Routing for analog chip designs at NXP Semiconductors

Marjan van den Akker[*] *Theo Beelen*[†] *Rob H. Bisseling*[*]
Bas Fagginger Auer[*] *Frederik von Heymann*[‡] *Tobias Müller*[§]
Joost Rommes[†]

Abstract

During the study week 2011 we worked on the question of how to automate certain aspects of the design of analog chips. Here we focused on the task of connecting different blocks with electrical wiring, which is particularly tedious to do by hand. For digital chips there is a wealth of research available for this, as in this situation the amount of blocks makes it hopeless to do the design by hand. Hence, we set our task to finding solutions that are based on the previous research, as well as being tailored to the specific setting given by NXP.

This resulted in an heuristic approach, which we presented at the end of the week in the form of a protoype tool. In this report we give a detailed account of the ideas we used, and describe possibilities to extend the approach.

1 Introduction

1.1 NXP Semiconductors

NXP Semiconductors N.V. (Nasdaq: NXPI) is a global semiconductor company and a long-standing supplier in the industry, with over 50 years of innovation and operating history. The company provides high-performance electronic chips to its customers, and produces these building on its expertise in the areas of RF, analog and digital circuits, power management, and security. These innovations are used in a wide range of automotive, identification, wireless infrastructure, lighting, industrial, mobile, consumer and computing applications. Headquartered in Europe, the company has approximately 28,000 employees working in more than 25 countries and posted sales of USD 3.8 billion in 2009.

1.2 Place and route for analog designs

The increasing demand for smaller, faster, and multi-functional electronic devices such as smart phones is one of the driving forces in the semiconductor industry. Combined with requirements on power usage, sustainability, and wireless functionality this

[*] Utrecht University
[†] NXP Semiconductors Netherlands N.V.
[‡] Delft University of Technology
[§] Centrum Wiskunde & Informatica, Amsterdam

is generating challenges in several domains. During the design of the layout of a chip, which is a representation of the chip in shapes in its physical layers (silicon, oxide, metal), one of the challenges is to place and route (wire) the circuit components in an optimal way. Aspects that define optimality may vary per design/application and are typically related to the area (convex hull) of the chip, the total wire length, and the unintended side-effects caused by the wires (crosstalk, i.e., electrical fields between wires). The place and route problem is further complicated by design rules and geometrical constraints.

The place and route problem has been studied for many years and mature solutions are available for digital designs [1, 4, 10, 11, 12, 15, 16, 17, 21], which typically consist of (almost) equally sized blocks and predefined routing channels. For analog designs, the main interest of NXP, the place and route problem is more complicated because blocks can vary in size and aspect ratio and may even overlap (so that there is no need for explicit routing), and because routing channels are typically not available but defined by the placement. Also, while typically several layers (metal) are available to route in, often one would like to limit this to as few layers as possible, and in some cases routing is even restricted to a single layer. Other objectives one can think of are to minimize the wire length and the number of turns in the wires, and typical constraints are that wires are either horizontal or vertical (only 90-degree turns) and should be at a certain minimum distance from each other. Furthermore, it is desirable to have a routing algorithm that is robust with respect to (small) changes in the layout, so that it can be used in parametrized designs to update the routing automatically when parameters change. This allows designers to quickly explore different physical design variants.

The challenge NXP set for the study week of 2011 was to develop an algorithm that, given a number of circuit blocks and their interconnections, computes an optimal layout including placement and routing.

1.3 Outline

In Section 2 we describe the precise task we discussed during the study week, and we give a detailed account of the partial results we were able to achieve, including the prototype of a tool that we think can simplify the work of chip designers. In Section 3 we give an overview of possible extensions and improvements of one particular aspect of our algorithm, which we believe could be a computational bottleneck for larger instances; Section 4 outlines a slightly more sophisticated algorithm than the one in Section 2, which also has the benefit of giving us lower bounds on the quality of the solutions it produces. We conclude the report with a summary of our results and an estimation of the success in terms of the original challenge.

2 Heuristic Method and Implementation

As illustration of the ideas developed for NXP during the study week, we implemented a heuristic for solving the problem as a `C++` program. An early version of this program was demonstrated during the final presentation session of the study week as well as a later version during a visit to NXP in Eindhoven (Figure 1).

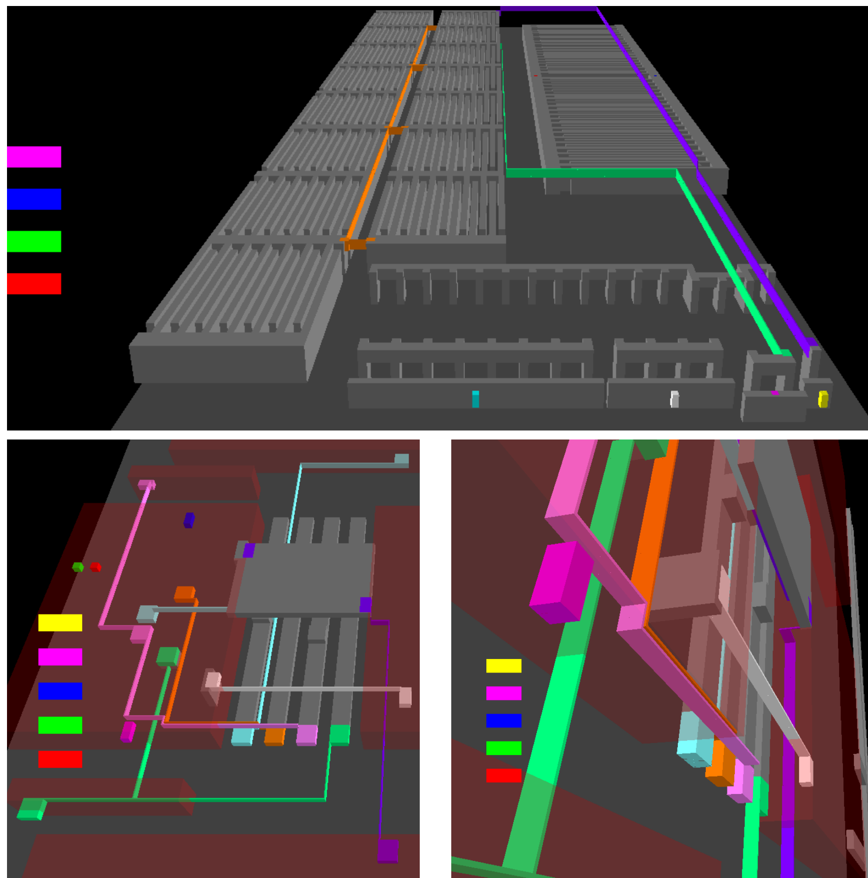

Figure 1: Screenshots from the demonstration at NXP, Eindhoven. The top picture shows generated routing for the `tunedCap` circuit and the bottom two pictures show routing for `difInvStage`. Both these circuits consist of 5 layers.

Because of the time constraints of the study week we chose to focus only on the routing aspect of NXP's problem: the program should be able to import a layout specified in NXP's circuit design software (Figure 2), where all the components have already been placed on the chip, and export routing in the form of wires connecting these components (thin wires in Figure 1). Wires are formed on the circuit by depositing metal in the production process.

The circuits are produced layer-by-layer, so we know beforehand that there is an $l \in \mathbb{N}$ specifying the number of layers ($l = 5$ in Figure 1) and a bounding box of the entire circuit board within which all components and wiring should be contained. We do not want to short-circuit different components on the circuit board by placing metal at the wrong places, therefore we receive for each layer a number of solid rectangles, in which no metal can be deposited (solid gray blocks in Figure 1). These rectangles are characterized by their lower-left and upper-right corners in $\mathbb{R}^2$, as well as the number of the layer in which they are present. The components in the circuit need to

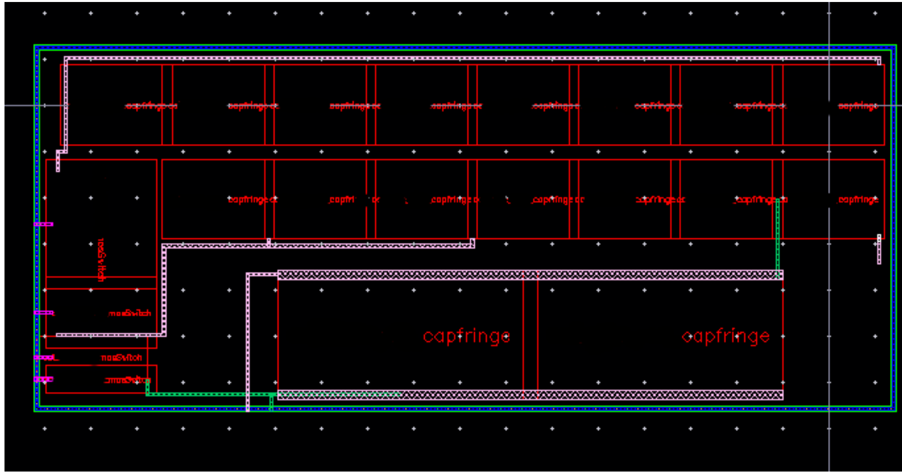

Figure 2: Circuit design software of NXP showing the layout of `tunedCap`.

be connected by conducting metal: we are given a list of *pins*, conducting rectangles belonging to a certain layer and having a certain color (the colored blocks in Figure 1). All pins sharing the same color should be connected by metal. To prevent accidental short-circuiting and interference, wires should have a minimum distance from each other; these minima can be different for different layers, so they are represented by a vector $\delta = (\delta_1, \ldots, \delta_l) \in \mathbb{R}^l_{>0}$. It is important to note that wires are not restricted to a single layer: wires can make jumps to other layers by *vias*.

Our program now needs to find out, given these parameters, where to deposit metal in the circuit such that

- for each pair of pins sharing the same color there exists a continuous metal path connecting the pins;
- the distance between two bits of deposited metal in the same layer k that are connecting pins with different colors is always $\geq \delta_k$;
- no metal is deposited in the solid areas and no metal is deposited outside of the circuit board.

These are hard constraints in the sense that any solution which does not satisfy all these criteria is unacceptable.

2.1 Optimization

If we only took the hard constraints into account, we could end up with very unfavorable and costly solutions (e.g., paths that needlessly use a lot of metal). Therefore we will, in addition to satisfying the hard constraints, try to minimize the following quantities:

- the total amount of metal that needs to be deposited (as depositing metal costs money and long paths increase the resistance, which increases power consumption);

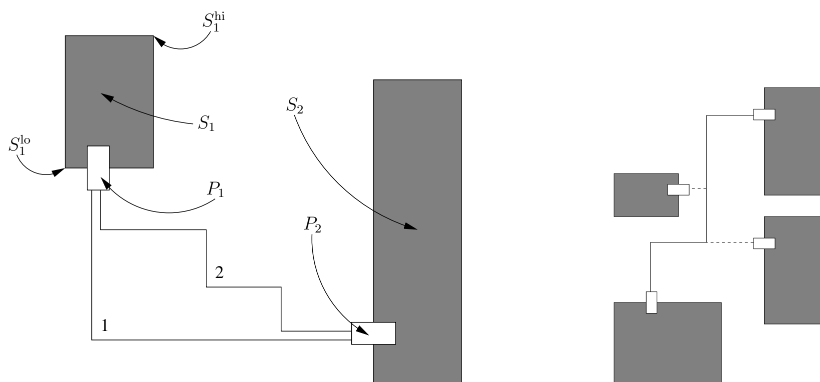

Figure 3: Left: two wires that use the same amount of metal, but turn a different number of corners. The blocks S_1 and S_2 are solid area bounding boxes, P_1 and P_2 are pin bounding boxes, and S_1^{lo} and S_1^{hi} are the lower-left and upper-right corners of S_1. Right: Construction of a Steiner tree (see Section 3) by first creating the shortest path between the pins farthest away from each other (solid line) and then connecting the remaining pins to this path (dashed line).

- the number of corners in each of the paths (as corners introduce interference and are more susceptible to production errors, compare paths 1 and 2 in Figure 3);

- the number of jumps between different layers (as producing vias is expensive).

Depending on the specific circuit for which a routing needs to be generated, there could be additional objectives, such as

- limiting the amount of metal deposited in a specific layer;

- discouraging paths from lying on top of each other;

- encouraging metal deposition in certain areas of a certain layer, while discouraging it in others;

- ...

Hence the optimization criteria should be as customizable and flexible for the circuit designers as possible.

The heuristic (Algorithm 12) we use to solve the problem described above is based primarily on Dijkstra's shortest path finding algorithm [6] (in our implementation we improved performance by using the A* algorithm [9]; compare [11] and, for possible improvements, [17]). Dijkstra's algorithm is particularly useful for incorporating flexible optimization goals by viewing the minimization of total distance as the minimization of a more abstract cost function in which the goals of the chip designers are incorporated. Hence we look for *cheapest paths* with respect to this cost function (an example of which is given in Algorithm 13), using Dijkstra's algorithm.

2.2 Dijkstra's algorithm

Dijkstra's algorithm [6] computes the shortest paths between a single source vertex s and all vertices v in a directed graph $G = (V, E)$ with non-negative weights on the edges representing distances or more general costs; for our routing problem, the weights represent costs of various types. The algorithm works as follows. For each vertex v, a temporary best distance $d(v)$ is maintained. The distance $d(v)$ is lowered each time a shorter path to v is encountered. The predecessor of v in that path is then registered, thus storing the shortest path, and not only its cost. At some stage in the computation, the distance reaches its final value, the shortest distance to v. Then, v is included in the set D of finished vertices. Initially, $d(s) = 0$, $d(v) = \infty$ for $v \neq s$, and $D = \emptyset$. It can be shown that the distance attains its final value once $d(v) = \min_{w \in V-D} d(w)$. Algorithm 11 presents Dijkstra's algorithm.

Algorithm 11 Dijkstra's Algorithm.

Input: Directed graph (V, E) with costs $c(v, w) \geq 0$ for every edge $(v, w) \in E$, source vertex $s \in V$.

Output: The cost $d(v)$ for reaching vertex v from s, and a predecessor in a shortest path to v, for every $v \in V$.

1: **for all** $v \in V$ **do**
2: $\quad d(v) \leftarrow \infty$;
3: $\quad \mathit{pred}(v) \leftarrow v$;
4: $d(s) \leftarrow 0$;
5: $D \leftarrow \emptyset$;
6: **while** $D \neq V$ **do**
7: $\quad v \leftarrow \text{argmin}\{d(w) \ : \ w \in V - D\}$;
8: $\quad D \leftarrow D \cup \{v\}$;
9: $\quad$ **for all** w with $(v, w) \in E$ **do**
10: $\qquad$ **if** $d(v) + c(v, w) < d(w)$ **then**
11: $\qquad\quad d(w) \leftarrow d(v) + c(v, w)$;
12: $\qquad\quad \mathit{pred}(w) \leftarrow v$;

The A* algorithm is a more efficient variant of Dijkstra's algorithm, which makes use of knowledge about a target vertex t that we want to reach, such as a lower bound $l(v)$ on the cost of reaching t. In our case, we will use the minimum distance to be covered on a three dimensional grid (see the next section) as a lower bound, ignoring other costs. The bound is called *consistent* if $l(v) \leq c(v, w) + l(w)$ for all $(v, w) \in E$. For our problem, the bound is consistent because $c(v, w)$ includes the distance cost of routing from v to w. Algorithm 11 can be changed from Dijkstra to A* by writing $t \notin D$ instead of $D \neq V$ on line 6 and $d(w) + l(w)$ instead of $d(w)$ on line 7.

2.3 Implementation

Algorithm 12 outlines the heuristic employed by the prototype tool. In the algorithm we deposit metal of certain colors in order to be able to differentiate between wires connecting different sets of pins. Two deposited wires that do not share the same color should always be separated by the minimum separation distance as specified

by δ. For bounding boxes we use the following notation: if B is a bounding box, then $B^{\mathrm{lo}}, B^{\mathrm{hi}} \in \mathbb{R}^2$ are the lower-left and upper-right corner of the bounding box, respectively. The layer where the bounding box is present is indicated by $B^{\mathrm{layer}} \in \mathbb{N}$.

Algorithm 12 Discretization and Heuristic Wiring.

Input: Number of layers $l \in \mathbb{N}$, minimum wire separation $\delta \in \mathbb{R}^l_{>0}$, circuit board bounding box B, solid area bounding boxes $S_1, S_2, \ldots$, pin bounding boxes $P_1, P_2, \ldots$.

Output: Discretized grid in which the metal depositions have been marked.

1: Let $\rho \leftarrow$ grid spacing based on δ or a specified resolution;
2: Create three-dimensional grid G of $\lceil (B^{\mathrm{hi}} - B^{\mathrm{lo}})/\rho \rceil$ by l cells;
3: **for all** solid area bounding boxes S **do**
4: Mark all cells lying between $\lfloor (S^{\mathrm{lo}} - B^{\mathrm{lo}})/\rho \rfloor$ and $\lceil (S^{\mathrm{hi}} - B^{\mathrm{lo}})/\rho \rceil$ in layer S^{layer} as inaccessible for all colors;
5: **for all** pins P **do**
6: Mark all cells lying between $\lfloor (P^{\mathrm{lo}} - B^{\mathrm{lo}})/\rho \rfloor$ and $\lceil (P^{\mathrm{hi}} - B^{\mathrm{lo}})/\rho \rceil$ in layer P^{layer} as metal with color P^{color};
7: Mark all cells within distance $\lceil \delta_{P^{\mathrm{layer}}}/\rho \rceil$ of P as inaccessible, except for color P^{color};
8: Sort pins by their color into nets and then the nets by increasing bounding box volume;
9: **for all** nets $\mathcal{P}$ in which all pins share the same color $\mathcal{P}^{\mathrm{color}}$ **do**
10: Find $P_1, P_2 \in \mathcal{P}$ such that the distance between the centers of P_1 and P_2 is maximal;
11: Create a cheapest path $T \subseteq G$ from P_1 to P_2 traversing only cells accessible for color $\mathcal{P}^{\mathrm{color}}$;
12: Similarly create cheapest paths from T to all $P \in \mathcal{P}$ not connected to T and add these paths to T;
13: **for all** cells $c \in T$ **do**
14: Mark c as metal with color $\mathcal{P}^{\mathrm{color}}$;
15: Mark all cells in layer c^z with distance $\leq \lceil \delta_{c^z}/\rho \rceil$ to c as inaccessible, except for color $\mathcal{P}^{\mathrm{color}}$;
16: Output G;

To simplify the problem we first discretize it (line 2) to a regular three-dimensional grid G, with l layers and spacing ρ within each layer. G will be the graph in which we perform Dijkstra's path-finding algorithm. Cells $c \in G$ have three coordinates $(c^x, c^y, c^z) \in \mathbb{Z}^3$, where $1 \leq c^z \leq l$ is the layer in which the cell resides. To ensure that we never deposit metal in solid regions, we mark these in G first by discretizing the bounding boxes and flagging the cells contained in them as inaccessible for metal from any pin. We then proceed at line 5 to add all pins, marking the cells contained in them as metal of the pin's color and ensuring that no metal belonging to pins with a different color can be deposited near the pin (as this could violate the minimum separation criterion).

After the solid regions and pins have been marked, we generate paths connecting all the pins sharing the same color at line 8. First we cluster the pins together such that we have *nets* of pins all sharing the same color. Note that the algorithm will yield

different results if we connect the pins contained in these nets in a different order, hence we will fix the ordering by sorting the nets of pins of the same color by the volume of the bounding box containing the pins. This idea originates from the fact that connecting all the short paths first will lead to less conflicts between paths later than first connecting all the long paths (which could potentially cut off short paths).

Then we will connect, for each net of pins sharing the same color, the pins belonging to this net. If there are at most two pins in a net, we can directly use Dijkstra's algorithm to find the cheapest path connecting them. However, if we have more than two pins, we need to generate a *Steiner tree* (see Section 3) connecting all of them. In our heuristic, this is done by first creating a path between the two pins that are farthest apart, and then using this long path as a 'trunk' to which the remaining, unconnected, pins connect as branches via Dijkstra's algorithm (see the right panel of Figure 3). A path can only be created along cells that are accessible for the particular color of the current path; this to ensure that the minimum separation distance is always maintained.

Algorithm 13 Routing Cost Function for Dijkstra's Algorithm.

Input: Neighboring cells c_{-1}, c_0, and c_1 in the grid G where c_{-1} is the predecessor of c_0.

Output: The cost k to use cell c_1 to continue the path.

1: Initialize $k \leftarrow 0$;
2: **if** c_1 is not marked as metal **then**
3: We need to deposit metal: $k \leftarrow k + k^{\text{metal}}$;
4: **if** $c_1^z \neq c_0^z$ **then**
5: We need to create a via: $k \leftarrow k + k^{\text{via}}$;
6: **if** $\|c_1 - c_{-1}\|^2 = 2$ **then**
7: We turn a corner if we continue this way: $k \leftarrow k + k^{\text{corner}}$;
8: ...
9: Output k;

Algorithm 13 gives a simple example cost function for Dijkstra's algorithm where we consider continuing an existing path going through cells $\ldots \to c_{-1} \to c_0$ to c_0's neighbor c_1 (neighbor in the sense that $\|c_1 - c_0\| = 1$). The cost can be influenced by varying three parameters: k^{metal}, k^{corner}, and k^{via}. This allows the designer to indicate whether he finds minimizing the length of the wires (increasing k^{metal}), minimizing the number of corners (increasing k^{corner}), or minimizing the number of vias (increasing k^{via}) more important. In Figure 1 the colored bars on the left are similar cost modifiers, from bottom to top: cost to deposit metal, cost to turn a corner, cost to create a via, cost to run over an existing wire, and cost to run over a solid block. By extending the cost function and permitting the designers to vary the associated weights, a number of different routing suggestions is easily obtained from the program. Note that the multiplicative factors such as k^{metal} can also be made to depend on the position of c_0 or c_1, permitting the designer to make certain layers or certain regions of layers more or less attractive for the wires to traverse; this and other costs can be added at line 8.

The program prototype has been demonstrated to circuit designers of NXP in Austria and its source code has been provided to NXP.

3 Some Ideas for Steiner Trees

As we mentioned above, if we have more than two pins in one net, we cannot just use Dijkstra's algorithm to connect them. If we consider the pure problem of only one set of pins to be connected, this is an instance of the *Steiner tree problem*. Given a set of vertices, called terminals (which will be our pins), and another set of vertices, called Steiner points (grid points that are not blocked), a *Steiner tree* is any tree that contains all terminals and (some) Steiner points.

This section will give a brief account of possibilities to find a good Steiner tree in a grid (with obstacles). In particular this means that here we ignore the fact that it might not be possible (depending on the layout) to achieve an optimal routing solution by considering the nets successively. We defer this discussion to the next section.

As is true for most aspects of chip design, it is a computationally hard problem to find a smallest Steiner tree [7], and there is an abundance of strategies to get reasonably good approximations in acceptable time [2, 18]; and for more algorithms, see [10, 11, 12, 15]. Here we will restrict our attention to approaches that seem appealing because of their simple implementability and their compatibility with the strategy we chose for paths.

Most algorithms we found in the literature deal with the rectilinear version of the Steiner tree problem, i.e., where all terminals and Steiner points are given in a two-dimensional rectilinear grid. As we want to find Steiner trees in a three-dimensional grid (with obstacles), these results are to be taken with some caution, although we believe that on average they are close to what is to be expected for our setup. The idea presented in the previous section can be seen as a simplification of ideas from [10], where it is stated that we will get a tree that is at most a factor of $3/2$ away from a minimal Steiner tree, and on average much closer to it.

It should, however, be mentioned that the authors in [10] consider nets that include a source. In such a case it is usually desirable to minimize the distance of the other pins to the source, whereas our focus is more on the total (weighted) wire-length used in the tree. In the former case one typically gets star-shaped trees, whereas in the latter a caterpillar-structure is more likely.

As long as the net contains at most 6 pins, a *rectilinear* Steiner minimal tree in an obstacle-free grid can be found by going through all permutations of the order of pins, connecting them in these orders in the way described in Section 2.3. For larger nets, this approach will not always produce an optimal tree (not even in this special case of obstacle-free rectilinear trees), but there are good approximations available [14]. Still restricted to the rectilinear case, and given that the instances are typically relatively small in the case of analog chip design, one can also consider computing a truly minimal Steiner tree, using, e.g., GeoSteiner [19, 20]. For more information, the paper by Hentschke et al. [11] is a good survey on exact results and approximations for rectilinear Steiner trees, taking into account different priorities of optimization.

4 Integer Programming and Approximations

In this section we propose a mathematically more rigorous approach, which extends our heuristic from Section 2 but was too elaborate to incorporate in the prototype tool during the short time span of the study week.

This is a column generation approach (which can also be found in [10] in somewhat similar form), an approach that has successfully been applied to different real-world optimization problems (see [5]). While we give all formulations in terms of paths between pairs of pins, they work (with one limitation) also for larger nets.

4.1 Column Generation

Let us start with decomposing the problem into two levels. The top (or master) level is the following: given all nets and for each a pool of paths connecting them (possibly all such paths), we want to select one path for each net, such that the resulting wiring satisfies our hard constraints and is of high quality with respect to the optimization goals.

This is an integer linear program (see precise formulation below), which is known to be computationally hard to solve to optimality [13]. And while the analog design is not excessively large, here we get a huge number of variables: one for each pair of a net and a possible path for this net.

We note here that one can reduce the size of the graph involved by using less vertices and introducing different weights on the edges depending of the capacity of the space between the vertices. One could construct such a graph with a *Generalized Voronoi Diagram*. This method has been used for path planning in games (see, e.g., [8]), where characters have to move through a landscape and avoid obstacles. In chip design, electrons move through the wires and avoid components (except for the locations of their pins). We can define a Generalized Voronoi Diagram around the components. The edges in this diagram result in a collection of corridors going through the central areas of the open space between the components. In this way, they result in edges for our routing network. For each pin we add an edge representing the shortest line connecting that pin to the network. In general, this network will be smaller in terms of vertices and edges than the grid which is attractive from a computational point of view. Observe that in this setup multiple wires can go through an edge or vertex. One drawback of this is, however, that the solution is not yet a complete description of a physical layout. But one could first determine the corridor a net will use, and then solve sub-problems in this smaller grid. There are some more technicalities to be considered here, so we will just leave it as a suggestion for further considerations.

In any case, by far not all possible paths are of interest for us. In fact, most are unnecessarily long or could even include loops.

Thus, we restrict the master problem to a small pool of paths and introduce a second level, where we try to find good paths outside the pool (using the dual solution from the restricted problem) to improve the routing.

4.2 Formulation

Let (V, E) be the underlying network, i.e., the grid of Section 2 or an alternative network. We label the nets by $1, \dots, m$, and denote with $\mathcal{P}_i$ the set of all possible paths for net i.

Define further $l_{ip} = \sum_{e \in p} l_e$ as the length of path $p \in \mathcal{P}_i$ (the lengths l_e are the weights we give the edges, depending on the optimization goals, e.g. according to Algorithm 13 in Section 2.3), let

$$a_{ep} = \begin{cases} 1 & \text{if edge } e \text{ is in path } p; \\ 0 & \text{otherwise,} \end{cases}$$

and define a_{vp} in the same fashion for the vertices that are not in one of the nets. Finally, we define the edge capacity c_e^{edge} as the maximum number of wires routed over edge e, and similarly c_v^{vertex} as the maximum number of wires that are allowed to cross through vertex v. For the grid used in Section 2.3, all values c_e^{edge} and c_v^{vertex} are set to 1.

Our variables are $x_{ip} \in \{0, 1\}$ where $x_{ip} = 1$ indicates that path $p \in \mathcal{P}_i$ is selected for net i. Then our master Integer Linear Program (master ILP) is

$$\begin{array}{llcll} \min & \displaystyle\sum_{i=1}^{m} \sum_{p \in \mathcal{P}_i} l_{ip} x_{ip} & & & \\ \text{s.t.} & \displaystyle\sum_{p \in \mathcal{P}_i} x_{ip} & = & 1 & (i = 1, \ldots, m), \quad (1) \\ & \displaystyle\sum_{i=1}^{m} \sum_{p \in \mathcal{P}_i} a_{ep} x_{ip} & \leq & c_e^{\text{edge}} & \forall e, \quad (2) \\ & \displaystyle\sum_{i=1}^{m} \sum_{p \in \mathcal{P}_i} a_{vp} x_{ip} & \leq & c_v^{\text{vertex}} & \forall v \text{ not in a net}, \quad (3) \\ & x_{ip} & \in & \{0, 1\}. & \quad (4) \end{array}$$

Constraints (1) ensure that exactly one path is selected for every net. Constraints (2) and (3) ensure that the edge and vertex capacities are respected.

To deal with the large number of variables, we are going to solve the problem by column generation. We start with a limited subset of the variables and solve the LP-relaxation (i.e., $x_{ip} \geq 0$) for this subset only. For example, we could use the solutions from Section 2. This way we obtain the restricted master LP. Then we solve the pricing problem, i.e., we look for variables that are not yet included in the restricted master LP and can improve the solution.

If such variables are found, they are added to the restricted master LP, it is solved again, after which pricing is performed, and so on. If pricing does not find any new variables anymore, we know that the master LP has been solved to optimality.

Unfortunately, this solution is not likely to be an integer solution. We discuss methods for finding an integer solution in Section 4.4.

4.3 The Pricing Problem

From the theory of linear programming it is well-known that for a minimization problem increasing the value of a variable will improve the current solution if and only if its reduced cost is negative. The pricing problem then boils down to finding the variable with minimum reduced cost.

Let λ_i, π_e, and σ_v be the dual variables of the net, edge capacity, and vertex capacity constraints, respectively. Now the reduced cost of x_{ip} is given by

$$l_{ip} - \lambda_i - \sum_e a_{ep}\pi_e - \sum_v a_{vp}\sigma_v.$$

We are going to solve the pricing problems for each net separately. Note that a_{ep} and a_{vp} are the decision variables and that they have to form a path connecting the net. Clearly, the values a_{vp} (the vertices on the path) are determined by the values a_{ep} (the edges on the path). For each vertex v on the path we have cost $-\sigma_v$. Since on a path each vertex has degree 2 (except for the first and the last one, but these are in a net), we can remove the variables a_{vp} from the pricing problem by adding cost $-\frac{1}{2}\sigma_v$ to each edge adjacent to v. The reduced cost is now given by

$$\sum_e a_{ep}\left(l_e - \pi_e - \sum_{v\in e}\frac{1}{2}\sigma_v\right) - \lambda_i.$$

The pricing problem for net i thus reduces to finding a shortest path, where the edge lengths are modified by the dual variables.

From the theory of linear programming we know that $\pi_e \leq 0$ and $\sigma_v \leq 0$. Therefore, the cost of the edges are non-negative and the pricing problem can be solved by Dijkstra's algorithm.

For a net i with more than two pins, the $\mathcal{P}_i$ are all possible Steiner trees, and hence at this point of the algorithm we are looking for a Steiner minimal tree. To save CPU-time, we can approximate the minimal Steiner tree, and only determine the optimal Steiner tree in case the approximation does not find a solution with negative reduced cost. If we decide to only approximate, we still have a high probability to find a very good solution to the LP-relaxation.

4.4 Integer Solutions

As we mentioned in the introduction, this approach also gives us a measure of the quality of the routing solution, because the solution of the LP-relaxation (which we solved to optimality) is a lower bound on the costs of the routing. To actually produce an integer solution, we can apply different strategies, which are only shortly mentioned here.

- We can perform branch-and-price, i.e., apply branching and proceeding with column generation (see, e.g., [3]). This will not lead us away from optimality.
- We can apply an ILP solver to the restricted master problem, which in all likelihood will have a manageable amount of variables.
- We can apply heuristics based on the LP solution. For example, we can first fix all paths that were selected with value 1. Then we proceed by selecting one by one paths with maximal fractional value that fit (in term of vertex and edge capacities) with the set of paths that were already selected. If we end with a solution with unconnected pins, we complete the solution using the heuristic from Section 2.

As was the case for the heuristic for the prototype tool, this method can also be applied if part of the routing is fixed, as this is nothing else than removing certain edges from our grid.

Comparing the ILP method of this section with the heuristic of Section 2, we note as advantages of ILP that it provides quality guarantees, it can be combined with the heuristic, and that it can be used on a smaller graph than the grid in the heuristic, which may save computation time. An advantage of the heuristic is that it is easy to implement, and that it often gives a fast and satisfactory solution. Summarizing, we see this ILP method as a natural extension of the heuristic, to be implemented if the heuristic gets too slow, if the solutions don't seem adequate, or to determine a quality measure of the heuristic.

5 Conclusion

Given the limited duration of the study week and the complexity of the problems connected to chip design, we decided to focus on one aspect which we felt could ease the work of the chip designers at NXP. Hence we tried to find an algorithm for connecting nets in a predefined layout, which is as flexible and customizable as possible, facilitating the designer to choose priorities between the different aspects that should be optimized, which is stable under local changes (if needed), and which gives reliably the same answers for identical inputs. We were able to present our algorithm in the form of a prototype tool (see Section 2.3) which showcases all these aspects. Furthermore, we describe a more generalized approach which provides a quality measure of the solution and improves our strategy to deal with larger inputs (see Section 4.1).

The study week permitted us to get acquainted with a large branch of new and interesting mathematics, as well as provide NXP with a useful prototype solution (Figure 4) for their routing problem.

References

[1] C.J. Alpert, T.C. Hu, J.H. Huang, A.B. Kahng, and D. Karger. Prim-Dijkstra tradeoffs for improved performance-driven routing tree design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(7):890–896, 1995.

[2] S. Arora. Polynomial time approximation schemes for Euclidean TSP and other geometric problems. In *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science*, pages 2–11, 1996.

[3] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1996.

[4] Cid Carvalho De Souza and Celso Carneiro Ribeiro. Heuristics for the minimum rectilinear Steiner tree problem: new algorithms and a computational study. *Discrete Applied Mathematics*, 45(3):205–220, 1993.

NXP participates in challenge event with Dutch universities

The "Studygroup Mathematics with Industry" is a yearly event where academic mathematicians work for a week on problems presented by the industry. At this year's event, from January 24—28 at Vrije Universiteit Amsterdam, NXP participated by presenting a complex challenge: the automatic placement and routing in analog layouts. A team of eight PhD students and academic researchers presented a well-working prototype simulation tool by the end of the week. This event is one of many activities with universities that NXP supports and actively participates in.

A team of eight enthusiastic PhD students and academic researchers from TU Delft, Utrecht University, CWI and Groningen University worked on this problem, receiving input and feedback from Joost Rommes and Theo Beelen from NXP, and presented their results on Friday January 28. Besides valuable literature references, also a well-working prototype simulation tool was developed, based on advanced graph and optimization algorithms. This tool allows users to set real-life layout requirements (wire distance and spacing, minimize number of used layers, etc) and see results in 3D on the fly! Last but not least, their unbiased analysis of the problem lead to new insights that NXP can use to enhance the methodology.
Apart from the technical advances on the presented problems, the event was a fine opportunity to establish contacts between industry and academia, enabling to get acquainted with each other, and leading to challenging new research areas.

Figure 1: Bas Fagginger Auer (UU) explains the proposed solution for the routing problem of NXP

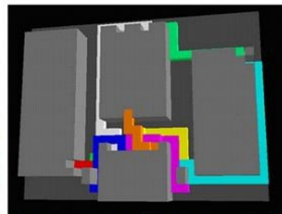

Figure 2: Impression of the interactive routing tool developed for NXP

The event was financially supported by University Relations of NXP.

For more information, please contact joost.rommes@nxp.com and theo.g.j.beelen@nxp.com

Figure 4: The work done during the study week was well received by NXP and was mentioned in their newsletter shortly after the final presentation.

[5] Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon, editors. *Column generation*, volume 5 of *GERAD 25th Anniversary Series*. Springer, New York, 2005.

[6] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959.

[7] M. R. Garey and D. S. Johnson. The Rectilinear Steiner Tree Problem is NP-Complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977.

[8] R. Geraerts. Planning short paths with clearance using explicit corridors. In *IEEE International Conference on Robotics and Automation*, pages 1997–2004, 2010.

[9] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[10] S. Held, B. Korte, D. Rautenbach, and J. Vygen. Combinatorial Optimization in VLSI design. In V. Chvátal and N. Sbihi, editors, *Combinatorial Optimization: Methods and Applications*. IOS Press, to appear.

[11] Renato F. Hentschke, Jaganathan Narasimham, Marcelo O. Johann, and Ricardo L. Reis. Maze routing Steiner trees with effective critical sink optimization. In *Proceedings of the 2007 international symposium on Physical design*, ISPD '07, pages 135–142, New York, NY, USA, 2007. ACM.

[12] Huibo Hou, Jiang Hu, and S.S. Sapatnekar. Non-Hanan routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(4):436–444, April 1999.

[13] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Plenum Press, New York, 1972.

[14] Christine R. Leverenz and Miroslaw Truszczynski. The rectilinear Steiner tree problem: algorithms and examples using permutations of the terminal set. In *Proceedings of the 37th annual Southeast regional conference (CD-ROM)*, ACM-SE 37, New York, NY, USA, 1999. ACM.

[15] Chih-Hung Liu, Shih-Yi Yuan, Sy-Yen Kuo, and Szu-Chi Wang. High-performance obstacle-avoiding rectilinear Steiner tree construction. *ACM Transactions on Design Automation of Electronic Systems*, 14:45:1–45:29, 2009.

[16] Dirk Müller, Klaus Radke, and Jens Vygen. Faster min–max resource sharing in theory and practice. *Mathematical Programming Computation*, 3:1–35, 2011. 10.1007/s12532-011-0023-y.

[17] S. Peyer, D. Rautenbach, and J. Vygen. A generalization of Dijkstra's shortest path algorithm with applications to VLSI routing. *Journal of Discrete Algorithms*, 7:377–390, 2009.

[18] J. Scott Provan. An approximation scheme for finding Steiner trees with obstacles. *SIAM J. Comput.*, 17(5):920–934, 1988.

[19] David Warme, Pawel Winter, and Martin Zachariasen. GeoSteiner [Computer Software]. www.diku.dk/hjemmesider/ansatte/martinz/geosteiner/. (version 3.1).

[20] Martin Zachariasen. Rectilinear full Steiner tree generation. *Networks*, 33(2):125–143, 1999.

[21] Hai Zhou. Efficient Steiner tree construction based on spanning graphs. In *Proceedings of the 2003 international symposium on Physical design*, ISPD '03, pages 152–157, New York, NY, USA, 2003. ACM.