge $e = (u, v)$, if some variable X_{e,n_1} is set, then no variable X_{e',n_2} can be set to true, where e' is any edge different to e_1 such that e and e' have one common vertex $n_2 \neq n_1$

$$X_{e_1,n_1} \implies \overline{X_{e_2,n_2}} \quad \forall e_1, e_2 \in E \times E : e_1 \neq e_2 \wedge \text{common}(e_1, e_2) \quad \forall n_1, n_2 \in N \times N : n_1 \neq n_2$$

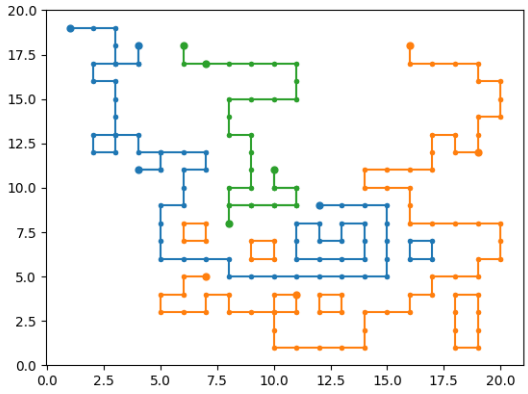

Figure 2: A solution with extra, unnecesary cycles

5 Wire length optimization

A desirable property for routings is that they minimize the total wire length. In our version of the problem, we will understand total wire length as the total amount of edges used by some net. That is, we are going to minimize the function $\sum_{e \in E} X_e$. Given that SAT has no direct mechanism to deal with objective functions, we will simply perform repeated calls to the SAT solver with an *at-most-k* constraint over the variables of the form X_e. More precisely, if we know that a solution of k edges exists, we will encode the constraint $AMO(k-1, X_e)$ and check if the problem is still solvable. A solution of cost k is optimal (at least for the chosen EMST) if there is no solution of cost $k-1$ or less.

6 Additional optimizations

This problem easily becomes intractable with big instances and/or instances with many nets. Some things can be done to *delay* the intractability. One of these things is to, given two endpoints u and v, restrict the set of candidate edges to their bounding box plus some small offset. That is, all edges (p, q) such that either p or q are *too far* from the bounding box formed by u and v are immediately discarded. Discarding an edge means that a unit clause that negates this edge will be added to the resulting formula. This helps in two things: it makes bigger problems tractable, and makes the SAT solver find smaller initial solutions. Our implementation includes this optimization, preventing variables $X_{e,n,s}$ that have vertices at distance more than 3 to any of the borders of the considered bounding box to be used.

7 Implementation details

Our implementation[1] is mainly written in C++, although there are some auxiliary Python scripts for plotting and other minor purposes. We have used picoSAT[2] as our SAT-Solver and the C++ library PBLib [3] to generate the SAT constraints. This library especially helps with the constraint that emulates the objective function when optimizing because it is capable to implement it in an incremental way, helping us to keep the formula size relatively small. Given that our implementation is more a toy than other thing, it includes a 300 second timeout for our solver that, if triggered, stops calling the solver and outputs the last best solution it found. Successive calls are performed on the same `Solver` instance, which means that previously learnt clauses are going to be reused.
The constraint that imposes that non-endpoint vertices have either zero or two has been implemented as combination of an at-most-two constraint plus a not-exactly-one. A not-exactly-one constraint on a set of n variables can be imple-

[1] https://github.com/srgrr/CellRouter
[2] http://fmv.jku.at/picosat/
[3] http://tools.computational-logic.org/content/pblib.php

mented with n clauses such that the ith clause has the form $l_1 \vee ... \vee \overline{l_i} \vee ... \vee l_n$ (which is what we get after applying De Morgan's laws to $\overline{(\overline{l_1} \wedge ... \wedge l_i \wedge ... \wedge \overline{l_n})}$).
All other constraints have been implemented with PBLib.
Problem instances are represented as text files with the following format:

```
numdims numnets dim[1] dim[2] ... dim[numdims]
numpoints[1]
points[1][1] points[1][2] ... points[1][numdims]
...
points[numpoints[1]][1] ... points[numpoints[1]][numdims] ...
...
numpoints[numnets]
....
```

A instance is parsed, and subnets are computed using the Euclidean Minimum Spanning Tree method. In case there is more than one EMST, a random one will be chosen, as the tie-breaker for edges with the same weight is, literally, `random() % 2`. Once we have defined the subnets, a basic SAT encoding is computed, and fed to the SAT Solver. If a solution is found, then the optimization process A summary this process can be found in figure 3. A more detailed description of the optimization process can be found in algorithm 1.

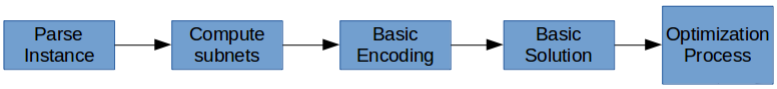

Figure 3: A summary of the whole routing process

8 Results

As we can see in figure 4, the behavior of the optimization phase is the most usual one in optimization problems: a solution with a cost near to the optimum is easily found, and most of the time is spent making minor improvements. This fact may serve us as a hint to add a timeout to our solver in order to avoid wasting huge amounts of time for very small improvements.
An interesting result is that it is easier for the solver to find solutions in dense instances than in sparse ones. This seems counterintuitive: more nets and vertices mean more variables and constraints, and therefore bigger formulas, which should be harder for the solver. However, we must take into account that more nets and vertices while maintaining the grid size also means less possible choices for each configuration. We believe that this is the reason that makes the solver work better on denser instances. An example of this behavior can be found in table 1. The dataset of this table is the same as in figure 5. This table also shows that the formula size grows very fast.

Algorithm 1 Optimization process

```
1:  procedure OptimizeRouting(E : A basic encoding for some routing instance)
2:      S ← solve(E)
3:      if S ≠ ∅ then
4:          removeUnnecessaryCycles(S)
5:          k ← getUsedEdges(S) −1
6:          proceed = true
7:          while proceed do
8:              addAtMostK(k, E)
9:              T ← solve(E)
10:             if T ≠ ∅ then
11:                 removeUnnecesaryCycles(T)
12:                 S ← T
13:                 k ← getUsedEdges(S) −1
14:             else
15:                 proceed ← false
16:         return S
17:     else
18:         return ∅
```

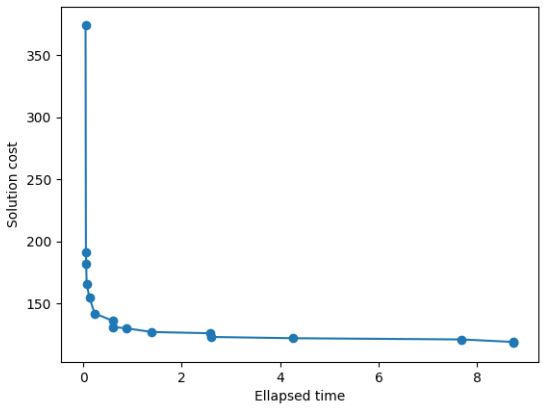

Figure 4: Evolution of the cost of the best found solution during time

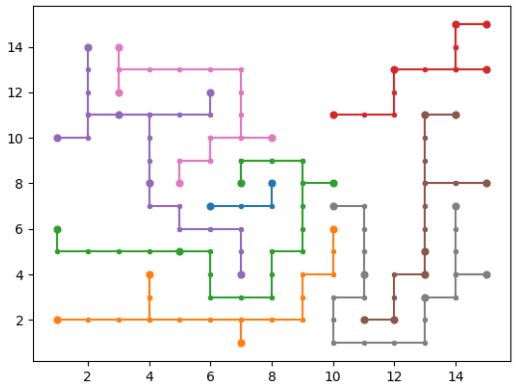

Figure 5: An optimal solution (for the chosen EMSTs) found by our solver for an instance consisting of a 15×15 grid with 8 nets of 2, 4, 4, 5, 6, 7, 4, and 5 vertices

Nets	Vars	Clauses	Time to reach optimum
1	2262	3757	0.000s
2	6966	19284	2.028s
3	11658	39932	10.461s
4	17796	69195	11.940s
5	25752	106308	36.040s
6	34716	151624	677.148s
7	39822	191448	68.465s
8	46362	239392	23.807s

Table 1: Formula size and required time to reach optimal solution as a function of the number of nets for a fixed grid

9 Conclusions and future work

This approach to the cell routing problem only works for small instances, given the nature of the SAT problem.
Our implementation only includes a subset of the features described in [1]. Real life instances may include additional constraints of very different nature. For example, it may be necessary to prevent wires from some net n_1 to be too near to wires from some other net n_2. This kind of constraints are called *design rules*, and our implementation cannot support them.
Another interesting feature we did not include in our final implementation was the highway variables. A highway variable H_i is a shortcut variable to a set of variables $X_{e,n,s}$ for some fixed $n \in N$ and for some $s \in S$ that form a path

between the two endpoints of the subnet s. The most common implementation consist of adding a set of implications $H_i \implies X_{e,n,s}$, and sometimes a constraint that imposes that at least one (exactly in practice) of all the H_i that belong to some given subnet n. This last constraint is imposed because otherwise the solver will tend to ignore the H_i variables, as would be not necessary to take them into account when constructing a solution. These variables and clauses may help to find solutions in a shorter amount of time.
There are also other aspects of this project that give room for some improvement, as implementation and design decisions.

References

[1] J. Cortadella, J. Petit, S. Gómez, and F. Moll, "A boolean rule-based approach for manufacturability-aware cell routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, pp. 409–422, March 2014.