Efficient Detailed Routing on Optimized Tracks

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Niko Sebastian Klewinghaus

aus

Leverkusen

Bonn, September 2021

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

Erstgutachter:	Herr Professor Dr. Jens Vygen
Zweitgutachter:	Herr Professor Dr. Dr. h.c. Bernhard Korte
Tag der Promotion:	20.12.2021
Erscheinungsjahr:	2022

If you're gonna play the game, boy
you better learn to play it right!

(Johnny Cash, original by Kenny Rogers)

Acknowledgments

I want to thank my supervisor Professor Dr. Jens Vygen for his constant support, great ideas and valuable guidance while writing this thesis.

I want to thank my colleagues at the Institute for Discrete Mathematics Dr. Markus Ahrens and Dr. Christian Schulte for the efficient joint work on BonnRouteDetailed. Many parts of BonnRouteDetailed that are mentioned in Chapter 3 are their work and without them BonnRouteDetailed would never have achieved the quality it has today.

I further want to thank Professor Dr. Dr. h.c. Bernhard Korte, Professor Dr. Stefan Hougardy and Professor Dr. Stephan Held for their support, valuable discussions and for the efficient and friendly working atmosphere at the Institute for Discrete Mathematics at the University of Bonn.

Many thanks go to Dr. Pascal Cremer, Dr. Dirk Müller, Felix Nohn, Dr. Philipp Ochsendorf, Dr. Thomas Petig, Stefan Rabenstein, Pietro Saccardi, Dr. Rudolf Scheifele, Dr. Jannik Silvanus and Mirko Speth for efficient collaboration, fruitful discussions, all the help they provided and the extremely friendly atmosphere.

I further want to specially thank Dr. Markus Ahrens for proof-reading large parts of this thesis. His comments and suggestions have been extremely valuable.

Also many thanks go to my always helpful and friendly colleagues at IBM, especially Laura Darden, Anne Heppner, Karsten Muuss, Smitha Reddy, Dr. Christian Schulte and Dr. Gustavo Téllez.

Not unmentioned should be the small things and great people that make life so much easier in times of hard work. You deserve thanks simply for your existence! The acrobats from the university acrobatics group, especially Angrit, Becci, Carsten, Felix, Imme, Irmi, Lina, Maria, Marius and Robert for their trust and enthusiasm, Alex, Maja, Sandra and Simon as well as Christine and Victoria for one of the greatest weeks that I ever experienced, the Spitzigrat in Switzerland for its very existence, Annika, Catha, Corrie and Lennart for great circus and all the others for everything!

Most importantly, I want to thank my girlfriend Maria, my sons Elrik and Lasse, my family and friends for their love and tolerance while I was busy writing this thesis.

Contents

Chapter 1

Introduction

VLSI design is the process of designing very large scale integrated circuits. This comprises both the definition of the logic behavior of the integrated circuit (called logic design) as well as the design of the physical layout (called physical design). VLSI design is a huge and extremely important field for applying discrete mathematics. Its enormous instance sizes make efficient automation inevitable and its numerous mathematically difficult problems [30], [20] make efficient automation hard (or even impossible) to obtain. Therefore the overall problem has to be divided into different subtasks and often approximation algorithms or even heuristic approaches have to be used.

Major steps of physical design are the placement of the individual circuits (called placement) and the layout of the connections between them (called routing). Input for routing are so-called nets, sets of connection points (so-called pins) that need to be connected. Routing further can be divided into two or three substeps, global routing, an optional step called track assignment and detailed routing. In many cases, wires are preferably placed on special coordinates, so-called tracks. The main contributions of this thesis concern detailed routing. We describe a novel algorithm for automatically computing track patterns for a set of different kinds of wires for detailed routing and show that it yields substantial improvements both in run time and quality of results. We further develop an axiomatic description of distance rules between wires of different nets, derive important properties and describe a highly optimized algorithm to check such diff-net rules, reducing run time by more than a factor two.

In the rest of this chapter, we briefly describe some further aspects of VLSI design and give an overview of the structure of this thesis. It is important that an integrated circuit can run correctly at the desired frequency. Although timing behavior needs to be respected already during placement and routing, it is often considered a separate step called timing optimization. In practice, some approximation of timing behavior is considered during most steps of both logic and physical design. The used timing models get more accurate during the design process and the effort spent to meet timing constraints increases. Typically, there are a number of steps dedicated exclusively to optimize remaining timing problems at the end of physical design. Furthermore, a number of special subtasks can be identified such as the design of clock networks distributing clock signals over the chip, the design of power distribution as well as the design of the primary inputs and outputs delivering signals to and from the integrated circuit. For more details on the

physical design process, we refer the reader for example to [32] or [5].

Current very large scale integrated circuits can have billions of transistors and many kilometers of wires. Therefore, they are designed in a hierarchical fashion. This means that the whole circuit is divided into smaller circuits which in turn are divided into smaller parts again. At each level of hierarchy, some connections may remain, thus creating a hierarchy of different instances containing other instances as subproblems. Each part is designed individually, meeting certain boundary conditions ensuring compatibility with the other parts. In the end, all the parts are combined together. During this hierarchical design process, it frequently happens that changes in one part of the integrated circuit require changes in other parts. Thus an iterative approach is used, designing all parts with increased level of detail and precision during each iteration.

Instances from different levels of the hierarchy tend to have different characteristics. Instances from the highest levels of hierarchy (so-called top-level instances) tend to have few but very long nets, a large area and many very wide wires. Usually, they do not contain any logic but only interconnections between their child instances. Instances at the lower levels of the hierarchy (so-called RLMs, random logic macros), have much smaller area, contain lots of logic gates and many nets and usually use only a limited amount of wide wires. In between top-level instances and RLMs are so-called integration-level instances which have intermediate features. Furthermore there might be instances called macros which contain special functionality like memory arrays or analog circuits and need to be designed differently. We consider such macros given and do not discuss their design any further.

In this thesis we focus on the routing task of physical design or, more precisely, the detailed routing task. This means, we assume the logic design of the integrated circuit finished and thus the logic structure fixed as well as the physical placement of the circuits and the primary inputs and outputs given. Further, we assume that power distribution structures are already designed and fixed as well as the global clock networks. The task is then to design all necessary connections on the chip in a way that they meet physical design rules as well as timing requirements.

Due to the enormous instance sizes and the inherent complexity of the task, routing is commonly done in two or even three steps. First global routing determines rough positions for the connections of all nets, called global routes, optimizing complex objectives like congestion and timing but largely ignoring local constraints. Some routing flows (e.g. [42]) then use an intermediate step, called track assignment [6], [10], [35], which assigns long wires of these global routes to individual detailed routing tracks, respecting some local rules but not yet respecting all design rules or connecting these long segments together. Finally, during detailed routing exact locations for all wires respecting all design rules are calculated. The global routes and potentially track assigned wires are used to limit the search space.

On some layers, wires need to be placed on a set of predefined coordinates, called tracks. On other layers, wires can be placed freely. However, restricting the router mostly to certain tracks (forming a routing grid) may be beneficial. Both grid-based (e.g. [33]) and grid-less approaches (e.g. [11], [14] and [15]) have been proposed to route on such layers. We are interested in the detailed routing step with only a global routing but no track assignment as input and we will restrict ourselves mostly to grid-based routing.

Instance sizes in detailed routing are enormous. On large instances, several million nets needs to be routed with hundreds of meters of wires on a graph with hundreds of billions of nodes. As an example, Figure 1.1 shows $\frac{1}{16000}$ of the lowest two routing layers of a large instance.

Furthermore, the detailed routing problem is hard in theory. It includes many NP-hard problems, for example finding a minimum-length rectilinear Steiner tree [17] or the vertex-disjoint paths problem [25], which remains NP-hard even on grid graphs [31]. There is even little hope for efficient approximation algorithms with a good approximation guarantee [12]. Therefore, in practice either subproblems are discussed or heuristic methods are employed.

The rest of this thesis is organized as follows. In Chapter 2, we introduce some notation that is needed for the following chapters, especially for Chapter 5. In Chapter 3, we briefly describe BonnRouteDetailed, the detailed routing solution developed by the Research Institute for Discrete Mathematics at the University of Bonn in joint work with IBM.

In Chapter 4, we describe a novel algorithm for automatically computing track patters for a set of different kinds of wires for detailed routing and how this algorithm can be used efficiently in practice in BonnRouteDetailed. We prove its correctness and discuss its run time.

In Chapter 5, we first develop an axiomatic description of diff-net rules and derive important properties. We develop two descriptions of an important class of diff-net rules and prove their equivalence. We prove for a number of practically important diff-net rules that they are diff-net rules in our theoretical sense and belong to this class. Then we describe a highly optimized algorithm to check such diff-net rules in BonnRouteDetailed.

In Chapter 6, we present experimental results of the algorithms developed in Chapter 4 and Chapter 5 as well as parallelization results of BonnRouteDetailed including these algorithms. Our automatic track pattern calculation yields substantial improvements both in run time and quality of results. Our optimized diff-net rule checking reduces the run time of BonnRouteDetailed by more than a factor two. Furthermore, BonnRouteDetailed parallelizes very well. The main routing step achieves a speedup of factor 40 with 64 threads on 14nm instances and a speedup of factor 32 with 64 threads on newer 7nm instances.

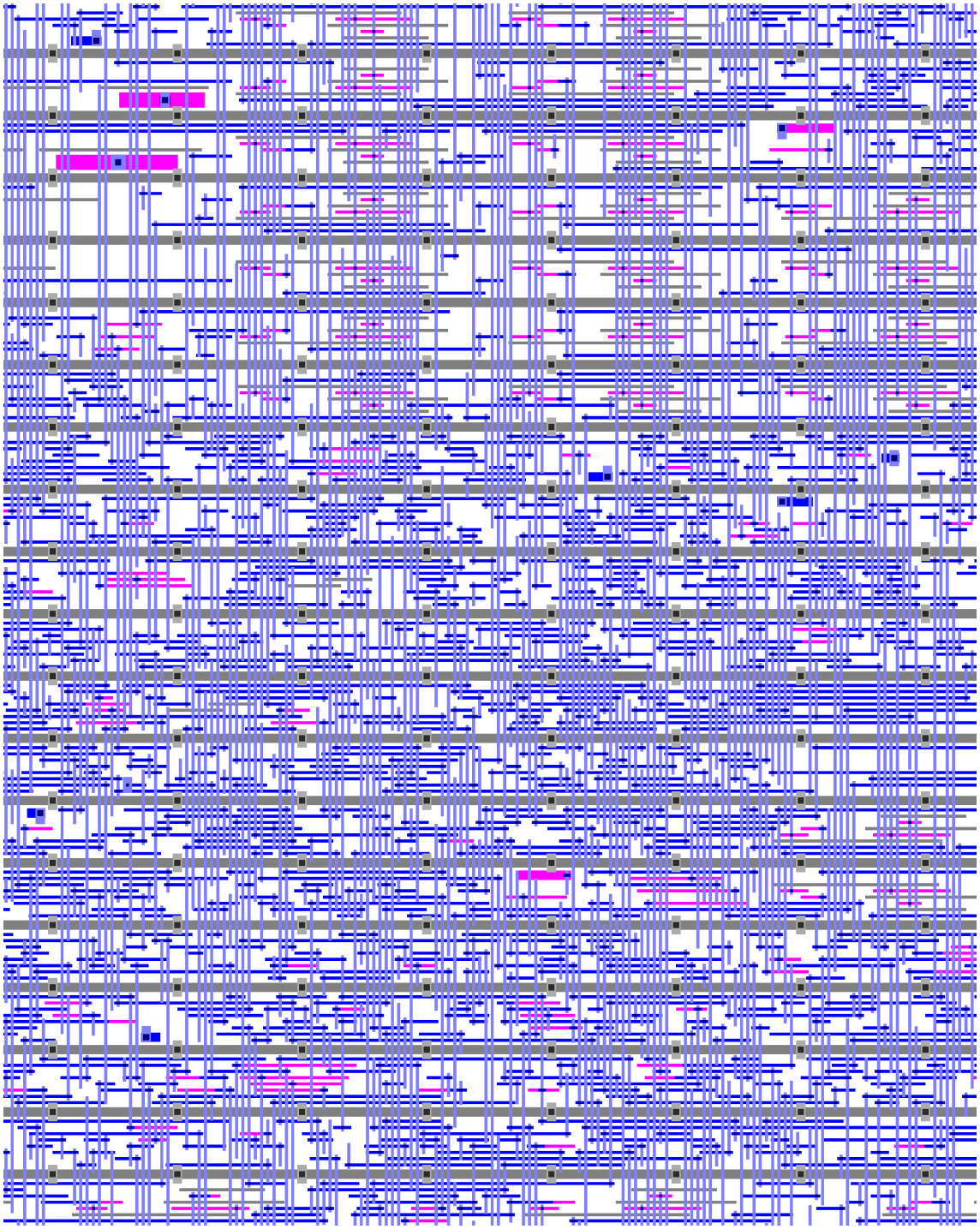


Figure 1.1: $\frac{1}{16000}$ of the lowest two routing layers of the instance L-14-A-2. Dark (light) blue shapes are wires on the lowest (second lowest) routing layers. Pink shapes are pins, gray shapes are blockages (including power rails) and black squares indicate vias connecting the lowest and the second lowest routing layers. All wires on the lowest routing layer run horizontally and all layers on the second lowest routing layer run vertically. Layers (including adjacent via layers) above and below these two routing layers are not shown.

Chapter 2

Definitions and Notation

We assume basic knowledge of combinatorial optimization. For an overview, we refer the reader to [29]. Let $\mathcal{B} := \{true, false\}$ be the set of Boolean values. We use $\wedge$ for Boolean 'and' and $\vee$ for Boolean 'or'. We may use these expressions and functions returning these expressions directly in all places where Boolean values are required.

We use the $\mathcal{O}$-notation to denote asymptotic running times.

By $\mathbb{N}$ we denote the set of all natural numbers including 0. By $\mathbb{Z}$ and $\mathbb{R}$ we denote the set of all integers (positive, zero and negative) and the set of all real numbers respectively.

Let X be any set. By $\mathfrak{P}(X)$ we denote the set of all subsets of X. Let $k \in \mathbb{N}$. By $\mathfrak{P}_k(X)$ we denote the set of all subsets of size k of X.

We assume throughout this dissertation that all sets that are not explicitly defined to be infinite are finite. For example, $\mathbb{N}$ and $\mathcal{R}$ (defined below) are explicitly infinite, but whenever we write something like 'let $R \subseteq \mathcal{R}$' we implicitly assume R to be finite. In particular all the sets occurring during the algorithms are finite and all the maxima and minima exist because they are taken over a finite set.

Let $x_1, x_2, y_1, y_2 \in \mathbb{R}$, $p \in \mathbb{R}$, $p \geq 1$. Define $d_x((x_1, y_1), (x_2, y_2)) := |x_2 - x_1|$, $d_y((x_1, y_1), (x_2, y_2)) := |y_2 - y_1|$, $d_p((x_1, y_1), (x_2, y_2)) := (|x_2 - x_1|^p + |y_2 - y_1|^p)^{\frac{1}{p}}$ and $d_{max}((x_1, y_1), (x_2, y_2)) := d_\infty((x_1, y_1), (x_2, y_2)) := max(|x_2 - x_1|, |y_2 - y_1|)$.

Let $l_{max} \in \mathbb{N}$ with l_{max} even and $\mathcal{L} := \{0, \ldots, l_{max}\}$ be the **set of layers**. Define $\mathcal{L}_{wiring} := \{l \in \mathcal{L} : l \text{ even}\}$ and $\mathcal{L}_{via} := \mathcal{L} \setminus \mathcal{L}_{wiring}$. $\mathcal{L}_{wiring}$ are called **wiring layers** and used for vertical or horizontal wires; $\mathcal{L}_{via}$ are called **via layers** and used for interconnecting adjacent wiring layers. To allow dense packing of wires, each wiring layer l has a **preferred direction** $pref(l) \in \{horizontal, vertical\}$ which means that most or all wires on that layer run in that direction. To allow for efficient routing, the preferred direction of the wiring layers alternates. We call wires running in the non-preferred direction **jogs**. To simplify the notation and w.l.o.g. we arbitrarily fix the preferred direction of layer 0 to be horizontal. Thus, for $l \in \mathcal{L}_{wiring}$ we get formally:

$$pref(l) := \begin{cases} horizontal & l \equiv 0 \bmod 4 \\ vertical & l \equiv 2 \bmod 4 \end{cases}$$

Let $\mathcal{X}_{min}, \mathcal{Y}_{min}, \mathcal{X}_{max}, \mathcal{Y}_{max} \in \mathbb{Z}$ with $\mathcal{X}_{min} \leq \mathcal{X}_{max}$ and $\mathcal{Y}_{min} \leq \mathcal{Y}_{max}$. The **chip area**

$\mathcal{A}$ is defined as $\mathcal{A} := [\mathcal{X}_{min}, \mathcal{X}_{max}] \times [\mathcal{Y}_{min}, \mathcal{Y}_{max}]$. Let

$$\mathcal{R} := \{[x_1, x_2] \times [y_1, y_2] : x_1, y_1, x_2, y_2 \in \mathbb{Z}, x_1 \leq x_2, y_1 \leq y_2\}$$

be the **set of all closed, axis-parallel rectangles**. In the following use the term 'rectangle' when we mean 'closed axis-parallel rectangle'. For convenience, define the **set of rectangles containing** $(0, 0)$:

$$\mathcal{R}_0 := \{r \in \mathcal{R} : (0, 0) \in r\}$$

as well as the **set of all zero- or one-dimensional rectangles**:

$$\mathcal{R}_{stick} := \{[x_1, x_2] \times [y_1, y_2] \in \mathcal{R} : x_1 = x_2 \text{ or } y_1 = y_2\}$$

For $r_1, r_2 \in \mathcal{R}$ let $r_1 + r_2 := \{v_1 + v_2 : v_1 \in r_1, v_2 \in r_2\}$ be the Minkowski sum. Note that $r_1 + r_2 \in \mathcal{R}$. Let $x, y \in \mathbb{Z}, r \in \mathcal{R}$. Define $(x, y) + r := r + (x, y) := [x, x] \times [y, y] + r$. Let $r = [x_{min}, x_{max}] \times [y_{min}, y_{max}] \in \mathcal{R}$. Define $x_{min}(r) := x_{min}$, $x_{max}(r) := x_{max}$, $y_{min}(r) := y_{min}$, $y_{max}(r) := y_{max}$. Define $mirror(r) := [-x_{max}, -x_{min}] \times [-y_{max}, -y_{min}]$. Let $r_1, r_2 \in \mathcal{R}, p \in \mathbb{R}, p \geq 1$. Define

$$d_x(r_1, r_2) := min\{d_x((x_1, y_1), (x_2, y_2)) : (x_1, y_1) \in r_1, (x_2, y_2) \in r_2\}$$

$$d_y(r_1, r_2) := min\{d_y((x_1, y_1), (x_2, y_2)) : (x_1, y_1) \in r_1, (x_2, y_2) \in r_2\}$$

$$d_p(r_1, r_2) := min\{d_p((x_1, y_1), (x_2, y_2)) : (x_1, y_1) \in r_1, (x_2, y_2) \in r_2\}$$

$$d_\infty(r_1, r_2) := d_{max}(r_1, r_2) := min\{d_\infty((x_1, y_1), (x_2, y_2)) : (x_1, y_1) \in r_1, (x_2, y_2) \in r_2\}$$

Let $R \subseteq \mathcal{R}$. Define the **bounding box** $bbox(R) \in \mathcal{R}$ of all rectangles in R: $bbox(R) := [min\{x_{min}(r) : r \in R\}, max\{x_{max}(r) : r \in R\}] \times [min\{y_{min}(r) : r \in R\}, max\{y_{max}(r) : r \in R\}]$.

In modern technologies, many layers on a chip cannot be manufactured with a single mask, but multiple masks need to be applied, each creating some of the desired shapes. Thus shapes need to be assigned to different masks. This is done by assigning a color to each shape which determines the mask by which the shape is manufactured. Formally, let $\mathcal{COL}$ be a finite **set of colors**. If there are no colors on a given layer, then one can simply assign the same color to all shapes on this layer.

Let $\mathcal{SC}$ be a finite set, called the **set of shape classes**. Each object on the chip is assigned a **shape class** determining together with its color which distance rules it needs to obey. Let $\mathcal{SC}_v \subseteq \mathcal{SC}$ be a subset of the shape classes which are exclusively used on via layers. We denote **all other shape classes** by $\mathcal{SC}_o := \mathcal{SC} \setminus \mathcal{SC}_v$. We use this distinction later because design rules on via layers can be substantially different from design rules on metal layers.

We sometimes need to distinguish between shapes used for routing, shapes that are parts of pins and shapes that serve as blockage. Therefore we define the set of possible **purposes** of a shape:

$$\mathcal{SP} := \{wire, pin, blockage\}$$

Now we can formally define what a shape is. A **shape** is an integral, axis-parallel, closed, rectangular area on a specific layer together with an associated shape class, a color and a purpose. Formally, define the **set of all shapes**

$$\mathcal{S} := \{(r, l, sc, c, sp) : r \in \mathcal{R}, l \in \mathcal{L}, sc \in \mathcal{SC}, c \in \mathcal{COL}, sp \in \mathcal{SP}\}$$

Let $s_1 = (r_1, l_1, sc_1, c_1, sp_1) \in \mathcal{S}, s_2 = (r_2, l_2, sc_2, c_2, sp_2) \in \mathcal{S}$. We say s_1 is a subshape of s_2 and write $s_1 \subseteq s_2$ if $r_1 \subseteq r_2, l_1 = l_2, sc_1 = sc_2, c_1 = c_2$ and $sp_1 = sp_2$. We say s_1 is a proper subshape of s_2 and write $s_1 \subsetneq s_2$ if s_1 is a subshape of s_2 and $r_1 \neq r_2$. Let $s = (r, l, sc, c, sp) \in \mathcal{S}$. Define $r(s) := r$, $l(s) := l$, $sc(s) := sc$, $c(s) := c$, $sp(s) := sp$, $attr(s) := (l, sc, c, sp)$. For simplicity of notation, define:

$$\mathcal{S}_w := \{s \in \mathcal{S} : sp(s) = wire\}$$

$$\mathcal{S}_p := \{s \in \mathcal{S} : sp(s) = pin\}$$

$$\mathcal{S}_b := \{s \in \mathcal{S} : sp(s) = blockage\}$$

Further, define $x_{min}(s) := x_{min}(r(s))$, $x_{max}(s) := x_{max}(r(s))$, $y_{min}(s) := y_{min}(r(s))$, $y_{max}(s) := y_{max}(r(s))$. Define $center(s) := (\frac{x_{min}(s)+x_{max}(s)}{2}, \frac{y_{min}(s)+y_{max}(s)}{2})$. Define $area(s) := (x_{max}(s)-x_{min}(s))(y_{max}(s)-y_{min}(s))$ and $diam(s) := max(x_{max}(s)-x_{min}(s), y_{max}(s) - y_{min}(s))$.

For $r \in \mathcal{R}, s \in \mathcal{S}$, define $r + s := s + r := (r + r(s), l(s), sc(s), c(s), sp(s))$. For $x, y \in \mathbb{Z}, s \in \mathcal{S}$, define $(x, y) + s := s + (x, y) := s + [x, x] \times [y, y]$.

Let $S \subseteq \mathcal{S}$. Define S **homogeneous** $:= \forall s_1, s_2 \in S : attr(s_1) = attr(s_2)$. Let now S be homogeneous, $S \neq \emptyset$ and $s_0 \in S$. Define $attr(S) := attr(s_0)$. Let $S \subseteq \mathcal{S}, r \in \mathcal{R}, x, y \in \mathbb{Z}$. Define $r+S := S+r := \{r+s : s \in S\}$ and $(x, y)+S := S+(x, y) := \{(x, y)+s : s \in S\}$. Let $s_1, s_2 \in \mathcal{S}, p \in \mathbb{R}, p \geq 1$. Define $d_x(s_1, s_2) := d_x(r(s_1), r(s_2))$, $d_y(s_1, s_2) := d_y(r(s_1), r(s_2))$, $d_p(s_1, s_2) := d_p(r(s_1), r(s_2))$, $d_\infty(s_1, s_2) := d_{max}(s_1, s_2) := d_\infty(r(s_1), r(s_2))$.

Wires on a chip are modeled by so-called wire and via sticks. Wire sticks represent wires within a wiring layer of the chip, vias interconnect wires on neighboring wire layers. A via is modeled by three shapes, one on the lower wiring layer the via connects, one on the upper wiring layer and one on the via layer in between. Wire and via models define the metal shapes that are induced by a wire or a via stick. Define

$$\mathcal{WM} := \{(r, sc) : r \in \mathcal{R}_0, sc \in \mathcal{SC}_o\}$$

$\mathcal{WM}$ is the **set of all wire models**. Let $wm = (r, sc) \in \mathcal{WM}$. Define $r(wm) := r$, $sc(wm) := sc$.

Similarly, define the **set of all via models**:

$$\mathcal{VM} := \{((r_b, sc_b), (r_m, sc_m), (r_t, sc_t)) : r_b, r_m, r_t \in \mathcal{R}_0, sc_b, sc_t \in \mathcal{SC}_o, sc_m \in \mathcal{SC}_v\}$$

Let $vm = ((r_b, sc_b), (r_m, sc_m), (r_t, sc_t))$. Define $r_b(vm) := r_b$, $sc_b(vm) := sc_b$, $r_m(vm) := r_m$, $sc_m(vm) := sc_m$, $r_t(vm) := r_t$ and $sc_t(vm) := sc_t$.

A **wire stick figure** (short: **wire stick**) represents an axis-parallel segment of a wire on a given layer. Formally, let

$$\mathcal{WS} := \{(r, l, m, c) : r \in \mathcal{R}_{stick}, l \in \mathcal{L}_{wiring}, m \in \mathcal{WM}, c \in \mathcal{COL}\}$$

be the **set of all wire sticks**. Similarly a **via stick figure** (short: **via stick**) represents a via connecting two neighboring wiring layers. Let

$$\mathcal{VS} := \{(x, y, l, m, c_b, c_m, c_t) : x, y \in \mathbb{Z}, l \in \mathcal{L}_{via}, m \in \mathcal{VM}, c_b, c_m, c_t \in \mathcal{COL}\}$$

be the **set of all via sticks**. Define the **set of all stick figures** (or short: **sticks**)

$$\mathcal{ST} := \mathcal{WS} \cup \mathcal{VS}$$

Let $ws = (r, l, (r_m, sc), c) \in \mathcal{WS}$. Define the shape of the wire stick ws in the following way: $shape(ws) := (r + r_m, l, sc, c, wire) \in \mathcal{S}$. To simplify the notation, define $shapes(ws) := \{shape(ws)\}$. Let $vs = (x, y, l, ((r_b, sc_b), (r_m, sc_m), (r_t, sc_t)), c_b, c_m, c_t) \in \mathcal{VS}$. Define $shape_b(vs) := ((x, y) + r_b, l - 1, sc_b, c_b, wire) \in \mathcal{S}$, $shape_m(vs) := ((x, y) + r_m, l, sc_m, c_m, wire) \in \mathcal{S}$ and $shape_t(vs) := ((x, y) + r_t, l + 1, sc_t, c_t, wire) \in \mathcal{S}$. Let further $l' \in \{l - 1, l, l + 1\}$. Define

$$shape_{l'}(vs) := \begin{cases} shape_b(vs) & l' = l - 1 \\ shape_m(vs) & l' = l \\ shape_t(vs) & l' = l + 1 \end{cases}$$

Define the set of shapes of vs: $shapes(vs) := \{shape_b(vs), shape_m(vs), shape_t(vs)\} \subseteq \mathcal{S}$.

In recent technology nodes on some layers wires need to be placed at special coordinates called tracks due to manufacturing reasons. On other layers, very complex design rules apply to combinations of three or more wires depending on their widths and relative geometry. These design rules can often be fulfilled automatically if wires of certain widths are only placed on certain coordinates. On some layers, wires could be placed arbitrarily, but it is beneficial to restrict routing tools to a certain subset of possible coordinates to facilitate efficient packing of wires, especially efficient packing of wires with different widths and spacing values. Therefore, we define a set of usable tracks on each layer (in theory these could be all coordinates although that would in most cases lead to very long run times and very poor results). If tracks are given a priori by technology restrictions, we simply use all possible coordinates for all possible wire models. If no such set is given by the technology, we show how to compute optimized sets of tracks for a number of wire models in Chapter 4. Again the tracks of the chip are simply the union of all these sets of tracks.

A set of tracks is a nonempty, finite set of coordinates. Define the **set of all sets of tracks**: $\mathcal{TR} := \{\{t_1, \ldots, t_k\} : k \in \mathbb{N}, k \geq 1, t_1, \ldots, t_k \in \mathbb{Z}\}$. A chip then contains for each wiring layer a set of tracks.

In general, it is not an easy task to find a good set of tracks if multiple wire models are involved. A future alternative might be to use the output of some trackless track-assignment algorithm (e.g. as described in [35]) to define tracks for detailed routing. Sometimes some wire models can only use a subset of the tracks on some layers.

On a chip there are typically some predefined structures and some areas that may not be used for routing. From a routing perspective, these are simply blockages, shapes that belong to the chip and induce distance rules for any metal shapes to be placed.

With $\mathcal{T}$ we denote a finite **set of threads** used for parallel computation.

Chapter 3

BonnRouteDetailed

BonnRouteDetailed [19] is a state-of-the-art, mostly track-based detailed routing tool with an emphasis on low run time, efficient parallelization and effective high density routing. It is the detailed routing tool of the BonnTools [28], the chip design solution developed at the Research Institute for Discrete Mathematics at the University of Bonn in joint work with IBM.

BonnRouteDetailed produces routings with short wire length, few vias, few detours and avoids a number of electrically or timing-wise undesirable configurations. It obeys a wide class of diff-net design rules exactly and avoids many same-net errors on a best effort basis. It can be used to produce very dense detailed routings meeting timing constraints but still containing a limited number of design rule violations. These can later be corrected by an industrial routing tool, leading to superior results than running any one of the two routing tools alone (see for example [40], [19] and [3]). Furthermore, BonnRouteDetailed automatically detects and reports a wide range of problems in its input and produces comprehensible statistics on its output. In this chapter we give an overview of key components and concepts of BonnRouteDetailed.

3.1 General Concept

BonnRouteDetailed is designed to be used as a bulk routing tool (and not an incremental router). It contains a number of optimizations precomputing certain data for the given chip which takes some time for precomputation but saves a lot of run time during actual routing. Five major components need to be mentioned here.

First, so-called **soft track patterns**. BonnRouteDetailed precomputes for all wire models that are not required to be located on certain given **hard track patterns** so-called soft track patterns, special coordinates where wires of these wire models are preferably placed. These track patterns are optimized to make efficient packing of a mixture of different wire models possible. BonnRouteDetailed can deviate from them to access pins or off-grid wires. We mention some more details in Section 3.3 and discuss the problem to compute good soft track patterns in detail in Chapter 4.

Second, the so-called **grid**. All shapes on the chip are stored in a geometrical data structure called the grid, allowing efficient geometrical queries. Because especially blockages are often represented in a redundant and inefficient way (for example there might be

multiple blockages overlapping or even blockages completely contained within each other), input shapes are preprocessed and represented in an optimized, overlap-free way [18]. The grid can be efficiently queried and updated in a highly parallelized way. We describe this data structure in more detail in Section 3.4.

Third, BonnRouteDetailed precomputes legality information (concerning diff-net rules) for the most common wire and via models for on-grid locations in the so-called **fast grid** (and updates this information during routing). This data structure is briefly mentioned in Section 3.6 and described in more detail in Section 5.3. This reduces overall run time significantly, because routing objects change a lot less often than legality information is queried.

Fourth, accessing all pins on the lowest layers is often challenging. To avoid pins from being blocked by routes for other nets, BonnRouteDetailed precomputes access paths for most pins before routing, escaping them to a specified layer with a dynamic program. This step selects one access path for as many considered pins as possible and guarantees that all these access paths can be used simultaneously. However, sometimes it is beneficial to deviate from these preselected access paths later on. Therefore, the router has the ability to compute further access paths on the fly during routing and discard the precomputed ones. If pins of the same net are very close together, it is preferable to connect them directly instead of computing access paths. BonnRouteDetailed already connects such pins during the dynamic program with very short and efficient connections. Precomputing short connections and escaping the remaining low-layer pins improves results significantly (see [2]). Further, BonnRouteDetailed also precomputes access paths for so-called blocked pins (pins that are not legally accessible due to diff-net rules). In practice such pins often occur due to unclean design data. While a chip is designed, many parts are changed simultaneously by different people to save time. Thus it frequently occurs that a change has already been made on some part of a chip but other parts have not yet been updated. Yet such a chip still needs to be routed and thus routing tools need to cope with all sorts of imperfect input such as blocked pins. To save run time and complexity in the core routing algorithms, BonnRouteDetailed precomputes access paths for such blocked pins, if possible making sure that the end points of the access paths are legally accessible. We describe the algorithms used for pin access in a little more detail in Section 3.7.

Fifth, the way BonnRouteDetailed models and deals with diff-net rules is highly optimized for a bulk routing setup where all involved wire models, nets and diff-net rules are known in advance. BonnRouteDetailed uses so-called **shape classes** to model which diff-net rules apply for a certain piece of metal. These shape classes and the rules that apply to them are precomputed. Many data structures like the fast grid and the grid and some temporary data structures rely on the fact that the set of all shape classes and rules is known such that the maximum distance where a shape (of a certain shape class) can influence legality of another shape can be bounded from above. Also precomputing checking data in the fast grid for the most common wire and via models assumes that the most frequent wire and via models are known in advance.

After data has been loaded, access paths and short connections have been precomputed and the grid and the fast grid have been initialized, nets are routed on a net-by-net basis one after another (but with multiple threads each routing one net at a time). Consequently, one very important aspect is the order in which the nets are routed. BonnRouteDetailed

precomputes a desired order described in more detail in Section 3.8. The most important criteria are the width of the default wire model of the net and the width of the special wire models used to access pins. While routing in a multi-threaded way, BonnRouteDetailed makes local changes to this precomputed order to reduce the probability of collisions occurring. We describe these aspects of parallelization in more detail in Section 3.5.

Another key concept is the idea that the router should always produce a connected routing for each net. Even if the task is impossible, because there is overcongestion or some area is cut off by a blockage or pins are blocked, BonnRouteDetailed in almost all cases produces a (potentially illegal) connection for the net. Only if input data is severely incomplete or inconsistent (like there is no wire model at all or pins lie outside the chip area), making it impossible to produce meaningful output, nets are left open. BonnRouteDetailed does never postpone a task. When a net is due to be routed, it is routed completely.

First, BonnRouteDetailed tries to route the net with (pretty restrictive) default parameters without changing any other nets. If this fails, BonnRouteDetailed tries to route the net while changing (ripping-up and re-routing) other nets. If this again fails, some parameters are successively relaxed, for example allowing to route in a larger area and in later steps also reducing measures to avoid same-net errors. If all these attempts fail, routes that violate diff-net rules to other nets are allowed. Ultimately, a path search that can violate any legality restrictions at certain costs is run. This framework is described in more detail in Section 3.9. One key assumption to the success of this framework is that a net that was successfully routed might be ripped-up and re-routed but is never disconnected again. Thus even though individual connections may change, any net that is connected always stays connected.

To avoid same-net errors there are a number of very powerful frameworks implemented in BonnRouteDetailed. Most importantly, BonnRouteDetailed includes a very efficient multi-label path search [1]. This enables BonnRouteDetailed not only to find shortest paths, but to find (not anymore necessarily shortest) paths with certain restrictions on the minimum segment length after a certain kind of segment was placed; for example one can specify that after a via the path needs to continue in preferred direction for a certain distance until a jog is allowed. This is a very powerful tool to avoid a wide class of same-net errors. Additionally, there is an elaborate post-processing function, fixing same-net errors in computed paths by local changes (for example by moving the position of jogs) and by extending segments by so-called endstyles, short segments of metal that are added to the end of certain wire sticks to fix design rule violations but that are not necessary for the connectivity of the path ([41]). BonnRouteDetailed can precompute for each pin a so-called pin shrink, that is the set of positions where the pin can be accessed legally with a wire of a given direction and wire model or a via with a given via model, thus restricting pin access and fulfilling same-net rules at pins. Last but not least, BonnRouteDetailed employs a concept called protections, allowing for each path search to precompute certain positions where a wire or via of a given wire or via model should not be placed. This is for example very helpful to avoid same-net errors between different vias or between vias and jogs at Steiner points in the connection for a net.

These four concepts are used to efficiently avoid same-net errors in the following way. Whenever a non-trivial pin is accessed, the pin shrink is respected. Before searching a

path, first a set of protections is computed to avoid same-net errors to already existing wiring of the same net. The number and kind of protections is determined by the current set of parameters, first it is tried with rather restrictive protections and if this fails, the number of protections is reduced. Then, a path is searched with a standard path search and post-processed. Often, the result is same-net error clean and can be used. If this is not the case, a minimum label system is computed that can fix as many of the remaining same-net errors as possible and a new path is computed with a multi-label path search and post-processed. This is iterated until all multi-label restrictions are used or the found path is same-net error clean. In the end, the path with the least number of same-net errors is taken. This setting is very beneficial because post-processing a path is very fast (and can fix already most same-net errors) and multi-label path searches are comparatively slow.

Another very important aspect of the quality of detailed routing is timing behavior. Design rule clean connections are useless if they do not meet timing requirements. BonnRouteDetailed does not measure timing directly. Instead it relies on two powerful concepts to ensure that computed connections meet timing constraints. First, it is designed to use global wires computed by BonnRouteGlobal which in turn is directly timing aware ([21], [39], [38], [22]). BonnRouteGlobal computes global routes that have certain timing properties and BonnRouteDetailed aims to implement detailed routes that are geometrically and topologically similar to the input global routes. Therefore, the rough structure of computed detailed routes is timing-wise well-designed. To this end, BonnRouteDetailed uses a complex data structure to store global and detailed routes initially described in [26]. It can guarantee that global and detailed routes are similar in some sense as well as enable good run times for huge nets with many components. We mention some more details on this data structure in Section 3.4 and refer to [26] for further insight. But for timing characteristics, also the very details matter. BonnRouteDetailed tries to ensure that its routes have locally good timing characteristics by avoiding certain problematic configurations. Some further details on this topic are mentioned in Section 3.12.

Last but not least, efficient parallelization is key to the success of BonnRouteDetailed. We mention more details in Section 3.5, but the main concept is to allow full parallelization for all major steps. The basic concept was introduced in [27] for the main parallel routing step and has since then been applied to many more steps of BonnRouteDetailed improving total parallel speedup even more. The main idea is to compute small sub-tasks in parallel and add them to central data structures as quickly as possible while locking as little as possible. When the solutions of sub-tasks computed in parallel conflict, BonnRouteDetailed recomputes one of the solutions rather than ensuring that sub tasks are independent (for example by geometric partitioning of the chip area like in [37]). Most of the key data structures of BonnRouteDetailed have been implemented in a way to enable such kinds of parallelization efficiently and often with very little additional programming effort.

One often underestimated factor for the practical success of BonnRouteDetailed is the fact that it can compute a powerful set of comprehensive statistics assessing the state of its in- and output and that it can automatically detect and report a wide number of problems in its input. In times of iterative and hierarchical design flows, these capabilities are, besides the ability to produce efficient routings in good run time, a key factor for effective chip design. Most instances initially contain a number of problems that prevent

clean detailed routing. Identifying and fixing such issues quickly is key in successfully designing very large scale integrated circuits efficiently in practice. We briefly mention some key points in Section 3.2.

In the following sections, we describe some of the key concepts of BonnRouteDetailed in more detail and refer to previous works for some other key concepts.

3.2 Data and Input Specification

In this section we briefly summarize the kind of input data that BonnRouteDetailed expects, mention some checks that are performed on input data and briefly discuss some assumptions that are made and how they are enforced.

Of course a detailed routing tool needs a geometric description of the chip to be routed. This includes the chip area, the number and type of routing layers, blockages and pins. It needs diff-net rules which are encoded in the form of shape classes and distance rules (see Chapter 2 or [40]). It further needs the information which kind of metal can be used to route connections which is encoded in wire and via models for each layer. Obviously also logical information is needed, in particular which pins need to be connected. This is encoded in the so-called nets. Each net contains a number of pins (usually exactly one source pin and a number of sink pins) which need to be connected as well as wire and via models that can be used to connect the pins. Special wire and via models that can be used in a limited region around the pins to attach to the pins may be provided as well as a range of assigned layers on which the majority of routing of the net should be located. Nets may already contain input wires, both modifiable ones that can be changed and locked ones that need to be kept as they are.

As described in Section 3.9 and Section 3.12, BonnRouteDetailed needs a complete global routing for each net to be routed. This means, for each net that is to be routed, the union of pins, input detailed routes and global routes should be connected. These global routes are used to determine the rough topology of the created detailed routing as well as to create routing areas for detailed path searches and thus the rough geometry of the detailed routing produced. BonnRouteDetailed might get (hard) track patterns for some wire models as input and it gets usually a large number of different same-net design rules that need to be obeyed.

BonnRouteDetailed makes a few assumptions about its input. One of the most important has already been mentioned, there should be a complete global routing. Input global and detailed routing should not contain any loops (with the exception of some precomputed structures used to add redundant vias at certain pins) and all leaves of the union of detailed routing and pins should be pins or locked detailed wires. Otherwise, some detailed routes are turned into global routes and some global routes are deleted to fulfill this assumption.

BonnRouteDetailed detects and reports many problematic situations in its input. Input track patterns are checked for sanity, in particular neighboring tracks are checked to be legal with respect to each other and warnings are issued if any odd situations are encountered. Wire and via models and diff-net rules are checked for basic requirements, for example there should be at least one diff-net rule between any kind of metal shapes. Pins are checked to be accessible and access wire codes are verified to be suitable for the pins

they are assigned to. Global routes are checked for completeness and redundant wires. Nets are checked to contain at least two pins (preferably exactly one source and at least one sink pin) and no obviously useless wiring. Shapes on colored and uncolored layers are verified to have the required attributes. Pins are verified to lie within the chip area and nets and pins to have usable wire and via models.

If any problems are detected, BonnRouteDetailed still routes the given chip but issues a comprehensive summary of warnings such that detected issues can be resolved efficiently. This kind of automatic input verification is, whenever possible, extremely valuable, especially because many aspects of input data are in practice created in a semi-automated fashion or even entirely hand-coded and seldom thoroughly covered by external automatic testing.

3.3 Track Patterns

As briefly mentioned in Section 3.1, BonnRouteDetailed is a mostly track-based routing tool. This means that it does not place detailed wires on arbitrary coordinates but mostly sticks to precomputed special coordinates for each wire model called tracks. We discuss the problem to compute these tracks in detail in Chapter 4. Currently, soft track patterns are computed in the beginning of BonnRouteDetailed, but we aim to compute track patterns earlier in the flow in the near future. Then, to estimate resource usage of different wire models, global routing tools can take into account exactly the track patterns that BonnRouteDetailed uses. This should lead to increased accuracy when predicting detailed routability based on global routing and thus better chip design flow stability and reliability as well as better resource usage by the global router and thus better detailed routing quality (e.g. fewer detours).

3.4 Storing Routing Objects

BonnRouteDetailed uses two main data structures to store routing data. The first data structure stores nets (and the wires of the nets). As described first in [26], the net data structure serves two goals. First, under some mild assumptions, it provides facilities to generate source and target locations and routing areas for path search in time, up to logarithmic terms, linear in the number of wires and vias and pin shapes that are currently stored in the net. Thus it enables BonnRouteDetailed to route very large nets efficiently. Second and maybe even more important, it guarantees that detailed routing solutions can not deviate too much from the input global routing. A sample net with global and detailed routing can be found in Figure 3.1.

To this end, the net data structure consists of a number of different data structures. First, it stores two graphs, one containing the union of global and input detailed wires and pins (the so-called **global wire graph**) and one containing all detailed wires and pins (the **detailed wire graph**). At each point in time the detailed wire graph represents the (detailed) connected components of a net. The global wire graph remains unchanged and represents the input global routing (plus all detailed wiring that was already there before global routing was done and thus is necessary to make the global wire graph connected). For technical reasons, the degree of all nodes in both graphs that do not represent pins is

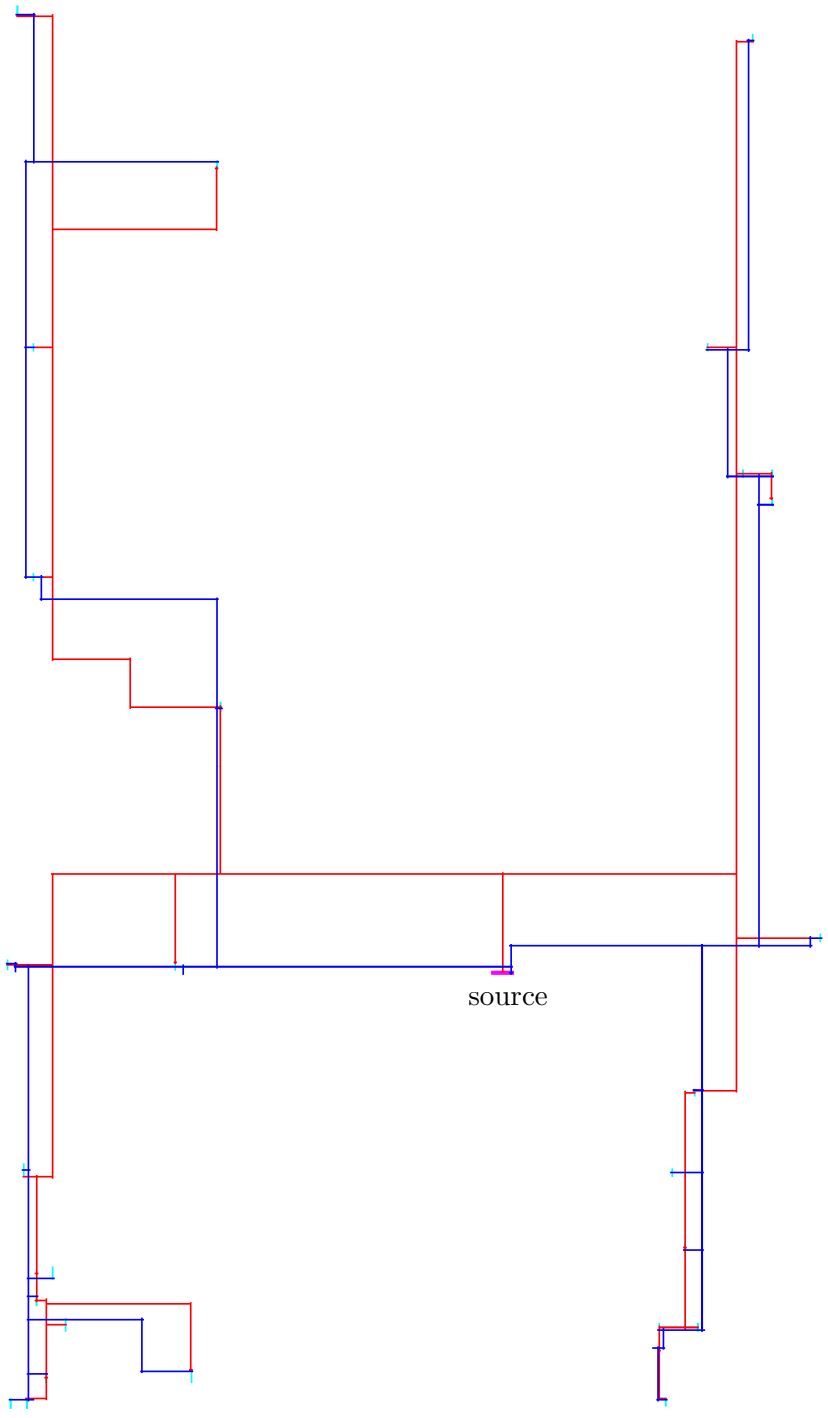


Figure 3.1: Detailed wiring (blue) and corresponding global wiring (red). The sink pins are drawn in cyan and the source pin is marked in magenta.

restricted to be at most three. A node of higher degree can be represented by a number of nodes of degree three and some dummy arcs. Furthermore, the net data structure stores references from certain nodes in the global wire graph to nodes in the detailed wire graph and vice versa. In particular, for each node of degree three and each node that represents a fixed object (a pin or a locked wire) in the detailed wire graph, BonnRouteDetailed stores a corresponding node in the global wire graph. When some detailed connection needs to be ripped-up, the violating segment can be found in the detailed wire graph. Then, the graph is traversed in both directions until nodes of degree three or fixed nodes are reached. Source and target nodes are created from the neighborhood of those nodes. Furthermore, the two corresponding nodes in the global wire graph define a unique path connecting them in the global wire graph. BonnRouteDetailed calculates the routing area to reconnect the ripped-up connection from this global route. If a replacement is found, BonnRouteDetailed replaces the ripped-up connection by the new one and updates the data structures. For each node in the global wire graph that has a counterpart in the detailed wire graph and has an incident edge that belongs to a global route that has not yet been detailed routed, the net data structure stores a corresponding detailed node.

When a component is to be connected (BonnRouteDetailed always connects components to the electrical source component for a number of reasons described in [26]) the node in the detailed wire graph that represents the component has by definition a corresponding global node. From that corresponding global node BonnRouteDetailed goes backwards towards the electrical source component until the first (different) global node that has a corresponding detailed node. BonnRouteDetailed creates the routing area from the global wires traversed and source and target areas from the two detailed nodes found. Again, if BonnRouteDetailed successfully finds a path, all data structures are updated and in particular corresponding detailed nodes are assigned to nodes of degree three along the path in the global wire graph. Furthermore, a number of minor data structures are stored, most notably a set of components (basically pointers into the components of the detailed wire graph) to access individual components quickly and for nodes in the global wire graph the closest detailed component (away from the electrical source) that can be reached from that node and its distance. This information is used to quickly decide which component to route next in case of multi-terminal nets. Furthermore, the net data structure stores the distance to the (electrical) source node at each global node. This enables BonnRouteDetailed to find the unique path between any two global nodes efficiently.

The second main data structure, the **grid**, stores all objects on the chip; that is blockages, pins and wires. Its main purpose is to allow fast regional queries, mainly to check diff-net distance rules. When assessing whether a given shape is legal at a given position, BonnRouteDetailed needs to find all nearby shapes as quickly as possible. Furthermore the grid has to be capable of highly parallel access (both to read data and to modify data). We describe parallelization requirements in more detail in Section 3.5 and thus concentrate here on the core data structure by itself.

The vast majority of objects on many chips are (at least at the end of detailed routing) detailed wires. Detailed wires have two properties that BonnRouteDetailed exploits to store them efficiently. First, they are naturally represented by stick figures and second these stick figures are in almost all cases located on tracks. BonnRouteDetailed first divides the whole chip area on each layer into smaller rectangular sub-areas in a way that

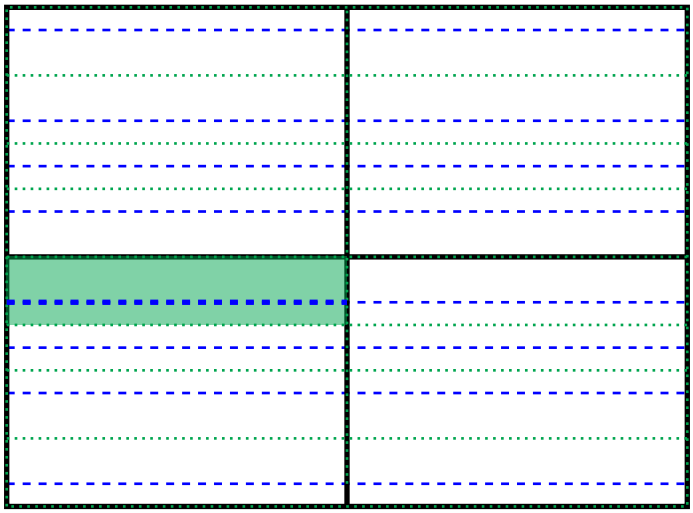


Figure 3.2: The structure of a grid with four sub-grids. The sub-grids are marked in black, tracks are drawn as dashed blue lines. The borders of the stripes are marked in green dotted lines. One stripe and the corresponding part of the corresponding track are highlighted.

each side of each sub-area is roughly some constant (which could depend on the layer but currently does not). Then within each sub-area (we call the corresponding part of the grid a sub-grid), wire sticks and other shapes are stored in so-called stripes. For each track that intersects a given sub-grid, there is exactly one stripe which spans the whole sub-grid in preferred direction and whose area ends in the middle between two tracks against preferred direction. Thus the whole chip area on each layer is partitioned into stripes which have roughly constant length in preferred direction and which have the property that (the sticks of) most detailed wires (excluding vias) are located within exactly one stripe. Figure 3.2 shows the global structure of the grid.

Wire sticks are stored in all stripes that they intersect (via sticks are typically stored in three stripes on three different layers) and all other shapes are also stored in all stripes that they intersect. Pin shapes are typically relatively small and thus intersect only a few stripes. Blockages can be substantially larger and thus sometimes intersect many stripes. BonnRouteDetailed preprocesses all blockage (and pin) shapes to maximize them along the preferred direction of the given layer to minimize the total number of blockage-stripe intersections. For more details concerning the algorithm to preprocess blockages, we refer to [18]. Typically, most blockages are very small and there are only a few large blockages. On most instances, the majority of objects in the grid are stored in only very few stripes and each object is only stored in very few stripes on average.

Within each stripe, BonnRouteDetailed stores all objects (sticks and shapes) in a simple array, ordered lexicographically by their minimum and maximum coordinate in preferred direction. For each object stored in a stripe, it is stored how many objects stored to the left intersect its minimum coordinate (objects on the chip can overlap). This simple implementation with an array instead of a balanced binary search tree has

experimentally been proven to be clearly superior. Insertion and deletion time is linear in the size of the array, but first, queries are much more frequent than modifications and second, the size of the array is bounded because the length of the stripe is bounded. Storing objects in an array instead of a balanced binary search tree benefits from the fact that in this way geometrically close objects are stored in only a few large blocks of main memory. Accessing data is much faster because there is less memory overhead and because memory caches can be used much more efficiently.

Querying all objects in a given area A is very fast. The grid stores the maximum extension of a wire or via model around its stick. To get all shapes intersecting A BonnRouteDetailed first extends A by these maximum extensions (because shapes need to be collected but sticks are stored). Then each relevant stripe can be accessed in constant time. Within a stripe, BonnRouteDetailed uses binary search to find the last object whose minimum coordinate lies within the search area (or whose minimum coordinate lies before the search area if no object with minimum coordinate within the search area exists.). Then it traverses all objects to the left until it reaches the end of the stripe or finds an object whose maximum coordinate does not lie within the search area anymore. At this point BonnRouteDetailed checks how many more objects to the left intersect this object and thus can possibly intersect the search area. Then BonnRouteDetailed traverses further to the left until all these objects have been found and checked. In practice, intersections are typically very short and only few objects are involved thus this requires little overhead. Note that even though objects can be stored in multiple stripes, BonnRouteDetailed makes sure to report them only once. Figure 3.3 illustrates an example query.

A more detailed description of a slightly different version of the grid data structure (with the most notable difference that it used balanced binary search trees instead of arrays) was described in [40] and a different version formerly used can be found in [34].

Furthermore, BonnRouteDetailed uses the so-called **fast grid** to store precomputed legality information. This data structure is quite similar to the grid, it also stores data for each track in each sub grid on each layer, but it does not store actual shapes or sticks but precomputed legality information for possible future sticks. To this end, it stores for each stripe an array of maximal intervals of identical precomputed data. The definition and use of this data is described in detail in Section 5.3.

3.5 Parallelization

With the work in [27] BonnRouteDetailed has switched its main parallelization paradigm from a region-based multi-sequential approach (originally described in [37]) to full parallelization. The concept of full parallelization means that all threads in parallel select a net, find a path for a component of the net and try to add the path to the grid. While adding paths to the grid, collisions are detected and eventually paths are recalculated if necessary. Therefore, the grid needs to be thread-safe in the following sense: Reading data and removing sticks needs to be thread-safe and there needs to be a thread-safe mechanism to either atomically add a set of sticks or reject it if it conflicts with other sticks already in the grid. Also the fast grid needs to support thread-safe read and update operations. Each thread needs to temporarily ignore some shapes (for example shapes that should be accessed by the path search as well as shapes that have been ripped-up to make space

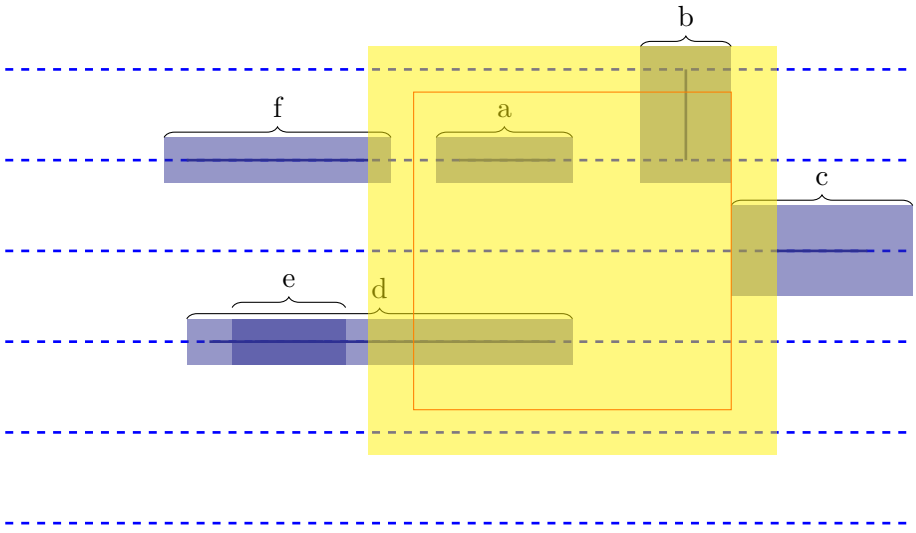


Figure 3.3: An example grid query. The query area A is marked as an orange rectangle. The extended query area is marked in yellow. In this case five stripes need to be traversed and objects a,b,c and d are reported. Note that object b is reported only once, exactly for the stripe that contains the vertical minimum of the intersection of the search area with the object. Object c is only found because the search area is extended. When traversing the third lowest stripe, object e is the first one whose maximum coordinate lies outside the search area. Together with object e the grid stores that one more object to the left intersects e. Thus the traversal is continued until the intersecting object (object d) is found (and in this case also reported). Object f is traversed but not reported, as its stick figure intersects the extended search area, but its shape does not intersect the search area.

for new shapes) and also to maintain a set of temporary additional shapes (for example routes that have already been computed for other nets but not yet been submitted). Furthermore, each thread needs a data structure storing where it can use the data from the fast grid (or more precisely to which extent it can use the data) or if it has to recalculate checking information from the grid and its temporary data structures.

These challenges are solved efficiently in the implementation of BonnRouteDetailed based on the PThreads library [8], [9]. Basic thread-safety of the grid and fast grid is ensured because each stripe (both of the fast grid and the grid) possesses an individual read-write lock which is automatically locked as required when the grid or the fast grid respectively is accessed or updated. The only tricky part is how to securely detect and handle so-called collisions, situations when the solution for one task of one thread conflicts with the solution of another thread for another task. Note that each solution can include routes for multiple nets if rip-up and re-route is involved. This part is solved in the following way: Once a solution is found, the total area that can potentially influence legality of any part of the solution is computed. This whole area is locked in the grid (in a globally fixed order so that no deadlocks can occur). Then every stick of the solution is checked for legality and if possible added to the grid. If anything illegal is detected, everything is reset. In the end, the whole area is unlocked. Afterward, non-critical parts of the update are carried out, old wires that have been ripped-up are removed from the grid and the fast grid is updated. Temporary shapes for each thread are stored in thread-local additional data structures based on quadtrees. These data structures are briefly introduced in Section 5.2 and discussed in more detail in [27]. For more information on quadtrees, we refer the reader for example to [7].

This approach gives huge speedups compared to former approaches for the main parallelized routing routine (see [27]). However, initially, only the main routing step was fully parallelized, but the techniques and infrastructure developed to that end also allow efficient parallelization of wider parts of the code. Furthermore, the order in which nets are processed was only driven by general routing requirements but not by parallelization needs and for multi-threaded rip-up and re-route, a very simple locking scheme was used. Therefore, since the original implementation of full parallelization a number of improvements have been made. We now briefly describe these improvements.

The greatest effect on the overall multi-threaded performance of BonnRouteDetailed has the successive parallelization (or elimination) of many more parts of the code. This was greatly facilitated by the work described in [27]. Some of the most important parts that have been parallelized are:

- preprocessing of nets (building the net data structure and assuring assumptions)
- circuit row pin access (see Section 3.7)
- preselecting access paths for blocked pins
- precomputing pin access criterion for the net order (see Section 3.8)
- initializing the fast grid
- numerous statistics before and after detailed routing

Therefore, by now most of the run time of BonnRouteDetailed is in parallelized parts and overall parallelization speedup is very good (see Section 6.4).

Another improvement is a modification of the order in which nets are processed. The general order in which nets are processed is described in detail in Section 3.8. For parallelization purposes, local modifications are applied to this order on the fly. In order to avoid collisions, it is beneficial if nets that are routed in parallel do not share the same area (although the parallelization framework explicitly can handle overlapping areas). Therefore, BonnRouteDetailed tries to minimize the overlap of the areas of different nets that are processed in parallel. On each layer, it uses a regular partition of the chip area in rectangular cells. For each net the set of cells that intersect parts of the net is precomputed. Whenever a net is processed, BonnRouteDetailed marks all cells that it intersects as currently occupied. Whenever BonnRouteDetailed starts to route a new net, it searches among the next k nets for the net with the least overlap to all nets that are currently being processed (while making sure that no net is postponed more than l times). This guarantees that the actual order is still similar to the original order (for sufficiently small k and l) but gives good reductions in the total number of collisions. Note that even if each net would be scheduled perfectly (that is with no overlap at all), collisions can still occur because during rip-up also nets in cells that are not intersected by the original net can be affected. Furthermore, it is usually not possible to schedule nets in a way that at no time no two active nets intersect the same cells for a reasonable number of threads while not disturbing the original order too much.

Experimental results show large reductions in the number of collisions and small reductions in the number of remaining design rule violations with many threads. Run time, however, is almost identical. With 64 threads, collisions in non-ripup path searches decrease by more than a factor ten and collisions in rip-up path searches by roughly a factor two. This is very plausible, because in normal path searches the affected area is known in advance and can be used to optimize scheduling whereas rip-up path searches can affect an arbitrary area by ripping-up other nets. With these changes, with 64 threads, BonnRouteDetailed produces on average about one collision per 1700 tasks. With 64 threads, remaining design rule violations are reduced by roughly one percent, mostly diff-net spacing violations and shorts are avoided. Figure 3.4 shows all active nets at some point in time and the corresponding cells.

The third improvement that has been made concerns the locking strategy during rip-up and re-route. The original approach used a very simple strategy. When trying to rip-up a net, it would try to lock the net and if that was not possible, it would discard the whole rip-up sequence as a collision and restart. This could lead to situations where multiple tasks were restarted as collisions while no task was successfully finished or added to the grid. This has been changed. During a rip-up-sequence, nets are not locked all the time anymore, but they are copied when they are first encountered and then the thread works on the copy. Thus locking nets can not fail anymore, because threads can wait for the locks and still deadlocks can not happen (for more information on deadlocks, see for example [13]). Therefore, collisions can occur only when trying to add results to the grid. In this case, some other thread has to have been successful for a collision to occur. Thus the whole framework is guaranteed to converge even if unlimited number of collisions per net are allowed (in practice BonnRouteDetailed still limits the number of collisions per

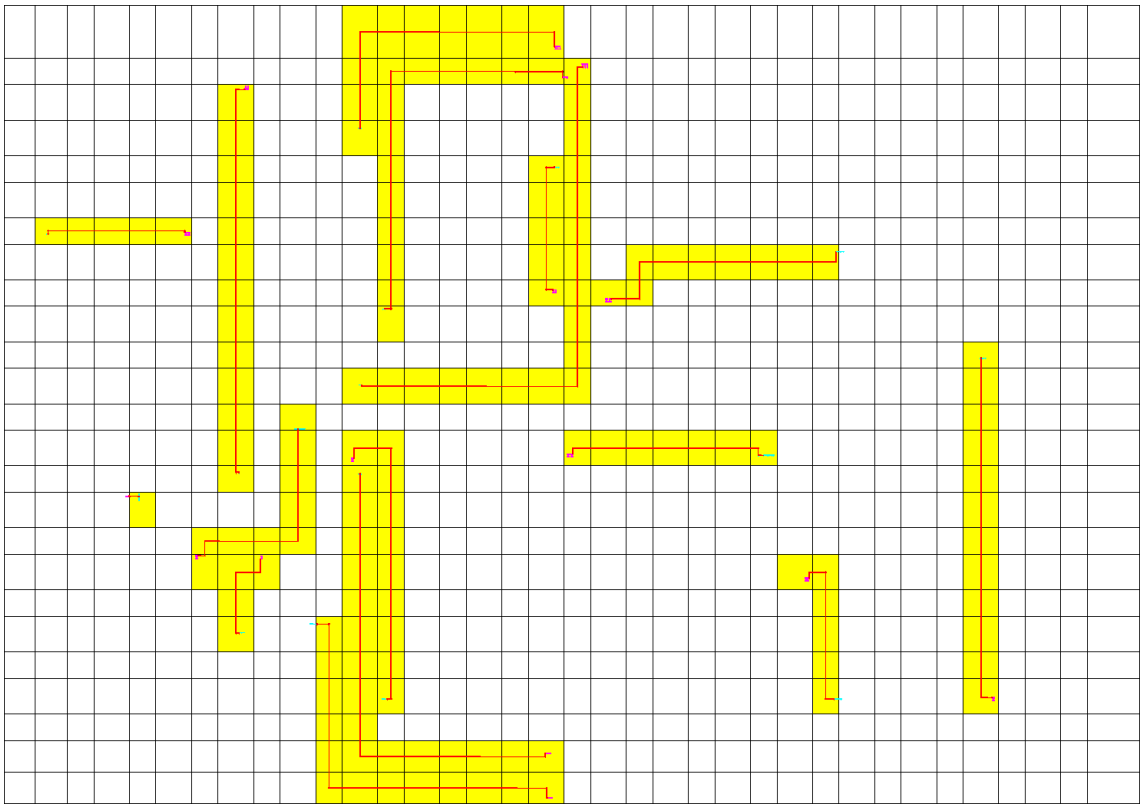


Figure 3.4: Active nets at some point in time with 16 threads. The cells used to decide which nets might conflict are drawn in black. Yellow cells are currently occupied by an active net. Red are global wires of active nets; magenta and cyan are the pins of the active nets.

net; after the limit is reached violations-allowing backup modes are activated to save run time on hopelessly overcongested designs).

Remark. In practice, BonnRouteDetailed saves some copies by not copying the original net of a rip-up sequence but this makes little difference because in the majority of cases the initial net does not contain any detailed wiring and thus can not be involved in any rip-up sequence anyway.

This change again reduces the number of collisions (especially during rip-up and reroute) and thus sometimes saves quite some run time. Both changes lead to less variability in the parallel detailed routing results and thus to more stability during the chip design flow.

3.6 Handling Diff-Net Rules

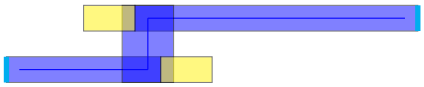
BonnRouteDetailed aims to produce diff-net-error clean output. To this end each piece of metal is assigned a so-called **shape class** which encodes distance rules that apply. We discuss which rules can be handled exactly and how to handle them efficiently in great detail in Chapter 5. In particular, in Section 5.3, we discuss how to precompute legality information stored in the fast grid to speed up frequent queries during path search. We show practical results for this framework in Section 6.3. We only mention some practically relevant exceptions here and refer to Chapter 5 and [40] for more details.

As described in Chapter 5, most diff-net rules can be modeled exactly. There are two practically relevant exceptions. The first one are run length dependent diff-net rules (see Definition 5.1.8 for a formal definition of run length). On some layers, some wire models need to obey larger distance rules if they have a common run length greater than some value r (with $r > 1$). This kind of diff-net rules can not be modeled exactly by our framework (see Lemma 5.1.13). But in practice, the value r is usually only moderately large such that a high percentage of pairs of wires that have common run length larger than zero and are close to each other have common run length larger than r anyway. Thus assuming the higher spacing requirements that apply for wires with run length larger than r for all wires is only slightly restrictive. Further, if track patterns are calculated for such a wire model, one has one track pattern for all wires of the wire model independent of their length (or common run length with other wires) anyway. Thus taking into account the larger spacing value for the wire model when calculating track patterns is usually required anyway. Thus this solution introduces only slight pessimism and makes sure that these diff-net rules are always fulfilled.

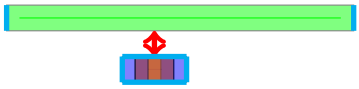
The second (and probably even more important) class of rules that can not directly be modeled exactly are so-called line end rules. These rules mean, roughly speaking, that if an edge between two convex vertices of a metal component is shorter than some threshold it needs to meet additional spacing requirements. There are three main cases when this rule occurs.

First, at the end of wires running in preferred direction whose width is small enough (line ends against preferred direction). Almost every end of such wires running in preferred direction has a line end and thus needs to meet the additional spacing requirements. Furthermore, assuming a line end against preferred direction at every position along a wire

(a) The most common case. Simple line ends against preferred direction are modeled exactly and spacing requirements are precisely obeyed.



(b) A rare but pessimistic case. If wires in preferred direction are continued with a wire against preferred direction, there is not necessarily a line end. The yellow areas are incorrectly blocked by the wire running in preferred direction (some or all of them is blocked by the wire running against preferred direction anyway).



(c) An incorrect case to be avoided. Line ends in preferred direction along wires are not modeled correctly and thus can lead to diff-net spacing violations.

(d) Line ends in preferred direction should be avoided. If there is space available, they can easily be fixed by an endstyle.

Figure 3.5: Modeling of line end rules. Blue and green indicate metal shapes on a routing layer (of different nets). Orange indicates via shapes. Red marks indicate violated design rules. Cyan edges indicate line ends.

in preferred direction usually blocks no additional shapes. Therefore, BonnRouteDetailed models this case slightly pessimistically by assuming a line end against preferred direction at the end of each narrow piece of wiring running in preferred direction (even if it is continued by another piece of wire running in preferred direction) (Figure 3.5(a)). There are only very few cases when this is really pessimistic, for example if the wire is continued with a wire running against preferred direction that prevents the resulting edge against preferred direction to be a line end (Figure 3.5(b)). Because jogs are seldomly used, this case occurs very rarely.

This technique can not handle so-called adjacent track line end rules, rules where a line end one some track interacts with line ends on neighboring tracks. If such rules are present, further techniques have to be employed. For more information about modeling line ends in BonnRouteDetailed, see [40].

The second case when line ends can occur are the side edges of short wires in preferred direction (line ends in preferred direction). This is in general an undesirable case anyway, because those wires in most cases block both neighboring tracks for further wiring (Figure 3.5(c)). Thus although not strictly illegal, BonnRouteDetailed tries to avoid to create this situation to save routing space. This can effectively be achieved by either adding endstyles to short wires running in preferred direction (Figure 3.5(d)) or, if this fails, by forcing wires running in preferred direction to be long enough via multi-label path search (basically the same mechanisms that can avoid min area violations for such wires).

The third case are vias or input wiring, but in these cases it is mostly known which edges are a line end and which edges are not a line end and thus the information if stricter rules apply to each edge can be encoded into the shape class of the pieces of metal.

With these techniques, BonnRouteDetailed achieves routings that are mostly diff-net error clean, apart from situations where the diff-net error aware path search framework fails completely (for example at blocked pins or in overcongested areas). In such situations, typically the input can, is and needs to be changed to enable clean detailed routing.

3.7 Pin Access

One of the trickiest parts of detailed routing is pin access. While struggling to simultaneously access all types of pins even in the densest regions, detailed routing tools need to pay special attention not only to diff-net rules but also to avoid same-net errors that can be created in combination with the pins and between wires of different wire and via models that are sometimes used to access pins. Furthermore, electrical characteristics need to be taken into account, especially close to the electrical source pin. BonnRouteDetailed has four mechanisms to efficiently access pins (and other off-track input metal).

The most important mechanism is the so-called circuit row pin access. We briefly summarize some key facts and refer for further details to [2]. The circuit row pin access is basically a dynamic program working in each circuit row to preselect one access path for each pin on the lowest layers (or already connecting close-by pins). First, a number of simple design rule clean candidate access paths and short connections are precomputed for each pin. Then, the dynamic program computes an optimal combination of these access paths for all pins within a single circuit row with respect to some objective function. Due to the nature of the dynamic program, a wide range of objective functions can be used. The objective function that BonnRouteDetailed uses incorporates the following criteria (in order of importance):

- maximize number of pins accessed
- maximize number of pins already connected
- use access paths going in the right direction
- spread access paths

Obviously, the main goal is to compute an access path for as many pins as possible (such that all access paths together are legal). Next, already connecting pins during the circuit row pin access leads to very efficient and short connections and is thus desirable. Furthermore, BonnRouteDetailed computes for each pin in which direction the rest of the net is located and tries to direct access paths already in that direction. Last, it is beneficial to spread the endpoints of the access paths in a way that no two endpoints are located too close together to make them easier accessible. The circuit row pin access in BonnRouteDetailed runs in $\mathcal{O}(n \log(n))$ time where n is the number of pins in the instance (that is all pins in the given circuit row) [2].

Figure 3.6 shows a small part of a sample solution found by the circuit row pin access. Note that all pins are accessed, there are already four connections made, all access paths point in a direction where something else of the net is located and all the vias concluding access paths end on different (vertical) tracks.

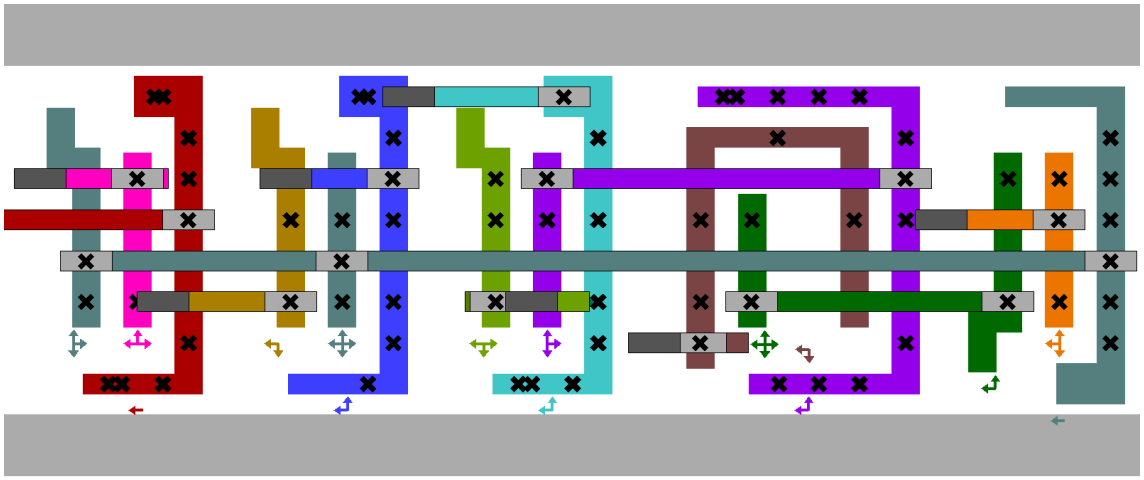


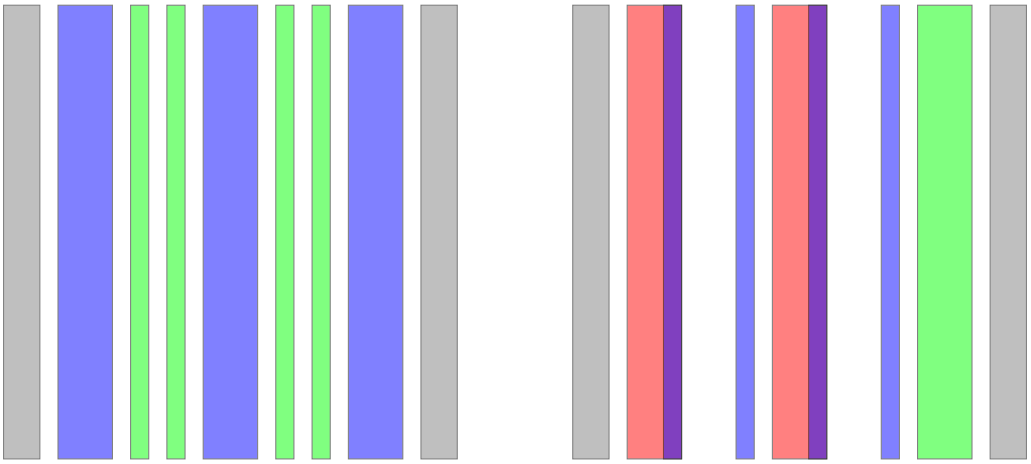
Figure 3.6: Circuit row pin access. The two grey horizontal stripes are power rails. Pins are colored according to the net they belong to. Black crosses mark legal via positions on the pins. Colored horizontal stripes are the computed pin access paths and short connections. Light gray rectangles indicate the vias accessing the pins and dark gray rectangles indicate vias to the next layer above. Small arrows indicate in which directions other parts of the nets are located. This picture was taken from [2].

Furthermore, BonnRouteDetailed has an on-the-fly pin access that can compute (potentially more complex) access paths for any pin (also higher-layer pins) during routing. The main reason for computing such access paths is that the path search in BonnRouteDetailed works on a grid graph and thus currently can not access pins that are not located on the path search grid. This version is on the one hand used for all off-grid pins that are not located on the lowest layers and on the other hand used if for some reason the preselected path for a low-layer pin can not be used. It works in a parametrized way, first, only simple and same-net-error clean paths are computed and then parameters are gradually relaxed to allow more complicated (but potentially also more error-prone) paths until some target number of paths per pin is reached. Undesirable paths (such as paths that contain or are likely to produce a same-net error) are assigned higher costs during path search.

Third, BonnRouteDetailed checks each pin prior to routing if it can be legally accessed with any path that the on-the-fly pin access can compute at all. If this is not the case, the pin is called blocked and BonnRouteDetailed precomputes a single (illegal) access path prior to routing. Furthermore, the pin is marked as blocked such that no attempts to access it legally are made anymore. Instead, further violation-mode paths are computed if necessary, ignoring certain design rules.

Fourth, BonnRouteDetailed has a module accessing off-track input wires with very simple access paths. Such off-track input wires should occur only rarely, but if they do, BonnRouteDetailed adds simple paths consisting of a via and potentially a short piece of wire in preferred direction to access them on the routing grid.

These mechanisms have been proven to be very efficient and increase the quality of results significantly (see [2]).



(a) Placing single wide wires after triple wide wire have been placed is trivial.

(b) Placing triple wide wires after single wide wires have been placed often requires lots of rip-up and re-route.

Figure 3.7: The order in which nets are routed is important. Wide wires need to be routed early to avoid many problems.

3.8 Net Ordering

The order in which nets are processed is very important. Nets that are routed early tend to make less detours and require fewer vias than nets that are routed late. Timing characteristics of early routed nets are significantly better because they have less detours and less undesired configurations such as layer and taper fuses (see Section 3.12 for a definition of layer and taper fuses). Further, the order of nets can greatly influence success of detailed routing as such and therefore the number of shorts and diff-net errors produced. Experiments showed that one simple key feature is most important for this fact. Nets with wide wire models need to be routed first. This is very plausible; it is much harder to find free space for a wide wire after most narrow wires have already been routed and the chip is already relatively full than it is to find space for a narrow wire after many wide wires have been routed (see Figure 3.7 for an example). Further criteria that can be taken into account are the timing criticality of a net (although critical nets tend to have wide wires so this usually coincides with the wire width of the net) and how hard it is to access the pins of a net. If there are only very few possibilities to access a pin of a net, it might be beneficial to route the net earlier than if there are hundreds of ways to access each pin, because if pins with few possibilities are routed late they are easily blocked by other nets while pins with many possibilities typically can still be accessed. For these reasons, BonnRouteDetailed orders nets by the following criteria lexicographically:

- average relative width and spacing of the default wire models of the net over all relevant layers
- average relative width and spacing of the pin access wire models of the pins of the net over all relevant layers

- some user-defined parameter indicating timing criticality of a net
- hardness to access the hardest to access pin of the net
- random value

Remark. In fact, for historical reasons, there are two different user-defined parameters indicating timing criticality per net which are used and BonnRouteDetailed uses one more minor criterion computed from the minimum over all components of the minimum of the x- and y-distance of the given component to the hardest to access component.

As discussed above, the width of the wire models (including spacing) is the most important criterion. Here, the spacing from the given wire model to itself is taken as it is usually a good indicator of the overall spacing requirements of a given wire model. Width and spacing are normalized by the minimum possible width and spacing on each layer to compare values for different layers. The default wire model is more important than the pin access wire models, because usually the majority of wiring of a net is comprised of the default wire model and only little is comprised of the access wire models. The user-defined parameters are taken into account next to give designers some control about the order of nets while still making sure that routing quality can not degrade too much by always considering widths of wire models before the user-defined parameters. Usually these user-defined parameters coincide with the width of the default wire models of a net anyway. The last important criterion is the hardness to access the pins of a net. BonnRouteDetailed calculates how many access paths the pin access code can find. If enough paths can be found without modifying any existing wiring, the pin is easy to access. If only very few paths can be found, the number of paths is relevant. If the pin can only be accessed by modifying existing wiring or even can not be accessed legally at all, it is considered very hard to access. Last but not least, if all these criteria are equal, BonnRouteDetailed computes a pseudo random number for each net to order them randomly. Experiments have shown that this is clearly superior to taking some arbitrary orderings like considering nets in the order they appear in the input if all criteria agree, probably because similar nets in the same region tend to be specified one after the other in the input thus degrading parallelization quality.

3.9 Path Search

As described in Section 3.5, each thread of BonnRouteDetailed routes one net at a time. For a given net, BonnRouteDetailed routes again one component at a time. BonnRouteDetailed always connects the closest component to the component that contains the electrical source. Distances are measured with respect to the global wire graph.

For each component to be routed, different path searches are tried. First, BonnRouteDetailed tries to route the component without rip-up and with very restrictive parameters concerning allowed layers, size of the routing area and protections. If for some reason no solution is found, BonnRouteDetailed tries the same parameters but allows rip-up and re-route. If again no path is found, BonnRouteDetailed relaxes the parameters and tries again. The relevant parameters are divided into three sets, parameters that concern the source component, parameters that concern the target component and parameters that

concern the connection globally. If BonnRouteDetailed detects a problem at a specific location, only the relevant parameters are relaxed. For example, if it fails to compute any access location at the source component, it only relaxes the parameters for the source component. BonnRouteDetailed alternates for each parameter set between normal and rip-up path searches until the most relaxed parameter set is reached. If still no path is found, BonnRouteDetailed consecutively activates three backup modes. First, it allows at high costs to conflict to other wires without ripping them up and re-routing them. Second, it allows to conflict to any metal object on the chip at very high costs. Last, it generates a connection from the global route by attaching it to the source and target component purely geometrically without checking any rules. Like this, it is guaranteed that BonnRouteDetailed computes a connection for each net unless some fundamental data is missing (such as if there is no wire model for a net at all).

For a given parameter set, BonnRouteDetailed calculates source and target locations, a routing area for each wire and via model, ignored shapes and protections. An example sequence of computed paths with their routing areas can be found in Figure 3.8. If rip-up is allowed, a rip-up sequence is started (see Section 3.10 for further details about rip-up and re-route).

Then, BonnRouteDetailed starts the multi-label framework (see Section 3.11 for further details about multi-labeling). Within the routing area, a local track graph is precomputed containing only the tracks for the relevant wire models (separately for different regions of the routing area because some wire models are typically only allowed in certain regions of the routing area) [1]. Additional coordinates can be added to locally access pins but currently this is not done. Then, a path with the least restrictive label system is searched and post-processed. If this path contains same-net errors, the least restrictive label system that can avoid the maximum number of the contained errors is calculated and a new path search is started. This is iterated until either the path is legal with respect to same-net errors or the most powerful label system is reached or no path is found. Then the path with the fewest errors is taken. Computed legality information from the last path search with the same parameters but a less restrictive label system are reused for the next path search to reduce run time.

The path search itself is a very fast multi-label path search [1] based on Dijkstra's algorithm [16] that supports costs per layer, direction and wire model. Further it supports additional costs for certain nodes, tracks and high net-dependent costs to rip-up nets or violate diff-net rules. Additional costs on some (but not all) tracks of some layers are used to spread wires efficiently in areas with low congestion. The path search uses future costs that compute for given cost per layer and direction (including costs for vias per via layer) a shortest path in the complete grid graph. The main task here is to calculate which layer should be considered for the future costs. For further information about the future cost, we refer to [23] and [1]. The future costs can not take into account rip-up costs, additional costs for individual nodes and additional costs for individual tracks.

For further information about the path search inside BonnRouteDetailed, we refer to [1].

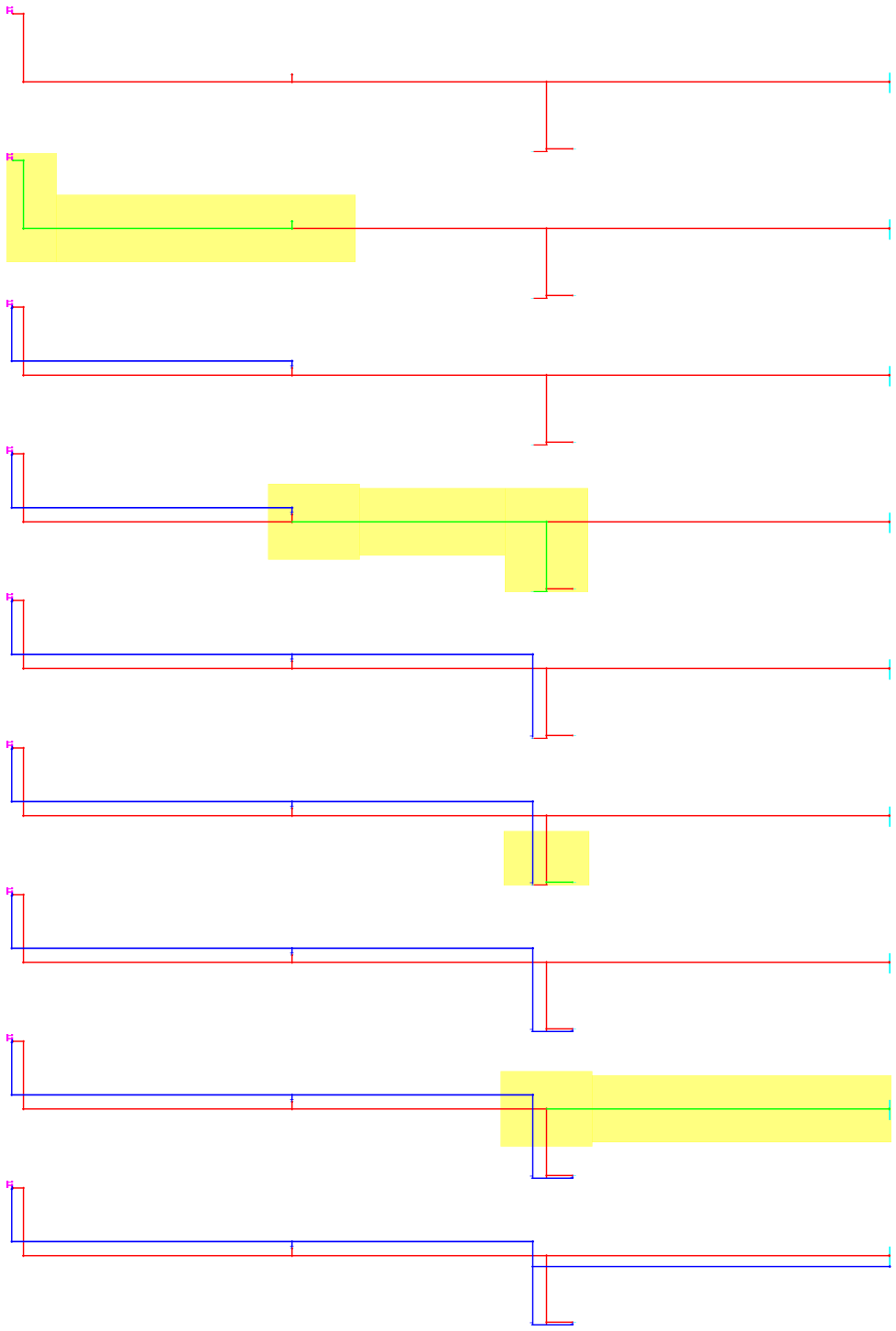


Figure 3.8: Routing a multi-terminal net. Global wires are drawn in red, detailed wires in blue. Source pins are colored magenta and sink pins cyan. Routing areas are colored yellow and the currently active part of the global route is green.

3.10 Rip-Up and Re-Route

As briefly mentioned in Section 3.1 and Section 3.9, sometimes it is necessary to change already routed nets in order to route some other net. To guarantee convergence, BonnRouteDetailed always maintains connectivity of already routed nets. This is done with the help of so-called rip-up sequences. When rip-up is required to route a net, a rip-up sequence is started. First, a path that can (at high costs) conflict to existing modifiable wiring of other nets is searched. If no such path exists, it is impossible to route the net without introducing conflicts to non-changeable objects. Rip-up thus fails and further measures to compute a connected routing are taken. If such a path is found, the state of the concerned net is copied for potential later restoration. Then, the set of all shapes of other nets that need to be removed and re-routed in order to make the new path legal is calculated. These shapes are processed one by one. First, the concerned net is identified and copied. This serves two purposes. First, if the rip-up-sequence fails, the original net is not modified. Furthermore, if BonnRouteDetailed does not work on the original net but only on a copy, it can be guarantee during parallel computation that in case of a collision, at least one result can successfully be applied. For any given shape that needs to be ripped-up and re-routed, the path that needs to be ripped-up and re-routed is calculated with the help of the net data structure briefly described in Section 3.4. Furthermore, a corresponding global route is found and source and target locations are calculated. Then a new path search is started. First, only legal paths are considered. If this succeeds, the conflict is successfully resolved. BonnRouteDetailed can apply the new path to the copied net and has one violation less to deal with. If it fails, BonnRouteDetailed tries to find a path that in turn can have violations to other nets. If such a path is found, the new violations are added to the set of shapes that need to be ripped-up and re-routed and the path is applied to the copy of the net. If no path is found, BonnRouteDetailed aborts the whole rip-up sequence and restores the original state of all nets. This only happens in rare circumstances, e.g. if a time or memory limit is encountered. Furthermore, if a rip-up sequence rips out too many nets, it is unlikely that it successfully finishes in reasonable run time. Thus there is a limit on the total number of connections that can be ripped-up by any rip-up sequence. If this limit is reached, BonnRouteDetailed also aborts the rip-up sequence.

To avoid cyclic rip-up sequences, node costs are temporarily increased for all nodes covered by conflicting ripped-up wire sticks. Therefore it is less likely that while re-routing a ripped-up route the old route is found again. These node costs are cleared after each component is routed.

To make this framework work in a parallel environment, it needs to be ensured that no global data structures are changed before the rip-up sequence is successfully completed. To ensure this, a number of additional data structures are necessary. BonnRouteDetailed write-locks the original net, so it can modify (and potentially reset) it during a rip-up sequence. BonnRouteDetailed copies all other involved nets and modifies only the copies until the rip-up sequence is successful. It does not modify the grid or the fast grid until the rip-up sequence is complete, but uses additional thread-local data structures $\mathcal{IS}_t$ and $\mathcal{AS}_t$ for each thread $t \in \mathcal{T}$ to store **temporary ignored** or **additional shapes** for each thread. Thus everything can easily be reset in case a rip-up sequence fails. If a rip-up

sequence is successful and all violations can be resolved, all areas that can affect legality of all computed shapes and all affected nets need to be locked simultaneously. If some net has been changed in the mean time, there is a collision and the rip-up sequence needs to be restarted. Likewise if some computed shape has become illegal in the meantime. Otherwise all computed changes can be applied and everything can be unlocked. Like this, it is made sure that either all changes are applied, increasing the number of successful connections by one, or everything is reset to the state before the rip-up sequence started. For further details on parallel implementation of rip-up and re-route, we refer to [27].

3.11 Handling Same Net Rules

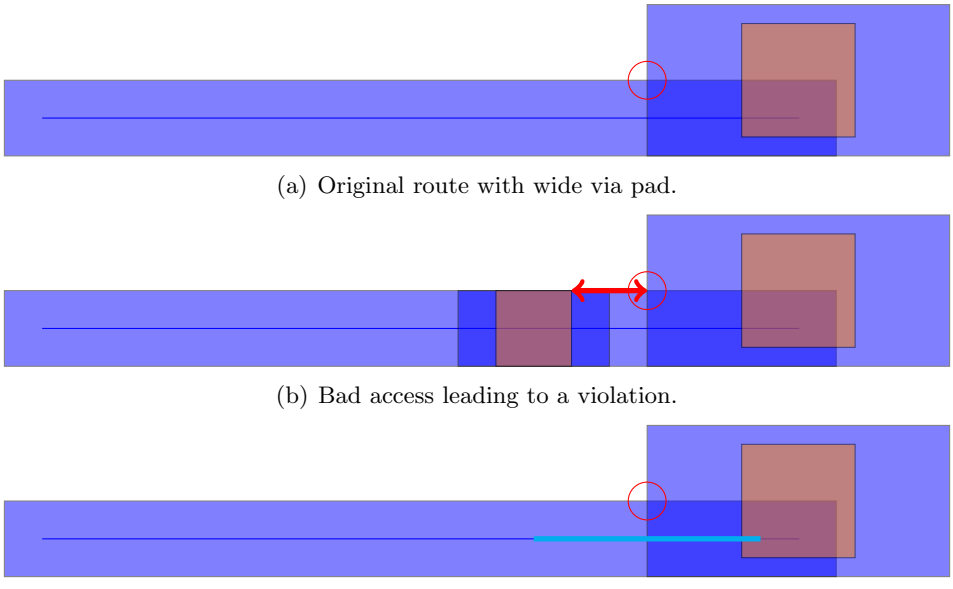
Handling same-net rules efficiently during detailed routing has become more and more important with each new technology. While physical structures on the chips have become smaller, design rules have become more and more complex. In the past, it was sufficient for BonnRouteDetailed to pay special attention to design rule clean pin access and post-process paths after they have been found to fulfill same-net design rules (see for example [19]). In recent technologies, this has no longer been true. Therefore, a number of more advanced mechanisms have been introduced to BonnRouteDetailed to handle same-net rules. We summarize the most important concepts here and refer to previous works for more details.

One of the oldest concepts implemented in BonnRouteDetailed is the so-called pin shrink. For each pin, direction and wire or via model, one can compute a subset of the points contained in the pin where accessing the pin with a wire or via of the given model in the given direction is same-net rule clean (for wires assuming that the piece accessing the pin is sufficiently long). BonnRouteDetailed has implemented pin shrinks for many same-net rules. Therefore, when accessing a pin, especially with a via, the set of points where the pin can be legally accessed is known. This reduces the number of same-net errors made during pin access greatly. Figure 3.9 shows an example for a very simple via pin shrink, the so-called contained pin shrink, calculating the positions where a via pad is completely contained in the pin shape. This pin shrink is extensively used especially on the lowest layer touched by routing, because one usually does not wish to add any additional metal on that layer due to the fact that there are extremely complex rules between metal on this layer and the layers below. Usually routing tools do not even know the exact shapes below this layer because there are very many shapes on the lowest layers and even storing them imposes a substantial overhead. Therefore pins on this layer are exclusively accessed by vias from above that are completely contained in the pin. Additionally, more complex kinds of pin shrinks are used on all routing layers.

Unfortunately, the use of a correct pin shrink alone is not enough to avoid same-net rule violations at pins and it does not help at all to avoid same-net rule violations at Steiner points. Therefore, same-net errors are avoided by another concept in certain situations. For example consider a route that contains a piece of wire running in preferred direction that is continued with a via whose pad is wider than the wire, shown in Figure 3.10(a). The metal component formed contains a concave vertex. In many technologies, placing a via too close to such a concave vertex is forbidden. Therefore, if the route shown in Figure 3.10(a) is part of a multi-terminal net, it might be accessed as shown in Figure 3.10(b) by a via

(a) Via pad (b) Pin (c) Pin shrink (gray: pin, blue: area where the stick of a via can be placed)

Figure 3.9: Via contained pin shrink



(a) Original route with wide via pad.

(b) Bad access leading to a violation.

(c) Computed protections.

Figure 3.10: Avoiding via-middle-to-concave-vertex errors with protections. Blue indicates metal shapes on a wiring layer, orange shapes are via middle shapes. Concave vertices of the metal component are marked with a red circle. The red error indicates the violation. Protections are marked in cyan.

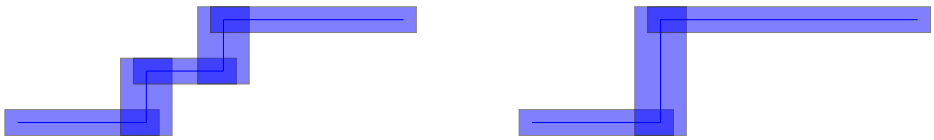
close to the existing concave vertex which would lead to a same-net error. To avoid such kind of errors, BonnRouteDetailed uses so-called protections. Before each path search, for each usable wire and via model, a set of points is calculated where placing a wire of the given wire or via model would very likely lead to a same-net error. Then during path search these positions are forbidden. Due to the fact that some of these protections are slightly over-restrictive, they are relaxed during later path searches if no path is found. Figure 3.10(c) shows example protections for the via-middle-to-concave-vertex rule.

The concepts described above deal with same-net errors at pins or Steiner points. Of course, same-net errors can also occur within a single path. To avoid such errors, BonnRouteDetailed employs two more concepts. First, paths are post-processed to avoid same-net errors. Two techniques are used. First, jogs are moved within the path to combine multiple short jogs to one longer one and to move jogs away from vias, because jog-via combinations often lead to a number of same-net errors. Second, endstyles are added to wires or to via pads in order to fulfill certain same-net design rules (or in one case avoid diff-net rule violations). The most important post-processing routines are:

- combine jogs (Figure 3.11(a))
- move jogs away from vias (Figure 3.11(b))
- fix min area violations with endstyles (Figure 3.11(c))
- remove line ends in preferred direction with endstyles (Figure 3.11(d))
- align via and wire shape with endstyles (Figure 3.11(e))
- move line ends against preferred direction away from vias to respect certain design rules (Figure 3.11(f))

All these post-processing routines are very fast and most same-net design rule violation within paths can already be resolved this way. For more information about endstyles and post-processing in paths in BonnRouteDetailed, see [41].

Some error classes are not handled by post-processing routines and some design rules that are handled are not always fixed. Therefore, if design rule violations remain, an even more powerful (but also slower) concept, multi-label path search, is employed. Multi-label path search in BonnRouteDetailed was initially described in [36] and further discussed and improved in [18], [3] and [1]. The basic concept is to use multiple labels per grid node to pose certain restrictions on segment length after different types of segments (wires running in preferred direction, jogs and vias) have been used. For example, min area errors can be avoided if after each via one needs to go a certain length (depending on the width of the wire model and the required min area value) in preferred direction before being allowed to place another via. Furthermore, so-called via bridges, two vias of the same net that are connected by a very short wire in preferred direction and whose middle shapes are too close together to be legal, can be efficiently avoided by multi-label path searches. Again, after placing a via, one needs to go a certain distance (in any direction) before being allowed to place a via on the layer where the former via was placed. Note that for this rule a via in the other direction is fine independent of the distance to the first via. Additionally, multi-label path searches can represent colored wires in an effective way



(a) Combine multiple jogs to a single one to reduce likelihood of several classes of design rule violations.

(b) Move jogs away from vias to reduce likelihood of several classes of design rule violations.

(c) Fix min area violations with endstyles.

(d) Remove line ends along preferred direction with endstyles (cyan indicates line ends).

(e) Align wire and via shapes with endstyles. In this case, two forbidden adjacent short edges errors are fixed by the modification.

(f) Move line ends against preferred direction away from vias to respect via middle to line end distance rules.

Figure 3.11: Post-processing paths in BonnRouteDetailed. Blue shapes are located on wiring layers, orange are via middle shapes. Red indicates design rule violations.

and thus avoid coloring problems. Furthermore, segments at the start and the end of a path can be forced to be long enough and running in preferred direction to avoid certain same-net errors at the start or end of a path. Currently, the most important purposes concerning same-net rules of multi-label path search are:

- color paths correctly
- avoid design rule violations at the end of paths
- avoid illegal via bridges
- avoid illegal via towers
- avoid illegal jog via combinations
- avoid min area errors
- avoid notch errors

BonnRouteDetailed does not always employ all kinds of multi-label restrictions, but chooses the multi-label system according to the errors present after post-processing a path. It chooses the least restrictive label system that can fix all present errors, finds a new path and if necessary iterates. In the end the best path found (the path with the least weighted number of errors with some experimentally tuned weights) is taken. For more information about avoiding design rule violations by multi-label systems in BonnRouteDetailed, see [1].

The concepts presented here can avoid most same-net design rule violations, but still a small number of such errors remain. Therefore, in practice, BonnRouteDetailed is used in combination with an industrial routing tool. BonnRouteDetailed is first run, producing very dense and efficient routing solutions in low run time, but leaving some design rule violations. Then the industrial routing tool is run, cleaning up the remaining violations and hopefully disturbing routing quality not too much (see [40], [19] and [3]).

3.12 Handling Timing Requirements

BonnRouteDetailed does not explicitly calculate timing information itself. Instead it uses a twofold strategy to deliver detailed routing with good timing characteristics. The first part relies on a good global routing. BonnRouteGlobal (and probably every major global routing tool) can produce timing aware global routings and is heavily used to optimize timing. Timing aware global wires (including wire models and layer assignments) are used as input to BonnRouteDetailed. BonnRouteDetailed tries to produce detailed routes that are (with regard to wire models, layers, topology and geometry) similar to the global routes. An example global and detailed routing can be found in Figure 3.12. In this way, good global timing characteristics are mostly maintained by the detailed routing. We refer to Section 3.4 and [26] for further details on the net data structure and routing algorithms enabling close global-detailed routing correspondence.

Secondly even if detailed routing is similar to the global routing, it must locally have good timing properties. When timing is concerned, one feature is predominantly important. Lots of capacitance topologically behind high resistance leads to long delays and

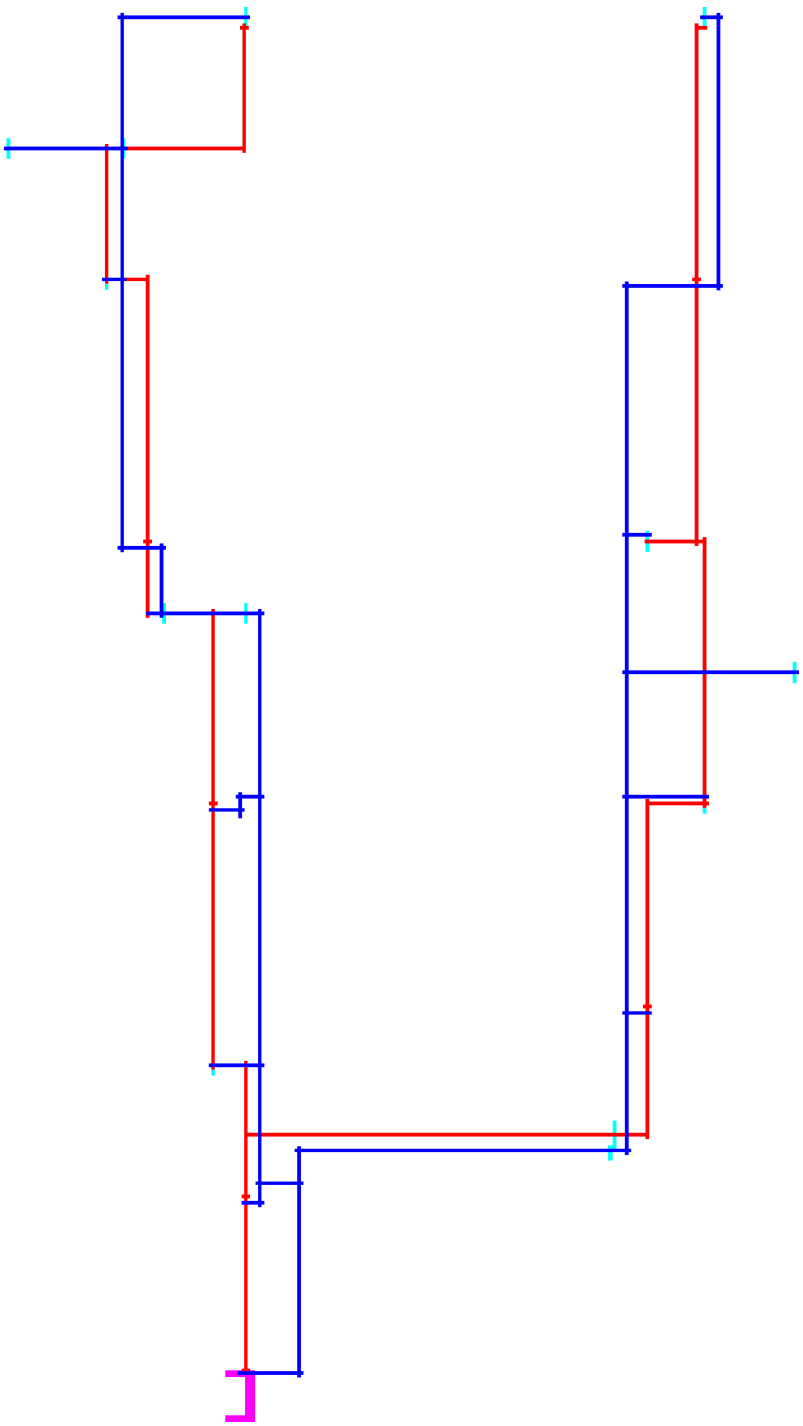
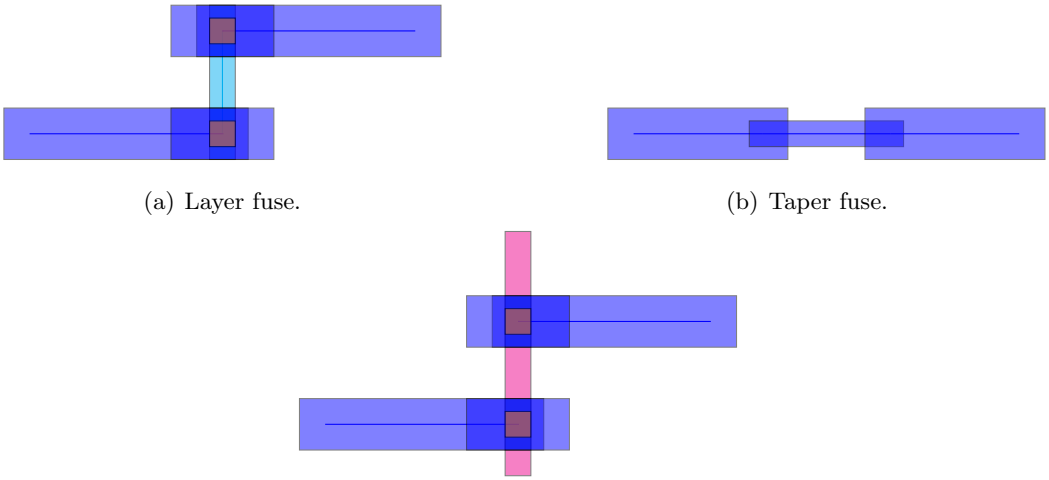


Figure 3.12: Detailed wiring (blue) and corresponding global wiring (red). The sink pins are drawn in cyan and the source pin is marked in magenta. Note the similarity of the global and the detailed routing structure.



(a) Layer fuse.

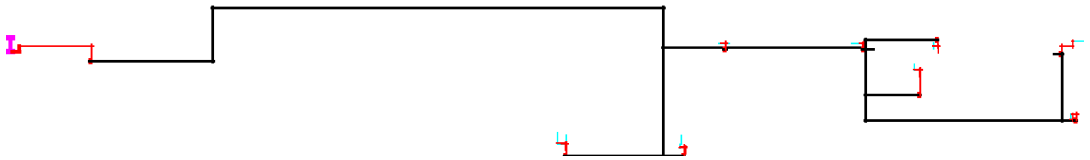
(b) Taper fuse.

(c) Pin fuse.

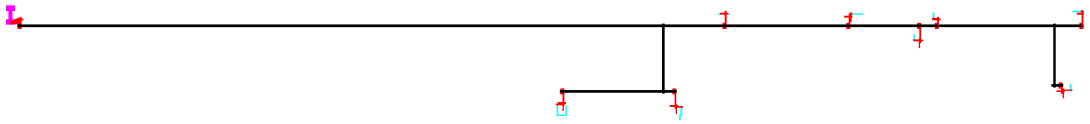
Figure 3.13: Timing-wise bad configurations. Blue are shapes on an assigned layer. Cyan are shapes on a lower layer. Orange are via middle shapes. Magenta are pin shapes on a lower layer.

thus to bad timing behavior. Capacitance is usually pretty much determined by global routing and unmodifiable features of a net. During detailed routing, BonnRouteDetailed can not change circuits or pins, thus input capacitance of the sink pins is fixed. Small detours typically lead to small changes in capacitance, therefore if the rough structure of the net is fixed, capacitance is basically predetermined. But local resistance very much is dependent on local features such as layers, wire models and via models. Even a very short piece of thin wire close to the source of a net can add a significant amount of delay due to the added resistance which has to drive most of the capacitance of the net. Likewise, a bad via or a wire on the wrong layer (which is thus thin) or a route going through a pin shape with high resistance can easily make the timing of a net degrade. Sometimes it is still unavoidable that low layers or thin wires are used to access the source pin of a net (if the pin is located on a low layer, somehow the route needs to go up and if the pin is narrow, there needs to be a legal transition to a wide wire).

Therefore, in the optimal case, source-sink-paths in the beginning transition to the assigned layers and wire width as quickly as possible, then remain on the assigned layers and with the assigned wire width and in the end somehow connect the sink pin (small deviations close to the sink pin usually are not so important because they drive much less capacitance). In particular, it is undesirable if within one source-sink-path the router switches down from the assigned layers and then back up to the assigned layers (Figure 3.13(a)), or if it switches from a wide to a thin and back to a wide wire model (Figure 3.13(b)). Furthermore it is often undesired if the path from the source to a sink contains another sink pin (Figure 3.13(c)). We call these cases **layer fuse**, **taper fuse** and **pin fuse** respectively. (Taper fuse because thin wire models usually are exclusively used to access pins, so-called taper wire codes). Therefore, there are two objectives for detailed routing. First, keep the usage of thin wire models and lower layers low (for nets



(a) Bad solution. Long piece of thin wiring at the source pin.



(b) Good solution.

Figure 3.14: Two detailed routing solutions for the same net. Black wires are thick and on high layers. Red wires are thin and on lower layers. The source pin is magenta, sink pins are cyan.

where timing matters). This is achieved by restricted routing areas and largely increased costs below the assigned layers. Therefore, long wires below the assigned layers and long thin wires become very unlikely. In areas where no pins need to be accessed, BonnRouteDetailed can even guarantee not to place any wire below the assigned layers. Figure 3.14 shows two routing solutions for a net. A good one and one with a long piece of thin wire on low layers right at the source pin leading to a timing degradation.

Second, BonnRouteDetailed has a number of mechanisms to avoid layer, taper and pin fuses. Pin fuses can completely be forbidden by selecting source and target locations appropriately. Layer fuses are made improbable by routing layer costs and by an appropriate choice of source and target locations but still can occur. If they do, BonnRouteDetailed tries to avoid them by multi-labelling (it is easy to create a label system that allows going up to and down from the assigned layers exactly once). Taper fuses can also be made unlikely by appropriate choice of source and target locations and small routing areas for access wire models. If they still occur, a post-processing routine tries to eliminate them by switching wire models along the path. Due to track pattern restrictions this can fail. In this case, this could also be fixed by multi-labelling (it can be more time consuming than fixing layer fuses because three or more different kinds of wire models can be involved) but this is currently not implemented in BonnRouteDetailed. Overall, BonnRouteDetailed leaves very few timing-wise undesired configurations.

Chapter 4

Computing Track Patterns

In this chapter, we deal with the problem to compute good track patterns for a given set of wire models. The terms tracks and track patterns both denote a set of coordinates that are used to place wires. We use the term tracks when we want to emphasize that we deal with a general set of coordinates and the term track pattern when we want to emphasize that we have a specially structured set of tracks (e.g. tracks that are repeating between each pair of power rails).

BonnRouteDetailed relies on a very powerful and optimized path search framework finding shortest paths in an implicitly given grid graph. For this model to work, BonnRouteDetailed requires special coordinates where most of the wires in preferred direction are placed, so-called tracks. On some layers, tracks are predefined by technology requirements (we call these **hard track patterns** because they need to be obeyed in any case). On many layers, there are no such restrictions a priori, but it is very beneficial to restrict the path search to predefined track patterns both to speed up the path search and to facilitate efficient packing of wires. We call such additional track patterns **soft track patterns** because they do not need to be obeyed everywhere; for example to access pins, short wires can be placed on different coordinates. Due to the large number of different wire models and layers on modern chips, it is both very laborious and very error-prone to define all such soft track patterns by hand. Additionally, if track patterns are computed automatically, they can be computed separately on each chip taking into account the individual mix of wire and via models. Furthermore, the choice of soft track patterns has a huge influence on routing quality. Therefore, in this section, we discuss the automated computation of good combinations of soft track patterns per chip. Our automatic soft track pattern generation reduces run time by 40% and 33% on 14nm and 7nm instances respectively and improves almost all measured metrics significantly. Detailed experimental results can be found in Section 6.2.

This chapter is organized as follows. In Section 4.1 we formalize the problem and specify the input. In Section 4.2 we discuss a number of simple track patterns and their shortcomings and develop objectives for optimized track patterns. In Section 4.3 we describe a dynamic program to calculate optimized track patterns efficiently, prove its correctness and discuss its theoretical run time. In Section 4.4 we discuss some practical aspects concerning optimized track patterns.

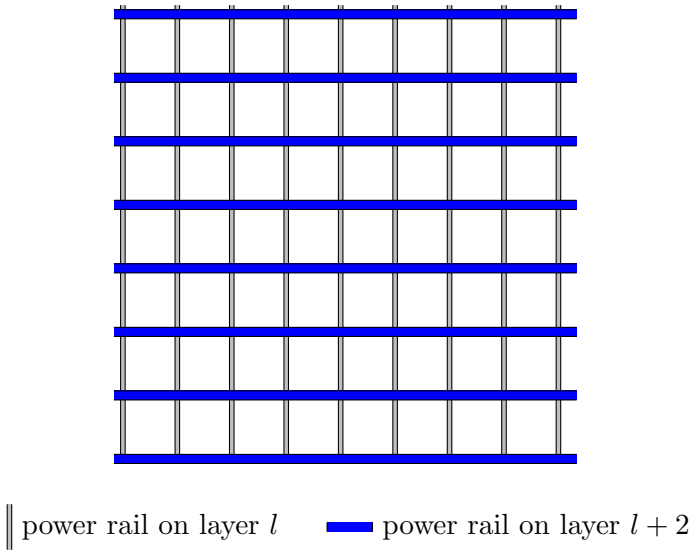


Figure 4.1: Power grid. Neighboring layers have alternating preferred direction. Usually power rails are placed in regular intervals. On each intersection of the power rails on neighboring layers there is a power via connecting the power rails.

4.1 Input and Problem Specification

Wiring layers have alternating preferred direction, therefore track patterns on adjacent wiring layers are basically independent. Thus for the purpose of this section we can restrict ourselves to deal with a single wiring layer $l \in \mathcal{L}_{wiring}$. The vast majority of instances has a uniform power distribution structure and almost all wires run in preferred direction. Therefore, we compute one uniform track pattern for each wire model for the whole instance. Consequently, it does not matter which direction is the preferred direction on layer l, we only consider coordinates in the non-preferred direction anyway. Although definitions and algorithms work independently of the preferred direction, for examples and descriptions we assume the preferred direction to be vertical for this chapter.

On most layers, power is distributed via so-called power rails. That means, there are stripes of metal spanning the whole length of the chip in preferred direction in regular intervals in non-preferred direction.

Because power distribution is both subject to different constraints and very critical for the design of a chip, it is determined long before detailed routing. Thus we assume power rails are fixed input and do not try to optimize them. Figure 4.1 shows typical power rails on two neighboring layers. Formally, we get two numbers, the **power rail width** $w_p \in \mathbb{N}$ and the **power rail pitch** $p \in \mathbb{N}$. W.l.o.g., we assume that the center of the first power rail is located exactly at coordinate zero. We further assume that w_p is even. This means that we have power rails at $[kp - \frac{w_p}{2}, kp + \frac{w_p}{2}]$ $(k \in \mathbb{N})$. Furthermore, we get a **set of relevant wire models** $WM \subseteq \mathcal{WM}$. We assume $WM = WM_o \cup WM_i$ with $WM_o \cap WM_i = \emptyset$ and $WM_o \neq \emptyset$. WM_o is the set of wire models we want to optimize, WM_i is an additional (possibly empty) set of wire models for which we have a track pattern given that we want

to take into account while optimizing. For a wire model $wm \in WM$ we define the **width** of wm:

$$w(wm) := \begin{cases} x_{max}(r(wm)) - x_{min}(r(wm)) & pref(l) = vertical \\ y_{max}(r(wm)) - y_{min}(r(wm)) & pref(l) = horizontal \end{cases}$$

We assume that for all $wm \in WM$ we have $w(wm)$ even. This is necessary to have a stick at the center of the wire model at an integer coordinate. If it is not the case, one can scale the coordinate system to achieve this assumption. Furthermore, we assume $x_{max}(r(wm)) = -x_{min}(r(wm))$ (if $pref(l) = vertical$) or the analogous statement in the horizontal case (this means, we assume that a stick lies in the center of its shape). If this is not the case, one can simply move the computed tracks to match the origin of the wire model. Moreover, we assume $w(wm) > 0$ for all $wm \in WM$.

In Chapter 5 we show how do deal with a wide range of distance rules between metal shapes. However, in this section we can restrict ourselves to a much simpler model. We are not interested in local rules occurring at the end of wires or between vias, the only relevant rules are those that apply between two long wires with a certain wire model (and thus shape class) running in preferred direction. In this case, almost always a single minimum distance value suffices. Therefore, we define minimum spacing values for each pair of wire models and for each wire model and the power rail. Formally, for $wm_1, wm_2, wm \in WM$ let $s(wm_1, wm_2) > 0$ be the **required spacing** between the shapes of two wires running in preferred direction with wire models wm_1 and wm_2 respectively and let $s_p(wm) > 0$ be the **required spacing** between the shape of a wire running in preferred direction with wire model wm and a power rail. We restrict ourselves to single-colored layers here (and thus do not discuss colors at all) because on most colored layers there are further restrictions on the track patterns and thus usually track patterns are hand-coded and obligatory anyway. We briefly mention how to extend our results to multi-colored layers in Section 4.4. Figure 4.2 illustrates power rails, width and spacing constraints.

Remark. Sometimes, vias on a given layer have wider pads than the wires they are used for. In this case it might be beneficial to take the width and spacing of the via pad into account when calculating track patterns. We leave this as an opportunity for future research.

Furthermore, on most chips, the frequency of different wire models varies a lot. Often, the most frequent wire model accounts for much more than half of the total wire length on a chip while the third or fourth most frequent wire model accounts for less than one percent. Therefore, while computing track patterns it makes sense to take into account the frequency of a given wire model. An inefficient track pattern for a less frequent wire model is much less harmful than an inefficient track pattern for the most frequent wire model. To this end, we estimate the total wire length for each wire model before routing and use the relative estimated wire length for our algorithm. Formally, for a wire model $wm \in WM$ let $f(wm) \in (0, 1]$ be the **relative estimated frequency** of that wire model.

Remark. On some chips, the relative estimated frequencies of wire models may vary significantly over different regions of the chip. In this case it might be beneficial to compute different track patterns for different regions. On the other hand, transitioning between different track pattern regions is highly non-trivial and might introduce severe local routing problems. Therefore, we do not pursue this approach.

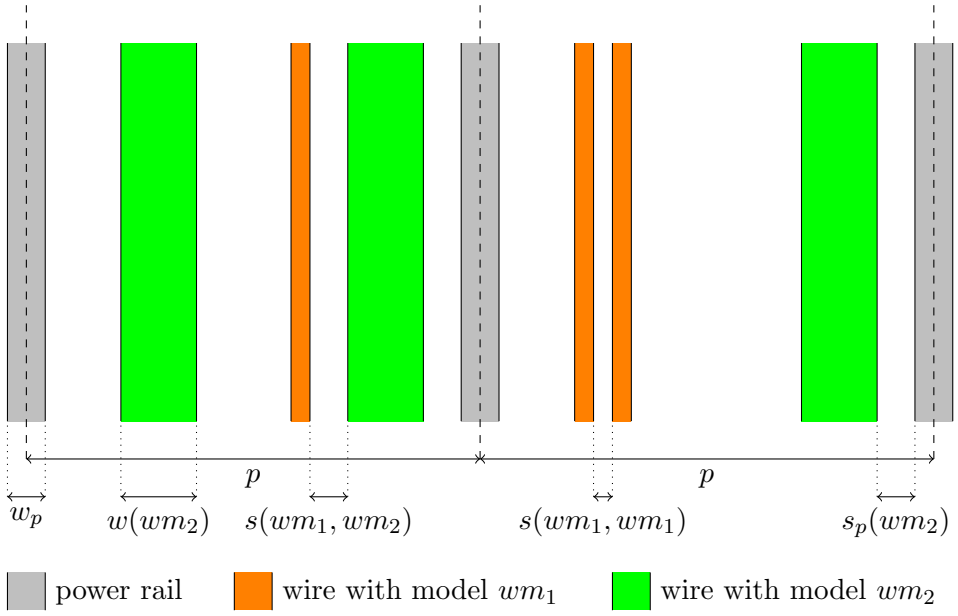


Figure 4.2: Input specification.

We explicitly exclude a relative estimated frequency of zero here, because the wire model would be irrelevant for optimization anyway and because this leads to numerical problems.

Remark. All input values should be of reasonable size, in particular not too close to zero to avoid numerical problems. We assume this is the case and do not discuss it any further.

Of course, we expect $\sum_{wm \in WM} f(wm) \leq 1$. We only assume less-or-equal instead of equal because we might only consider a subset of all wire models. For some set of wire models $WM_1 \subseteq WM$ we define $f(WM_1) := \sum_{wm \in WM_1} f(wm)$. In Section 4.4 we describe how we compute the relative estimated frequencies.

4.2 Simple Track Patterns

In this section we describe a number of natural simple track patterns and show their advantages and limitations. This also motivates the next section. To simplify the notation, we first define adjusted spacing values as the spacing values that are implied between the sticks of the wires and power rails (or in other words the tracks). Let $wm_1, wm_2, wm \in WM$. We define

$$\bar{s}(wm_1, wm_2) := s(wm_1, wm_2) + \frac{w(wm_1)}{2} + \frac{w(wm_2)}{2}$$

and

$$\bar{s}_p(wm) := s_p(wm) + \frac{w(wm)}{2} + \frac{w_p}{2}$$

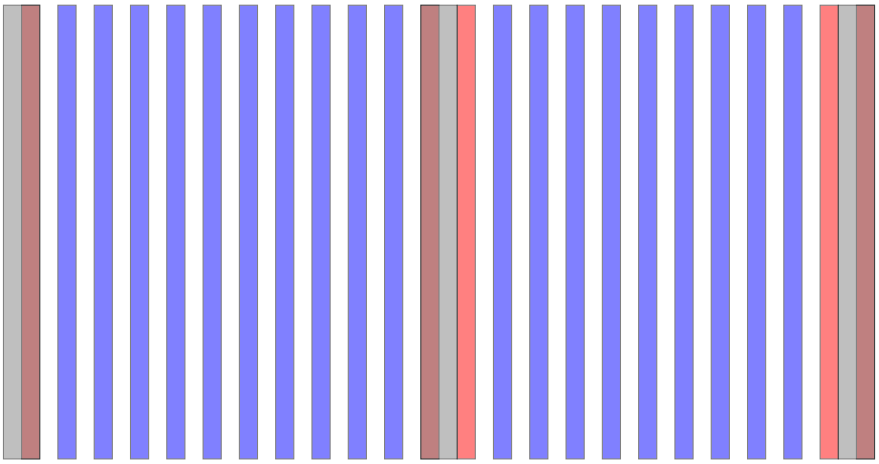


Figure 4.3: TP1 for single wide wires. Red wires are illegal due to the power rails. Dark red indicates (illegal) wires overlapping power rails. Note that in every second power bay one track is wasted.

Now, consider a wire model wm with width $w(wm)$ and spacing requirement $s(wm, wm)$. The simplest possible track pattern might be to start at any arbitrary coordinate t_0 and then place each track as close as possible to the previous track. This yields tracks at

$$\{t_0 + k\bar{s}(wm, wm) : k \in \mathbb{N}\} \tag{TP1}$$

Note that TP1 contains infinitely many tracks. In practice we only need the finitely many tracks that intersect the chip area. We do not restrict the number of tracks here for simplicity of notation.

If there were only wires with wire model wm on the chip, this would be perfect. Unfortunately this is never the case, as there are at least some power rails. Our simple tracks might or might not pack well with regard to the power rails, but in any case we now have some tracks that overlap with the power rails and thus can not be used (Figure 4.3).

A better way to define a simple track pattern is to consider the space between two power rails, pack as many wires as fit between them and then repeat the same pattern between the next power rails. In fact, from now on, we define all our track patterns between two consecutive power rails (and they are repeated between each two consecutive power rails).

For $wm \in WM$ we define $n(wm)$ to be the **number of tracks of wire model** wm that fit between two power rails. In the following, we assume $n(wm) \geq 1$. If this is not the case, the wire model is useless and can be discarded. One easily computes

$$n(wm) = \left\lfloor \frac{p - 2\bar{s}_p(wm)}{\bar{s}(wm, wm)} \right\rfloor + 1$$

For our wire model wm we get

$$\{kp + \bar{s}_p(wm) + l\bar{s}(wm, wm) : k \in \mathbb{N}, 0 \leq l < n(wm)\} \tag{TP2}$$

TP2 now gives us as many usable tracks on the chip for wire model wm as possible (Figure 4.4). Still, if wires of the wire model do not fit perfectly between the power rails,

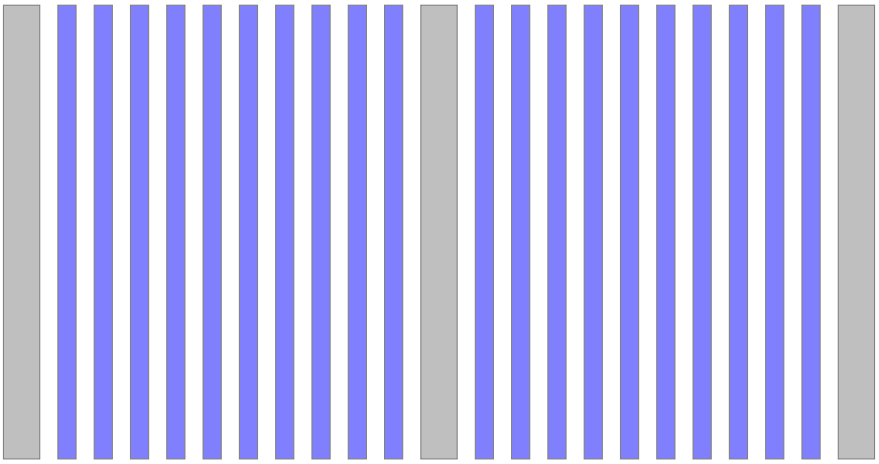


Figure 4.4: TP2 for single wide wires. In each power bay, the maximum number of tracks can be used and each track is legal with respect to the power rails.

we now pack all wires closest possible to the previous power rail as well as to each other and leave some open space next to the next power rail (Figure 4.5(a)). This has some advantages and some disadvantages. On the positive side, the additional space next to the power rail can sometimes be used to replace a thin wire by a thicker wire, thus losing only one thin track for placing a thicker wire. On the negative side, if we only place wires of wire model wm it results in better timing characteristics if we move wires as close to the power rails as possible and distribute the additional space between the wires evenly because power rails do not switch and thus can not induce noise in neighboring wires.

A simple alternative is to distribute the additional space equally between the wires (Figure 4.5(b)). To formally describe this track pattern, we define the **slack** $slack(wm)$ of a wire model $wm \in WM$ as the space between two power rails that is not needed to place $n(wm)$ wires:

$$slack(wm) := p - 2\bar{s}_p(wm) - (n(wm) - 1)\bar{s}(wm, wm)$$

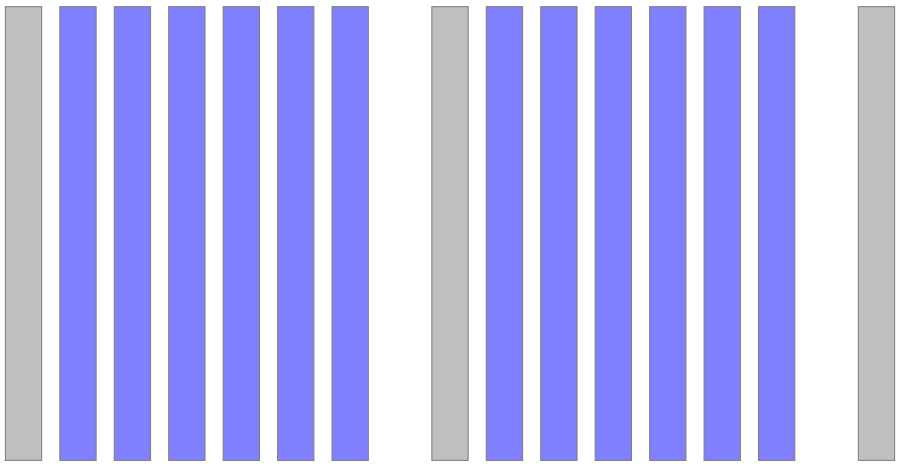
Note that we always have $0 \leq slack(wm) < \bar{s}(wm, wm)$. Now we can formally describe the track pattern:

$$\left\{ kp + \bar{s}_p(wm) + \left\lfloor l \left(\bar{s}(wm, wm) + \frac{slack(wm)}{n(wm) - 1} \right) \right\rfloor : k \in \mathbb{N}, 0 \leq l < n(wm) \right\} \qquad \text{(TP3)}$$

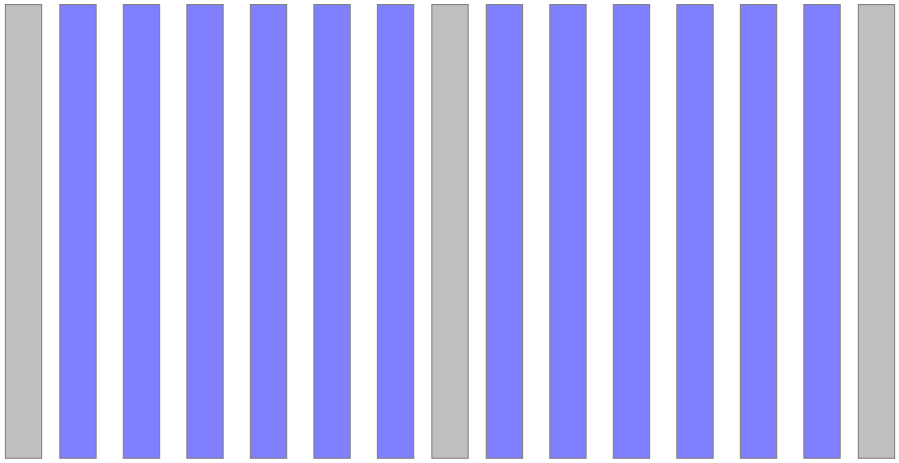
Remark. Note that in the definition of TP3 we arbitrarily choose to round non-integral numbers down. We need integral coordinates due to the definition above, but we can round in any way we please and if necessary or beneficial, we can scale the coordinate system to contain any rational coordinate that we like.

TP3 is in some sense optimal if only one wire model is used because it lexicographically maximizes the available number of tracks and average spacing between different tracks.

But this is not the only relevant objective. There are certainly many situations where the main requirement is to pack as many wires of a given wire model as possible between



(a) TP2



(b) TP3

Figure 4.5: TP2 and TP3 for double wide wires with single spacing. Both track patterns yield the same number of tracks, but for TP3 the extra spacing is evenly distributed between the signal wires.

two power rails, but on many instances, a certain mixture of wire models needs to be packed together (and even though the average relative usage of the different wire models usually is known quite precisely before detailed routing, the concrete mixture between any two power rails or even in different sections between the same power rails varies a lot). Therefore, optimally one would like to simultaneously optimize the total number of tracks of any mixture of different wire models that can be packed between two power rails. Of course one could place a track on each coordinate to reach this goal, but run time requirements certainly forbid this solution. In fact, there are even more objectives. First, it is beneficial to reduce interaction between different tracks, meaning any given track should block as few other tracks (potentially of different wire models) as possible. This is the case because reducing interaction or dependencies between different tracks makes it easier to free any given track for a given wire model if that is necessary after other wires have already been placed. This is in practice very relevant for the rip-up and re-route strategy described in Section 3.10. Furthermore, often more than one wire model can be used to route a given net. For example, additional wire models are beneficial to access pins that have a width that does not match the width of the default wire model of the net. In this case, it would be beneficial if the additional pin access wire models and the default wire model of the given net had the same track pattern such that the router can easily switch from the pin access wire model to the default wire model without having to use a via. Ideally, tracks should in some sense align with pins such that each pin can be legally accessed by a wire on a track.

For some of the objectives above it would be optimal to use the same track pattern for all wire models, for example TP3 with respect to the most frequent wire model.

It is questionable if track patterns should be optimized considering pin positions or if pins should be designed according to some optimized track patterns. If pins are given a priori (for example because they belong to some macro that was designed independently and has to be used as-is) it might be beneficial to take pin positions into account when defining track patterns. However, it is easy to create instances where it is impossible to define a reasonable number of tracks that can legally access each pin. Therefore, if pins are also subject to optimization, we propose to first optimize track patterns and then adjust pin positions to the computed tracks.

If we assume that pins will be adjusted to the track patterns, four objectives remain:

1. maximize number of legal tracks for each wire model and for all combinations
2. minimize dependency between different tracks
3. enable switching of wire models within path
4. spread tracks as much as possible to improve timing characteristics

Note that objective 2 automatically implies that we will not have too many tracks in total. Further note that objective 3 is of minor importance for at least two reasons. First, often the best way to switch wire models is at vias anyway for both timing and design rule reasons. Second, if this is not possible, a short piece of wiring in non-preferred direction can be inserted to switch tracks. Still this is non-trivial to do legally because of certain design rules. Therefore, we consider the third objective with a lower priority. Also note

that objective 4 often contradicts objective 1 which is much more important. Thus we use objective 4 only as a tie-breaker if it does not harm objective 1 and 2.

If more than one wire model is used, all of the above track patterns have severe weaknesses with respect to above objectives. TP1 does not take into account the structure of the power rails and thus contains tracks that are blocked by the power rails. For the other track patterns, consider for example the situation in Figure 4.6 and 4.7. There, we see eight perfectly packing 1x wires between 3x wide power rails. If we now consider a 1.5x wide wire model with 1.5x spacing to other wires and 1x spacing to power (only default spacing to power rails is a pretty common situation, because power does not switch and thus does not induce electrical noise in neighboring wires), we see that each of the previously mentioned track patterns has major weaknesses.

If we use the default tracks for the 1.5x wires as shown in Figure 4.6(a) we can trivially switch between 1x and 1.5x wires. But unfortunately, the leftmost and rightmost tracks are illegal due to the power rails. Furthermore, each two neighboring tracks block each other, so that we can place at most three 1.5x tracks between adjacent power rails (although five are possible with different tracks). Even worse, no matter which legal subset of size three of the 1.5x wires we choose, at least one of the 1x wires blocks two of the 1.5x wires simultaneously (compare Figure 4.7(a)).

If we use TP2 for the 1.5x wires (Figures 4.6(b) and 4.7(b)), we are able to pack five of them between adjacent power rails. However, now the 1.5x tracks and the 1x tracks are disjoint. Furthermore, two of the 1.5x wires block three 1x wires simultaneously and four of the 1x wires block two of the 1.5x wires simultaneously. Additionally, we placed all of the additional space next to a power rail where it helps timing least.

TP3 fixes the last problem, the additional space is now equally distributed in between the 1.5x wires but still two of the 1.5 wires block three 1x wires and four of the 1x wires block two of the 1.5x wires (see Figure 4.6(c) and 4.7(c)).

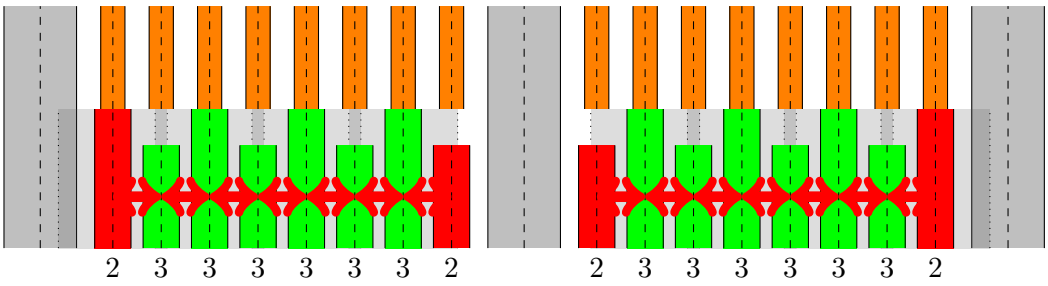
In general, it is not possible to fulfill all these objectives at the same time, but in Section 4.3 we show how we can do significantly better than this.

4.3 Optimized Track Patterns

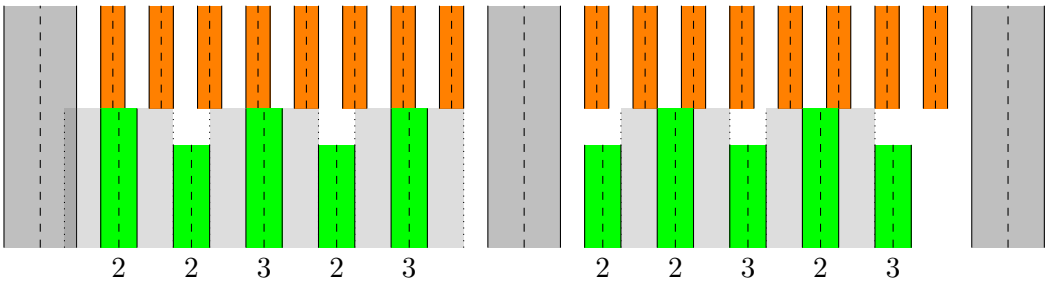
In the last section, we have discussed some simple track patterns and formulated objectives for good (combinations of) track patterns. We will now describe how to compute good track patterns.

The main idea is to optimize a number of track patterns between two adjacent power rails simultaneously with a dynamic program. We start next to one power rail and traverse the space between the power rails (say from left to right), at each point computing all relevant partial track pattern combinations with tracks up to that point. The fact that tracks far enough to the left of the current coordinate can not influence any tracks to the right of it makes it possible to reduce the number of candidates to consider significantly. We design an efficiently computable objective function, make some well-founded assumptions and describe techniques that allow us to prune a large number of candidates away at each point.

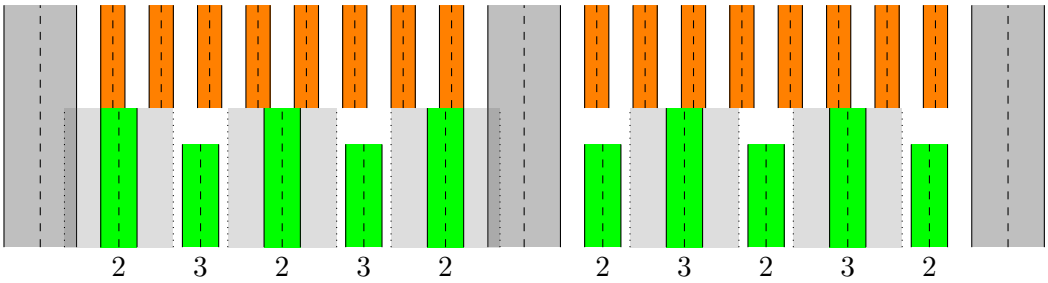
The main assumption is that no two tracks of the same track pattern should block each other. This seems very reasonable; one of our main objectives is to reduce dependency



(a) Use 1x tracks for 1.5x wires.

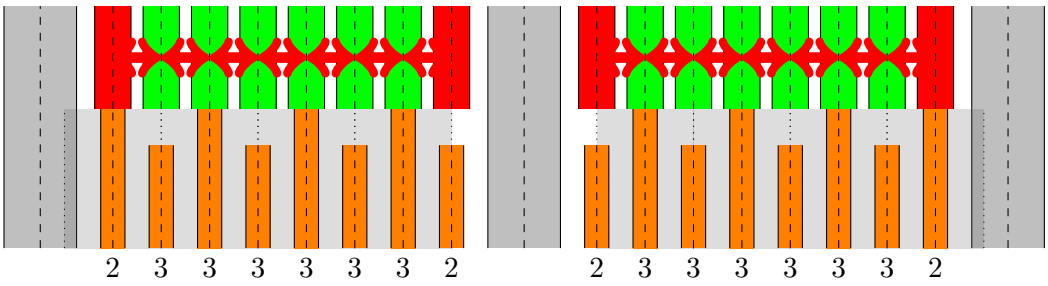


(b) Use TP2 for 1.5x wires.

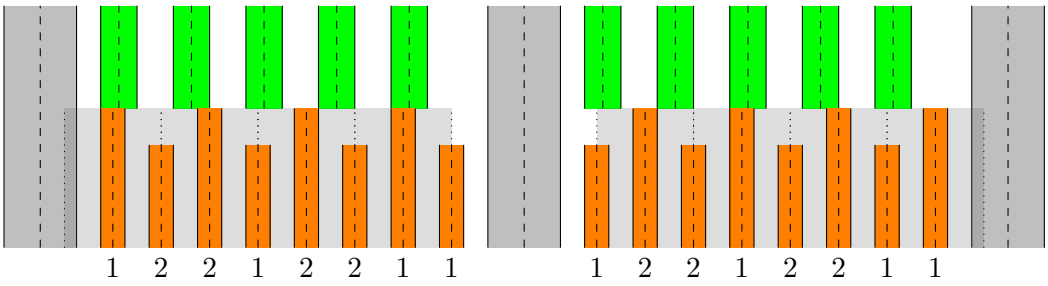


(c) Use TP3 for 1.5x wires.

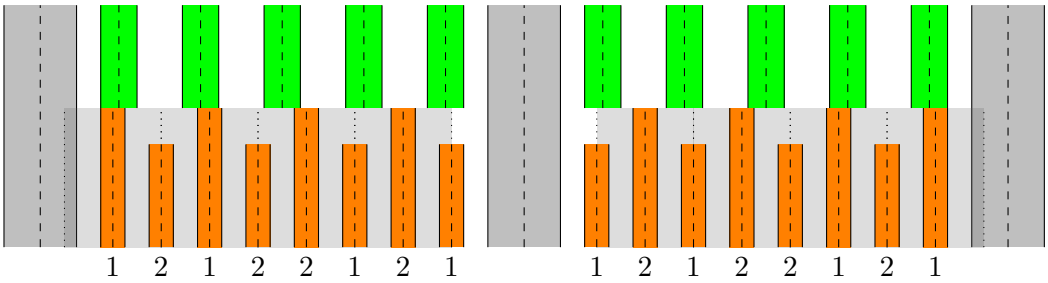
Figure 4.6: Suboptimal simple track patterns for a 1.5x wide wire with 1.5x spacing: Dark gray are power rails, orange are single-wide (1x) default wires with the canonical track pattern, green are 1.5x wide wires with 1.5x spacing, red wires conflict with the power rails, red arrows mark conflicting tracks of the same track pattern, light gray marks the area where a 1.5x wire blocks other wires and the numbers indicate how many 1x tracks are blocked by each 1.5x track.



(a) Use 1x tracks for 1.5x wires.



(b) Use TP2 for 1.5x wires.



(c) Use TP3 for 1.5x wires.

Figure 4.7: Suboptimal simple track patterns for a 1.5x wide wire with 1.5x spacing: Dark gray are power rails, orange are single-wide (1x) default wires with the canonical track pattern, green are 1.5x wide wires with 1.5x spacing, red wires conflict with the power rails, red arrows mark conflicting tracks of the same track pattern, light gray marks the area where a 1x wire blocks 1.5x wires and the numbers indicate how many 1.5x tracks are blocked by each 1x track.

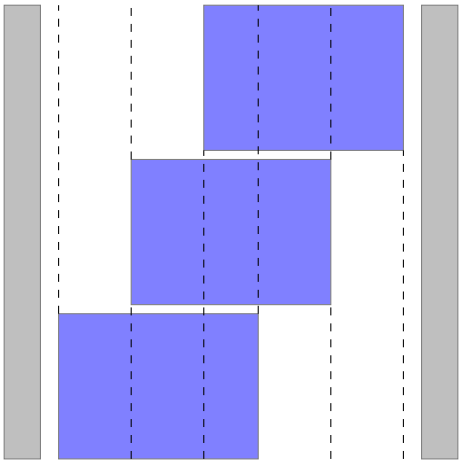


Figure 4.8: Overlapping alternative tracks for a very wide wire model.

between different tracks. At least in the simplest case with only one track pattern, (and probably the most important; it occurs very often that in some section between two power rails only wires of one wire model are placed) we should be optimal.

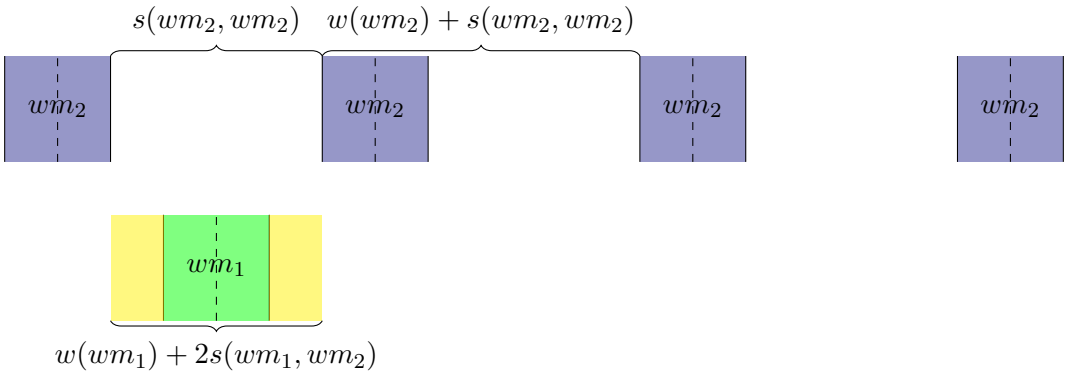
Remark. One could argue that for very infrequent or very wide wire models it might be beneficial to allow a number of different alternative tracks that block each other (like shown in Figure 4.8), but experiments show that this is not beneficial at all.

The second assumption is that for each track pattern, we should only lose very few tracks compared to a densest possible packing of that wire model. This is also very natural, because otherwise it is likely that we encounter situations that are unroutable with our track patterns although they are easily routable with a simple track pattern. In practice it turned out to be best to lose at most one track of each wire model (but this is not inherent to the algorithm, it can trivially be modified to allow more tracks lost). Now, we can describe our dynamic program. For simplicity, we introduce some notation (and in practice, these values are precomputed before the dynamic program starts to reduce run time).

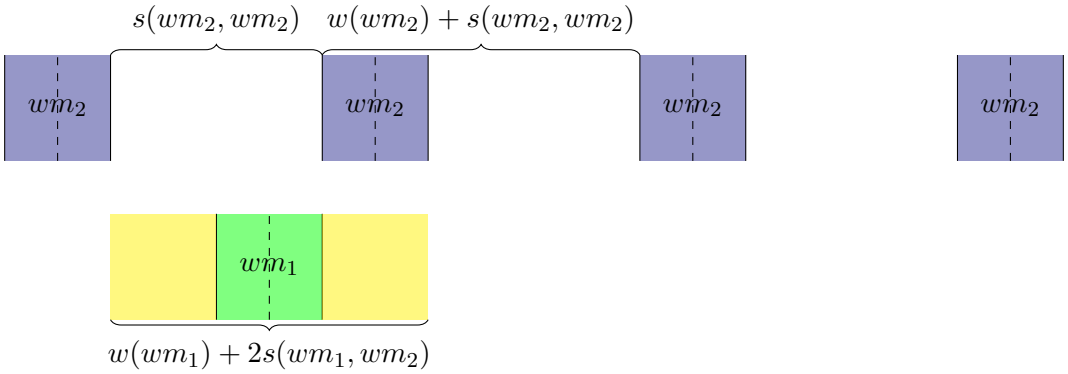
We never place any track that conflicts with a power rail, so for each wire model $wm \in WM$ we know the first and last position where we can place a track, we call them $t_f(wm)$ and $t_l(wm)$ and have: $t_f(wm) := \bar{s}_p(wm)$ and $t_l(wm) := p - \bar{s}_p(wm)$. When evaluating how many tracks of some wire model a track of another wire model blocks, we compare the actual value to the minimum possible value given that both wire models are packed densest possible. For two wire models $wm_1, wm_2 \in WM$ we denote the minimum number of tracks of wire model wm_2 that a track of wire model wm_1 can block if the tracks of wire model wm_2 are packed densest possible (without regard of the power rails) by $b_{min}(wm_1, wm_2)$. One easily computes (compare Figure 4.9):

$$b_{min}(wm_1, wm_2) = \left\lceil \frac{\max(w(wm_1) + 2s(wm_1, wm_2) - s(wm_2, wm_2), 0)}{w(wm_2) + s(wm_2, wm_2)} \right\rceil$$

We want to use $b_{min}(wm_1, wm_2)$ as a reference value to normalize the number of blocked



(a) $w(wm_1) + 2s(wm_1, wm_2) \leq s(wm_2, wm_2)$: wm_1 can legally be placed between two tracks of wm_2 that are packed at minimum distance. Therefore, no track is blocked.



(b) $w(wm_1) + 2s(wm_1, wm_2) > s(wm_2, wm_2)$: wm_1 blocks an interval of width $w(wm_1) + 2s(wm_1, wm_2)$ for wm_2. To minimize the number of wm_2 tracks in this interval, it is optimal to align the left border of this interval with the right border of a wm_2 track. The first $s(wm_2, wm_2)$ units of this interval do not contain a wm_2 track. After this, each $w(wm_2) + s(wm_2, wm_2)$ units a new wm_2 track is blocked.

Figure 4.9: Computing $b_{min}(wm_1, wm_2)$.

tracks to compare them for different wire models. This fails if it is zero or negative, therefore we define a modified version $\tilde{b}_{min}(wm_1, wm_2) := \max(1, b_{min}(wm_1, wm_2))$.

Remark. In practice, the spacing between two wire models $wm_1, wm_2 \in WM$ is almost always the maximum of the two spacings of the two wire models ($s(wm_1, wm_2) = \max(s(wm_1, wm_1), s(wm_2, wm_2))$) which implies $\tilde{b}_{min}(wm_1, wm_2) = b_{min}(wm_1, wm_2)$.

Now, we can define a **candidate**. A candidate consists of a (possibly empty) set of legal tracks for each wire model to optimize. Let $wm \in WM$. First, we define such a set of legal tracks:

$$\mathcal{TPC}_{wm} := \{(t_1, \ldots, t_n) : n \in \mathbb{N}, t_f(wm) \leq t_1, t_n \leq t_l(wm), t_i + \bar{s}(wm, wm) \leq t_{i+1} \quad (1 \leq i < n)\}$$

For each wire model $wm \in WM_i$, we assume that we have a track pattern given; we denote them by $T_{wm} \in \mathcal{TPC}_{wm}(wm \in WM_i)$. Then, we define the **set of all possible candidates** for our given set of wire models $\mathcal{TPC}$:

$$\mathcal{TPC} := \{(T_{wm})_{wm \in WM_o} : T_{wm} \in \mathcal{TPC}_{wm}(wm \in WM_o)\}$$

Further, we define some notation. Let $wm \in WM$ and $T = (t_1, \ldots, t_n) \in \mathcal{TPC}_{wm}$. We define $lt(T) := t_n$, $n(T) := n$ and for $1 \leq i \leq n$: $t_i(T) := t_i$. In particular, if $T = ()$ define $n(T) := 0$. Moreover let $cand = (T_{wm})_{wm \in WM_o}$. For $wm \in WM_o$ define $T_{wm}(cand) := T_{wm}$.

Now we will define our objective function. We take into account two competing objectives. First, each wire model should get as many tracks as possible and second, each track of each wire model should block as few tracks of each other wire model as possible (including the predefined tracks for the wire models in WM_i). First, we define a function indicating if two tracks for two wire models block each other. Let $wm_1, wm_2 \in WM, t_1, t_2 \in \mathbb{N}$. Define

$$\chi_{wm_1,wm_2}(t_1, t_2) := \begin{cases} 1 & |t_1 - t_2| < \bar{s}(wm_1, wm_2) \\ 0 & else \end{cases}$$

Now, we define formally how many tracks of some track pattern for some wire model a track of another wire model blocks. For $wm_1, wm_2 \in WM, t \in \mathbb{N}, T_2 = (t_1, \ldots, t_n) \in \mathcal{TPC}_{wm_2}$ define

$$b(wm_1, t, wm_2, T_2) := \sum_{i=1,\ldots,n} \chi_{wm_1,wm_2}(t, t_i)$$

And for $wm_1 \in WM, T_1 = (t_1, \ldots, t_n) \in \mathcal{TPC}_{wm_1}, wm_2 \in WM, T_2 \in \mathcal{TPC}_{wm_2}$ define

$$b(wm_1, T_1, wm_2, T_2) := \sum_{i=1,\ldots,n} b(wm_1, t_i, wm_2, T_2)$$

Now, for two track patterns, we define how many tracks are blocked on average by one of the wire models. To make blocked tracks comparable between different wire models, we consider the number of blocked tracks normalized by the adjusted minimum number of blocked tracks defined above. Thus it is considered worse if a wire model blocks an

additional track that would normally only block one track than if a wire model blocks six instead of five tracks.

For $wm_1 \in WM, T_1 \in \mathcal{TPC}_{wm_1}, wm_2 \in WM, T_2 \in \mathcal{TPC}_{wm_2}$ define

$$\bar{b}_{rel}(wm_1, T_1, wm_2, T_2) := \frac{b(wm_1, T_1, wm_2, T_2)}{n(T_1)\tilde{b}_{min}(wm_1, wm_2)} \tag{4.1}$$

For a candidate we can now define the average relative number of tracks blocked by computing the weighted average over all combinations of wire models. We weight everything by the relative frequency of the wire models here, because if more frequent wire models block more tracks as well as blocking more tracks of more frequent wire models is worse than for less frequent wire models. Note that the terms for the number of tracks that a wire model blocks of its own tracks do neither harm nor help but for simplicity of notation we keep them. For $cand = (T_{wm})_{wm \in WM_o} \in \mathcal{TPC}$ define

$$\bar{b}_{rel}(cand) := \frac{1}{(f(WM))^2} \sum_{wm_1 \in WM} \sum_{wm_2 \in WM} f(wm_1) f(wm_2) \bar{b}_{rel}(wm_1, T_{wm_1}, wm_2, T_{wm_2}) \tag{4.2}$$

Note that by slight abuse of notation, for $wm_1 \in WM_i$ or $wm_2 \in WM_i$ by T_{wm_1} or T_{wm_2} we mean the given precomputed track patterns defined above.

We can now define our objective function *obj*: Let $cand = (T_{wm})_{wm \in WM_o} \in \mathcal{TPC}$. First we define a term for the number of tracks that each wire model loses.

$$o_n(cand) := \frac{1}{f(WM_o)} \sum_{wm \in WM_o} \frac{n(wm) - n(T_{wm})}{n(wm)} f(wm)$$

Note that we normalize by the maximum possible number of tracks here. Losing one track of many should be less expensive than losing one of few. Further we again weight by the relative frequency because losing a track for a very frequent wire model is worse than for a wire model that is rarely used.

Then we simply use a weighted sum of the two terms as **objective function**.

$$obj(cand) := w_n o_n(cand) + (\bar{b}_{rel}(cand) - 1)$$

Where $w_n \in \mathbb{R}, w_n \geq 0$ is a parameter governing the trade-off between the two different objectives, maximizing the number of tracks for each wire model and minimizing the dependency between different tracks. Note that we subtract one from the average relative number of tracks blocked because conceptually we want to measure the number of additional tracks blocked although this of course makes no difference as it is constant for all candidates. Our goal now is to minimize our objective function and thus minimize the weighted average of the number of tracks lost and the additional number of tracks blocked. Further, we do not want to lose too many tracks for any given wire model, because that could easily lead to situations where routing is impossible with our optimized track patterns but is easy with trivial track patterns. For our description, we will assume that losing at most one track for each wire model is acceptable, but this can trivially be adapted (for each wire model individually) if needed. To simplify the description of the

algorithm, we define the **set of track pattern candidates losing at most one track for each wire model**.

$$\mathcal{TPC}_1 := \{(T_{wm})_{wm \in WM_o} \in \mathcal{TPC} : n(T_{wm}) \geq n(wm) - 1 \ (wm \in WM_o)\}$$

The task is now to find a candidate $cand \in \mathcal{TPC}_1$ with $obj(cand)$ minimum.

We start by describing a very simple dynamic program enumerating all possible candidates and then subsequently optimizing our algorithm until it is efficient enough for practical use.

Our dynamic program traverses the space between two power rails from left to right and at each coordinate c stores a set of (partial) track pattern candidates that contain tracks up to that point. We call this set $CAND(c)$. Let $t_f := \min_{wm \in WM_o} t_f(wm)$ and $t_l := \max_{wm \in WM_o} t_l(wm)$. These are the first and the last track that we need to consider for any wire model. Therefore, we can restrict our dynamic program to work on the interval $[t_f, t_l]$. Further, we have a set of **final (complete) track pattern candidates** which we denote by *FINAL*. Algorithm 4.3.1 formalizes this most simple version of our algorithm. It begins with the empty candidate and at the leftmost relevant coordinate t_f. At each coordinate c until t_l, it considers every candidate and adds every possible set of new tracks at the current coordinate to it. The generated new candidates are added to the set of relevant candidates $CAND(c+1)$ at the next coordinate. Finally, it chooses the best candidate that has enough tracks for each wire model among all the final candidates.

Algorithm 4.3.1: ComputeOptimizedTrackPatternsSimple

Input: $l \in \mathcal{L}_{wiring}$, WM_o, $WM_i \subseteq \mathcal{WM}$

```
1  CAND(t_f) := {(())_{wm∈WM_o}}
2  CAND(c) := ∅ (t_f < c ≤ t_l + 1)
3  for c = t_f to t_l do
4      foreach cand = (T_wm)_{wm∈WM_o} ∈ CAND(c) do
5          WM_p := {wm ∈ WM_o : TrackLegal(wm, T_wm, c)}
6          foreach WM_n ⊆ WM_p do
7              cand_new := AddTracks(cand, WM_n, c)
8              CAND(c+1) := CAND(c+1) ∪ {cand_new}
9  FINAL := CAND(t_l + 1) ∩ TPC_1
10 cand_best := argmin_{cand∈FINAL} obj(cand)
11 return cand_best
```

The subroutines AddTracks and TrackLegal are defined in Algorithms 4.3.3, 4.3.4 and 4.3.2. Algorithm 4.3.2 decides if a track at a certain position can be added to a partial set of tracks for a given wire model because it does not conflict with the existing tracks and the power rails.

Algorithm 4.3.2: TrackLegal

Input: $wm \in WM_o, T_{wm} = (t_1, \ldots, t_n) \in \mathcal{TPC}_{wm}, c \in \mathbb{N}$

return $t_f(wm) \leq c \leq t_l(wm)$ and ($T_{wm} = ()$ or $c \geq t_n + \bar{s}(wm, wm)$)

Algorithm 4.3.3 and 4.3.4 add tracks for the given set of wire models to the given partial track pattern candidate.

Algorithm 4.3.3: AddTrack

Input: $wm \in WM_o, T_{wm} = (t_1, \ldots, t_n) \in \mathcal{TPC}_{wm}, c \in \mathbb{N}$

1 **return** $T_{wm}^{new} := (t_1, \ldots, t_n, c)$

Algorithm 4.3.4: AddTracks

Input: $cand = (T_{wm})_{wm \in WM_o} \in \mathcal{TPC}$, $WM_n \subseteq WM_o, c \in \mathbb{N}$

1 $cand_{new} := \left(T_{wm}^{new} := \begin{cases} \text{AddTrack}(wm, T_{wm}, c) & wm \in WM_n \\ T_{wm} & wm \notin WM_n \end{cases} \right)_{wm \in WM_o}$

2 **return** $cand_{new}$

Note that Algorithm 4.3.1 implicitly also requires the spacing values, frequencies and power rail data as input.

Lemma 4.3.1. *Algorithm 4.3.1 computes an optimal track pattern candidate losing at most one track for each wire model. More formally, it computes a* $cand \in \mathcal{TPC}_1$ *with* $obj(cand)$ *minimal.*

Proof. Algorithm 4.3.2 makes sure that only track pattern candidates are considered throughout Algorithm 4.3.1. Line 9 makes sure that any track pattern candidate considered as the final result loses at most one track for each wire model. Therefore, Algorithm 4.3.1 indeed returns a track pattern candidate that loses at most one track for each wire model (if anything at all). Due to Line 10 it returns an optimal one of the candidates in *FINAL* (if anything at all).

Algorithm 4.3.1 traverses all relevant coordinates from left to right and adds at each coordinate every possible set of tracks to every computed candidate, thus enumerating every possible track pattern candidate. Because we assume that for each wire model at least one track fits between two consecutive power rails, there is at least one track pattern candidate (losing no tracks for every wire model). Therefore the output of Algorithm 4.3.1 is well defined, which concludes the proof. □

Now we will describe a number of modifications to Algorithm 4.3.1 making it more efficient. First, we note that Algorithm 4.3.1 considers many irrelevant candidates which contain too few tracks for some wire model. To prevent this, we define the **slack** $slack(T_{wm}, c)$ of a (partial) candidate $T_{wm} = (t_1, \ldots, t_n) \in \mathcal{TPC}_{wm}$ for some wire model $wm \in WM$ and some current coordinate $c \in \mathbb{N}$ with $t_f(wm) \leq c \leq t_l(wm) + 1$ and $T_{wm} = ()$ or $t_n + \bar{s}(wm, wm) \leq c$ by adding up the slack between the slacks to the left of and including c assuming an additional track at coordinate c (free space to the right of c is not counted). Thus, for a non-empty candidate, we get $t_1 - t_f(wm) + (\sum_{i=2}^{n} (t_i - (t_{i-1} + \bar{s}(wm, wm)))) + c - (t_n + \bar{s}(wm, wm)) = c - t_f(wm) - n\bar{s}(wm, wm)$ and for an empty candidate simply get $c - t_f(wm)$. Therefore, we define:

$$slack(T_{wm}, c) := c - t_f(wm) - n(T_{wm})\bar{s}(wm, wm)$$

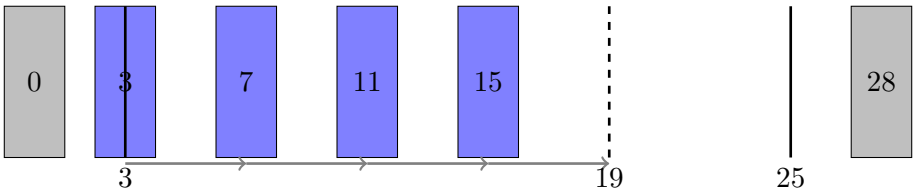
To simplify some proofs later one, we define $slack(T_{wm}, c)$ by the same formula for any coordinate.

The purpose of this is that we can calculate in advance for each wire model, how much additional space (slack) between any tracks can be allocated in total. If more slack

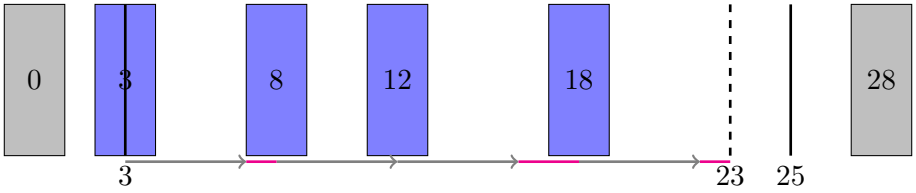
is allocated between some of the tracks, then not enough tracks will fit in total. Thus, by keeping track of the slack that was already allocated, we can decide at which point we have to add a new track of a given wire model to our (partial) candidate because otherwise it can never be completed to a complete candidate with enough tracks. In the beginning of the algorithm, we calculate for each wire model wm the total allowed slack: $allowed\_slack_{wm} := slack(wm) + \bar{s}(wm, wm)(wm \in WM_o)$ (we allow track patterns with up to one track less than the maximum possible number, therefore we add $\bar{s}(wm, wm)$ here; if we want to allow more or less tracks lost, we can adapt this formula). Then, when deciding for which wire models we add tracks to a given candidate $cand = (T_{wm})_{wm \in WM_o}$ at coordinate c, we calculate the set of wire models for which we have to add tracks at this coordinate because otherwise we will exceed our slack budget: $WM_a := \{wm \in WM_p : slack(T_{wm}, c) = allowed\_slack_{wm}\}$. We consider only those subsets WM_n of WM_p such that $WM_a \subseteq WM_n$. In addition, because the algorithm now keeps track of the slack of (partial) candidates, all candidates in $CAND(t_l+1)$ now automatically contain enough tracks for each wire model. Figure 4.10 illustrates the slack for some examples. Our updated procedure can be found in Algorithm 4.3.5. We will now prove that this modification does not affect the correctness of our algorithm.

Lemma 4.3.2. *A set of tracks $T_{wm} = (t_1, \ldots, t_n) \in \mathcal{TPC}_{wm}$ for some wire model $wm \in WM$ with a current coordinate $c \in \mathbb{N}$ with $t_f \leq c \leq t_l(wm)$, $t_n < c$ and $slack(T_{wm}, c) = allowed\_slack_{wm}$ can not be completed to $T_{wm} = (t_1, \ldots, t_n, \ldots, t_{n(wm)-1}) \in \mathcal{TPC}_{wm}$ if no track is placed at coordinate c.*

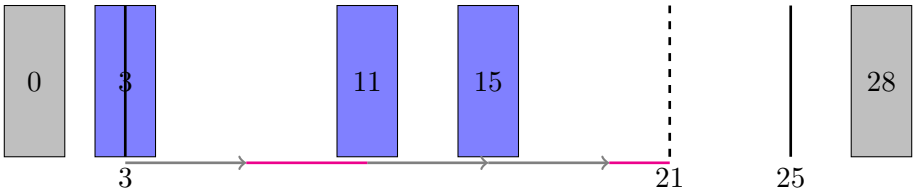
Proof. From $slack(T_{wm}, c) = allowed\_slack_{wm}$ we get by using the definitions: $c = slack(wm) + \bar{s}(wm, wm) + t_f(wm) + n(T_{wm})\bar{s}(wm, wm)$. First, we note that we still need to place additional tracks. If this was not the case (that is to say, if we had $n(T_{wm}) \geq n(wm)-1$), then we would get $c \geq p - \bar{s}_p(wm) + \bar{s}(wm, wm) = t_l(wm) + \bar{s}(wm, wm)$ which is a contradiction. Thus we need to place $n(wm) - 1 - n(T_{wm})$ additional tracks. If we do not place any track at coordinate c, the next track can not be placed before $c+1$. The last one of these can not be placed before $c + 1 + (n(wm) - 1 - n(T_{wm}) - 1)\bar{s}(wm, wm)$. Therefore, the last track can not be placed before: $c+1+(n(wm)-1-n(T_{wm})-1)\bar{s}(wm, wm) = slack(wm) + \bar{s}(wm, wm) + t_f(wm) + n(T_{wm})\bar{s}(wm, wm) + 1 + (n(wm) - 1 - n(T_{wm}) - 1)\bar{s}(wm, wm) = t_f(wm) + slack(wm) + \bar{s}(wm, wm)(n(wm) - 1) + 1 = p - \bar{s}_p(wm) + 1 = t_l(wm) + 1$. But a track at this position (or at any position further to the right) is not legal with respect to the power rail, which concludes the proof. □



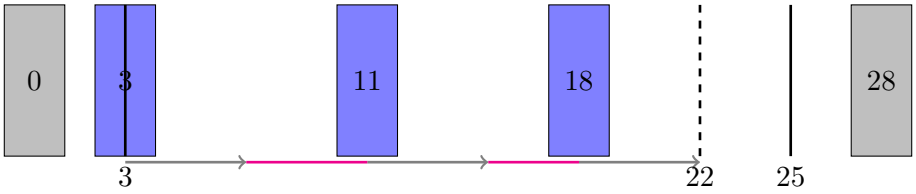
(a) No slack has been spent.



(b) Four units of slack have been spent.



(c) The total allowed slack has been spent. All further tracks have to be placed closest possible (at the current coordinate 21 and at the last possible coordinate 25).



(d) Too much slack has been spent. This candidate can not be completed to a candidate containing enough tracks anymore and thus is not relevant.

Figure 4.10: Examples for the slack of a (partial) candidate. Gray shapes are the power rails. Blue shapes represent the tracks for our wire model. The dashed line indicates the current coordinate. Black lines mark the first and last legal coordinate. Gray arrows indicate the required spacing and the slack is illustrated in magenta. The width of our wire model and the power rails is 2, the required spacing to the power rails is 1 and the required spacing between the blue shapes is 2. 6 tracks fit between the power rails, the slack of this wire model is 2. The allowed slack is 6.

Algorithm 4.3.5: ComputeOptimizedTrackPatternsWithSlack

Input: $l \in \mathcal{L}_{wiring}$, WM_o, $WM_i \subseteq \mathcal{WM}$

1 $CAND(t_f) := \{(())_{wm \in WM_o}\}$
2 $CAND(c) := \emptyset$ $(t_f < c \leq t_l + 1)$
3 $allowed\_slack_{wm} := slack(wm) + \bar{s}(wm, wm)$ $(wm \in WM_o)$
4 **for** $c = t_f$ **to** t_l **do**
5 **foreach** $cand = (T_{wm})_{wm \in WM_o} \in CAND(c)$ **do**
6 $WM_p := \{wm \in WM_o : \text{TrackLegal}(wm, T_{wm}, c)\}$
7 $WM_a := \{wm \in WM_p : slack(T_{wm}, c) = allowed\_slack_{wm}\}$
8 **foreach** $WM_n \subseteq WM_p : WM_a \subseteq WM_n$ **do**
9 $cand_{new} := \text{AddTracks}(cand, WM_n, c)$
10 $CAND(c+1) := CAND(c+1) \cup \{cand_{new}\}$
11 $FINAL := CAND(t_l + 1)$
12 $cand_{best} := argmin_{cand \in FINAL} obj(cand)$
13 **return** $cand_{best}$

Lemma 4.3.3. *Algorithm 4.3.5 works correctly.*

Proof. Lemma 4.3.2 shows that the modifications in Line 7 and Line 8 of Algorithm 4.3.5 are correct. It is necessary to add a track to *cand* for each wire model in WM_a at coordinate c, otherwise *cand* can never be completed to a candidate with enough tracks for each wire model.

It remains to show that each candidate in $CAND(t_l + 1)$ contains sufficiently many tracks for each wire model to prove that the modification of Line 11 is correct. This is true because the algorithm only adds (partial) candidates to $CAND(c)$ that can be completed to a candidate containing sufficiently many tracks for each wire model by adding tracks at coordinates larger or equal to c (and partial candidates in $CAND(c)$ do not contain any tracks at coordinate c). As no tracks can be added to any candidate at coordinates larger or equal $t_l + 1$, this implies that all candidates in $FINAL$ have sufficiently many tracks for all wire models. The statement is trivially true for the empty candidate added to $CAND(t_f)$ in Line 1.

To prove the statement for the other (partial) candidates, we first prove for a wire model $wm \in WM_o$ by induction for each $t_f \leq c \leq t_l(wm) + 1$ that for every partial candidate $cand = (T_{wm})_{wm \in WM_o} \in CAND(c)$, we have $slack(T_{wm}, c) \leq allowed\_slack_{wm}$. We will show this claim by induction over c. The statement is trivial for $c = t_f$. For the induction step, we note that the slack of a candidate from $CAND(c)$ concerning any wire model can be increased by at most one when considering the candidate (potentially augmented by some tracks) at coordinate $c + 1$. Due to the fact that in the case that the slack was equal to the allowed slack, a track was added for the given wire model which reduces the slack (given that $\bar{s}(wm, wm) > 0$ which we assume), the slack can never be greater than the allowed slack.

Now for a wire model $wm \in WM_o$ we consider the coordinate $t_l(wm) + 1$. If each candidate $cand = (T_{wm})_{wm \in WM_o}$ in $CAND(t_l(wm) + 1)$ has enough tracks for wire model wm, so do the final candidates. We have $slack(T_{wm}, t_l(wm) + 1) \leq allowed\_slack_{wm}$. By using the definitions of the slack and the allowed slack, this yields: $n(T_{wm}) \geq n(wm) +$

$\frac{1}{\bar{s}(wm,wm)} - 2$. Because everything (apart from the fraction) is integral, we get $n(T_{wm}) \geq n(wm) - 1$, which concludes the proof. □

Next, we prune dominated candidates during our algorithm. To this end, we will replace Line 10 of Algorithm 4.3.5 by a more sophisticated version, only adding the new candidate if there is no old candidate that is clearly better and pruning old candidates that are clearly inferior to our new candidate. Instead of simply adding $cand_{new}$ to $CAND(c+1)$, we replace $CAND(c+1)$ by a call of UpdateCandidates($CAND(c+1), cand_{new}, c+1$) where UpdateCandidates is defined in Algorithm 4.3.6.

Algorithm 4.3.6: UpdateCandidates

Input: $CAND \subseteq \mathcal{TPC}, cand_{new} \in \mathcal{TPC}, c \in \mathbb{N}$

1 **if** $\exists cand \in CAND$: BetterOrEqual($cand, cand_{new}, c$) **then**
2 **return** $CAND$
3 **else**
4 **return** $\{cand \in CAND$: not BetterOrEqual($cand_{new}, cand, c$)$\} \cup \{cand_{new}\}$

It remains to define when some (partial) candidate dominates another (partial) candidate (with respect to some current coordinate c). A candidate dominates another one if for each set of tracks located to the right of c that can legally be added to the dominated candidate it can also be legally added to the dominating candidate and the objective function of the completed dominating candidate is better or equal than the objective function of the completed dominated candidate. The meaning of the coordinate c here is that we consider the (partial) candidates complete up to (excluding) c.

When comparing two partial candidates, one problem arises. We do not yet know how much tracks completions of the two partial candidates will have for certain wire models. In particular, different completions can have a different number of tracks for some wire models. Therefore, we do not yet know the exact relative weight of blocked tracks between different combinations of wire models in the final candidates. Therefore, we can not easily compare (partial) objective functions of our partial candidates (we do not know the final value of $n(T_1)$ in (4.1)).

To overcome this problem, there are several possibilities. First, one could compare for each combination of wire models the number of mutually blocked tracks and only prune a candidate if all of them are not smaller. In this case, for any final number of tracks the resulting objecting function would not be worse. Slightly better, instead of comparing $|WM_o||WM|$ different values, it suffices to compare $|WM_o|$ different values. We can rearrange the terms in the definition of $\bar{b}_{rel}(cand)$ in a way that we group everything that is multiplied by the number of tracks of a wire model together. Then we can compare for each wire model in WM_o the term multiplied by the number of tracks of that wire model. Like this we only need to compare $|WM_o|$ different values (and can potentially prune more partial candidates away). Still, the total number of candidates to be considered at any coordinate can not be bounded by the maximum number of local track configurations.

Other possibilities would be to change the objective function such that it does not include the final number of tracks or to apply heuristic pruning, losing optimality of the calculated solution. Both of these solutions in practice give good results but are from a

theoretical point of view not satisfactory.

The solution that we chose is very simple, very fast in practice and retains the fact that in our final algorithm, we can bound the total number of candidates considered at any coordinate by the maximum number of local track configurations possible. Due to the fact that we consider only a small number of wire models together, we can run our algorithm for every combination of final number of tracks individually. Because we allow at most one track to be lost for every wire model, we need to run our algorithm at most $2^{|WM_o|}$ times (which is constant if we consider the total number of track patterns to be optimized constant). This approach is in practice much faster than the first alternative described above, computes an optimal solution and does not impact our final theoretical run time (given that the number of track patterns to optimize is constant). To implement this approach, we add for each track pattern to optimize the exact total desired number of tracks $n_t(wm)$ for wire model wm as input to our algorithm. We then can use a modified version of (4.1) and (4.2) by replacing $n(T_1)$ in (4.1) by $n_t(wm_1)$. We call the modified version of (4.2) $\bar{bp}_{rel}(cand)$. Then we can simply call our algorithm for all relevant combinations of $n_t(wm)$ values and take the best result.

Algorithm 4.3.7 gives a formal definition which we now explain in detail.

Algorithm 4.3.7: BETTEROREQUAL

Input: $cand_{better} = (T^b_{wm})_{wm \in WM_o}, cand_{worse} = (T^w_{wm})_{wm \in WM_o} \in \mathcal{TPC}, c \in \mathbb{N}$

if $\bar{bp}_{rel}(cand_{better}) > \bar{bp}_{rel}(cand_{worse})$ **then**
  return *false*
foreach $wm \in WM_o$ **do**
  if $n(T^b_{wm}) \neq n(T^w_{wm})$ **then**
    return *false*
  if $n(T^b_{wm}) > 0$ and $lt(T^b_{wm}) > lt(T^w_{wm})$ and $lt(T^b_{wm}) + \bar{s}(wm, wm) > c$ and $n(T^w_{wm}) < n_t(wm)$ **then**
    return *false*
foreach $wm_w \in WM_o$ **do**
  $nptw := max(\text{NEXTPOSSIBLETRACK}(wm_w, T^w_{wm_w}), c)$
  if $n(T^w_{wm_w}) < n_t(wm_w)$ **then**
    foreach $wm_b \in WM_o \setminus \{wm_w\}$ **do**
      if $\exists i \in \{1, \ldots, n(T^b_{wm_b})\}, |t_i(T^b_{wm_b}) - nptw| < \bar{s}(wm_b, wm_w)$ and $t_i(T^b_{wm_b}) > t_i(T^w_{wm_b})$ **then**
        return *false*
return *true*

The subroutine NEXTPOSSIBLETRACK can be found in Algorithm 4.3.8. If there is no track yet, it returns the first coordinate legal with respect to the power rail. Note that in this case it is not necessary to check legality of that track due to our assumption that for each wire model at least one track does legally fit between two consecutive power rails. If there are already some tracks, it returns the first coordinate legal with respect to the rightmost track if a track can be placed there. If no legal track can be placed anymore at all, it returns $t_l + 1$ to indicate this.

Algorithm 4.3.8: NEXTPOSSIBLETRACK

Input: $wm \in WM_o, T = (t_1, \ldots, t_n) \in \mathcal{TPC}_{wm}$

```
1 if T = () then
2     return t_f(wm)
3 else
4     if t_n + s̄(wm, wm) ≤ t_l(wm) then
5         return t_n + s̄(wm, wm)
6     else
7         return t_l + 1
```

We now describe the cases when Algorithm 4.3.7 does not declare a candidate to be dominating another one. (We do not have to prove these cases for the correctness of our algorithm, but we want to give some reasons why we believe some cases are not likely to be possible to be pruned away. We have to prove the other direction though.)

If the average relative number of blocked tracks is higher (Line 1), the candidate can not be dominating because the objective function of completed candidates can be higher.

Remark. Note that we could possibly prune even more candidates away, because we run our algorithm multiple times for different combinations of desired total numbers of tracks. Therefore, after the first run, we have an upper bound for the objective function, which we could use to prune away any candidate that can not be better anymore. However, experiments show that our algorithm already is very fast for any practically relevant instance, therefore we did not pursue this possibility any further.

If for any wire model, the two candidates have different numbers of tracks, we will not declare one of them dominating (Line 4). If a candidate has fewer tracks, the corresponding part of its objective function will be larger. Due to the fact that we determined the final number of tracks for each wire model before the algorithm, only candidates with the same number of tracks for all wire models can be completed (with the same completion) to valid final candidates.

Remark. Note that it would also be possible to prune candidates with fewer tracks, if all other conditions are fulfilled, but this would complicate our algorithm and proofs further. In practice, this case occurs only rarely, therefore we omit this optimization.

If for any wire model, both candidates have at least one track, and for the candidate in question, the track is further to the right, and it affects tracks to the right of c and additional tracks of that wire model can be added to the other candidate, then the candidate in question can not dominate the other one (Line 6). In this case, one can add another track for this wire model at coordinate c for the other candidate but not for the candidate in question, so it can not be dominating by definition.

Last but not least, we also need to take into account the effect of already placed tracks on the objective function with regard to future tracks. In particular, if any track that can still influence legality of any yet to be placed track is further to the right, we will not declare a candidate dominating (Lines 8 to 13). In a little more detail, if for some wire model we can still add a track to the potentially dominated candidate (Line 10) and we find any

other wire model and a track of the potentially dominating candidate that forbids that next track and where the potentially dominated candidate has the corresponding track further to the left (Line 12) then we do not declare the candidate to be dominating the other one.

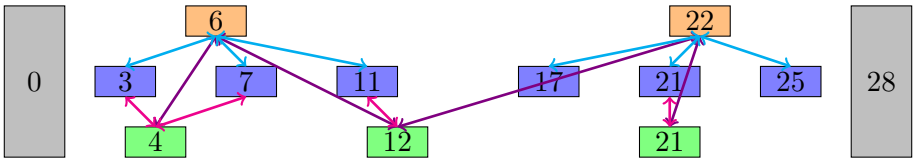
To illustrate the relevance of this more complicated case, we now present an example where taking into account the third last track of a candidate is necessary to get the correct solution. Consider two power rails with a power rail pitch of 28 and let the power rails have a width of 2. Consider three different wire models wm_0, wm_1 and wm_2. Let the width of all three wire models be 2. Let the spacing requirements be as follows: $s(wm_0, wm_0) = 14$, $s(wm_0, wm_1) = 9$, $s(wm_0, wm_2) = 13$, $s(wm_1, wm_1) = 2$, $s(wm_1, wm_2) = 2$ and $s(wm_2, wm_2) = 6$. Let further the required spacings to power be: $s_p(wm_0) = 4$, $s_p(wm_1) = 1$ and $s_p(wm_2) = 2$. For this example, we require each track pattern to have the maximum number of tracks possible (which are 2, 6 and 3 here for the three wire models). Further, we assume that wm_0 is much more frequent than the other two, e.g. let wm_0 be responsible for 80% of the estimated total wire length and the other two for 10% each. We set $w_n = 2$ (as we also do in practice). In this setting, one optimal solution is shown in Figure 4.11(a). Figure 4.11(b) and 4.11(c) show two (partial) candidates which are considered during execution of our algorithm at coordinate 21. All their rightmost tracks are equal, but the partial objective function of the candidate in Figure 4.11(c) is better because there is one conflict less between tracks of wire models wm_1 and wm_2. If the algorithm would not take into account all tracks that still can influence tracks that might be placed in the future, it would prune the candidate in Figure 4.11(b). Unfortunately, it would then never find the optimal solution shown in Figure 4.11(a). The best solution that can be constructed using the candidate in Figure 4.11(c) (actually the only one as it is unique) is shown in Figure 4.11(d). It has one conflict less between wire models wm_1 and wm_2 but one conflict more between wire models wm_0 and wm_1. Due to our choice of the estimated relative frequencies of the wire models, its objective function value is worse than the one of the optimal solution shown in Figure 4.11(a).

The modified version of Algorithm 4.3.5 can be found in Algorithm 4.3.9.

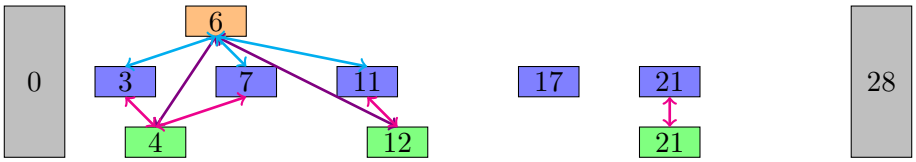
Algorithm 4.3.9: ComputeOptimizedTrackPatternsWithPruning

Input: $l \in \mathcal{L}_{wiring}$, WM_o, $WM_i \subseteq \mathcal{WM}, n_t(wm) \leq n(wm)\ (wm \in WM_o)$

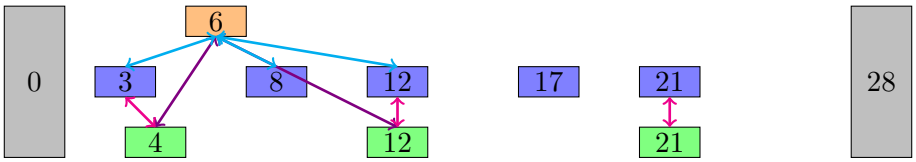
1 $CAND(t_f) := \{((\,))_{wm \in WM_o}\}$
2 $CAND(c) := \emptyset\ (t_f < c \leq t_l + 1)$
3 $allowed\_slack_{wm} := slack(wm) + \bar{s}(wm, wm)(n(wm) - n_t(wm))\ (wm \in WM_o)$
4 **for** $c = t_f$ **to** t_l **do**
5 **foreach** $cand = (T_{wm})_{wm \in WM_o} \in CAND(c)$ **do**
6 $WM_p := \{wm \in WM_o : \text{TrackLegal}(wm, T_{wm}, c), n(T_{wm}) < n_t(wm)\}$
7 $WM_a := \{wm \in WM_p : slack(T_{wm}, c) = allowed\_slack_{wm}\}$
8 **foreach** $WM_n \subseteq WM_p : WM_a \subseteq WM_n$ **do**
9 $cand_{new} := \text{AddTracks}(cand, WM_n, c)$
10 $CAND(c+1) := \text{UpdateCandidates}(CAND(c+1), cand_{new}, c+1)$
11 $FINAL := CAND(t_l + 1)$
12 $cand_{best} := argmin_{cand \in FINAL} obj(cand)$
13 **return** $cand_{best}$



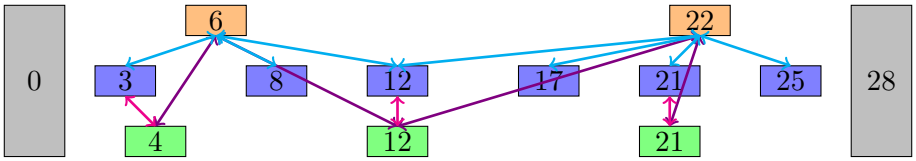
(a) An optimal solution. It has 6 conflicts between wm_0 and wm_1, 4 conflicts between wm_1 and wm_2 and 4 conflicts between wm_0 and wm_2.



(b) A (partial) candidate with worse objective value. It has 3 conflicts between wm_0 and wm_1, 4 conflicts between wm_1 and wm_2 and 2 conflicts between wm_0 and wm_2.



(c) A (partial) candidate with better objective value. It has 3 conflicts between wm_0 and wm_1, 3 conflicts between wm_1 and wm_2 and 2 conflicts between wm_0 and wm_2.



(d) The candidate with better objective value completed (the completion is unique in this case). It has 7 conflicts between wm_0 and wm_1, 3 conflicts between wm_1 and wm_2 and 4 conflicts between wm_0 and wm_2.

Figure 4.11: Example of track patterns where considering more than the last track is necessary. Gray shapes are the power rails. Orange shapes represent tracks for wire model wm_0, blue shapes represent tracks for wire model wm_1 and green shapes represent tracks for wire model wm_2. Conflicts between wire models wm_0 and wm_1 are indicated in cyan, conflicts between wm_1 and wm_2 in magenta and conflicts between wm_0 and wm_2 in purple.

It remains to prove that we do not prune too much.

Lemma 4.3.4. *Algorithm 4.3.9 works correctly. Any (partial) candidate pruned out by Algorithm 4.3.6 is not necessary to find an optimal solution.*

Proof. First, we note that we have correctly adapted our algorithm to only consider candidates with the desired number of tracks for each wire model. We have adapted the calculation of the slack for each wire model accordingly (Line 3) and the algorithm only can add tracks until the desired number is reached (Line 6).

Now, it suffices to show that if Algorithm 4.3.7 returns true for two candidates $cand_{better}$, $cand_{worse} \in \mathcal{TPC}$ and a coordinate c then $cand_{worse}$ is not needed for an optimal solution at coordinate c. In order to show this, we show that in this situation, any completion of the (partial) candidate $cand_{worse}$ can instead be applied to the (partial) candidate $cand_{better}$ giving an objective value that is at most as large. We have to show two things.

First, any completion of $cand_{worse}$ can indeed be applied to $cand_{better}$ instead. In other words, any track that is located at coordinate c or further to the right and can be added to $cand_{worse}$ can also be added to $cand_{better}$. This can only be violated, if for a wire model wm it is possible to add another track to $cand_{worse}$ and the last track of $cand_{better}$ is further to the right than the last track of $cand_{worse}$ and the last track of $cand_{better}$ forbids a track at coordinate c or further to the right to be added. In this case Algorithm 4.3.7 would return $false$ in Line 7.

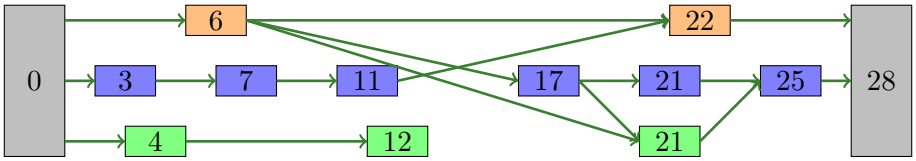
Second, we need to show that when adding the same tracks to $cand_{better}$ instead of $cand_{worse}$, the objective function can not increase. Due to Line 4, both completed candidates have the same number of tracks for all wire models. The corresponding part of the objective function will consequently be equal. The part of the objective function corresponding to the average relative number of blocked tracks can be written as a large weighted sum. Because the relative frequencies of the wire models are constant and both completed candidates have the same number of tracks for each wire model, all the weights are equal. The summands can be divided into three classes. First, terms that correspond to combinations of tracks blocking each other such that both tracks are precomputed tracks or part of the partial candidates $cand_{worse}$ and $cand_{better}$ respectively. This part of the sum is smaller or equal for $cand_{better}$ because Algorithm 4.3.7 did not return $false$ in Line 2. Second, terms that correspond to combinations of tracks blocking each other such that both tracks are part of the completion or one track is part of the completion and the other is part of a precomputed track pattern. This part of the sum is trivially equal because we apply the same completion to both candidates. Third, terms that correspond to one track that is part of the completion and another track that is part of the (partial) candidates $cand_{worse}$ and $cand_{better}$ respectively. For this part of the sum it suffices to show that each track of the completion blocks fewer or the same number of tracks of any wire model if it is added to $cand_{better}$ as it blocks when being added to $cand_{worse}$. If a track t for wire model wm' of the completion blocked more tracks of a wire model wm if added to $cand_{better}$ than it does if added to $cand_{worse}$ then there needs to be a $k \in \mathbb{N}$ such that the kth track for wire model wm of $cand_{better}$ is further to the right than the kth track for wire model wm of $cand_{worse}$. Additionally, it needs to be possible to add another track for wire model wm' to $cand_{worse}$ at position t and the kth track for wire model wm of $cand_{better}$ needs to make the track t illegal. But in this case, Algorithm 4.3.7 would have returned $false$ in Line 13, which concludes the proof. □

Last, we optimize our algorithm by noting that we do not need to consider each (partial) candidate at each coordinate and that we can omit some candidates already during creation because they are dominated by another candidate considered earlier. This optimization is based on two ideas. First, it is not necessary to try to add another track for a wire model at a coordinate where it is forbidden by the last track for that wire model or the power rail. Therefore, we do not need to consider a candidate again at any coordinate left of the minimum of the next allowed positions for all wire models. Consequently, we do not add each newly generated candidate to the set of candidates at coordinate $c + 1$ but at the maximum of $c+1$ and the minimum over all wire models of the next coordinate where we can place another track.

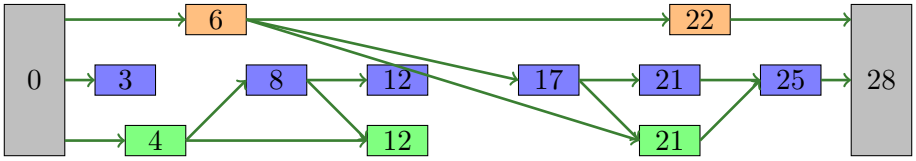
Secondly, there is a certain class of candidates that are never needed for an optimal solution (in the sense that when we exclude this class, there always remains an optimum solution). Whenever in a (partial) candidate any track can be moved by one unit to the left without generating any new conflict (to any track of the same wire model, another wire model or the power rail), this candidate is not necessary to generate an optimum solution. It can always be replaced by the version where the track is moved to the left (in the sense that any completion of this (partial) candidate can be applied to the modified version instead and this can only decrease the objective value). Consequently, if for a candidate we add an arc between any two tracks (not necessarily of the same wire model) which do not have a conflict but which can not be moved closer together without generating a conflict and we do the same for pairs of tracks and power rails (directed from the one further to the left to the one further to the right), we only need to keep those candidates where all tracks have a path from the left power rail or a predefined track. Any track that can not be reached from the left power rail and the predefined tracks can be moved at least one unit to the left without generating any new conflict. Figure 4.12 shows three examples of this concept. Figures 4.12(a) and 4.12(b) show the tight constraints for the two examples from Figure 4.11(a) and 4.11(d). Note that every track does have a path from the left power rail in both cases. Figure 4.12(c) shows an example of a (partial) candidate where some tracks can be moved to the left.

This observation leads to two more optimizations we can make. First, when generating new candidates at a certain coordinate, we only need to add tracks for wire models (among those for which it is legal to add tracks at all) for which at least one constraint to the left is tight (otherwise, if no constraint to the left is tight, we can omit to add a track if there is sufficient slack left, otherwise we have proven that the candidate which we are extending was not needed at all and we can skip it completely).

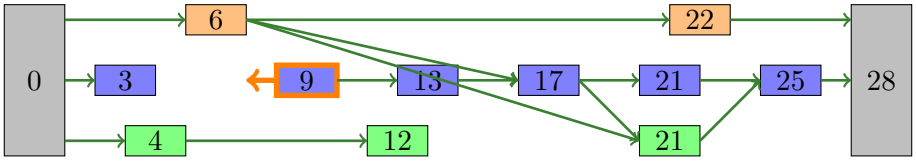
To implement this, we modify the way we generate the sets WM_n of wire models to add new tracks for. We define $WM_l := \{wm \in WM_o : \textsc{TrackLegal}(wm, T_{wm}, c), n(T_{wm}) < n_t(wm)\}$, the set of all wire models for which it is legal to add a track at coordinate c (and for which we still do not have enough tracks). Then we define the subset of wire models for which a track at coordinate c has at least one tight constraint to the left $WM_p := \{wm \in WM_l : \textsc{HasTightConstraint}(wm, cand, c)\}$ and the subsets of wire models with tight slack: $WM_a := \{wm \in WM_l : slack(T_{wm}, c) = allowed\_slack_{wm}\}$ (for which we have to add a track at coordinate c to get the required total number of tracks). The function HasTightConstraint can be found in Algorithm 4.3.10. Then we add tracks for all sets WM_n which are a superset of WM_a and at the same time a



(a) The solution from Figure 4.11(a).



(b) The candidate from Figure 4.11(d).



(c) A candidate which we can omit because the track marked in orange can be moved to the left without introducing any new conflicts.

Figure 4.12: Only candidates where all tracks have a path of tight constraints from the left power rail (or a predefined track) need to be considered. The instance is the same as in Figure 4.11. Green arrows show tight constraints.

subset of WM_p. Note that we do not necessarily have $WM_a \subseteq WM_p$. If this is not the case, then no such set WM_n exists which is intended. In this case, the candidate we started with was redundant and can be omitted.

Algorithm 4.3.10: HASTIGHTCONSTRAINT

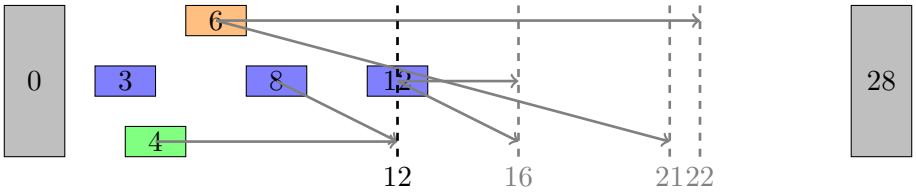
Input: $wm \in WM_o, cand = (T_{wm'})_{wm' \in WM_o} \in \mathcal{TPC}, c \in \mathbb{N}$

1 **return** $\exists wm' \in WM, i \in \mathbb{N}, 1 \leq i \leq n(T_{wm'}), t_i(T_{wm'}) + \bar{s}(wm', wm) = c$ or $c = t_f(wm)$

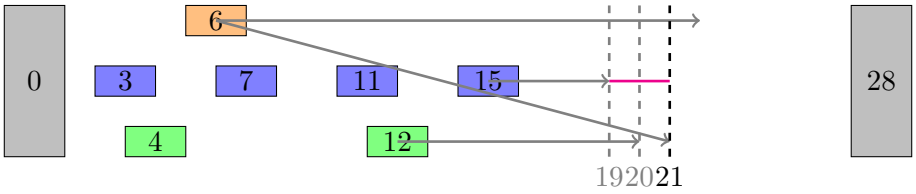
Lemma 4.3.5. *Algorithm 4.3.10 correctly determines if a track added at coordinate c for wire model wm to the (partial) candidate cand has a tight constraint to the left.*

Proof. There are two possible kinds of tight constraints, distances to the left power rail and distances to other tracks placed to the left of c. Algorithm 4.3.10 checks both of them correctly. Note that in the Algorithm there is no difference between the distance to the last track of the same wire model (if it exists), to tracks of other wire models in WM_o and the distances to tracks of wire models in WM_i). □

Second, we can further optimize the position where we insert new candidates. We do not need to take the maximum of $c + 1$ and the minimum over all wire models of the coordinate where the next track is legal. We can for each wire model consider the next coordinate where we can place a track and it does have at least one tight constraint to the left and take the minimum of these. There are some details to consider here. First, there might be a wire model for which no such coordinate exists. In this case, this wire model is not relevant for the next coordinate to consider, no track can be placed for it anymore. Second, there might be a wire model for which the next coordinate at which we could place a track legally and it would have a tight constraint to the left would inevitably lead to a larger than allowed slack (or we can not place any track with a tight constraint to the left anymore but do not have enough tracks yet). In this case, the candidate we are considering is not relevant at all and can be omitted. Figure 4.13(b) shows an example for this case. Third, we might already have placed enough tracks for some wire model. Then, like in the first case, the wire model is not relevant, no track can be placed for it anymore. Algorithm 4.3.11 formalize the calculation of the next relevant coordinate for a given (partial) candidate. It returns ∞ if the given candidate is irrelevant and should be omitted. Consequently, if Algorithm 4.3.11 returns ∞, the generated candidate is discarded. If it returns $t_l + 1$, the generated candidate is a final candidate and otherwise the candidate set at the returned coordinate is updated with the generated candidate. Figure 4.13(a) shows an example how the next relevant coordinate is calculated.



(a) Example how the next relevant coordinate is determined. This candidate is created at coordinate 12, but does not need to be considered before coordinate 16 again. For wm_0 and wm_1, no track can be placed before coordinate 16, and for wm_2, a track can be placed at any coordinate larger than 12, but it would not have a tight constraint to the left before coordinate 16.



(b) Example of a (partial) candidate which is discovered to be irrelevant at coordinate 21. Consider wm_1. The allowed slack for wm_1 in our example instance is 2 (because we do not allow any track to be lost). At coordinate 19, another track for wm_1 is legal, but the already spend slack is 0, so a candidate not placing a track is created. It needs to be reconsidered at coordinate 20, where the next tack of wm_2 can be placed. wm_2 also has sufficient slack left, so a candidate with no track is created. This candidate is reconsidered at coordinate 21 due to the distance between wm_0 and wm_2. At this point, there is no possible solution for wm_1 anymore. We can not place a track for wm_1, because it would not have a tight constraint to the left and thus should have been placed further to the left. It is also not possible not to place a track for wm_1, because the current slack is 2 which is the allowed slack. Therefore this candidate is discarded and no further candidates are generated from it. Note, that in general it is not trivial to discard such candidates even earlier (for example at coordinate 19 in this case), because at coordinate 19 we do not know if we will place another track of a different wire model before coordinate 21 which will then generate a tight constraint for wm_1 at coordinate 21. We could sort out some candidates earlier by considering the minimum adjusted spacing between different wire models and thus proving that no other track that will be placed can generate a tight constraint for a coordinate close enough, but this is probably too expensive to be beneficial.

Figure 4.13: Two examples concerning the optimizations introduced in Algorithm 4.3.12. The instance is the same as in Figure 4.11. Gray shapes are the power rails. Orange shapes are tracks for wire model wm_0, blue shapes are tracks for wire model wm_1 and green shapes are tracks for wire model wm_2. Gray arrows show relevant constraints. Magenta lines indicate used slack. The black dashed line indicates the current coordinate, gray dashed lines mark other relevant coordinates.

Algorithm 4.3.11: NEXTRELEVANTCOORDINATE

Input: $cand = (T_{wm})_{wm \in WM_o} \in \mathcal{TPC}, c \in \mathbb{N}$

```
1  result := t_l + 1
2  foreach wm ∈ WM_o do
3      if n(T_wm) = n_t(wm) then
4          result := min{result, t_l + 1}
5      else if NextPossibleTrack(wm, T_wm) > c then
6          result := min{result, NextPossibleTrack(wm, T_wm)}
7      else
8          ntc := min{c' ∈ ℕ : c < c', HasTightConstraint(wm, cand, c')}
9          if slack(T_wm, min{ntc, t_l(wm) + 1}) > allowed_slack_wm then
10             return ∞
11         if ntc ≤ t_l(wm) then
12             result := min{result, ntc}
13         else
14             result := min{result, t_l + 1}
15 return result
```

Lemma 4.3.6. *Algorithm 4.3.11 correctly returns the next coordinate for which adding a new track (at a position larger than c) to the candidate cand needs to be considered (in the sense that not inserting any tacks before the coordinate returned by Algorithm 4.3.11 does not make Algorithm 4.3.12 incorrect). If Algorithm 4.3.11 returns $t_l + 1$, no tracks can be added to cand anymore. If Algorithm 4.3.11 returns ∞, cand was a redundant candidate that is not relevant for the solution of Algorithm 4.3.12 at all.*

Proof. There are five cases that can happen for each wire model wm. First, it can already have enough tracks. In this case (Line 4), no further track can be added for this wire model and the candidate is marked final by returning $t_l + 1$ if no smaller value is computed for the other wire models. Second, it is not legal to place a track for wire model wm at coordinate c. In this case (Line 6), the next relevant coordinate for this wire model is the next coordinate where a track can be placed (and a track there automatically has a tight constraint to the left). Note, that if it is not possible to place any track for wm anymore, the candidate is complete with regard to wm and NEXTPOSSIBLETRACK returns $t_l + 1$, marking the candidate final if no smaller value is computed for the other wire models. If a track for wm can legally be placed at coordinate c, then the next greater coordinate than c for which a track would have a tight constraint to the left is relevant. The third case (Line 10) occurs when at the next such coordinate (or more precisely, at the minimum of the next such coordinate and $t_l(wm) + 1$) the slack of the candidate would be too large. In this case, the given (partial) candidate can not be completed to any relevant final candidate anymore and thus can be discarded. Here, we need to take the minimum with $t_l(wm) + 1$, because $t_l(wm)$ is the last legal coordinate to place a track of wire model wm, therefore at $t_l(wm) + 1$, all tracks for wire model wm have been determined. If we would not take the minimum, then free space after $t_l(wm)$ would also be considered although this area is already blocked by the right power rail and thus does not add to the slack. In

the fourth case (Line 12), the next coordinate with a tight constraint to the left is small enough so that a track can be placed legally. Then for wm this is the next coordinate to consider. The fifth and last case (Line 14) occurs, when the next coordinate with a tight constraint to the left if larger then $t_l(wm)$. In this case, the given candidate is final with regard to wm (it has enough tracks, otherwise case two would have occurred). As above, we consider $t_l + 1$ in this case again for this wire model. If the algorithm does not return ∞ to indicate that the given candidate was irrelevant, it returns the minimum of the calculated values for all wire models. If the candidate was final with respect to all wire models, it is final and $t_l + 1$ is returned. Otherwise the next relevant coordinate is returned. □

The updated (and final) version of our dynamic program can be found in Algorithm 4.3.12.

Algorithm 4.3.12: ComputeOptimizedTrackPatterns

Input: $l \in \mathcal{L}_{wiring}$, WM_o, $WM_i \subseteq \mathcal{WM}, n_t(wm) \leq n(wm)$ $(wm \in WM_o)$

1 $CAND(t_f) := \{(())_{wm \in WM_o}\}$
2 $CAND(c) := \emptyset$ $(t_f < c \leq t_l)$
3 $FINAL := \emptyset$
4 $allowed\_slack_{wm} := slack(wm) + \bar{s}(wm, wm)(n(wm) - n_t(wm))$ $(wm \in WM_o)$
5 **for** $c = t_f$ **to** t_l **do**
6 **foreach** $cand = (T_{wm})_{wm \in WM_o} \in CAND(c)$ **do**
7 $WM_l := \{wm \in WM_o : \text{TrackLegal}(wm, T_{wm}, c), n(T_{wm}) < n_t(wm)\}$
8 $WM_p := \{wm \in WM_l : \text{HasTightConstraint}(wm, cand, c)\}$
9 $WM_a := \{wm \in WM_l : slack(T_{wm}, c) = allowed\_slack_{wm}\}$
10 **foreach** $WM_n \subseteq WM_p : WM_a \subseteq WM_n$ **do**
11 $cand_{new} := \text{AddTracks}(cand, WM_n, c)$
12 $c_n := \text{NextRelevantCoordinate}(cand_{new}, c)$
13 **if** $c_n \neq \infty$ **then**
14 **if** $c_n \leq t_l$ **then**
15 $CAND(c_n) := \text{UpdateCandidates}(CAND(c_n), cand_{new}, c_n)$
16 **else**
17 $FINAL := FINAL \cup \{cand_{new}\}$
18 $cand_{best} := argmin_{cand \in FINAL} obj(cand)$
19 **return** $cand_{best}$

Now we can prove that our final algorithm works correctly.

Theorem 4.3.7. *Algorithm 4.3.12 computes a candidate* $cand = (T_{wm})_{wm \in WM_o} \in \mathcal{TPC}$ *such that for all* $wm \in WM_o$ *we have* $n(T_{wm}) = n_t(wm)$ *with minimal objective function.*

Proof. Due to Lemma 4.3.4 we only need to prove that the modifications marked in blue do not make the algorithm incorrect. The modification in Line 8 has the effect that Algorithm 4.3.12 only considers (partial) candidates such that each track has at least one tight constraint to the left. This is sufficient as discussed above. In Line 12, the next coordinate where a track can potentially be added to $cand_{new}$ (only considering

coordinates such that each track has at least one tight constraint to the left) is correctly calculated according to Lemma 4.3.6. If $c_n = \infty$, $cand_{new}$ can be discarded according to Lemma 4.3.6. If $c_n \leq t_l$, the set of candidates at c_n is updated, otherwise $cand_{new}$ is a final candidate. Note, that *FINAL* is directly updated, therefore the minor change in Line 2 is correct. There is one more detail to take care of. Due to the fact that the algorithm does not consider every candidate at each coordinate anymore, the slack can increase by more than one between two times when an (updated) candidate is considered. Line 9 of Algorithm 4.3.11 makes sure that candidates whose slack gets too large are not processed anymore the next time they are considered for some wire model. Therefore, all final candidates have sufficiently many tracks for each wire model, which concludes the proof. □

Next, we analyze some aspects of Algorithm 4.3.12 concerning its run time. First we note that Algorithm 4.3.12 can easily be implemented in a much faster (and memory efficient) way than the simple description above.

In the simple description above, Algorithm 4.3.7 computes $\bar{bp}_{rel}$ values for the two candidates from scratch considering all already placed tracks until the left-hand side power rail. This is not necessary. These values can be precomputed and incrementally updated when a new candidate is created from an existing one. Then they are stored at the candidate to be accessed in constant time during Algorithm 4.3.7. We do not describe the incremental calculation of the objective function in detail here because calculations are straightforward. When adding a new track, one simply needs to add all terms of the sums that depend on the newly created track (with all the relevant constant coefficients applied).

Furthermore, when creating a new track pattern candidate, a naive implementation needs to copy all the tracks back to the left-hand side power rail. This can also be optimized significantly. In each candidate, we store for each track pattern the number of tracks, the remaining (or already used) slack, the precomputed parts of the objective function and an index in a global array of track descriptions, pointing to the description of the last (rightmost) track of that track pattern. Such a track description then contains the coordinate of the track, the index of the description of the next track to the left (or an invalid index if there is no such track) and a reference counter how often this track description is referenced by later track descriptions. Furthermore, we have only one global array of track descriptions and a stack of free entries in this array. Whenever a reference count of a track description drops to zero, we can add the corresponding index to the stack of free entries (also implemented by a simple array) to reuse the memory when we create the next new track description. Because all the data is stored in two simple large arrays, memory overhead is very small and access times are very good. This significantly reduces memory allocation and usage. When creating a new candidate we only need to allocate a constant amount of memory plus memory for the newly created tracks. Total memory usage is reduced because multiple new candidates can reuse the same track descriptions. Furthermore, we can access all relevant information (the number of tracks, the coordinate of the rightmost track and the coordinates of a small number of rightmost tracks) in constant time / in time linear in the number of tracks examined.

Additionally, the set *FINAL* never needs to be computed or stored. We can instead store only the currently best candidate and its objective value and update these whenever

a new candidate would be added to *FINAL*.

In the simple description above, Algorithm 4.3.12 stores a set of track pattern candidate for each coordinate between t_f and t_l which makes its run time dependent on the coordinate system used. This is not necessary, one can instead store only the sets that are non-empty in a map indexed by the coordinates the sets belong to (and once a given coordinate c has been processed, the set $CAND(c)$ is no longer needed and can be removed). This removes the theoretic dependence on the coordinate system completely, but in practice storing a few thousand empty sets does not induce any measurable run time overhead. Therefore our practical implementation does not use this optimization.

Concerning the overall run time of our algorithm, there are a few points we want to mention. First, as stated above, it is practically not dependent on the coordinate system used (and can be implemented such that this is also true in a strict sense). This is the case because every coordinate at which the algorithm will ever attempt to place any track is the sum of a number of input values (to be precise, of one adjusted spacing to the left power rail and then a number of adjusted spacing values between wire models or of the coordinate of a predefined track and a number of adjusted spacing values between wire models). This is true because only coordinates are considered where at least one wire model has a tight constraint to the left. Consequently, the set of coordinates that need to be considered (which influences the total run time) heavily depends on the structure of the adjusted spacing constraints (and predefined track patterns). For example, if all the adjusted spacing constraints are an integer multiple of some base value and there are no predefined track patterns, then every coordinate ever considered will also be an integer multiple of the base value. If on the other hand adjusted spacing values are relatively prime and some of them are very small, then most coordinates will be relevant. In practice, in many cases the relevant adjusted spacing values have a large common divisor and therefore most coordinates are irrelevant, greatly speeding up the algorithm.

The number of candidates that need to be considered at any relevant coordinate depends highly on the largest spacing value, or more precisely on the ratio of the largest and the smallest spacing value (and, of course, on the number of wire models to optimize track patterns for). The higher this ratio is, the more combinations of tracks are possible within the range where they still can influence future tracks. It does not depend on the distance between the power rails, so if the wire models to optimize and all adjusted spacing values are fixed, Algorithm 4.3.12 runs in time linear in the distance between the power rails. In practice, on our instances the run time of Algorithm 4.3.12 is negligible (compare Section 6.2).

After we have described the main algorithm, proven its correctness and briefly examined its run time behavior, we need to discuss some practical extensions of the core algorithm described above.

First, as we have mentioned above, it is easy to modify the algorithm such that for some track patterns more than one or no track at all can be lost compared to the maximum number of tracks possible for the corresponding wire model. Experiments show that allowing to lose more than one track is not beneficial. The average efficiency to pack mixtures of different wire models increases only slightly but the chance that local unroutable situations are created increases drastically. Designers tend to expect that they can pack the maximum number of wires of each wire model between any two power rails (and the

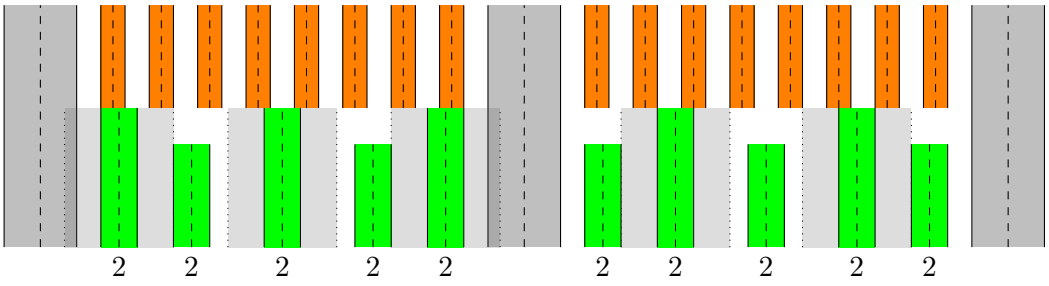
assumption that close to the maximum number can indeed be packed seems to be reasonable). Thus we never allow to lose more than one track. For the same reason we define a threshold such that for wire models with the maximum number of tracks possible below this threshold the algorithm is guaranteed to lose no tracks. Note that in many cases this does not completely fix the track pattern but still leaves some space for optimization because many of the wide wire models that are affected by this threshold do not pack optimally between two power rails anyway.

Second, as mentioned above, an as uniform as possible spreading of the tracks of any track pattern has some benefits to timing and noise (with the exception of the tracks next to the power rails, these can be moved close to the power rails as power never switches). Thus, we use a heuristic post-processing routine, spreading tracks more evenly if this does not change the objective value. This routine works as follows. For each track pattern, it iterates from right to left over each track, checking if it can improve spreading locally. For the first and last tracks, it checks if there is a different coordinate for the track with the same objective value but closer to the power rail. For the other tracks, it checks if there is a different coordinate for the track with the same objective value but a more even spacing to both sides. This procedure is repeated until no track can be moved anymore. If considering only one track pattern alone, this results in perfect spreading. When optimizing two or more wire models at once, there are often few possibilities to improve spreading without increasing the objective function.

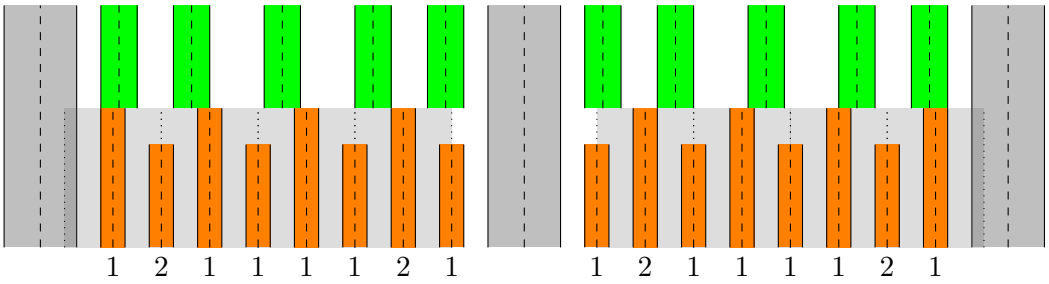
Remark. We do not consider to optimize this post-processing routine, because the trivial version does the job and its run time is negligible.

Concluding this section, we show some track patterns computed by our algorithm. Figure 4.14 shows the track patterns computed for the wire models that we already discussed in Section 4.2 and for which we have shown and analyzed track patterns generated by simpler algorithms in Figure 4.6 and 4.7. Note that our algorithm simultaneously optimized track patterns for the 1x and 1.5x wires that are shown in Figure 4.14 and also for 2x wires that are shown in Figure 4.16. In Figure 4.14(a) we can see that both wire models have the optimum number of tracks (five and eight respectively). Furthermore, each of the 1.5x tracks blocks only two 1x tracks (note that with each of the simpler approaches at least two of the 1.5x tracks blocked three default tracks simultaneously). Conversely, in Figure 4.14(b) we see that for our optimized tracks, all but two of the 1x tracks block only one 1.5x track. With the simpler approaches, at least four of the 1x tracks blocked two 1.5x tracks. Thus in this simple example, our optimized tracks are in each metric as least as good as each of the simple track patterns and often significantly better (apart from the minor objective of evenly spreading the tracks, this is slightly worse for the 1.5x tracks). This shows that already considering only two track patterns at a time we can get huge benefits by optimizing the track patterns.

If we consider more track patterns, we see the same effect. Sometimes, there are basically no options, but if there are, our algorithm can reduce the number of tracks blocked significantly. Figure 4.15 and Figure 4.16 show track patterns TP3 and the optimized track patterns generated by our algorithm for 2x wires (with respect to 1x and 1.5x wires). 1.5x and 2x wires block the same number of tracks of each other, but the two 2x tracks in the center between the power rails block three instead of four 1x tracks and only two of the 1x tracks block two 2x tracks instead of four.



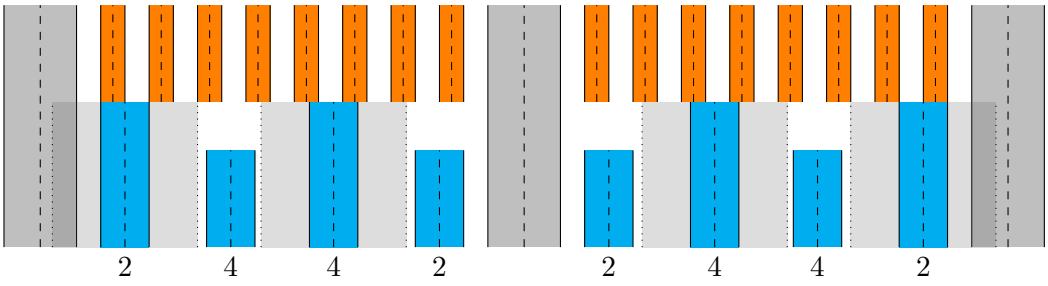
(a) Number of 1x tracks blocked by 1.5x wires.



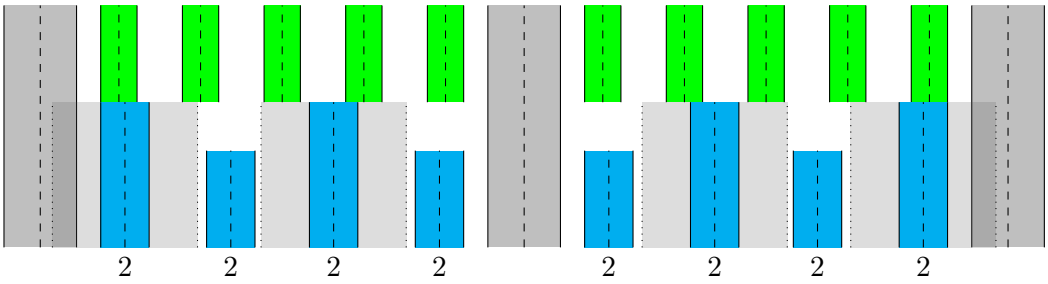
(b) Number of 1.5x tracks blocked by 1x wires.

Figure 4.14: Optimized tracks for 1x and 1.5x wires.

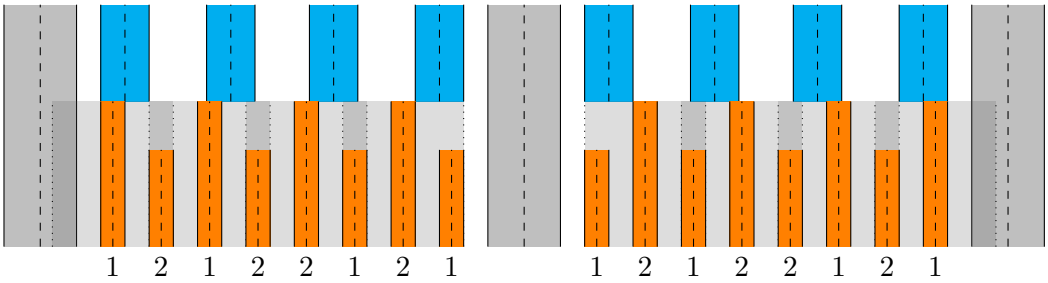
We will now show one more example illustrating benefits and effects if more than two track patterns are considered simultaneously. Our algorithm provides a powerful framework to test different settings. For example, one could consider using TP3 for the most common wire model and then optimize each track pattern separately while considering the number of blocked tracks of the default track pattern. Figure 4.17, 4.18 and 4.19 show the track patterns TP3, the track patterns if TP3 is taken for the most common wire model and each other is optimized separately and the result if all three track patterns are optimized simultaneously for 1x wires with 1.5x spacing, 1.5x wires and 2x wires (note that this example comes from a differently designed layer, therefore power rails are set up differently and thus the problem is entirely different to the examples above). As expected, the intermediate approach gives some but not all the benefits of the simultaneously optimized track patterns. For example for the 2x wires, in Figure 4.17(a) we can see that two of the 2x tracks block three 1x tracks, in Figure 4.18(a) only one of the 2x tracks block three 1x tracks and in Figure 4.19(a) all 2x tracks block only two 1x tracks. Looking at all combinations of track patterns, the intermediate approach is clearly superior than simply using TP3, but optimizing three track patterns simultaneously is in turn clearly superior to the intermediate approach in this example. Furthermore, there is no reason why the intermediate approach should ever be better than our optimized approach. Due to the fact that the run time of all approaches are negligible compared to the run time of BonnRouteDetailed, we do not consider the intermediate approach any further.



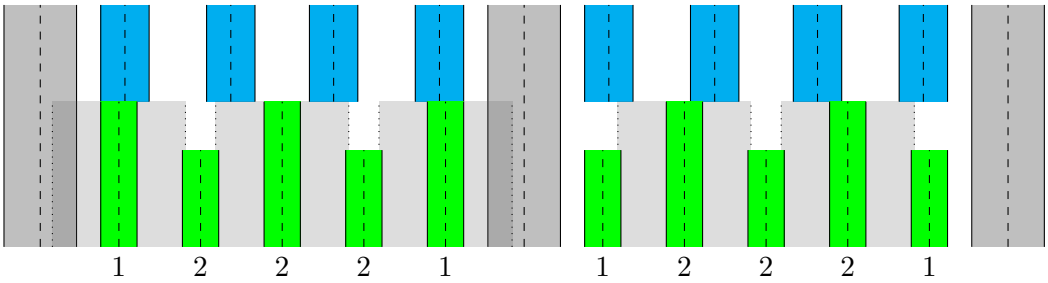
(a) Number of 1x tracks blocked by 2x wires.



(b) Number of 1.5x tracks blocked by 2x wires.

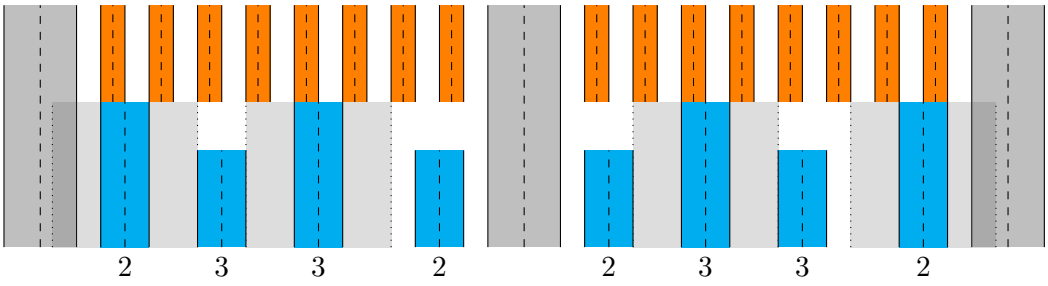


(c) Number of 2x tracks blocked by 1x wires.

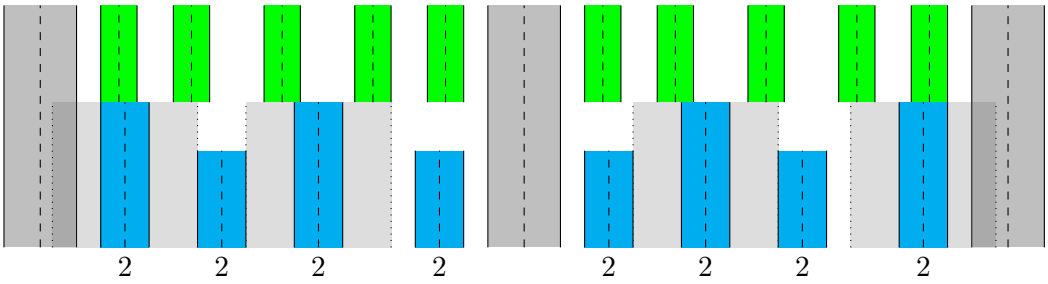


(d) Number of 2x tracks blocked by 1.5x wires.

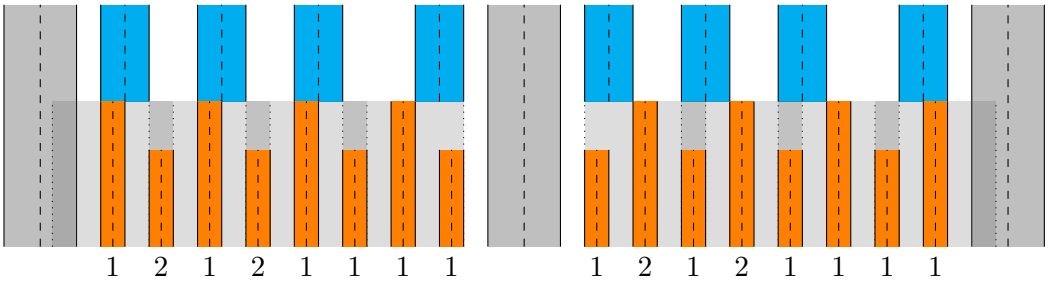
Figure 4.15: Use TP3 for 2x, 1.5x and 1x wires.



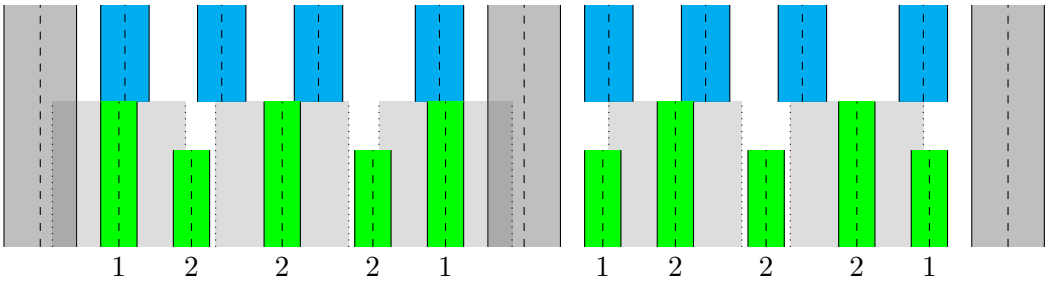
(a) Number of 1x tracks blocked by 2x wires.



(b) Number of 1.5x tracks blocked by 2x wires.

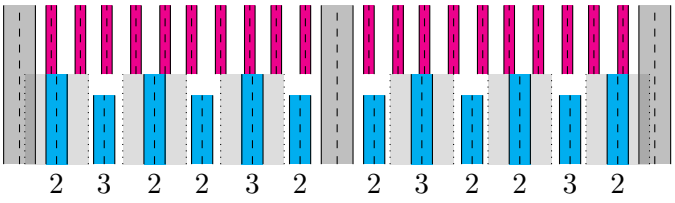


(c) Number of 2x tracks blocked by 1x wires.

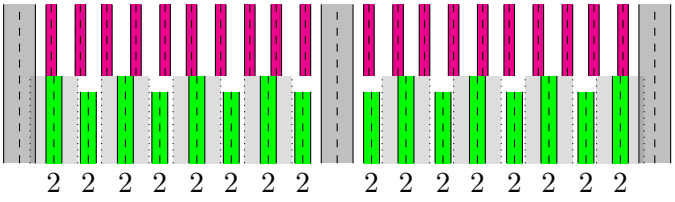


(d) Number of 2x tracks blocked by 1.5x wires.

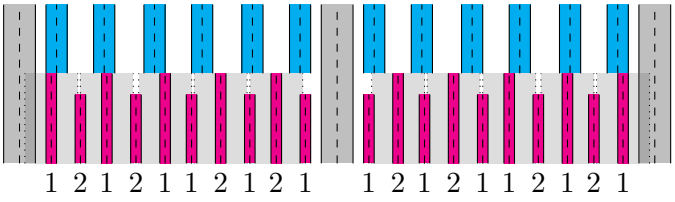
Figure 4.16: Optimized tracks for 2x, 1.5x and 1x wires.



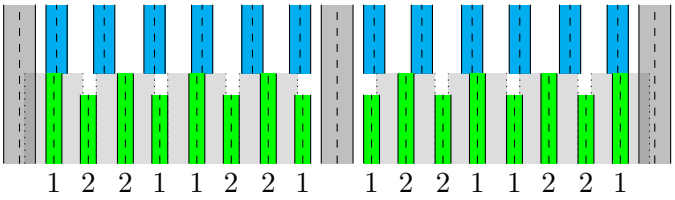
(a) Number of 1x tracks blocked by 2x wires.



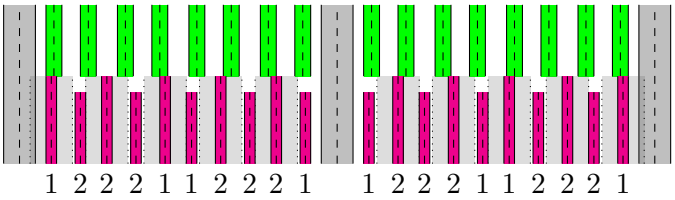
(b) Number of 1x tracks blocked by 1.5x wires.



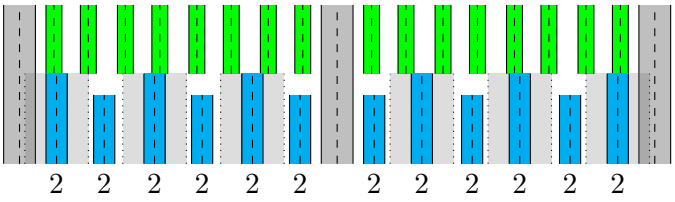
(c) Number of 2x tracks blocked by 1x wires.



(d) Number of 2x tracks blocked by 1.5x wires.

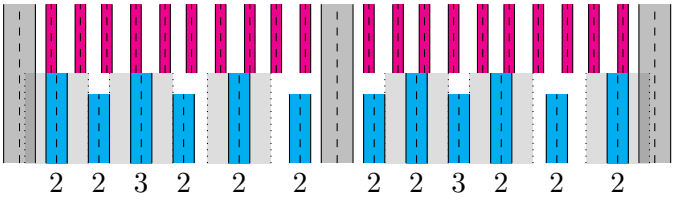


(e) Number of 1.5x tracks blocked by 1x wires.

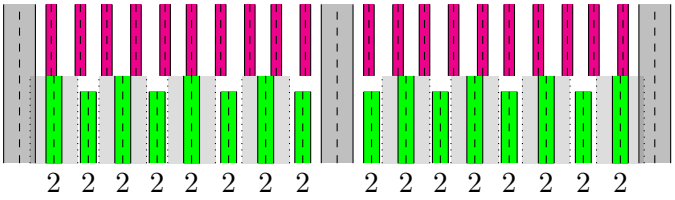


(f) Number of 1.5x tracks blocked by 2x wires.

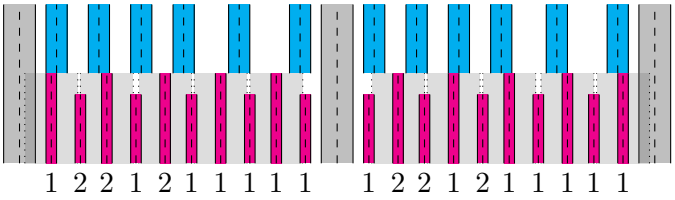
Figure 4.17: Use TP3 for 1x wires with 1.5x spacing, 1.5x wires and 2x wires.



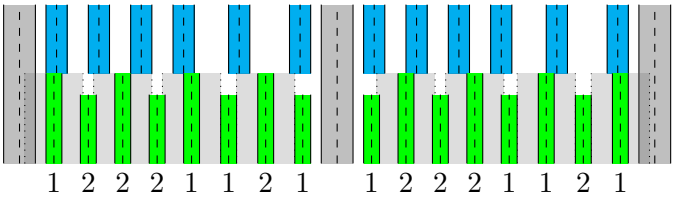
(a) Number of 1x tracks blocked by 2x wires.



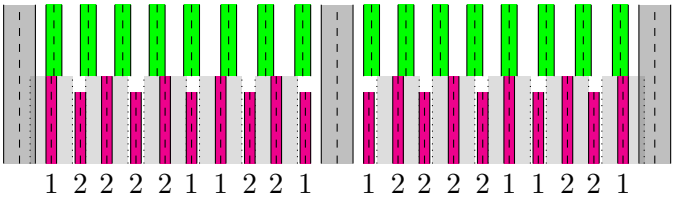
(b) Number of 1x tracks blocked by 1.5x wires.



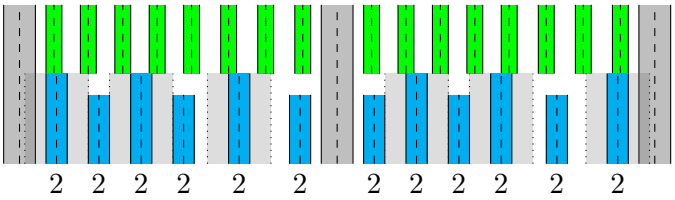
(c) Number of 2x tracks blocked by 1x wires.



(d) Number of 2x tracks blocked by 1.5x wires.

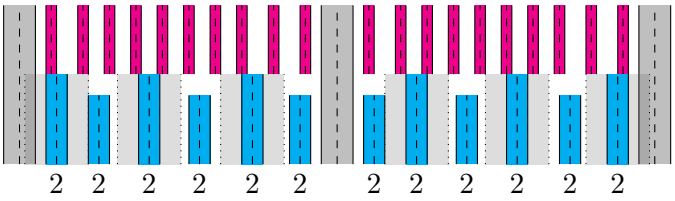


(e) Number of 1.5x tracks blocked by 1x wires.

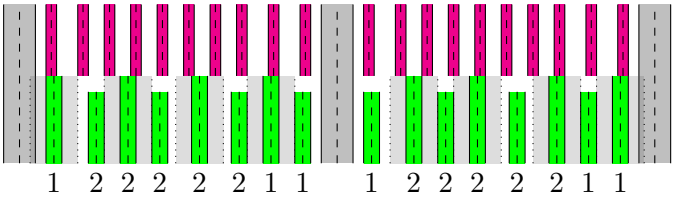


(f) Number of 1.5x tracks blocked by 2x wires.

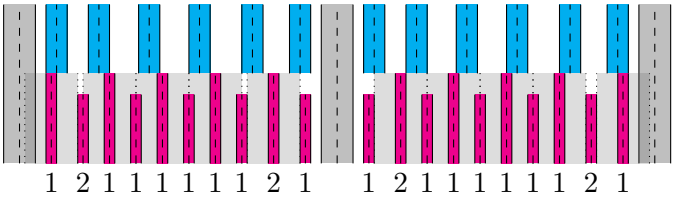
Figure 4.18: Use TP3 for 1x wires with 1.5x spacing and optimize tracks for 1.5x wires and 2x wires separately while taking into account tracks for 1x wires with 1.5x spacing.



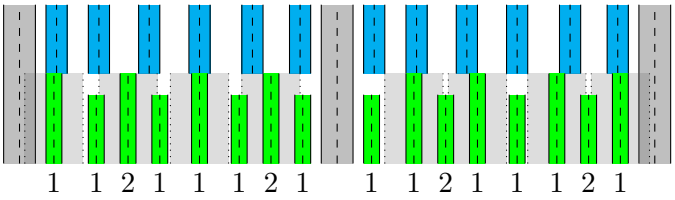
(a) Number of 1x tracks blocked by 2x wires.



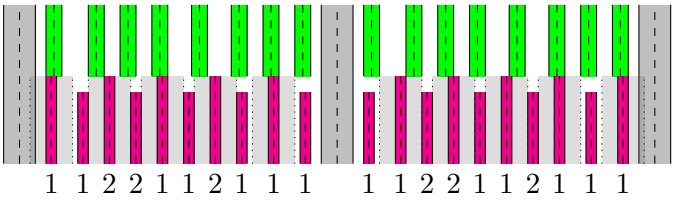
(b) Number of 1x tracks blocked by 1.5x wires.



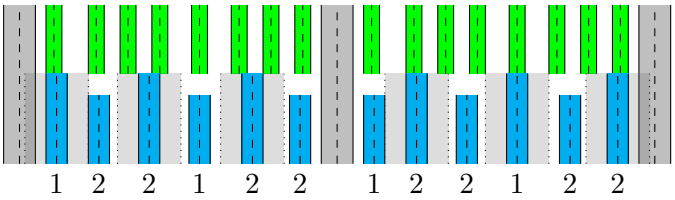
(c) Number of 2x tracks blocked by 1x wires.



(d) Number of 2x tracks blocked by 1.5x wires.



(e) Number of 1.5x tracks blocked by 1x wires.



(f) Number of 1.5x tracks blocked by 2x wires.

Figure 4.19: Optimized tracks for 1x wires with 1.5x spacing, 1.5x wires and 2x wires.

4.4 Some Practical Considerations

In this section we discuss some further practical considerations concerning our track pattern generation algorithm. First and most importantly, we describe how we use Algorithm 4.3.12 in practice. Our algorithm can easily optimize up to three track patterns simultaneously while considering a larger number of already precomputed track patterns as side constraints in very good run time. In almost all cases, four track patterns can be optimized simultaneously in very good run time and often, depending on the involved spacing values, five or more track patterns can be optimized simultaneously in an acceptable run time. However, on the instances we use for testing there are only very few cases where four or more different wire models are used to a relevant extend on any layer. Even on the instances where on some layers more than three wire models are used, the overall results improve only very slightly, if at all, if optimizing more than three wire models simultaneously. The combination of track patterns calculated improves (in the sense that for some pairs of wire models, the number of blocked tracks decrease) but these improvements effect only a very small fraction of the overall routing on these instances such that the effects on the overall results are negligible. Therefore the setting that we use is as follows: We sort all wire models by the frequency they occur on the chip. Then we first optimize the three most frequent wire models simultaneously (if there are three wire models with a relevant portion of the total wire length each, otherwise we use less) and then optimize each remaining wire model (in the order of decreasing frequency) alone while considering the five most frequent wire models as side constraints. We could of course also optimize wire models four to six simultaneously, but experiments indicated that this does not improve results significantly. On the testbed we use, for the vast majority of testcases and layers there are only few wire models used to a substantial amount anyway.

As discussed in Section 4.2, there are competing objectives. Most relevantly we need to mention the ability to switch between different wire models and the ability to pack a mixture of different wire models efficiently here. Obviously, on a dense chip, the ability to pack wires efficiently is very important. Equally obviously, on a very sparse chip, the ability to pack wires efficiently does not matter and the ability to switch different wire models predominates. Furthermore, the importance of the different objectives also depends on the definition of pins, the assignment of layers to nets and the definition of special wire codes that are used to access pins. Also, wire models that are in terms of total wire length only very rarely used have very little influence on the total packing of wire models but can be responsible for a significant amount of necessary wire model changes. Therefore, if such rare wire models do not pack very efficiently or block many tracks it has little influence on overall routing results. On the other hand it might be important that they can be easily combined with other wire models. On the other hand, a wire model that is responsible for most of the total wire length needs to pack efficiently on a dense design. This is the motivation for one further modification to our algorithm that we made. We have a (technology dependent) threshold such that for wire models with total relative estimated wire length above the threshold we compute an optimized track pattern and for wire models with total relative estimated wire length below the threshold we assign the track pattern of the default (most frequent) wire model. This ensures that paths can easily switch between all these infrequent wire models and the

most frequent wire model while harming global packing efficiency only very marginally. This approach has the additional benefit that it limits the total number of tracks generated (of all track patterns combined). This is important, because some of the data structures used in BonnRouteDetailed (especially the grid) use the union of the tracks of all track patterns to store objects on the chip. This becomes inefficient if there are very many different tracks which are used for only very few wires. Experiments have shown that this approach is superior to always optimizing all wire models with our algorithm.

The objective function in the algorithm (and the choice of wire models to optimize first) heavily depends on the estimated total frequency of the wire models. There are a number of ways to obtain such estimates before routing. BonnRouteDetailed currently uses an estimate based on global wires. For each net, it gets one default wire model per layer. Therefore it counts on each layer for each net the global wire length on that layer for the associated wire model. This does not take into account the special wire models used to access pins. It is unclear how to consider those and if considering them yields any improvements at all. If such detailed information is not yet available (for example because these track patterns should already be computed before global routing such that the global router can use them to approximate resource usage of different wire models more accurately), simpler models can be used. For example, for each net the bounding box wire length could be used and distributed somehow (in the simplest case evenly) over the layers assigned to the net. Which estimate is best depends on the kind of information available and on the accuracy needed, which in turn heavily depends on the criticality of the chip and on the mix of wire models used.

Another important point to mention is that one should consider the stability of the computed track patterns during the chip design work flow. Typically, during the design of a chip, there are multiple iterations of rerouting the chip from scratch, analyzing the result, changing something and iterating. After that, usually most of the routing is fixed and only small portions are rerouted in each iteration to fix any remaining problems. The potential issue to consider here is that during these iterations often the frequencies of different wire models change. This leads to the risk that track patterns suddenly change between iterations which might disrupt the design flow heavily and lead to unpredictable results due to relations between track patterns and pins for example. Furthermore, if track patterns change during the later phases when most of the routing is already fixed, new wires on new track patterns might fit very badly between already fixed wires on old track patterns.

There are a number of ways to prevent these problems. One easy way is to compute track patterns once and store them with the design. Then one can keep the same track patterns as long as they are reasonable and only recompute them if a lot of wire model usages change. This has the advantage that the designer has complete control about the creation and change of the track patterns but at the same time the disadvantage that it requires manual decisions. Another option is to compute track patterns each time a chip is routed, but take into account the positions of already present wires and potentially blockages and pins, preferring tracks that match existing structures. This has the advantage that it requires no manual interaction and fixes at least the second half of the problem. If most of the wires are already routed on some given tracks and these are considered, the algorithm can be tuned to find the same track patterns again with high

probability (unless there is a very good reason). This maintains full flexibility of the tracks generated while routing the complete chip from scratch but at the same time ensures a high level of stability later on.

One should note that considering existing wires should work smoothly if they were created by the same router with the same constraints, but considering pin and or blockage positions is in general a lot more tricky. Often, there are too few pins and / or blockages that they make any significant difference on many layers and they are designed with different objectives in mind and fixed comparatively early in the flow. Thus adapting tracks to pins and or blockages can lead to poor results because the tracks that fit to the pins inherently pack badly. We made various experiments taking pin positions into account but could achieve no benefit at all and often even got worse results. Thus we recommend to consider existing wires only or to store track patterns at the chip.

Reconsidering the example from Figure 4.14, one can notice that some of the 1x tracks are worse than the others, because they block more 1.5x tracks. This is a very common phenomenon, in most cases some tracks block more tracks of other wire models than others. Thus one could think that it is a good idea to prefer better tracks during routing over worse tracks by introducing different costs. On the other hand, this is another type of costs that is added to the already complex mix of different costs for different objectives. Even though potentially improving packing of tracks on average, this can also lead to longer (and thus suboptimal) routes. We made a number of experiments which showed no benefit. Positive and negative effects seemed to roughly even out if costs were chosen at a suitable level. Thus we propose to discard this idea since it gives no clear practical benefit but complicates code and algorithms.

Above we assume that on the layers considered there are evenly spaced power rails and no colors. Both assumptions may be wrong in practice, but this difficulty can easily be overcome. First, if we have colors on a layer, usually there are predefined track patterns anyway that are designed to fulfill complex restrictions on colored metal. If this is not the case, we can easily adapt our algorithm to store a color along with each track and thus can create colored track patterns. This increases the number of candidates but is no fundamental problem. We did not test this because we are not aware of any instances where track patterns on colored layers can be subject to automated optimization. Second, we do not necessarily have uniformly spaced power rails. There might be power rails with different spacings. In this case, we can trivially apply our algorithm for each different power rail spacing separately and use the computed tracks only in the corresponding power bays. Furthermore, there might be power staples instead of power rails, but this does not really make a difference as in practice there are usually very many such power staples on only few coordinates in non-preferred direction. We can then treat these rows of power staples just like a power rail. Usually it is not possible to route any meaningful wire in preferred direction between two power staples anyway. Only in the case when power staples are not arranged in rows in preferred direction, our algorithm is not directly applicable.

It remains to discuss the choice of the various parameters which we introduced above. We have done a lot of tuning of the individual parameters and finally chose the following values. For our choice of instances, $w_n = 2$ gives best results independent of the technology of the chips. We do not consider a wire model in the first optimizing step optimizing up

to three track patterns if it is used for less than one per cent of total wire length. We do not allow any track to be lost for a given wire model if less than five tracks are possible. Furthermore, we use our optimized track patterns only if a wire model is used for at least five (for 14nm instances) respectively a half per cent (for 7nm instances) of the total wire length, otherwise we assign the track pattern of the default wire model. This choice is very likely to change, because current instances are at least to some extent tuned to former sets of track patterns, thus increasing the negative effects of different track patterns especially at pins and access wire models. Once the whole design of the chips gets tuned to the new track patterns we expect to be able to use optimized track patterns also for wire models which are less frequent. This expectation is backed up by the fact that tuning on older and more mature 14nm instances showed a much larger cut-off value to be optimal than tuning on newer and less optimized 7nm instances. Unfortunately it is not practical to design a whole chip from start to finish with both choices of track patterns due to the massive amount of manual work required.

4.5 Conclusion

In this section, we have discussed the problem to generate good (soft) track patterns automatically, identifying objectives and restrictions and discussing a number of simple approaches. Then we have developed a fast algorithm computing optimal combinations of track patterns, proving its optimality and briefly discussing its asymptotic run time. Furthermore, we have discussed a number of practical implications and use cases. We show its practical effectiveness and run time in Section 6.2.

Chapter 5

Checking Diff-Net Rules

The metal shapes of different nets need to obey certain minimum distance rules. In the simplest form, the shapes of different nets may not touch each other. Touching shapes would be electrically connected and thus cause wrong behavior of the chip. In practice also very close shapes have a high probability to end up electrically connected due to variations during manufacturing and are thus forbidden.

In former technologies, often a simple minimum distance rule d was sufficient, e.g. all $s_1, s_2 \in \mathcal{S}$ that belong to different nets must fulfill:

$$d_2(s_1, s_2) \geq d$$

With each new technology, the produced metal structures become smaller and the fabrication processes become more complicated. This introduces more and more complex rules for the metal shapes of a chip. Typically, the smaller the involved structures, the more complicated are the rules. Thus the most complex rules occur on the lowest layers, especially on the layers containing the transistors of a chip, but also the lowest routing layers which mostly contain circuit internal wiring and pins.

For circuit design and for designing specialized and highly optimized structures like memory elements, it seems inevitable to model many complex rules exactly. For routing signal nets however, it is often acceptable to simplify the most complex rules. By introducing slightly stronger restrictions, it is often possible to simplify diff-net spacing rules a lot. In practice such techniques have proven to be very helpful to reduce routing run time, for example so-called line end minimum distance rules can be modeled very well by simpler rules in practice [40]. As another example, rules forbidding certain combinations of three or more wires with certain widths at certain distances next to each other can be automatically fulfilled by forcing wires of certain widths to be placed on certain track patterns.

In this chapter, we first model diff-net rules formally in Section 5.1 and derive some important properties. We develop two descriptions of an important class of diff-net rules and prove their equivalence. We prove for a number of practically important diff-net rules that they are diff-net rules in our theoretical sense and belong to this class. Then we describe how they are handled efficiently by BonnRouteDetailed in two steps. First, we describe in Section 5.2 a basic and simple procedure to check diff-net rules. Second, we develop in Section 5.3 a more complex and highly optimized framework suitable for

extensive diff-net rule checking during path search. Our optimized framework for diff-net rule checking reduces the run time of the main detailed routing step of BonnRouteDetailed by factor 2.4 and the total run time of BonnRouteDetailed by factor 2.2. We present more detailed experimental results showing the effectiveness of the presented methods in Section 6.3.

5.1 Diff-Net Rules

For us, a diff-net rule is a function taking two shapes and returning if they are legal with respect to each other that fulfills certain requirements.

Definition 5.1.1. *We call a function* $dr : \mathcal{S} \times \mathcal{S} \to \mathcal{B}$*:*

- ***symmetric**, iff:* $\forall s_1, s_2 \in \mathcal{S} : dr(s_1, s_2) = dr(s_2, s_1)$
- ***local**, iff:* $\exists b \in \mathbb{N} : \forall s_1, s_2 \in \mathcal{S}, d_{max}(s_1, s_2) > b : dr(s_1, s_2) = true$
- ***invariant under translation**, iff:* $\forall s_1, s_2 \in \mathcal{S}, (x, y) \in \mathbb{Z}^2 : dr(s_1, s_2) = dr(s_1 + (x, y), s_2 + (x, y))$

For such a local function we call the smallest such b its **locality constant**. For certain restrictions on the shapes the diff-net rule is applied to, we call the corresponding b the locality constant with respect to these restrictions. We also define the locality constant of a set of such local functions as the maximum of the locality constants of the functions. Formally:

Definition 5.1.2. *Let* $dr : \mathcal{S} \times \mathcal{S} \to \mathcal{B}$ *be local and symmetric,* $sc, sc_1, sc_2 \in \mathcal{SC}, l \in \mathcal{L}$*. Define:*

$$b(dr) := \min\{b \in \mathbb{N} : \forall s_1, s_2 \in \mathcal{S}, d_{max}(s_1, s_2) > b : dr(s_1, s_2) = true\}$$

$$b_{sc,l}(dr) := \min\{b \in \mathbb{N} : \forall s_1, s_2 \in \mathcal{S}, l(s_1) = l(s_2) = l, sc(s_1) = sc, \\ d_{max}(s_1, s_2) > b : dr(s_1, s_2) = true\}$$

$$b_{sc_1,sc_2,l}(dr) := \min\{b \in \mathbb{N} : \forall s_1, s_2 \in \mathcal{S}, l(s_1) = l(s_2) = l, sc(s_1) = sc_1, sc(s_2) = sc_2, \\ d_{max}(s_1, s_2) > b : dr(s_1, s_2) = true\}$$

Let DR be a set of such local functions. Define:

$$b(DR) := max\{b(dr) : dr \in DR\}$$

$$b_{sc,l}(DR) := max\{b_{sc,l}(dr) : dr \in DR\}$$

$$b_{sc_1,sc_2,l}(DR) := max\{b_{sc_1,sc_2,l}(dr) : dr \in DR\}$$

Now we can define diff-net rules formally:

Definition 5.1.3. *A **diff-net rule** is a symmetric, local function $dr : \mathcal{S} \times \mathcal{S} \to \mathcal{B}$ that is invariant under translation. Define the **set of all diff-net rules**:*

$$\mathcal{DR} := \{dr : \mathcal{S} \times \mathcal{S} \to \mathcal{B} : dr \text{ symmetric, local and invariant under translation}\}$$

A diff-net rule directly gives us a function on two sets of shapes in the following way. Let $dr \in \mathcal{DR}, S_1, S_2 \subseteq \mathcal{S}$. Then define

$$dr(S_1, S_2) := \bigwedge_{s_1 \in S_1, s_2 \in S_2} dr(s_1, s_2)$$

This means, two sets of shapes are legal with respect to each other if each shape in the first set is legal with respect to each shape in the other set.

Efficient path search algorithms can only handle diff-net rules that fulfill some further constraints, at least on routing layers. We will now model these requirements.

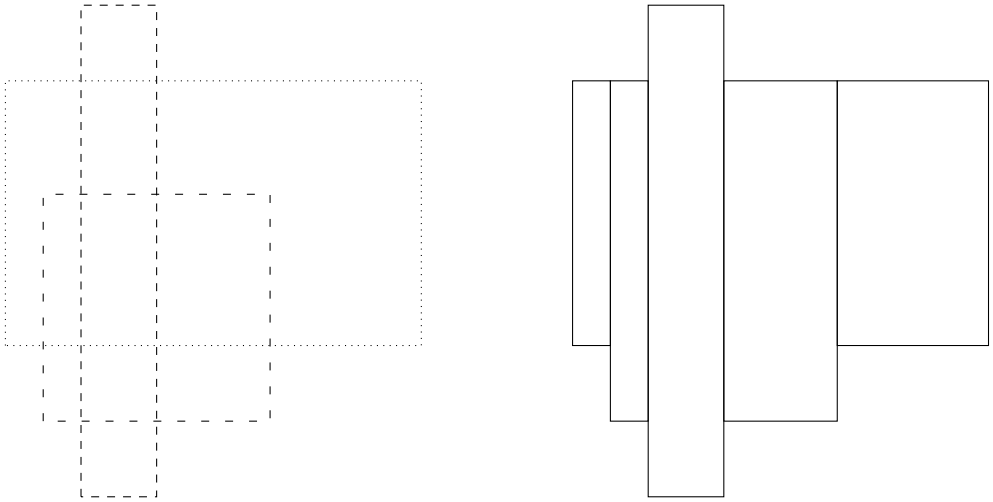
Definition 5.1.4. *We call a diff-net rule dr*

- ***invariant under representation**, iff: $\forall S_1, S_1', S_2 \subseteq \mathcal{S}$, $\bigcup_{s \in S_1} r(s) = \bigcup_{s \in S_1'} r(s)$, $S_1 \cup S_1'$ homogeneous : $dr(S_1, S_2) = dr(S_1', S_2)$*
- ***monotone**, iff: $\forall s_1, s_1', s_2 \in \mathcal{S}, s_1' \subseteq s_1, dr(s_1, s_2) = true : dr(s_1', s_2) = true$*
- ***consistent**, iff: $\forall s_1, s_2 \in \mathcal{S}, diam(s_1) > 1, dr(s_1, s_2) = false : \exists s_1' \in \mathcal{S}, s_1' \subsetneq s_1, dr(s_1', s_2) = false$*

Thus, monotone means that if a shape is legal with regard to another shape then any subshape also is. Consistent means, if some shape with longest edge length larger than 1 is illegal with respect to another shape then at least one part is illegal too. Invariant under representation means that if some metal area is represented in different ways by shapes, this does not influence legality with respect to other shapes. An illustration of this concept can be found in Figure 5.1.

Remark. Note that in the definition of consistent, we exclude shapes with longest edge length smaller or equal to 1. Shapes with the largest edge length smaller than 1 do not have any proper subshapes. Shapes with the largest edge length equal to 1 do have proper subshapes but any set of proper subshapes represents a different metal area (in a way, they can not be split into parts). Therefore we do not require that there is a proper subshape that is illegal with regard to another shape if the shape is illegal with regard to that other shape.

Lemma 5.1.5. *Monotone and consistent are independent, meaning that there are both diff-net rules that are consistent and not monotone as well as diff-net rules that are monotone and not consistent.*



S_1 S_1'

Figure 5.1: For any diff-net rule that is invariant under representation, S_1 should be legal with regard to a set of shapes S_2 if and only if S_1' is.

Proof. Let $d \in \mathbb{N}, k \in \mathbb{N}, k \geq 2$. Consider the following functions:

$$dr_1(s_1, s_2) := \begin{cases} \mathit{false} & d_{max}(r(s_1), r(s_2)) < b, area(s_1) \geq k, area(s_2) \geq k \\ \mathit{true} & \text{else} \end{cases}$$

$$dr_2(s_1, s_2) := \begin{cases} \mathit{false} & d_{max}(r(s_1), r(s_2)) < b, area(s_1) \leq k, area(s_2) \leq k \\ \mathit{true} & \text{else} \end{cases}$$

Both are symmetric, local with locality constant $b-1$ and invariant under translation, thus both are diff-net rules.

Consider dr_1: Let $s_1, s_1', s_2 \in \mathcal{S}, s_1' \subseteq s_1, dr_1(s_1, s_2) = \mathit{true}$. This means, $area(s_1) < k$ or $area(s_2) < k$ or $d_{max}(r(s_1), r(s_2)) \geq b$. Because we have $d_{max}(r(s_1'), r(s_2)) \geq d_{max}(r(s_1), r(s_2))$ and $area(s_1') \leq area(s_1)$ we also get $dr_1(s_1', s_2) = \mathit{true}$. Thus dr_1 is monotone.

Consider $s_1, s_2 \in \mathcal{S}, d_{max}(s_1, s_2) < b, area(s_1) = k, area(s_2) = k$. Then we have $dr_1(s_1, s_2) = \mathit{false}$, and because $k \geq 2, diam(s_1) > 1$. But we also have $\forall s_1'' \subsetneq s_1 : area(s_1'') < area(s_1) = k$. Thus $dr_1(s_1'', s_2) = \mathit{true}$. Thus dr_1 is not consistent.

Consider dr_2: Let $s_1, s_2 \in \mathcal{S}, diam(s_1) > 1, dr_2(s_1, s_2) = \mathit{false}$. Then we have $d_{max}(r(s_1), r(s_2)) < b, area(s_1) \leq k, area(s_2) \leq k$. Let $s_1' \subsetneq s_1$ with $d_{max}(r(s_1'), r(s_2)) = d_{max}(r(s_1), r(s_2))$. Then we get $area(s_1') < area(s_1) \leq k$. Thus we get $dr_2(s_1', s_2) = \mathit{false}$. dr_2 is consistent.

On the other hand, let $s_1, s_2, s_1' \in \mathcal{S}, d_{max}(r(s_1), r(s_2)) < b, area(s_2) = k, area(s_1) = k+1, s_1' \subsetneq s_1$. Then we have $dr_2(s_1, s_2) = \mathit{true}$ and $area(s_1') < area(s_1) = k+1$ and thus $area(s_1') \leq k$. Thus we get $dr_2(s_1', s_2) = \mathit{false}$ and dr_2 is not monotone. □

Theorem 5.1.6. *A diff-net rule is invariant under representation if and only if it is monotone and consistent.*

Proof. Let $dr \in \mathcal{DR}$ be invariant under representation. Let $s_1, s_1', s_2 \in \mathcal{S}$, $s_1' \subseteq s_1$, $dr(s_1, s_2) = \mathit{true}$. We have $r(s_1) \cup r(s_1') = r(s_1)$ and $\{s_1, s_1'\}$ homogeneous. Thus by the definition of invariance under representation we get $dr(\{s_1, s_1'\}, \{s_2\}) = dr(\{s_1\}, \{s_2\}) = \mathit{true}$. Further $\mathit{true} = dr(\{s_1, s_1'\}, \{s_2\}) = dr(s_1, s_2) \wedge dr(s_1', s_2) = dr(s_1', s_2)$. Thus dr is monotone.

Now let $s_1, s_2 \in \mathcal{S}, \mathit{diam}(s_1) > 1, dr(s_1, s_2) = \mathit{false}$ (if such s_1, s_2 do not exist, dr is trivially consistent). Then we have, $r(s_1) = \bigcup_{s \subseteq s_1, \mathit{diam}(s) \leq 1} r(s) = \bigcup_{s \subsetneq s_1, \mathit{diam}(s) \leq 1} r(s)$. Because dr is invariant under representation and by definition, we get $\mathit{false} = dr(s_1, s_2) = dr(\{s \subsetneq s_1, \mathit{diam}(s) \leq 1\}, \{s_2\}) = \bigwedge_{s \subsetneq s_1, \mathit{diam}(s) \leq 1} dr(s, s_2)$. Thus $\exists s \subsetneq s_1 : dr(s, s_2) = \mathit{false}$. Therefore, dr is consistent.

Now, let dr be monotone and consistent. Let $S_1, S_1', S_2 \subseteq \mathcal{S}$, $\bigcup_{s \in S_1} r(s) = \bigcup_{s \in S_1'} r(s)$ and $S_1 \cup S_1'$ homogeneous. Assume $dr(S_1, S_2) \neq dr(S_1', S_2)$. W.l.o.g. we can assume $dr(S_1, S_2) = \mathit{false}$ and $dr(S_1', S_2) = \mathit{true}$. We have: $dr(S_1, S_2) = \bigwedge_{s_1 \in S_1, s_2 \in S_2} dr(s_1, s_2)$. Thus there are $s_1 \in S_1, s_2 \in S_2$ with $dr(s_1, s_2) = \mathit{false}$. Now we have two cases. Either $\mathit{diam}(s_1) \leq 1$. Otherwise, we have $\mathit{diam}(s_1) > 1$. Because dr is consistent we get: $\exists s_1' \subsetneq s_1 : dr(s_1', s_2) = \mathit{false}$. We further have $\mathit{area}(s_1') < \mathit{area}(s_1)$ or $\mathit{diam}(s_1') < \mathit{diam}(s_1)$. By repeating this argument iteratively, we have: $\exists s_1'' \subseteq s_1, \mathit{diam}(s_1'') \leq 1, dr(s_1'', s_2) = \mathit{false}$. Now because $\bigcup_{s \in S_1} r(s) = \bigcup_{s \in S_1'} r(s)$, $\mathit{diam}(s_1'') \leq 1$ and $S_1 \cup S_1'$ homogeneous we have $\exists s_1''' \in S_1' : s_1'' \subseteq s_1'''$. If $dr(s_1''', s_2) = \mathit{true}$, we also had $dr(s_1'', s_2) = \mathit{true}$ because dr is monotone. Thus we have $dr(s_1''', s_2) = \mathit{false}$. By definition of the diff-net rule for sets it follows $dr(S_1', S_2) = \mathit{false}$. This is a contradiction, thus we have $dr(S_1, S_2) = dr(S_1', S_2)$ and dr is invariant under representation. □

The following lemma shows that diff-net rules that are invariant under representation are completely defined by its values on shapes of diameter less or equal 1.

Lemma 5.1.7. *Let $dr \in \mathcal{DR}$, let dr be invariant under representation, $S_1, S_2 \subseteq \mathcal{S}$, S_1 homogeneous. Then*

$$dr(S_1, S_2) = \bigwedge_{s \in \mathcal{S}, \mathit{attr}(s) = \mathit{attr}(S_1), \mathit{diam}(s) \leq 1, r(s) \subseteq \bigcup_{t \in S_1}(r(t))} dr(\{s\}, S_2)$$

Proof. Let $S_1' := \{s \in \mathcal{S}, \mathit{attr}(s) = \mathit{attr}(S_1), \mathit{diam}(s) \leq 1, r(s) \subseteq \bigcup_{t \in S_1}(r(t))\}$. Then we have $\bigcup_{s \in S_1} r(s) = \bigcup_{s \in S_1'} r(s)$. By the definition of invariance under representation we then get $dr(S_1, S_2) = dr(S_1', S_2)$. By the definition of diff-net rules for sets we further get $dr(S_1', S_2) = \bigwedge_{s_1 \in S_1', s_2 \in S_2} dr(s_1, s_2) = \bigwedge_{s_1 \in S_1'} \bigwedge_{s \in \{s_1\}, s_2 \in S_2} dr(s, s_2) = \bigwedge_{s_1 \in S_1'} dr(\{s_1\}, S_2)$ □

We will now look at the most frequent restrictions that are imposed on shapes of different nets in practice and examine if they are diff-net rules in our mathematical sense and which constraints they fulfill. To do so, we need to introduce the notion of **run length** between two shapes. Some examples for this concept can be found in Figure 5.2.

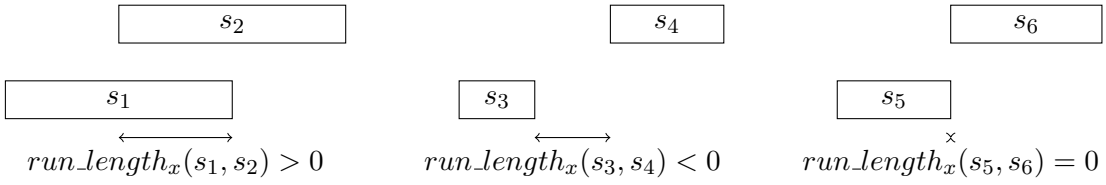


Figure 5.2: Run length examples.

Definition 5.1.8. *Let $s_1, s_2 \in \mathcal{S}$. Define*

$$run\_length_x(s_1, s_2) := min(x_{max}(s_1), x_{max}(s_2)) - max(x_{min}(s_1), x_{min}(s_2)).$$

Analogously, define

$$run\_length_y(s_1, s_2) := min(y_{max}(s_1), y_{max}(s_2)) - max(y_{min}(s_1), y_{min}(s_2)).$$

Let

$$run\_length(s_1, s_2) := max(run\_length_x(s_1, s_2), run\_length_y(s_1, s_2)).$$

The probably most common and simplest possible diff-net rule is the restriction that shapes need to have a specific **Euclidean distance**. This is an example of a wide class of diff-net rules:

Definition 5.1.9. *Let $p \in \mathbb{R} \cup \{\infty\}, 1 \leq p \leq \infty, d \in \mathbb{N}, s_1, s_2 \in \mathcal{S}$, then define*

$$dist_d^p(s_1, s_2) := \begin{cases} false & d_p(s_1, s_2) < d \\ true & else \end{cases}$$

This means, shapes need to have at least distance d measured by the p-norm.

Remark. Note that according to this definition the rule applies between shapes of any shape classes, colors and on any layers. Lemma 5.1.14 shows that we can also restrict any rule to apply only to shapes on the same layer and of certain shape classes and color combinations. In fact, analogously to Lemma 5.1.14, we can also restrict any diff-net rule to any combination of layers if necessary.

Lemma 5.1.10. *For all $p \in \mathbb{R} \cup \{\infty\}, 1 \leq p \leq \infty, d \in \mathbb{N}$, $dist_d^p$ is a diff-net rule that is invariant under representation.*

Proof. $dist_d^p$ is symmetric. Because d_p is invariant under translation, this is also true for $dist_d^p$. Further, $dist_d^p$ is local with locality constant $d-1$: Let $s_1, s_2 \in \mathcal{S}, d_{max}(s_1, s_2) \geq d$. Then we have $d \leq d_{max}(s_1, s_2) \leq d_p(s_1, s_2)$ and therefore $dist_d^p(s_1, s_2) = true$. Thus $dist_d^p$ is a diff-net rule. By Theorem 5.1.6, it suffices to show that $dist_d^p$ is monotone and consistent. Let $s_1, s_1', s_2 \in \mathcal{S}, s_1' \subseteq s_1$. Then $d_p(s_1', s_2) \geq d_p(s_1, s_2)$. Thus $dist_d^p$ is monotone. Let $s_1, s_2 \in \mathcal{S}, diam(s_1) > 1$. Then $\exists s_1' \in \mathcal{S}, s_1' \subsetneq s_1, d_p(s_1', s_2) = d_p(s_1, s_2)$. Thus $dist_d^p$ is also consistent. □

Another very common type of diff-net rules are **horizontal** and **vertical rules**. These rules formalize that shapes must have a specific distance in x- respectively y-direction if they have run-length at least some $a \in \mathbb{Z}$ in y- respectively x-direction. Formally:

Definition 5.1.11. *Let $d \in \mathbb{N}, d > 0, a \in \mathbb{Z}, s_1, s_2 \in \mathcal{S}$.*

$$hor_{a,d}(s_1, s_2) := \begin{cases} false & run\_length_y(s_1, s_2) \geq a, d_x(s_1, s_2) < d \\ true & else \end{cases}$$

Analogously, define

Definition 5.1.12.

$$ver_{a,d}(s_1, s_2) := \begin{cases} false & run\_length_x(s_1, s_2) \geq a, d_y(s_1, s_2) < d \\ true & else \end{cases}$$

Lemma 5.1.13. *For all $d \in \mathbb{N}, d > 0, a \in \mathbb{Z}$, $hor_{a,d}$ and $ver_{a,d}$ are monotone diff-net rules. $hor_{a,d}$ and $ver_{a,d}$ are consistent and thus invariant under representation if and only if $a \leq 1$.*

Proof. We prove the lemma for $hor_{a,d}$. The proof for $ver_{a,d}$ is completely analogous. $hor_{a,d}$ is obviously symmetric. Because d_x as well as run_length are invariant under translation, $hor_{a,d}$ is invariant under translation. Further, $hor_{a,d}$ is local with locality constant $max\{d-1, -a\}$. Thus $hor_{a,d}$ is a diff-net rule.

Now we show that $hor_{a,d}$ is monotone. Let $s_1, s_1', s_2 \in \mathcal{S}, s_1' \subseteq s_1, hor_{a,d}(s_1, s_2) = true$. We have $run\_length_y(s_1, s_2) < a$ or $d_x(s_1, s_2) \geq d$. Because $run\_length_y(s_1', s_2) \leq run\_length_y(s_1, s_2)$ and $d_x(s_1', s_2) \geq d_x(s_1, s_2)$, we also have $hor_{a,d}(s_1', s_2) = true$. Thus $hor_{a,d}$ is monotone.

Let now $a \leq 1$. We need to show that $hor_{a,d}$ is consistent. Let $s_1, s_2 \in \mathcal{S}, diam(s_1) > 1, hor_{a,d}(s_1, s_2) = false$. Thus we have $run\_length_y(s_1, s_2) \geq a$ and $d_x(s_1, s_2) < d$. There are two cases.

First, assume $x_{max}(s_1) \neq x_{min}(s_1)$. At least one x-coordinate of s_1 has x-distance smaller d to s_2, say x_0. Let $s_1' := ([x_0, x_0] \times [y_{min}(s_1), y_{max}(s_1)], l(s_1), sc(s_1), c(s_1), sp(s_1))$. We have $s_1' \subsetneq s_1$, $run\_length_y(s_1', s_2) = run\_length_y(s_1, s_2)$ and $d_x(s_1', s_2) < d$. Thus we have $hor_{a,d}(s_1', s_2) = false$.

Now, assume $x_{max}(s_1) = x_{min}(s_1)$. Because of $diam(s_1) > 1$ we have $y_{max}(s_1) \geq y_{min}(s_1) + 2$. Because $a \leq 1$, some part of s_1 of vertical length at most 1 is responsible for $run\_length_y(s_1, s_2) \geq a$. If $a = 1$, a sub shape of vertical length 1 is necessary, otherwise a sub shape of vertical length 0 suffices. In any case, there is a sub shape of vertical length 1 with run length $\geq a$. Formally: $\exists s_1' \subseteq s_1 : y_{max}(s_1') - y_{min}(s_1') = 1, x_{min}(s_1') = x_{min}(s_1), x_{max}(s_1') = x_{max}(s_1)$. For this sub shape we also have $s_1' \subsetneq s_1$ and $d_x(s_1', s_2) < d$. Thus $hor_{a,d}(s_1', s_2) = false$. Thus, $hor_{a,d}$ is consistent.

Let now $a > 1$. We show that $hor_{a,d}$ is not consistent. Let $s_1, s_2 \in \mathcal{S}$ two shapes with $run\_length_y(s_1, s_2) = a$, the same y-coordinates, zero length in x-direction and $d_x(s_1, s_2) < d$. Then we have $hor_{a,d}(s_1, s_2) = false$. Then for all $s_1' \subsetneq s_1$, s_1' also has zero length in x-direction but is shorter than s_1 in y-direction. Thus $run\_length_y(s_1', s_2) < run\_length_y(s_1, s_2) = a$. Thus $hor_{a,d}(s_1', s_2) = true$. Thus $hor_{a,d}$ is not consistent. □

We now need the notion of **color dependency**. In technologies where multiple colors are used on a layer, diff-net rules depend on the colors of the shapes. More precisely, there are different diff-net rules for shapes of different colors than for shapes of the same color.

Further there might be rules that apply independently of the colors of the shapes. We model this formally in the following way: Let

$$\mathcal{CD} := \{same, diff, arbitrary\}$$

Let $cd \in \mathcal{CD}, col_1, col_2 \in \mathcal{COL}$. We define

$$cd(col_1, col_2) := \begin{cases} true & col_1 = col_2, cd = same \\ false & col_1 \neq col_2, cd = same \\ true & col_1 \neq col_2, cd = diff \\ false & col_1 = col_2, cd = diff \\ true & cd = arbitrary \end{cases}$$

Remark. Technically, we could also make rules dependent on both colors directly instead of force that rules depend only on the fact if colors are equal or different. But we want to stress the inherent symmetry of production processes producing each color with the same technique and thus force symmetry in the colors.

For simpler notation we define the following function deciding if two shapes match certain specifications: Let $s_1, s_2 \in \mathcal{S}, l \in \mathcal{L}, sc_1, sc_2 \in \mathcal{SC}, cd \in \mathcal{CD}, sp_1, sp_2 \in \mathcal{SP}$. Then

$$\begin{aligned} applies(s_1, s_2, l, sc_1, sc_2, cd, sp_1, sp_2) := l(s_1) = l(s_2) = l \wedge \{sc(s_1), sc(s_2)\} = \{sc_1, sc_2\} \\ \wedge\, cd(col(s_1), col(s_2)) \wedge \{sp(s_1), sp(s_2)\} = \{sp_1, sp_2\} \end{aligned}$$

The following lemma shows that restricting any diff-net rule to a single combination of shape classes, layer, color dependency and shape purposes again gives a diff-net rule. This enables us to store and consider diff-net rules for each combination of shape class, layer, color dependency and purposes individually.

Lemma 5.1.14. *Let $dr \in \mathcal{DR}$ be a diff-net rule. Let $l \in \mathcal{L}, sc_1, sc_2 \in \mathcal{SC}, cd \in \mathcal{CD}$ and $sp_1, sp_2 \in \mathcal{SP}$. Let $s_1, s_2 \in \mathcal{S}$. Then the following also defines a diff-net rule:*

$$dr'(s_1, s_2) := \begin{cases} dr(s_1, s_2) & applies(s_1, s_2, l, sc_1, sc_2, cd, sp_1, sp_2) \\ true & else \end{cases}$$

Further, if dr is invariant under representation, so is dr'.

Proof. dr' is symmetric because dr and $applies$ are symmetric in s_1, s_2.

dr' is invariant under translation because dr is invariant under translation and $applies$ does not depend on $r(s_1)$ and $r(s_2)$.

Let b be the locality constant of dr restricted to shape classes $\{sc_1, sc_2\}$, layer l, color dependency cd and purposes $\{sp_1, sp_2\}$ (meaning that dr is only applied to shapes s_1, s_2 such that $\{sc(s_1), sc(s_2)\} = \{sc_1, sc_2\}$, $cd(col(s_1), col(s_2))$, $l(s_1) = l(s_2) = l$ and $\{sp(s_1), sp(s_2)\} = \{sp_1, sp_2\}$). Then dr' is local with locality constant b.

Thus dr' is a diff-net rule.

Let now dr be invariant under representation. Then dr is monotone and consistent and it suffices to show that dr' is also monotone and consistent. Let $s_1, s_1', s_2 \in \mathcal{S}, s_1' \subseteq$

$s_1, dr'(s_1, s_2) = true$. Because $s_1' \subseteq s_1$, we have $applies(s_1, s_2, l, sc_1, sc_2, cd, sp_1, sp_2) \Leftrightarrow applies(s_1', s_2, l, sc_1, sc_2, cd, sp_1, sp_2)$. If $applies(s_1', s_2, l, sc_1, sc_2, cd, sp_1, sp_2) = false$, we have $dr'(s_1', s_2) = true$. Otherwise, we have $applies(s_1, s_2, l, sc_1, sc_2, cd, sp_1, sp_2) = true$ and because $dr'(s_1, s_2) = true$ also $dr(s_1, s_2) = true$. Because dr is monotone we get $dr(s_1', s_2) = true$ and thus $dr'(s_1', s_2) = true$. Thus dr' is monotone.

Let now $s_1, s_2 \in \mathcal{S}, diam(s_1) > 1, dr'(s_1, s_2) = false$. Because $dr'(s_1, s_2) = false$, we have $applies(s_1, s_2, l, sc_1, sc_2, cd, sp_1, sp_2) = true$ and $dr(s_1, s_2) = false$. Because dr is consistent, $\exists s_1' \in \mathcal{S}, s_1' \subsetneq s_1, dr(s_1', s_2) = false$. Because by definition we also have $applies(s_1', s_2, l, sc_1, sc_2, cd, sp_1, sp_2) = true$ we get $dr'(s_1', s_2) = false$ and dr' is also consistent. □

The following lemma shows that we can take the Minkowski sum with an arbitrary rectangle (depending on the shape class, layer and color) before applying a diff-net rule and still obtain a diff-net rule. This includes moving shapes by a constant offset and extending shapes by a constant in each direction and can be used for modeling a number of more complicated restrictions in practice.

Lemma 5.1.15. *Let $dr \in \mathcal{DR}$ be a diff-net rule. Let $r : \mathcal{L} \times \mathcal{SC} \to \mathcal{R}$. Let $s_1, s_2 \in \mathcal{S}$ Then the following also defines a diff-net rule:*

$$dr'(s_1, s_2) := dr(s_1 + r(l(s_1), sc(s_1)), s_2 + r(l(s_2), sc(s_2)))$$

Further, if dr is invariant under representation, so is dr'.

Proof. dr' is symmetric because dr is.

Let $s_1 \in \mathcal{S}, s_2 \in \mathcal{S}, (x, y) \in \mathbb{Z}^2$. Then we have $dr'(s_1 + (x, y), s_2 + (x, y)) = dr(s_1 + (x, y) + r(l(s_1), sc(s_1)), s_2 + (x, y) + r(l(s_2), sc(s_2))) = dr((s_1 + r(l(s_1), sc(s_1))) + (x, y), (s_2 + r(l(s_2), sc(s_2))) + (x, y)) = dr(s_1 + r(l(s_1), sc(s_1)), s_2 + r(l(s_2), sc(s_2))) = dr'(s_1, s_2)$. Thus dr' is invariant under translation.

Let $m := \max_{l \in \mathcal{L}, sc \in \mathcal{SC}} \{x_{max}(r(l, sc)), y_{max}(r(l, sc)), -x_{min}(r(l, sc)), -y_{min}(r(l, sc))\}$. Then let the locality constant of dr be b. Let $s_1, s_2 \in \mathcal{S}$ with $d_{max}(s_1, s_2) > b + 2m$. Then we have $d_{max}(s_1 + r(l(s_1), sc(s_1)), s_2 + r(l(s_2), sc(s_2))) > b + 2m - 2m = b$. Thus we get $dr(s_1 + r(l(s_1), sc(s_1)), s_2 + r(l(s_2), sc(s_2))) = true$ and therefore $dr'(s_1, s_2) = true$. Thus dr' is also local with locality constant at most $b + 2m$.

Let now dr be invariant under representation. Let $S_1, S_1', S_2 \subseteq \mathcal{S}, \bigcup_{s \in S_1} r(s) = \bigcup_{s \in S_1'} r(s), S_1 \cup S_1'$ homogeneous . Define $T_2 := \{s_2 + r(l(s_2), sc(s_2)) : s_2 \in S_2\}$. Let $r := r(l(S_1), sc(S_1)) = r(l(S_1'), sc(S_1'))$. This is well defined because $S_1 \cup S_1'$ is homogeneous. Let $T_1 := S_1 + r, T_1' := S_1' + r$. We have $\bigcup_{t \in T_1} r(t) = \bigcup_{t \in T_1'} r(t)$ because the same holds for S_1, S_1' and the union swaps with the Minkowski sum. Thus we have $dr'(S_1, S_2) = \bigwedge_{s_1 \in S_1, s_2 \in S_2} dr(s_1 + r(l(s_1), sc(s_1)), s_2 + r(l(s_2), sc(s_2))) = \bigwedge_{s_1 \in S_1} dr(s_1 + r, T_2) = dr(T_1, T_2) = dr(T_1', T_2) = dr'(S_1', S_2)$. Thus dr' is invariant under representation. □

Remark. Usually, the above lemma is used for $r : \mathcal{L} \times \mathcal{SC} \to \mathcal{R}_0$ such that the original shape is a subshape of the checked shape.

Remark. Shrinking shapes by some distance in any direction and then applying an invariant under representation diff-net rule usually does not give a diff-net rule that is invariant under representation.

Lemma 5.1.16. *Let dr_1 and dr_2 be two diff-net rules, $s_1, s_2 \in \mathcal{S}$. Then $dr(s_1, s_2) := dr_1(s_1, s_2) \wedge dr_2(s_1, s_2)$ defines a diff-net rule. Further, if dr_1 and dr_2 are invariant under representation, so is dr.*

Proof. dr is symmetric because dr_1 and dr_2 are.

dr is invariant under translation, because dr_1 and dr_2 are.

Let b_1 and b_2 be the locality constants of dr_1 and dr_2. Then dr is local with locality constant $\max\{b_1, b_2\}$.

Let now dr_1 and dr_2 be invariant under representation. It suffices to show that dr is monotone and consistent. Let $s_1, s_1', s_2 \in \mathcal{S}, s_1' \subseteq s_1, dr(s_1, s_2) = \mathit{true}$. Then we have $dr_1(s_1, s_2) = \mathit{true}$ and $dr_2(s_1, s_2) = \mathit{true}$. Thus we get $dr_1(s_1', s_2) = \mathit{true}$ and $dr_2(s_1', s_2) = \mathit{true}$ and therefore $dr(s_1', s_2) = \mathit{true}$. dr is monotone.

Let now $s_1, s_2 \in \mathcal{S}, diam(s_1) > 1, dr(s_1, s_2) = \mathit{false}$. We have $dr_1(s_1, s_2) = \mathit{false}$ or $dr_2(s_1, s_2) = \mathit{false}$. W.l.o.g we can assume $dr_1(s_1, s_2) = \mathit{false}$. Then $\exists s_1' \in \mathcal{S}, s_1' \subsetneq s_1, dr_1(s_1', s_2) = \mathit{false}$. But then we also have $dr(s_1', s_2) = \mathit{false}$ and dr is consistent. □

One more commonly used type of diff-net rules are **center to center rules**.

Definition 5.1.17. *Let $p \in \mathbb{R} \cup \{\infty\}, 1 \leq p \leq \infty, d \in \mathbb{N}, s_1, s_2 \in \mathcal{S}$ and define:*

$$c2c_d^p(s_1, s_2) := \begin{cases} \mathit{false} & d_p(center(s_1), center(s_2)) < d \\ \mathit{true} & \mathit{else} \end{cases}$$

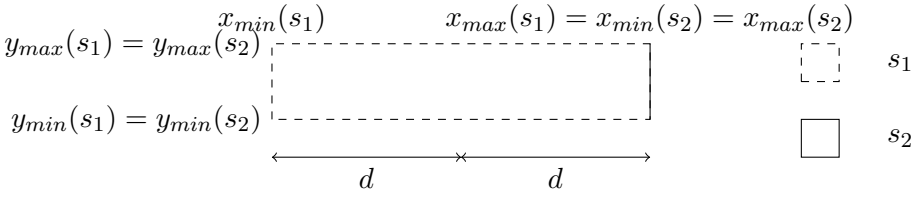
These rules are usually used only on via layers for a very good reason, as the following lemma shows.

Lemma 5.1.18. *For all $p \in \mathbb{R} \cup \{\infty\}, 1 \leq p \leq \infty, d \in \mathbb{N}, d > 0$, $c2c_d^p$ is a diff-net rule. $c2c_d^p$ is not monotone for any d.*

Proof. $c2c_d^p$ is symmetric and invariant under translation.

It is also local, because $d_p(center(s_1), center(s_2)) \geq d_{max}(s_1, s_2)$.

$c2c_d^p$ is not monotone for any $d > 0$: Let $s_1, s_2 \in \mathcal{S}$ be two shapes with $y_{min}(s_1) = y_{min}(s_2)$, $y_{max}(s_1) = y_{max}(s_2)$, $x_{max}(s_1) = x_{min}(s_2) = x_{max}(s_2)$, $x_{min}(s_1) + 2d = x_{max}(s_1)$ and $attr(s_1) = attr(s_2)$. Then we have $s_2 \subseteq s_1$ and $c2c_d^p(s_1, s_2) = \mathit{true}$ because $d_p(center(s_1), center(s_2)) = d$. But we have $d_p(center(s_2), center(s_2)) = 0 < d$ and thus $c2c_d^p(s_2, s_2) = \mathit{false}$. Therefore, $c2c_d^p$ is not monotone.



□

Above, we have seen that the conjunction of two invariant under representation diff-net rules is again a diff-net rule that is invariant under representation. The following lemma shows that the same is not true for the disjunction.

Lemma 5.1.19. *Let dr_1 and dr_2 be two diff-net rules, $s_1, s_2 \in \mathcal{S}$. Then $dr(s_1, s_2) := dr_1(s_1, s_2) \vee dr_2(s_1, s_2)$ defines a diff-net rule. Further, if dr_1 and dr_2 are monotone, so is dr. dr is not necessarily consistent even if dr_1 and dr_2 are.*

Proof. dr is symmetric and invariant under translation because dr_1 and dr_2 are. Let b_1 and b_2 be the locality constants of dr_1 and dr_2. Then dr is local with locality constant $b \leq \min\{b_1, b_2\}$. Thus dr is a diff-net rule.

Let dr_1 and dr_2 be monotone. Let $s_1, s_1', s_2 \in \mathcal{S}, s_1' \subseteq s_1, dr(s_1, s_2) = \mathit{true}$. W.l.o.g we have $dr_1(s_1, s_2) = \mathit{true}$. Because dr_1 is monotone, we have $dr_1(s_1', s_2) = \mathit{true}$ and thus by definition $dr(s_1', s_2) = \mathit{true}$. Thus dr is monotone.

Let $sc_1, sc_2 \in \mathcal{SC}$, $r : \mathcal{L} \times \mathcal{SC} \to \mathcal{R}$, $r(l, sc) := \begin{cases} [0,0] \times [0,0] & sc = sc_1 \\ [0,0] \times [5,5] & sc = sc_2 \end{cases}$. Consider the following diff-net rules: $dr_1(s_1, s_2) := dist_2^2(s_1 + r(l(s_1), sc(s_1)), s_2 + r(l(s_2), sc(s_2)))$, $dr_2 := dist_2^2$. dr_1 and dr_2 are both consistent. Let $l \in \mathcal{L}, c \in \mathcal{COL}, sp \in \mathcal{SP}$. Define $s_2 = ([0,0] \times [0,1], l, sc_2, c, sp) \in \mathcal{S}$, $s_1 = ([0,0] \times [2,4], l, sc_1, c, sp) \in \mathcal{S}$. We have $diam(s_1) = 2 > 1$ and $dr(s_1, s_2) = dr_1(s_1, s_2) \vee dr_2(s_1, s_2) = \mathit{false}$. But all sub shapes of s_1 which are not s_1 itself are: $([0,0] \times [2,2], l, sc_1, c, sp)$, $([0,0] \times [3,3], l, sc_1, c, sp)$, $([0,0] \times [4,4], l, sc_1, c, sp)$, $([0,0] \times [2,3], l, sc_1, c, sp)$, $([0,0] \times [3,4], l, sc_1, c, sp)$. Thus we have $\forall s_1' \subsetneq s_1 : dr(s_1', s_2) = \mathit{true}$. dr is not consistent. □

Summarizing we now have a powerful framework to specify diff-net rules. We can specify multiple rules for each combination of shape classes, layer and for same or different colors individually. We can use horizontal, vertical and Euclidean rules as well as add arbitrary extensions to any rule. This covers many rules used in practice on real designs. Some other commonly used rules can be modeled efficiently by these rules without making relevant errors in practice (see [40]).

In the next section we will see how we can use this framework to implement checking of diff-net rules very efficiently based on the assumptions we made here. We see how these assumptions play together with a number of optimizations commonly used during detailed routing in particular and VLSI-design in general and why they are necessary for computing shortest paths to route nets during detailed routing in Section 5.2.

5.2 Simple Diff-Net Rule Checking in BonnRouteDetailed

In this section we describe how diff-net rules are handled in BonnRouteDetailed. First we describe a simple way to handle diff-net rules suitable for not run time critical uses. In the next chapter we develop an optimized interface suitable for extensive use, e.g. during path search.

BonnRouteDetailed uses an optimized data structure called the grid to store shapes and sticks. We denote by $\mathcal{SG} \subseteq \mathcal{S}$ the set of shapes that are currently stored in the grid. For further details on this data structure we refer to [40]. BonnRouteDetailed furthermore uses a very efficient parallelization framework which was first described in [27]. This requires for each thread $t \in \mathcal{T}$ two **sets of temporary shapes**, one set of shapes that are currently not considered for checking diff-net rules (e.g. because they are pins of the same net that is currently routed and should not be checked against or

because they have been ripped-up but not submitted yet) denoted by $\mathcal{IS}_t \subseteq \mathcal{S}_w \cup \mathcal{S}_p$ and one set of shapes that currently have to be considered additionally for diff-net rule checking (mainly consisting of new routes for different nets that have been found during a rip-up sequence but not submitted yet) denoted by $\mathcal{AS}_t \subseteq \mathcal{S}_w$. Let $l \in \mathcal{L}$. We define $\mathcal{SCIS}_t^l := \{sc(s) : s \in \mathcal{IS}_t, l(s) = l\}$ and $\mathcal{SCAS}_t^l := \{sc(s) : s \in \mathcal{AS}_t, l(s) = l\}$. We use these sets of shape classes to efficiently check diff-net rules in Section 5.3. We can easily store these sets and update them when $\mathcal{IS}_t$ or $\mathcal{AS}_t$ are changed.

The by far most common use case of diff-net rule checking during detailed routing is checking if a possible new shape meets all diff-net rules with respect to the already existent shapes. We now define **legality of a shape** $s \in \mathcal{S}$ **with respect to thread** $t \in \mathcal{T}$ (with respect to $\mathcal{SG}$, $\mathcal{IS}_t$ and $\mathcal{AS}_t$ and diff-net rules $DR \subseteq \mathcal{DR}$):

Definition 5.2.1.

$$s \ legal_t := \forall dr \in DR, s' \in (\mathcal{SG} \setminus \mathcal{IS}_t) \cup \mathcal{AS}_t : dr(s, s') = true$$

This means that a shape is legal with respect to thread t if it is compatible with each other shape currently on the chip (with consideration of $\mathcal{IS}_t$ and $\mathcal{AS}_t$). We both use the notation legal_t and equivalently write legal with respect to thread t. We define **legality of a shape** (without respect to any thread):

Definition 5.2.2.

$$s \ legal := \forall dr \in DR, s' \in \mathcal{SG} : dr(s, s') = true$$

We use this thread-independent notation in Section 5.3.

Likewise, we define a **stick** $st \in \mathcal{ST}$ to be **legal (with respect to thread** t**)**, if each of its shapes is legal:

Definition 5.2.3.

$$st \ legal_t := \forall s \in shapes(st) : s \ legal_t$$

$$st \ legal := \forall s \in shapes(st) : s \ legal$$

Sometimes, a given net can not be routed without changing already routed nets. In these cases, diff-net checking can ignore modifiable shapes of other nets. Still, diff-net rules need to be checked correctly for all shapes that can not be modified.

Definition 5.2.4.

$$s \ legal_t \ with \ rip\text{-}up := \forall dr \in DR, s' \in (\mathcal{SG} \setminus \mathcal{IS}_t) \cup \mathcal{AS}_t, sp(s') \neq wire : dr(s, s') = true$$

$$s \ legal \ with \ rip\text{-}up := \forall dr \in DR, s' \in \mathcal{SG}, sp(s') \neq wire : dr(s, s') = true$$

Definition 5.2.5.

$$st \text{ } legal_t \text{ } with \text{ } rip\text{-}up := \forall s \in shapes(st) : s \text{ } legal_t \text{ } with \text{ } rip\text{-}up$$

$$st \text{ } legal \text{ } with \text{ } rip\text{-}up := \forall s \in shapes(st) : s \text{ } legal \text{ } with \text{ } rip\text{-}up$$

Thus, a **shape is legal with rip-up** if it can be made legal by removing any number of wire shapes from the chip.

Remark. In practice, some wires may not be modified during detailed routing. Definitions 5.2.4 and 5.2.5 (and the following algorithms) can easily be adapted to handle these cases. We omit such details to simplify our notation.

By modeling diff-net rules according to Definition 5.1.3 we have made some assumptions about the structure of practical diff-net rules. We have assumed that diff-net rules are always symmetric. This is very natural. The order in which two shapes are given to the rule does not influence legality. We have further assumed that diff-net rules are invariant under translation. This is also a natural requirement coming from practical use cases. For example when a chip is designed hierarchically, legality of a configuration of shapes should not depend on the local coordinate system. Legality of shapes within a macro with regard to each other should not depend on the placement of the macro. Third we have assumed that each diff-net rule is local (and implicitly assume that the locality constants are small). This is necessary to make efficient detailed routing possible. If a shape can influence other shapes far away, checking diff-net rules becomes too expensive and parallelization becomes impossible. Thus these basic restrictions seem to be well chosen and for the same reasons they are widely fulfilled in practice. Unfortunately we need some further restrictions to make efficient detailed routing possible.

While constructing new wiring for a net, paths are searched on a certain graph constructed by checking legality of wire sticks for each edge individually (and this is necessary to make efficient path search possible). The path has to be legal if and only if each edge is. In different words, legality of the path should only depend on the metal it consists of, not on the representation by individual wire sticks (but still it needs to be checked by checking its individual shapes). For example if a wire stick is split into two substicks, the whole stick should be legal if and only if both substicks are. Illustrations of this requirement can be found in Figure 5.3 and 5.4. There is one exception, via middle shapes are naturally represented in only one way within a path. Furthermore, vias of the same net usually also have to fulfill diff-net rules, so different via shapes can never merge, they always stay separate, single, rectangular shapes.

These requirements are fulfilled if and only if all diff-net rules (apart from those that are only used for via middle shapes) are invariant under representation.

Requiring diff-net rules (apart from rules for via middle shapes) to be invariant under representation has further benefits. Blockages, pins or any metal (apart from via middle shapes which look very simple anyway, they are just one rectangle) can safely be represented in any way by shapes, as long as the total area (of parts with a given color, shape class and purpose) stays the same. For example, one can represent them in an overlap

Figure 5.3: Wire sticks might be split when they are accessed by another stick.

Figure 5.4: The grid used for path search influences checked shapes.

free form or maximize representation in a specific direction to store them more efficiently (see for example [40], [18], [34] and [43]).

Therefore, BonnRouteDetailed can in principal support all diff-net rules that are invariant under representation (as defined in Definition 5.1.3 and 5.1.4) and that can be efficiently stored and evaluated for two given shapes as well as arbitrary diff-net rules that can be efficiently stored and evaluated for via middle shapes. In practice, a small number of different types of such rules occur and are currently supported, mainly Euclidean distance rules with an optional extension for both shapes and horizontal and vertical rules with an optional extension. Furthermore, for via layers rules between the centers of two vias are used. Supporting a new type of rule is very easy, as long as it meets the aforementioned requirements.

Formally, let DR be the set of all diff-net rules on a chip. All these diff-net rules should be given in a way that they can be evaluated in constant time for any two given shapes. Furthermore, all diff-net rules that are not only used for via middle shapes need to be invariant under representation. The set DR of relevant diff-net rules is given as input to BonnRouteDetailed.

Remark. Note that for the rest of this chapter we assume for simplicity of notation that there are no restrictions between shapes on different layers (such as for example inter-layer via rules). If there are such restrictions, these can be added in a straightforward way.

In order to access relevant diff-net rules efficiently, one defines and stores diff-net rules per layer and per combination of shape classes separately. This is possible because of Lemma 5.1.14 and Lemma 5.1.16. Therefore, when checking diff-net rules for two shapes, one needs to check only diff-net rules that apply for the combination of shape classes and layer. In the following we will omit this detail but of course in practice it is important to

check only relevant diff-net rules to achieve a good run time.

We will now describe a first simple method to check diff-net rules. The general procedure to check a shape (or stick) for legality with respect to a thread t is very easy: For a given shape s, calculate the area that can influence legality of this shape (this is always bounded because all diff-net rules are by definition local). Then collect all shapes within this area from the grid. Finally, for each of these collected shapes (say s') and for each relevant diff-net rule, check if s and s' are legal_t with respect to each other. A formal description of this procedure can be found in Algorithm 5.2.1.

Algorithm 5.2.1: ShapeLegal

Input: $s \in \mathcal{S}, t \in \mathcal{T}, DR \subseteq \mathcal{DR}, ripup\_allowed \in \mathcal{B}$

```
1 b := b_{sc(s),l(s)}(DR)
2 r := r(s) + [−b, b] × [−b, b]
3 S := {s' ∈ (SG \ IS_t) ∪ AS_t : r(s') ∩ r ≠ ∅, l(s') = l(s)}
4 foreach s' ∈ S : not ripup_allowed or sp(s') ≠ wire do
5     foreach dr ∈ DR do
6         if dr(s, s') = false then
7             return false
8 return true
```

Remark. In practice, the values $b_{sc(s),l(s)}(DR)$ can be precomputed for all possible layers and shape classes such that line 1 only takes constant time. Further, as stated above, we do not need to loop over all diff-net rules in DR, considering those relevant for the given shapes (determined by their shape classes and layer) is sufficient.

To check a stick for legality with respect to a thread t, one simply checks each shape of the stick separately as shown in Algorithm 5.2.2.

Algorithm 5.2.2: StickLegal

Input: $stick \in \mathcal{ST}, t \in \mathcal{T}$, $DR \subseteq \mathcal{DR}, ripup\_allowed \in \mathcal{B}$

```
1 foreach s ∈ shapes(stick) do
2     if not ShapeLegal(s, t, DR, ripup_allowed) then
3         return false
4 return true
```

Remark. In practice, one should also check if a stick is not too close to the chip border, lies on an allowed track and runs in an allowed direction. These checks are rather trivial, so we omit them for simplicity.

Remark. In case of rip-up it might be useful to know the set of shapes that need to be ripped-up. For example shapes of more critical nets might be more expensive to rip-up. It is straightforward to modify Algorithm 5.2.1 and 5.2.2 such that they also output the set of shapes that need to be ripped-up to make a stick or shape legal.

Before we prove correctness of Algorithm 5.2.2 and Algorithm 5.2.1, we formulate and prove a simple lemma that we will use in the proof of the following theorem and in the next section.

Lemma 5.2.6. *Let $s \in \mathcal{S}$, $DR \subseteq \mathcal{DR}$.*

(a) Let $b := b_{sc(s),l(s)}(DR)$ and $r := r(s) + [-b, b] \times [-b, b]$. Let $s_o \in \mathcal{S}$ with $l(s_o) = l(s)$ and $r(s_o) \cap r = \emptyset$. Then for all $dr \in DR$ we have $dr(s, s_o) = true$.

(b) Let further $sc \in \mathcal{SC}$. Let further $b_{sc} := b_{sc(s),sc,l(s)}(DR)$ and $r_{sc} := r(s) + [-b_{sc}, b_{sc}] \times [-b_{sc}, b_{sc}]$. Let further $s'_o \in \mathcal{S}$ with $l(s'_o) = l(s)$ and $sc(s'_o) = sc$ and $r(s'_o) \cap r_{sc} = \emptyset$. Then for all $dr \in DR$ we have $dr(s, s'_o) = true$.

In words, all shapes outside r are legal with respect to s and all shapes outside r_{sc} with shape class sc are legal with respect to s.

Proof. Let s, DR, sc, b, b_{sc}, r, r_{sc}, s_o and s'_o as in the lemma. We first note, that because we have $r(s_o) \cap r = \emptyset$, it follows that we have $d_{max}(s, s_o) > b$. Analogously, we also get $d_{max}(s, s'_o) > b_{sc}$. The statement of the lemma follows directly from the definition of $b_{sc(s),l(s)}(DR)$ and $b_{sc(s),sc,l(s)}(DR)$. □

Now we prove the correctness of the algorithms.

Theorem 5.2.7. *Algorithm 5.2.1 and Algorithm 5.2.2 correctly check if a stick or shape is legal with respect to thread t (according to Definition 5.2.3 and 5.2.1) if ripup_allowed is false and correctly check if a stick or shape is legal with respect to thread t with rip-up (according to Definition 5.2.5 and 5.2.4) if ripup_allowed is true.*

Proof. First, consider the case *ripup_allowed* = *false*. Due to the assumption that there are no restrictions between different layers, we need to check against shapes on the same layer only. Let $s \in \mathcal{S}, t \in \mathcal{T}, DR \subseteq \mathcal{DR}$ be the input of Algorithm 5.2.1 and let $r \in \mathcal{R}$ as calculated in line 2 of Algorithm 5.2.1. Algorithm 5.2.1 checks correctly against all shapes s' in $(\mathcal{SG} \setminus \mathcal{IS}_t) \cup \mathcal{AS}_t$ such that $r(s') \cap r \neq \emptyset$ with respect to all diff-net rules $dr \in DR$. It remains to show that all other shapes do not need to be checked. Consider input shape s and a shape $s' \in (\mathcal{SG} \setminus \mathcal{IS}_t) \cup \mathcal{AS}_t$ with $r(s') \cap r = \emptyset$. We can further assume $l(s) = l(s')$ because of the assumption that there are no restrictions between shapes on different layers. Let b as defined in line 1 of Algorithm 5.2.1. Lemma 5.2.6 (a) with $s_o := s'$ gives directly $\forall dr \in DR : dr(s, s') = true$. Thus we do not need to check against s' and Algorithm 5.2.1 works as desired.

Correctness of Algorithm 5.2.1 in the case *ripup_allowed* = *true* follows from the fact that the only difference to the case *ripup_allowed* = *false* is that all shapes with shape purpose *wire* are ignored. Precisely these are also excluded in Definition 5.2.4.

Algorithm 5.2.2 correctly checks if each shape of the stick is legal. □

In this section, we have seen what kind of diff-net rules can be supported by BonnRouteDetailed directly and why and how shapes and sticks can be checked for legality in a straightforward way.

5.3 Optimized Diff-Net Rule Checking in BonnRouteDetailed

In this section, we present a more complex but highly optimized approach to check diff-net rules used for run time critical code, such as the main path search of BonnRouteDetailed.

During path search, the vast majority of calls of the checking module are asking for information about legality (with respect to some thread t) of wire sticks corresponding to an edge of the track graph with a small number of different wire / via models colored by the track color. Therefore, it is very beneficial if these frequent calls can be processed very fast. Thus, BonnRouteDetailed uses an optimized data structure, called the fast grid, to precompute and store information for these queries efficiently.

For a former similar approach, see [34], and [27] for aspects concerning its parallelization. Our approach has a number of advantages over the version described in [34]. The most important difference is that the approach in [34] makes the assumption that wires in non-preferred direction are legal if both of their endpoints are. This was true in former technologies but is more and more frequently violated in recent designs. In particular, the optimized track patterns we describe in Chapter 4 always violate this assumption because they do not contain tracks intersecting the power rails. Wires in non-preferred direction crossing the power rails are never legal (apart from nets connecting to power of course), but the two point-wise wires on their endpoints often are. Further, if optimized track patterns contain fewer tracks than potentially possible between two power rails, the assumption of [34] is also frequently violated. Our approach does not need such an assumption at all. Another difference is that we implement our approach based on the grid data structure described in Section 3.4, whereas [34] uses a fundamentally different approach to store metal shapes. Unlike [34], we discuss the handling of colored wires and aspects of efficient parallelization, incorporating our fast grid data structure in the parallelization framework first described in [27]. To this end, we store more detailed information in the fast grid (four states instead of three) and choose a different implementation (based on arrays of arrays instead of binary search trees).

The basic concept works as follows: For each edge of the track graph, we precompute and store information with which wire or via models the edge can be used thread-independently (that means not taking into account the shapes in $\mathcal{AS}_t$ and $\mathcal{IS}_t$).

We will now describe this data structure in detail and then design an algorithm for computing thread-dependent legality from this precomputed thread-independent information very fast. First, we formalize the track graph and related notation.

Definition 5.3.1. *Let TR_l ($l \in \mathcal{L}_{wiring}$) be the **tracks** of a chip. For simplicity of notation, we define: $TR_{-2} := \emptyset$ and $TR_{l_{max}+2} := \emptyset$. Then we define the **track graph** $G = (V, E)$ as follows:*

$$V := \bigcup_{l \in \mathcal{L}_{wiring}} V_l$$

$$E := \bigcup_{l \in \mathcal{L}_{via}} E_l^v \cup \bigcup_{l \in \mathcal{L}_{wiring}} E_l^x \cup E_l^y$$

where for $l \in \mathcal{L}_{wiring}$ define:

$$V_l := \begin{cases} (x, y, l) : y \in TR_l, x \in TR_{l-2} \cup TR_{l+2} & pref(l) = horizontal \\ (x, y, l) : x \in TR_l, y \in TR_{l-2} \cup TR_{l+2} & pref(l) = vertical \end{cases}$$

and:

$$E_l^y := \{\{(x, y_{min}, l), (x, y_{max}, l)\} \in \mathfrak{P}_2(V) : y_{min} < y_{max}, \nexists y' : y_{min} < y' < y_{max}, (x, y', l) \in V\}$$

$$E_l^x := \{\{(x_{min}, y, l), (x_{max}, y, l)\} \in \mathfrak{P}_2(V) : x_{min} < x_{max}, \nexists x' : x_{min} < x' < x_{max}, (x', y, l) \in V\}$$

For $l \in \mathcal{L}_{via}$ define:

$$E_l^v := \{\{(x, y, l-1), (x, y, l+1)\} : (x, y, l-1), (x, y, l+1) \in V\}$$

For simplicity of notation, define:

$$E^v := \bigcup_{l \in \mathcal{L}_{via}} E_l^v$$

$$E^w := \bigcup_{l \in \mathcal{L}_{wiring}} E_l^x \cup E_l^y$$

Let $e \in E$, $e = \{(x_{min}, y_{min}, l_{min}), (x_{max}, y_{max}, l_{max})\}, x_{min} \leq x_{max}, y_{min} \leq y_{max}, l_{min} \leq l_{max}$. For $e \in E^w$ we have $l_{min} = l_{max}$ and can thus define $l(e) := l_{min}$. For $e \in E^v$ we have $l_{min} + 2 = l_{max}$, $x_{min} = x_{max}$ and $y_{min} = y_{max}$ and can thus define $x(e) := x_{min}$, $y(e) := y_{min}$ and $l(e) := l_{min} + 1$. For $e \in E^w$, define further $r(e) \in \mathcal{R}_{stick}$ by $r(e) := [x_{min}, x_{max}] \times [y_{min}, y_{max}]$.

The vertices of this track graph are the points where a track on some wiring layer intersects a track on a neighboring wiring layer. The set of edges consists of three types of edges: via edges, horizontal and vertical edges. Via edges connect vertices on neighboring wiring layers (this means, we have a via edge wherever tracks on neighboring layers intersect). Vertical and horizontal edges connect vertices on the same layer that differ only in either x- or y-coordinate and do not have any vertex in between. Vertical and horizontal edges are called wire edges. An illustration of the track graph can be found in Figure 5.5.

On colored layers, most wires are colored in a simple way. For each wire and via model and each track, a preferred color is defined. Then almost all wires and vias are colored by this preferred color. Only if local coloring problems arise (for example due to input metal not following this coloring scheme), a small number of wires and vias are colored differently. Therefore, the vast majority of legality queries during the path search uses the preferred colors. In the fast grid, we store information only for the preferred colors to save memory. The few queries that use non-preferred colors can be answered by calculating legality from the grid.

To formalize this approach, we define for each edge and wire or via model the colors used by the fast grid. In practice these will not depend on the individual edge but only on the model and the track on which the edge is located on.

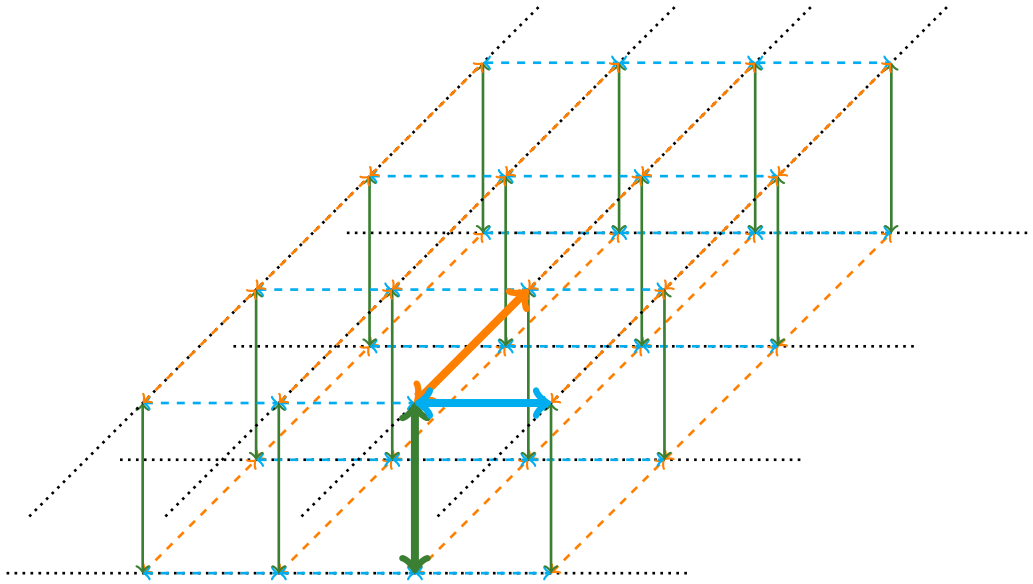


Figure 5.5: Track graph for two wiring and one via layers. Tracks are marked by black dotted lines, via edges and edges in x- and y-direction are drawn as green, cyan and orange dashed arrows respectively. For each type, one exemplary edge is highlighted.

Definition 5.3.2. *Let $e_w \in E^w$ and $wm \in \mathcal{WM}$. Let $fgc(e_w, wm) \in \mathcal{COL}$ be the **color used by the fast grid** for the edge e_w with wire model wm.*

*Let further $e_v \in E^v$ and $vm \in \mathcal{VM}$. Let $fgc_b(e_v, vm)$, $fgc_m(e_v, vm)$ and $fgc_t(e_v, vm) \in \mathcal{COL}$ the **colors used by the fast grid** for the edge e_v with via model vm for the bottom, middle and top shape.*

For the fast grid, we define the **wire stick corresponding to an edge of the track graph** in the following way:

Definition 5.3.3. *Let $e_v \in E^v$ and $vm \in \mathcal{VM}$. Define $stick(e_v, vm) \in \mathcal{VS}$:*

$$stick(e_v, vm) := (x(e_v), y(e_v), l(e_v), vm, fgc_b(e_v, vm), fgc_m(e_v, vm), fgc_t(e_v, vm))$$

Let further $e_w \in E^w$ and $wm \in \mathcal{WM}$. Define $stick(e_w, wm) \in \mathcal{WS}$:

$$stick(e_w, wm) := (r(e_w), l(e_w), wm, fgc(e_w, wm))$$

Now, we continue to describe how we optimize diff-net rule checking in BonnRouteDetailed. In order to use precomputed information in the most efficient way, we do not only precompute if a stick is legal (with respect to the current state of the chip and independent of the threads) but also if this can possibly change in the future. There are four alternatives. They correspond to three different types of shapes that can make a given stick illegal (and the fact that it can also be legal of course). Some shapes can be removed

from the chip (such as most shapes corresponding to wires). Some shapes can not be removed, but they are not relevant in some cases (such as pin shapes, they are not relevant when checking sticks of the same net that access them), we say they can be ignored. Some shapes are always relevant and can not be modified (such as most blockages).

Definition 5.3.4. *Define the* ***set of possible extended legality states****:*

$$\mathcal{ELS} := \{legal, illegal\_by\_removable, illegal\_by\_ignorable, always\_illegal\}$$

Legal means that something is currently legal. This can of course change when new sticks are added to the chip. The second state, *illegal_by_removable* means that something is currently illegal, but every object that makes it illegal could be removed from the chip. The third state *illegal_by_ignorable* means that something is illegal, everything that makes it illegal can possibly be ignored (or removed from the chip) but not everything that makes it illegal can be removed from the chip. The last state *always_illegal* means that something is always illegal and that can never change. To simplify notation, define a total order on $\mathcal{ELS}$ by the following relations: $legal < illegal\_by\_removable < illegal\_by\_ignorable < always\_illegal$. Furthermore, we define a function mapping a shape purpose to the appropriate extended legality state:

Definition 5.3.5. *Let* $msp : \mathcal{SP} \to \mathcal{ELS}$ *be defined by*

$$msp(pin) := illegal\_by\_ignorable$$

$$msp(wire) := illegal\_by\_removable$$

$$msp(blockage) := always\_illegal$$

The intention of this definition is that the extended legality state of a shape is the maximum over all shapes s' that make it illegal of $msp(sp(s'))$.

Now we can define the **extended legality state** of a shape:

Definition 5.3.6. *Let* $DR \subseteq \mathcal{DR}$ *be all diff-net rules of a chip. Define* $els : \mathcal{S} \to \mathcal{ELS}$ *by*

$$els(s) := \begin{cases} legal & s \text{ legal} \\ max_{s' \in \mathcal{SG}: \exists dr \in DR: dr(s,s') = false} msp(sp(s')) & s \text{ not legal} \end{cases}$$

Remark. In practice, there are other reasons why a shape may be always illegal. For example it is too close to the chip border, lies on a forbidden track or goes in the wrong direction. For these cases, the shape can easily be marked as always forbidden. We omit details here to simplify the notation.

Remark. In practice, sometimes wires are fixed on a chip, meaning that they may not be modified. These can be treated like pins but for simplicity of notation we do assume here that all wires may be modified.

We extend the definition of the extended legality state to sticks in the obvious way:

Definition 5.3.7. *Let* $DR \subseteq \mathcal{DR}$ *be all diff-net rules of a chip. Define* $els : \mathcal{ST} \to \mathcal{ELS}$ *by*

$$els(st) := \max\{s \in shapes(st) : els(s)\}$$

On many chips, only very few wire and via models are used for most of the wiring while the other wire and via models are used for very little wiring. We select a small set of the most frequently used wire and via models to store precomputed information for in order to reduce memory usage and optimize performance. If wire or via models are used very often, the reduction in checking run time highly overcompensates the required additional run time to compute and update the data. For rarely used wire and via models, the overhead to store precomputed information is larger than the benefit in checking run time. Thus we define on each wiring layer a set of wire models and on each via layer a set of via models to store precomputed information for.

Definition 5.3.8. *Let* $fgwm : \mathcal{L}_{wiring} \to \mathfrak{P}(\mathcal{WM})$ *such that for* $l \in \mathcal{L}_{wiring}$, $fgwm(l)$ *are the* ***wire models on layer*** l ***for which fast grid information is stored****. Let* $fgvm : \mathcal{L}_{via} \to \mathfrak{P}(\mathcal{VM})$ *such that for* $l \in \mathcal{L}_{via}$, $fgvm(l)$ *are the* ***via models on layer*** l ***for which fast grid information is stored****.*

Remark. In practice, the set of wire models used in preferred direction and in the other direction differs on some layers. We omit this distinction to simplify the notation. It can be incorporated into the algorithms in a straight-forward way.

Now we can define the information that we store in the fast grid (we call it the **fast grid information**).

Definition 5.3.9. *Let* $e_w \in E^w$ *and* $wm \in fgwm(l(e_w))$. *Define* $fgi(e_w, wm) \in \mathcal{ELS}$ *by:*

$$fgi(e_w, wm) := els(stick(e_w, wm))$$

Let further $e_v \in E^v$ *and* $vm \in fgvm(l(e_v))$. *Define* $fgi(e_v, vm) \in \mathcal{ELS}$ *by:*

$$fgi(e_v, vm) := els(stick(e_v, vm))$$

For each edge in the track graph and each wire or via model that we consider for the fast grid, the fast grid stores the extended legality information of the stick corresponding to the edge (with the colors used for the fast grid).

If the considered wire or via model is clear from the context, we omit it and use terms like the stick of an edge, the fast grid information of an edge and the shapes of an edge (which are the shapes of the stick of the edge).

Remark. In practice, we store the fast grid information slightly differently. At each vertex of the track graph, we store information for some adjacent edges. At a vertex, we store information for the wire edges in positive x- and y-direction (in relation to the vertex). For a via edge, we separate the information in two parts. At the top vertex we store extended legality information for the top pad and at the bottom vertex we store extended legality information for the bottom pad and the via middle shape. To get the full extended legality information of the edge, we simply take the maximum of the two stored values.

For each track on each layer in each sub-grid we store a sorted array of maximum intervals such that for all vertices in each interval the fast grid information is equal. In this way, the fast grid information is effectively compressed. The fast grid information is often constant for many consecutive vertices along a given track. This is the reason why

we split via edges, the extended legality information for the bottom pad is often constant in long intervals on the track on the lower layer of the via whereas the extended legality information of the top pad is often constant in long intervals on the track on the higher layer of the via which runs in orthogonal direction to the track on the lower layer. Therefore the length of the intervals can be maximized by splitting via edges. The choice on which layer we store the information for the via middle shapes is arbitrary. Figure 5.6 shows an example of split fast grid information for vias and the combined legality information.

We chose an array rather than a balanced binary search tree to store the intervals. Due to the constant maximum size of the array (because the sub-grids have a constant maximum size) and the fact that changes to the data are much less frequent than queries, an array results in better run time. Further, a balanced search tree has a considerable memory overhead which an array does not have.

Remark. To further reduce memory consumption, BonnRouteDetailed uses a highly optimized handcrafted implementation of the fast grid that can only store data for a constant number of wire / via models on each layer. We choose the wire / via models with the highest estimated wiring length on each layer but only if they are used by at least some percentage of the total wiring on the given layer. This has proven to be more efficient than supporting an arbitrary number of wire / via models on most real-world instances. On all layers of all our test instances, more than 99% of the wiring length consist of wires for which precomputed information is available. See Section 6.3 for experimental results.

We will now describe in detail how the fast grid information can be computed and kept up to date during detailed routing. We start with an empty chip, thus everything is legal. We initialize fast grid information accordingly. Algorithm 5.3.1 formalizes this. It takes as an additional argument a region that it should initialize. This can be set to the whole chip area to initialize data for the whole chip but we will later reuse this function to update some data locally and thus need this parameter.

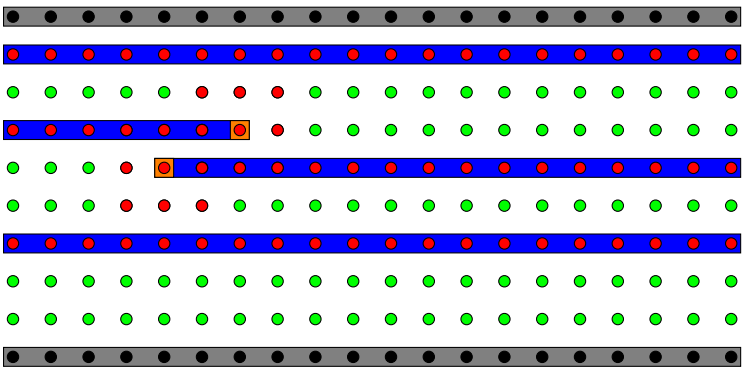
Algorithm 5.3.1: InitializeEmptyFastGrid

Input: $L \subseteq \mathcal{L}$, $r \in \mathcal{R}$

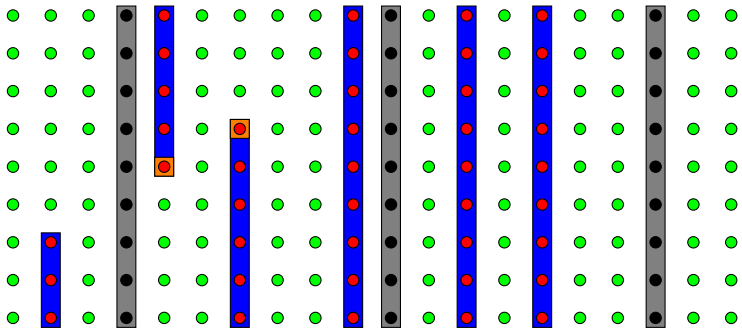
```
1 foreach e_w ∈ E^w such that l(e_w) ∈ L and r(e_w) ∩ r ≠ ∅ do
2     foreach wm ∈ fgwm(l(e_w)) do
3         fgi(e_w, wm) := legal
4 foreach e_v ∈ E^v such that l(e_v) ∈ L and (x(e_v), y(e_v)) ∈ r do
5     foreach vm ∈ fgvm(l(e_v)) do
6         fgi(e_v, vm) := legal
```

Remark. Note that in practice some positions are blocked by the chip border or because some wire and via models are not allowed at certain positions. We can easily adapt Algorithm 5.3.1 to reflect these restrictions.

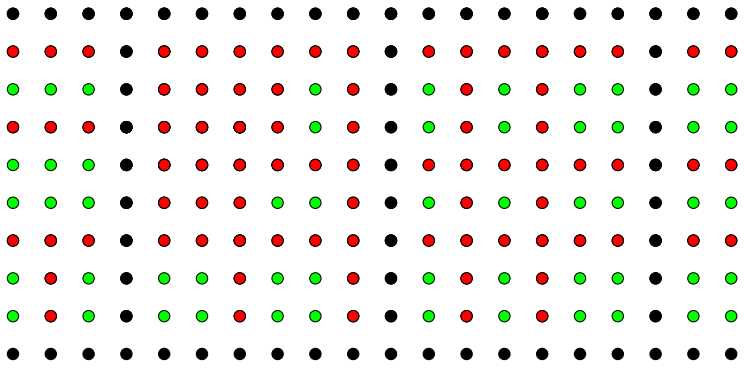
Remark. Note that in practice due to the way we store the fast grid information, in Algorithm 5.3.1 it is not necessary to explicitly set the fast grid information for each edge, but for the intersection of each track on each layer in each sub-grid with the given area we set one interval with the corresponding information.



(a) Legality of the bottom and middle shapes: 16 intervals (stored on the lower layer).



(b) Legality of the top shapes: 23 intervals (stored on the upper layer).



(c) Legal via positions: 94 horizontal intervals or 108 vertical intervals.

Figure 5.6: Fast grid legality information for vias. Splitting the legality information for vias into two parts and storing them separately results in significantly fewer intervals. Blue rectangles are removable shapes. Gray rectangles are power rails. Orange rectangles indicate vias between the two relevant layers. The small circles show the (partial) fast grid information for via edges on the given layer for a single via model. Green indicates *legal*, red indicates *illegal_by_removable* and black indicates *illegal_by_ignorable*.

After initializing the empty chip, we iteratively add each metal shape to the fast grid and update its values accordingly. When we add a shape, we find all edges of the track graph where the new shape makes corresponding sticks illegal and update the stored legality information there. As long as involved diff-net rules have small locality constants, this can be implemented efficiently. Algorithm 5.3.2 gives a formal description of this procedure. Note that Algorithm 5.3.2 also takes a region as an additional argument. It only updates values of edges that intersect that region. This is later needed for efficiency reasons. If there are no restrictions what to update, it can simply be set to the whole chip area.

Algorithm 5.3.2: FastGridAdd

Input: $s \in \mathcal{S}, DR \subseteq \mathcal{DR}, L \subseteq \mathcal{L}, r_u \in \mathcal{R}$

1 $L' := \begin{cases} \{l(s)-1, l(s), l(s)+1\} \cap \mathcal{L} & l(s) \in \mathcal{L}_{wiring} \\ \{l(s)\} & l(s) \in \mathcal{L}_{via} \end{cases}$

2 **foreach** $l \in L \cap L'$ **do**

3 **if** $l \in \mathcal{L}_{wiring}$ **then**

4 **foreach** $wm \in fgwm(l)$ **do**

5 $b := b_{sc(s),sc(wm),l}(DR)$

6 $r := (r(s) + [-b, b] \times [-b, b] + mirror(r(wm))) \cap r_u$

7 **foreach** $e_w \in E^w$ such that $l(e_w) = l$ and $r(e_w) \cap r \neq \emptyset$ **do**

8 **if** $\exists dr \in DR : dr(s, shape(stick(e_w, wm))) = false$ **then**

9 $fgi(e_w, wm) := max(fgi(e_w, wm), msp(sp(s)))$

10 **else**

11 **foreach** $vm \in fgvm(l)$ **do**

12 $sc_{l(s)}(vm) = \begin{cases} sc_b(vm) & l = l(s)+1 \\ sc_m(vm) & l = l(s) \\ sc_t(vm) & l = l(s)-1 \end{cases}$

13 $r_{l(s)}(vm) = \begin{cases} r_b(vm) & l = l(s)+1 \\ r_m(vm) & l = l(s) \\ r_t(vm) & l = l(s)-1 \end{cases}$

14 $b := b_{sc(s),sc_{l(s)}(vm),l(s)}(DR)$

15 $r := (r(s) + [-b, b] \times [-b, b] + mirror(r_{l(s)}(vm))) \cap r_u$

16 **foreach** $e_v \in E^v$ such that $l(e_v) = l$ and $(x(e_v), y(e_v)) \in r$ **do**

17 **if** $\exists dr \in DR : dr(s, shape_{l(s)}(stick(e_v, vm))) = false$ **then**

18 $fgi(e_v, vm) := max(fgi(e_v, vm), msp(sp(s)))$

Remark. As stated above, in line 8 and line 17 it suffices to consider all diff-net rules that are relevant for the given shape classes, colors and layers of the shapes instead of all diff-net rules.

Remark. In practice, in line 7 and line 16 of Algorithm 5.3.2 it is not necessary to consider each edge in the track graph individually. Instead, for each track in preferred direction

and each diff-net rule one can calculate in constant time the interval on this track that is forbidden by the added shape and the given diff-net rule for a certain type of edge. This interval can then be used to update the stored array of intervals with equal fast grid information directly.

Lemma 5.3.10. *Let $s \in \mathcal{S}$, $DR \subseteq \mathcal{DR}$, $L \subseteq \mathcal{L}$ and $r_u \in \mathcal{R}$. Let further $e_w \in E^w$, $l(e_w) \in L$, $r(e_w) \cap r_u \neq \emptyset$, $wm \in fgwm(l(e_w))$, $e_v \in E^v$, $l(e_v) \in L$, $(x(e_v), y(e_v)) \in r_u$, $vm \in fgvm(l(e_v))$.* FastGridAdd *updates $fgi(e_w, wm)$ and $fgi(e_v, vm)$ correctly if s is added to $\mathcal{SG}$. In particular, if $L = \mathcal{L}$ and $r_u = \mathcal{A}$, all precomputed checking information is correctly updated.*

Remark. In case that L in Lemma 5.3.10 and in the input of Algorithm 5.3.2 only contains wiring or via layers, the algorithm only updates fast grid information on those wiring or via layers. In this case via or wire edges e_v or e_w as in Lemma 5.3.10 do not exist and the corresponding part of the lemma does not apply. The other part of the lemma is still valid.

Proof. To prove the statement, we need to show that *fgi* is correctly updated for all combinations of wire edges and wire models as well as all combinations of via edges and via models. We do this in two steps. First, we show that if a combination of edge and model is considered in line 8 or line 17, then the corresponding fast grid information is correctly updated (a). Second, we show that in line 8 and 17 of Algorithm 5.3.2 all relevant combinations of edge and model are considered (b).

a) To show that each considered combination of edge and model is correctly updated, we first consider the definition of *fgi*, the fast grid information. The fast grid information of an edge is the maximum of the extended legality states of all the shapes of the edge. These extended legality states in turn are the maximum of $msp(sp(s'))$ over all s' in the grid such that s' makes the shape of the edge illegal (or *legal* if no such shape exists). So in other words, the fast grid information of the edge is the maximum of $msp(sp(s'))$ over all shapes s' that make the stick of the edge illegal (or *legal* if the stick of the edge is legal). Therefore, if a new shape s is added to the grid $\mathcal{SG}$, the fast grid information of any edge can only increase. More specifically, it will remain the same if the stick of the edge is legal with respect to the new shape and it will become the maximum of its original value and $msp(sp(s))$ if the stick of the edge is illegal with respect to s.

There is one more detail that we need to consider. Due to the assumption that there are no restrictions between shapes on different layers, for an edge it suffices to check the shape that is on the same layer as the new shape s (if it exists). All other shapes of the edge are legal with respect to the new shape s anyway. This is exactly what is done in lines 8, 9, 17 and 18 of Algorithm 5.3.2. If there is a diff-net rule that makes the shape on $l(s)$ of the edge illegal, update the stored fast grid information to the maximum of its original value and $msp(sp(s))$.

b) Now we show that all relevant combinations of edge and model are considered in line 8 and line 17 of Algorithm 5.3.2. First, we note that the correct set of layers is considered. If the new shape is a shape on a via layer, then only legality of via edges on the same layer can be affected. If s is a shape on a wiring layer, then legality of edges on the same wiring layer and of via edges on adjacent via layers can be affected because

exactly these edges have shapes on $l(s)$. Therefore, the set L' calculated in line 1 of Algorithm 5.3.2 is the correct set of layers to consider edges on.

Next, we consider the update restrictions L and r_u. If they are the whole set of layers and the whole chip area, they have no effect and can be ignored. Otherwise, they restrict the set of edges that are considered. More precisely, exactly all edges that are not on a layer contained in L and all edges which do not intersect r_u are omitted during the update. This is consistent with the lemma, which only makes a statement about edges on a layer contained in L and intersecting r_u. Therefore, we can restrict ourselves to the case where $L = \mathcal{L}$ and $r_u = \mathcal{A}$.

Next, we note that for each relevant layer, all via / wire models for which fast grid information is stored on that layer are considered.

It remains to show that on each layer and for each wire / via model, edges in the correct area are considered. Note that for an edge that is considered, only the shape on the layer $l(s)$ of the new shape is relevant. In line 12 and line 13 of Algorithm 5.3.2 the shape class and the overhang of the shape over the stick of the considered edges is correctly calculated. In line 5 and line 14, b is defined as the b of the diff-net rules with respect to the shape class and layer of the new shape s and the shape class of the edges on layer $l(s)$. Lemma 5.2.6 (b) states that for the new shape s with shape class $sc(s)$ and on layer $l(s)$, all shapes with the considered shape class outside of $r(s) + [-b, b] \times [-b, b]$ are legal with respect to s and thus do no need to be considered. It follows that only edges which intersect $r(s)+[-b, b]\times[-b, b]+mirror(r(wm))$ or $r(s)+[-b, b]\times[-b, b]+mirror(r_{l(s)}(vm))$ respectively can be relevant when updating the fast grid information for the new shape s. Therefore, Algorithm 5.3.2 updates all relevant edges, which concludes the proof. □

When we remove a shape, more work has to be done. We can not simply update nearby grid locations just by looking at the removed shape. We do not have the information if a location that was previously forbidden by the removed shape is still forbidden by a different shape or if it is now legal. Therefore, we first need to calculate the region that was affected by the removed shape. We call this region the update area. Then we calculate the region, called the collect area, where shapes can affect the update area. Then we find all shapes in the collect area, re-initialize the fast grid data in the update area and recalculate the values for the update area by re-adding each collected shape (and only updating values inside the update area). Figure 5.7 illustrates the removal of a via shape.

This of course takes much more run time than adding a shape but shapes are added much more often than removed. This procedure is described formally in Algorithm 5.3.3. In practice, on our testbed, there are roughly eight times as many shapes added to the fast grid as removed and the total run time for all remove operations is factor 1.5 larger than the total run time for all add operations. Both run times are negligible compared to the rest of the detailed router.

Algorithm 5.3.3: FastGridRemove

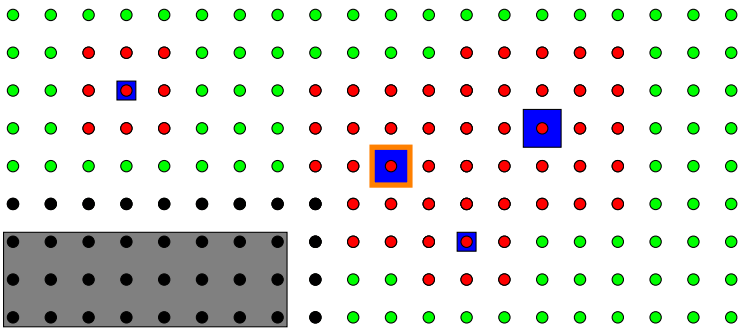
Input: $s \in \mathcal{S}, DR \subseteq \mathcal{DR}$

1 **if** $l(s) \in \mathcal{L}_{wiring}$ **then**
2 $\quad L_{update} := \{l(s)-1, l(s), l(s)+1\} \cap \mathcal{L}$
3 $\quad L_{collect} := \{l(s)-2, \dots, l(s)+2\} \cap \mathcal{L}$
4 $\quad r_{max} := bbox(\{r(wm) : wm \in fgwm(l(s))\} \cup \{r_b(vm) : vm \in fgvm(l(s)+1)\} \cup \{r_t(vm) : vm \in fgvm(l(s)-1)\})$
5 $\quad SC_{update} := \{sc(wm) : wm \in fgwm(l(s))\} \cup \{sc_b(vm) : vm \in fgvm(l(s)+1)\} \cup \{sc_t(vm) : vm \in fgvm(l(s)-1)\}$
6 **else**
7 $\quad L_{update} := \{l(s)\}$
8 $\quad L_{collect} := \{l(s)-1, l(s), l(s)+1\} \cap \mathcal{L}$
9 $\quad r_{max} := bbox(\{r_m(vm) : vm \in fgvm(l(s))\})$
10 $\quad SC_{update} := \{sc_m(vm) : vm \in fgvm(l(s))\}$
11 $b := max\{b_{sc(s),sc,l(s)}(DR) : sc \in SC_{update}\}$
12 $r_{update} := r(s) + [-b, b] \times [-b, b] + mirror(r_{max})$
13 InitializeEmptyFastGrid(L_{update},r_{update})
14 $ST_{update} := \{stick(e, vm) : e \in E^v, l(e) \in L_{update}, (x(e), y(e)) \in r_{update}, vm \in fgvm(l(e))\} \cup \{stick(e, wm) : e \in E^w, l(e) \in L_{update}, r(e) \cap r_{update} \neq \emptyset, wm \in fgwm(l(e))\}$
15 $S_{update} := \bigcup_{st \in ST_{update}} \{shapes(st)\}$
16 **foreach** $l \in L_{collect}$ **do**
17 $\quad r_{collect} := bbox(\{r(s') + [-b, b] \times [-b, b] : s' \in S_{update}, l(s') = l, b = b_{sc(s'),l}(DR)\})$
18 $\quad$ **foreach** $s' \in \mathcal{SG} : l(s') = l, r(s') \cap r_{collect} \neq \emptyset$ **do**
19 $\quad\quad$ FastGridAdd(s', DR, L_{update}, r_{update})

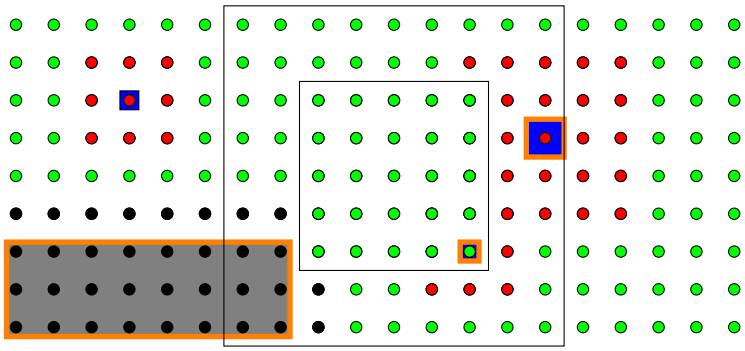
Remark. In practice, we do not compute the sets ST_{update} and S_{update} of sticks and shapes explicitly. We only need them to determine the collect areas. Due to the regular structure of the grid, the collect areas can easily be calculated without explicitly enumerating all sticks and shapes in the update area.

Lemma 5.3.11. *Algorithm 5.3.3 updates the precomputed checking information correctly. More precisely: Let $s \in \mathcal{S}$, $DR \subseteq \mathcal{DR}$, $e_w \in E^w$, $wm \in fgwm(l(e_w))$, $e_v \in E^v$, $vm \in fgvm(l(e_v))$.* FastGridRemove *updates $fgi(e_w, wm)$ and $fgi(e_v, vm)$ correctly if s is removed from $\mathcal{SG}$ (assuming that in line 18 the removed shape is not in the grid anymore).*

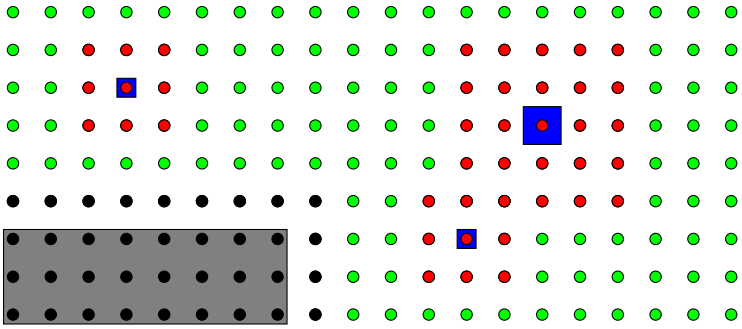
Proof. Algorithm 5.3.3 works in two steps. First, it calculates an update area and update layers such that the removed shape s can only influence the fast grid information of edges that intersect the update area and lie on the update layers. It re-initializes all the fast grid information in that area on these layers for an empty grid. Second, it determines the set of collect layers and collect areas where shapes may influence fast grid information of edges in the update area on the update layers. Then it calls FastGridAdd for all shapes intersecting the collect areas on the collect layers but only updates fast grid information



(a) Fast grid information before the shape marked in orange on a via layer is removed.



(b) Fast grid information after the shape on a via layer is removed and the update area is reinitialized. Shapes marked in orange need to be re-added to the fast grid.



(c) Fast grid information after all relevant shapes have been re-added to the fast grid.

Figure 5.7: Removal of a via shape on a via layer. This picture illustrates the different steps of Algorithm 5.3.3. Blue rectangles are removable via shapes. The gray rectangle is a blockage shape. The small circles show the fast grid information for via edges on the given layer for a single via model. Green indicates *legal*, red indicates *illegal_by_removable* and black indicates *always_illegal*. The black rectangles in (b) indicate the update and the collect area.

inside the update area and on the update layers. Like this, all information inside the update area on the update layers is correctly recomputed (because Algorithm 5.3.2 is correct) and any other fast grid information is not changed (which is correct because it is still valid). Now we prove that the first step (a) and the second step (b) are executed correctly by the algorithm.

a) First, we prove that the update area and layers are calculated correctly. If the new shape s is located on a via layer, it can only influence the fast grid information of via edges on the same layer. If the new shape is located on a wiring layer, it can influence the fast grid information on the same wiring layer and adjacent via layers. Therefore, L_{update} is set correctly in line 2 and line 7.

In line 5 and line 10, the set SC_{update} of all shape classes on layer $l(s)$ of all fast grid wire / via models is calculated. Lemma 5.2.6 (b) states that all shapes with shape classes in SC_{update} which do not intersect $r(s) + [-b, b] \times [-b, b]$ (with b as in line 11 of the algorithm) are legal with respect to s. Because r_{max} (as calculated in line 4 and line 9) is the maximum overhang of the shape over the stick of any fast grid model on layer $l(s)$, it follows that the extended legality state of all edges (with fast grid wire / via models and the colors used by the fast grid) that do not intersect r_{update} is not influenced by s. Thus fast grid information outside r_{update} or on any other layers than L_{update} does not need to be recalculated. Hence the update area and layers are calculated correctly.

b) Now we prove that the collect layers and areas are calculated correctly. To update the fast grid information of a wire edge, only shapes on the same layer are relevant. To update the fast grid information of a via edge, shapes on the via layer and the two adjacent wiring layers are relevant. Therefore, for each via layer in the set of layers to update L_{update} also the two adjacent wiring layers have to be added to the set of layers to collect shapes on $L_{collect}$. $L_{collect}$ is correctly calculated in line 3 and line 8.

In line 14 and line 15 all sticks and shapes of all edges for which the fast grid information needs to be updated are collected in the sets ST_{update} and S_{update}. Because the fast grid information of an edge depends only on the extended legality state of all its shapes, exactly all shapes that can influence legality of any of the shapes in S_{update} need to be re-added to the fast grid.

Next, we note that in the loop in line 16, all layers on which shapes can be contained in S_{update} are traversed. According to Lemma 5.2.6 (a), in line 17 the bounding box of the areas in which any shape can influence legality of any of the shapes in S_{update} is computed. In other words, all shapes outside the calculated region $r_{collect}$ can not influence legality of any of the shapes in S_{update} and thus can not influence any of the fast grid information that needs to be recomputed. Last, in line 18 and line 19, all shapes in the calculated region are added to the fast grid, which concludes the proof. □

Now we need a way to use the precomputed fast grid information to determine legality of wire sticks during detailed routing. Note that the precomputed fast grid information is thread-independent, meaning it does not reflect the thread specific shapes in $\mathcal{AS}_t$ and $\mathcal{IS}_t$. Therefore, when checking legality of a shape with respect to some thread t one can only use fast grid information for locations where legality is not influenced by shapes in $\mathcal{AS}_t$ and $\mathcal{IS}_t$ (either because there is no nearby shape in $\mathcal{AS}_t$ and $\mathcal{IS}_t$ or because legality is already determined independently of $\mathcal{AS}_t$ and $\mathcal{IS}_t$). The following two algorithms provide

thread-dependent legality information for wires and vias based on the thread-independent precomputed fast grid data. First, we describe an algorithm computing thread-dependent legality information for vias and discuss it in detail.

Algorithm 5.3.4: LEGALVIA

Input: $t \in \mathcal{T}, DR \subseteq \mathcal{DR}, e \in E^v$, $vm \in \mathcal{VM}$, $ripup\_allowed \in \mathcal{B}$

```
1  if vm ∈ fgvm(l(e)) then
2      fgi := fgi(e, vm)
3      S := shapes(stick(e, vm))
4      fga_a := true
5      fga_i := true
6      foreach s ∈ S do
7          b_a := max{b_{sc(s),sc,l(s)}(DR) : sc ∈ SCAS_t^{l(s)}}
8          b_i := max{b_{sc(s),sc,l(s)}(DR) : sc ∈ SCIS_t^{l(s)}}
9          fga_a := fga_a ∧ ({s' ∈ AS_t : l(s') =
             l(s), r(s') ∩ (r(s) + [−b_a, b_a] × [−b_a, b_a]) ≠ ∅} = ∅)
10         fga_i := fga_i ∧ ({s' ∈ IS_t : l(s') = l(s), r(s') ∩ (r(s) + [−b_i, b_i] × [−b_i, b_i]) ≠
             ∅} = ∅)
11     if fgi = always_illegal or (fgi = illegal_by_ignorable and fga_i) or
         (fgi = illegal_by_removable and fga_i and not ripup_allowed) then
12         return false
13     else if not ((fgi = legal and fga_a) or (ripup_allowed and
         fgi = illegal_by_removable and fga_a)) then
14         return StickLegal(stick(e, vm), t, DR, ripup_allowed)
15     else
16         return true
17 else
18     return StickLegal(stick(e, vm), t, DR, ripup_allowed)
```

Theorem 5.3.12. *Let $t \in \mathcal{T}$, $DR \subseteq \mathcal{DR}$, $e \in E^v$, $vm \in \mathcal{VM}$, $ripup\_allowed \in \mathcal{B}$. Let $ripup\_allowed = false$ ($ripup\_allowed = true$).*

Then LEGALVIA$(t, DR, e, vm, ripup_allowed)$ $=$ *true if and only if $stick(e, vm)$ is $legal_t$ (with rip-up).*

Proof. First, we note that if $vm \notin fgvm(l(e))$, the theorem follows from Theorem 5.2.7. Therefore we can restrict to the case that $vm \in fgvm(l(e))$.

On a high level, Algorithm 5.3.4 works as follows: First, it queries the precomputed fast grid information (a). This is not necessarily sufficient, because the precomputed fast grid information is thread-independent and the query is supposed to answer the question of legality with respect to thread t. Therefore, the algorithm checks if there are any temporary shapes of thread t in $\mathcal{AS}_t$ or $\mathcal{IS}_t$ that can potentially influence legality of the queried edge (b). Last, it is carefully decided if the precomputed checking information is valid or if a query to STICKLEGAL needs to be made and the corresponding result is returned (c).

a) In line 2 the precomputed fast grid information is queried. By definition, this value is the extended legality state of e if $\mathcal{AS}_t$ and $\mathcal{IS}_t$ are empty.

b) Next, $\mathcal{AS}_t$ and $\mathcal{IS}_t$ need to be taken into account. The algorithm iterates over all shapes s of $stick(e, vm)$ and calculates the maximum distance where any shape in $\mathcal{AS}_t$ and $\mathcal{IS}_t$ can influence the legality of s (taking into account the shape class of s and the set of shape classes currently present in $\mathcal{AS}_t$ and $\mathcal{IS}_t$). It stores in fga_a and fga_i if any shape in $\mathcal{AS}_t$ and $\mathcal{IS}_t$ is close enough to s that it can influence legality of s. This is calculated correctly by Lemma 5.2.6 (b). Therefore, after the loop beginning in line 6 is complete, fga_a and fga_i store correctly, if any shape in $\mathcal{AS}_t$ and $\mathcal{IS}_t$ can influence legality of any shape in S.

c) Last, we need to verify that the correct return value is determined based on the information computed in lines 2 to 10 (or that StickLegal is called when it is necessary). First, consider the three cases in line 11 in which the algorithm returns *false* in line 12. If $fgi = always\_illegal$ $\mathcal{AS}_t$ and $\mathcal{AS}_t$ do not matter at all, the edge e is illegal in any case. If $fgi = illegal\_by\_ignorable$, e is illegal by something that could be ignored (but not be ripped-up) by the current thread t, but because $fga_i = true$ we know that we have checked that this is not the case, there is nothing close enough that is currently ignored. Thus the precomputed fast grid information is still valid and we can also return *false*. If $fgi = illegal\_by\_removable$, e is illegal by something that can be removed. Because everything that can be removed also can be ignored (by definition), it could be ignored. Because $fga_i = true$ we know that this is not the case. If we further have $ripup\_allowed = false$ we can again return *false* (otherwise the answer might be *true* or *false* depending on the state of $\mathcal{AS}_t$).

Second, if the algorithm calls StickLegal in line 14, the result is trivially correct by Theorem 5.2.7.

Third we need to consider the two cases when the condition in line 13 is false and the algorithm returns *true* in line 16. If $fgi = legal$ and $fga_a = true$, the precomputed fast grid information means that without considering $\mathcal{AS}_t$ and $\mathcal{IS}_t$ e is legal and $fga_a = true$ means that considering the shapes in $\mathcal{AS}_t$ does not change that. Thus it is correct to return *true*. If $ripup\_allowed = true$ and $fgi = illegal\_by\_removable$ and $fga_a = true$, $fga_a = true$ means that no additional shapes in $\mathcal{AS}_t$ influence legality of e. $fgi = illegal\_by\_removable$ means that any shape that makes e illegal can be ripped up. Because we also have $ripup\_allowed = true$, it is safe to return *true*. Thus in each case the algorithm returns the correct value and the proof is complete. □

Next, for the sake of completeness, we present an algorithm computing thread-dependent legality information for wires. It is very similar to the algorithm for vias. Therefore, we do not discuss it in detail.

Algorithm 5.3.5: LEGALWIRE

Input: $t \in \mathcal{T}, DR \subseteq \mathcal{DR}, e \in E^w$, $wm \in \mathcal{WM}$, $ripup\_allowed \in \mathcal{B}$

if $wm \in fgwm(l(e))$ **then**
  $fgi := fgi(e, wm)$
  $r := r(shape(stick(e, wm)))$
  $b_a := max\{b_{sc(wm),sc,l(e)}(DR) : sc \in \mathcal{SCAS}_t^{l(e)}\}$
  $b_i := max\{b_{sc(wm),sc,l(e)}(DR) : sc \in \mathcal{SCIS}_t^{l(e)}\}$
  $fga_a := (\{s \in \mathcal{AS}_t : l(s) = l(e), r(s) \cap (r + [-b_a, b_a] \times [-b_a, b_a]) \neq \emptyset\} = \emptyset)$
  $fga_i := (\{s \in \mathcal{IS}_t : l(s) = l(e), r(s) \cap (r + [-b_i, b_i] \times [-b_i, b_i]) \neq \emptyset\} = \emptyset)$
  if $fgi = always\_illegal$ or ($fgi = illegal\_by\_ignorable$ and fga_i) or ($fgi = illegal\_by\_removable$ and fga_i and not $ripup\_allowed$) **then**
    return *false*
  else if not (($fgi = legal$ and fga_a) or ($ripup\_allowed$ and $fgi = illegal\_by\_removable$ and fga_a)) **then**
    return STICKLEGAL$(stick(e, wm), t, DR, ripup_allowed)$
  else
    return *true*
else
  return STICKLEGAL$(stick(e, wm), t, DR, ripup_allowed)$

Theorem 5.3.13. *Let $t \in \mathcal{T}$, $DR \subseteq \mathcal{DR}$, $e \in E^w$, $wm \in \mathcal{WM}$, $ripup\_allowed \in \mathcal{B}$. Let $ripup\_allowed = false$ ($ripup\_allowed = true$).*

Then LEGALWIRE$(t, DR, e, wm, ripup_allowed) = true$ *if and only if* $stick(e, wm)$ *is* $legal_t$ *(with rip-up).*

Proof. The proof is analogous to the proof of Theorem 5.3.12 if one considers that a wire edge has only one corresponding shape instead of the three that a via edge has. □

Remark. Algorithms 5.3.4 and 5.3.5 only compute legality information for wires and vias colored by the colors used by the fast grid. If a different color needs to be used, the legality information can easily be computed by STICKLEGAL from the grid.

Remark. In case of rip-up it might be useful if costs of a node depend on the set of shapes that need to be ripped-up. Algorithms 5.3.5 and 5.3.4 can be modified such that they also output the set of shapes that need to be ripped-up to make a stick or shape legal_t. However, in this case STICKLEGAL needs to be called also if the precomputed checking data could be used and indicates that something is legal with rip-up to determine the correct set of shapes. This leads to some increase in run time. Precomputing and storing the sets of shapes that need to be ripped-up unfortunately would use too much memory to be feasible.

Remark. In practice, our optimized diff-net rule checking works slightly differently. Instead of having two individual functions querying data for individual wires and vias separately (like Algorithm 5.3.4 and 5.3.5) we use one function querying all legality information

for a set of relevant wire and via models at a given vertex at once. This has several advantages. First, the path search needs information for all adjacent edges of a given vertex for all wire and via models relevant for the current net anyway. Querying all necessary information at once saves the overhead of finding the interval in the array multiple times instead of only once. Also, the queries to the temporary additional and temporary ignored shapes also have to be done only once. Further, when computing legality information for the path search algorithm, we compute an interval in preferred direction for which the legality information is constant. Due to the fact that in most cases the path search algorithm labels multiple consecutive vertices on a track, this improves the run time of the path search because computing legality information for multiple vertices once is faster than computing it for each node individually. The path search then stores the returned legality information for all vertices that it is valid for instead of querying it for each vertex.

Taking the maximum interval with identical information stored in the fast grid to determine the interval returned has some disadvantages in some cases. If the fast grid information for all relevant wire and via models can be used directly, it is beneficial to return an interval as long as possible. However, the longer the interval considered, the smaller is the chance that its legality is not influenced by the temporary additional and ignored shapes. Additionally, the longer the considered interval, the more expensive is querying the temporary additional and ignored shapes. Further, if the fast grid information can not be used for some reason, querying the grid is more expensive for a longer interval. Therefore we use the following strategy in practice: We take the interval stored in the fast grid, but cut it off max_length vertices before and after the query location. If the fast grid information can not be used but the grid has to be queried, we restrict the interval to contain only a single vertex, the query location, to avoid querying the grid more than necessary. Experimental results show that 20 is a suitable choice for max_length (see Section 6.3).

In this section we have seen how to efficiently define and store precomputed checking information, how to update it and how it can be used to implement a highly optimized interface for a fast path search algorithm. In the next chapter we prove our claims about practical performance by showing experimental results.

Chapter 6

Experimental Results

In this chapter, we present experimental results on a large set of real-world instances for the main concepts presented in Chapter 4 and Chapter 5 as well as parallelization speedups. In Section 6.1 we describe our testbed, metrics and methodology. In Section 6.2 and Section 6.3 we describe results for our automated track pattern generation and our efficient diff-net rule checking respectively. Our automated soft track pattern generation improves almost all metrics significantly and our optimized diff-net rule checking improves overall run time by more than a factor two. Furthermore, in Section 6.4 we present experimental results with different numbers of used threads showing excellent parallelization speedups of BonnRouteDetailed with our algorithms.

6.1 Testbed, Setup and Metrics

In this section, we describe our set of test instances as well as our experimental setup and metrics. All tests were done on two identical machines running CentOS 7 linux and having two 32-core AMD EPIC 7601 processors running at 2.2 GHz with 512 GB of RAM. AMD core performance boost was switched off so that all cores run at the same frequency for the parallelization experiments no matter how many cores are used. All experiments use exactly the same chip data and a fixed global routing as input. For the tests in Section 6.2 and 6.3, we ran all four tested versions at the same time on the same machine with 16 threads each. For the parallelization tests in Section 6.4, we ran each instance with each number of threads separately while nothing else was running on the machine. We used the same code for running single- and multi-threaded, but also measured an optimized single-threaded version which did not contain any locks or other parallelization utilities. As proposed in [24], we always report absolute run times as well as relative speed-ups. We define the average speed-up as the speed-up of the sum of the run times rather than some mean of the individual speed-ups. Observed speedups in this setting are typically worse than speedups observed in real productive runs, because in practice, single-threaded runs would not be run alone on a machine but load on machines would automatically be balanced, reducing performance of single-threaded runs. The theoretical limit on the speedup (the number of threads) is not achievable in this setting, because cache and other resources on the machine are shared between all threads and do not scale with the number of threads.

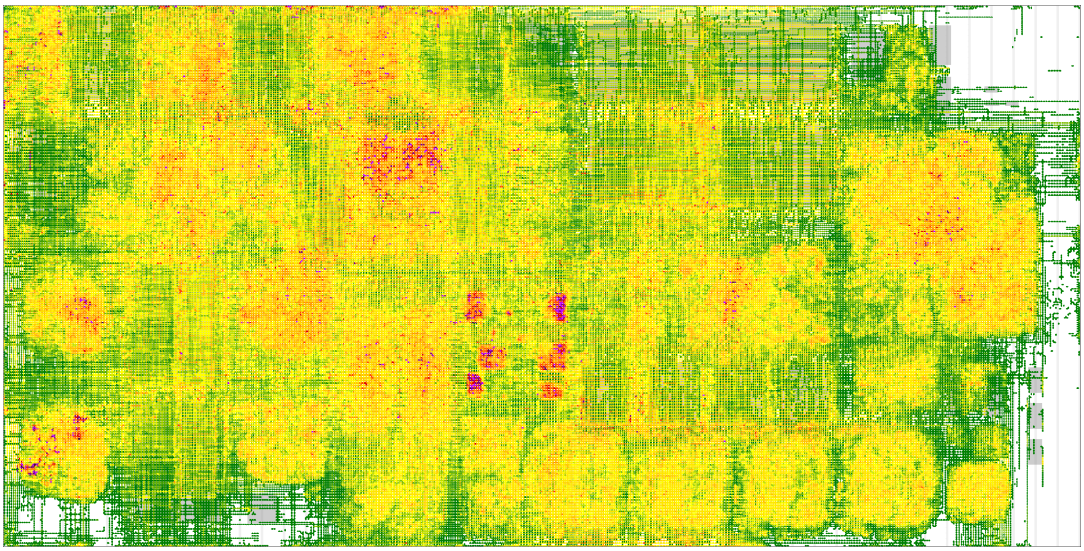
We use a large number of different instances from 14nm and 7nm technology with different sizes, levels of congestion and levels of maturity to test our algorithms. Furthermore, we validate some of our results on a set of newer 7nm instances. We did not use any of these verification testcases to tune our algorithms. They only became available after we had finished our implementation and tuning. All our test instances are real-world designs provided by our cooperation partner IBM. Our main testbed is shown in Table 6.1 and includes 60 instances in total, of which 39 are manufactured in 14nm technology and 21 in 7nm technology. The portion of non-default wires is measured after detailed routing and includes any wire with a non-default width or any non-default spacing to any other wire. Such wires are typically used for timing reasons and make detailed routing considerably harder. The portion of nets assigned to wide layers means the portion of all nets that are supposed to be routed mainly on layers with default wires wider than the default wires on the lowest routing layer. This figure is a rough indicator of how many nets have some timing criticality.

Instances range from very small RLMs to huge instances with millions of nets and many routing layers. In total, our test instances have 18 367 089 nets, ranging from 713 to 2 089 586 nets. They have between 5 and 16 routing layers and areas from 0.00031 mm^2 to over 2 mm^2. They range from timing-wise completely uncritical instances with almost only default wires to very critical and complex instances with over 25% wide wires and a high percentage of nets assigned to the upper layers for routing. Routing congestion ranges from completely uncritical to highly congested. Figure 6.1 shows two congestion maps of a critical and an easy instance. Most of our 14nm testbed is in a very mature state, especially the instances RLM-14-A and RLM-14-B, but also the instances L-14-A and L-14-C. The instances L-14-B are a little less clean, but still much cleaner than most of the 7nm instances. The 7nm instances come from very early phases of the design cycle and thus show many more design problems such as overcongested areas, inaccessible pins and other design errors. Due to the fact that we used some of the instances in our main testbed also for tuning parts of our algorithms and the circumstance that by the time we were done executing tests on our main testbed newer and much cleaner 7nm testcases were available, we further test on a smaller testbed of 22 much more mature 7nm instances shown in Table 6.2. For these tests, we used the same version of the algorithms described in Chapter 4 and 5 but a slightly newer version of other parts of BonnRouteDetailed. These verification testcases also include a wide variety of instances, ranging from 733 to 1 735 036 nets and from 8 to 16 routing layers. They were captured late in the design flow and thus include very few design problems. Some techniques to avoid design rule violations inside BonnRouteDetailed that are not discussed in detail in this thesis have been improved in the meantime, thus in these tests comparatively few design rule violations remain.

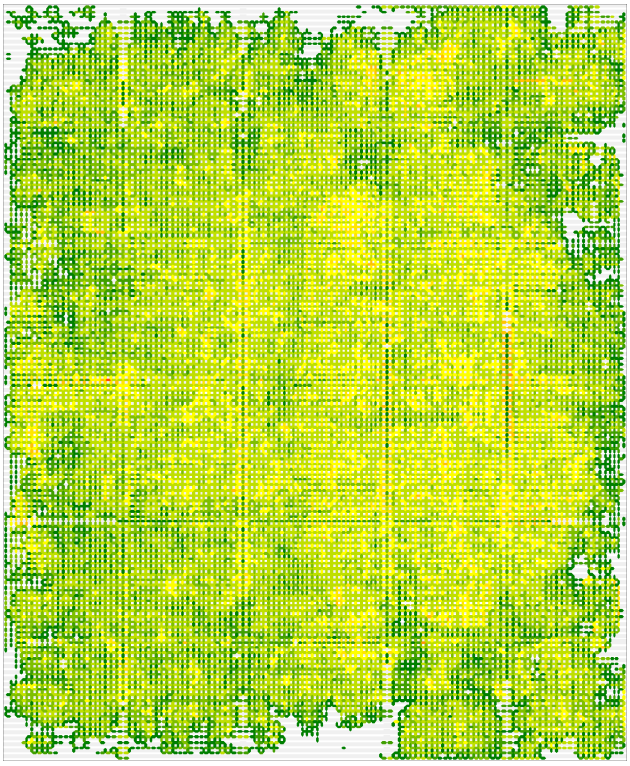
For our tests we measure a number of metrics, which we briefly describe now. First is the total run time of BonnRouteDetailed because that is of key interest to the designers using our tools. The faster tools run, the faster chips can be designed. We also measure the run time of the main detailed routing step (without several loading, pre- and post-processing steps) separately, because that is the part of BonnRouteDetailed that is effected by our changes. Second, we measure total memory consumption. This is less important than run time, because less memory consumption does not directly translate into increased productivity, but too high memory consumption is problematic if there are not enough

Instance	Technology	Number of Nets	Chip Area (mm^2)	Number of Wiring Layers	Portion of Non-Default Wires	Portion of Nets Assigned to Wide Layers
RLM-14-A-1	14nm	1 807	0.00170	5	8.16%	5.09%
RLM-14-A-2	14nm	12 817	0.01156	5	3.79%	1.01%
RLM-14-A-3	14nm	25 038	0.02015	7	7.02%	2.79%
RLM-14-A-4	14nm	30 753	0.17003	7	11.58%	13.45%
RLM-14-A-5	14nm	40 981	0.02831	8	1.47%	7.53%
RLM-14-A-6	14nm	56 459	0.03828	9	5.76%	4.72%
RLM-14-A-7	14nm	65 989	0.24946	7	7.88%	7.68%
RLM-14-A-8	14nm	73 729	0.04542	7	6.98%	3.46%
RLM-14-A-9	14nm	79 833	0.09269	7	8.37%	6.25%
RLM-14-A-10	14nm	84 255	0.12845	7	7.04%	5.14%
RLM-14-A-11	14nm	139 278	0.16767	8	7.97%	4.89%
RLM-14-A-12	14nm	142 243	0.23446	13	6.75%	11.43%
RLM-14-A-13	14nm	185 697	0.11608	8	3.76%	5.39%
RLM-14-A-14	14nm	216 422	0.38358	7	5.92%	4.57%
RLM-14-A-15	14nm	367 214	0.38287	9	5.02%	9.45%
RLM-14-B-1	14nm	11 042	0.00931	5	8.64%	3.35%
RLM-14-B-2	14nm	13 256	0.02760	7	3.58%	5.79%
RLM-14-B-3	14nm	18 799	0.01887	7	22.19%	9.53%
RLM-14-B-4	14nm	20 243	0.01908	5	18.03%	6.16%
RLM-14-B-5	14nm	23 637	0.02713	7	14.00%	9.76%
RLM-14-B-6	14nm	29 330	0.02202	7	10.89%	8.08%
RLM-14-B-7	14nm	33 878	0.03043	7	10.78%	5.49%
RLM-14-B-8	14nm	35 779	0.02591	7	11.79%	7.44%
RLM-14-B-9	14nm	39 558	0.04234	5	7.10%	2.37%
RLM-14-B-10	14nm	61 537	0.04622	9	1.55%	9.66%
RLM-14-B-11	14nm	52 819	0.04813	7	14.99%	10.71%
RLM-14-B-12	14nm	55 328	0.05308	7	14.50%	8.12%
RLM-14-B-13	14nm	72 995	0.05190	9	10.81%	11.05%
RLM-14-B-14	14nm	105 645	0.08061	7	6.58%	3.29%
L-14-A-1	14nm	421 316	0.89129	15	26.48%	17.65%
L-14-A-2	14nm	1 778 320	2.00383	15	17.10%	13.17%
L-14-A-3	14nm	1 580 264	1.59441	15	18.66%	8.94%
L-14-B-1	14nm	1 190 206	1.43917	15	9.01%	7.98%
L-14-B-2	14nm	1 310 380	1.57811	15	12.00%	8.18%
L-14-B-3	14nm	1 610 456	1.51498	15	9.56%	7.55%
L-14-B-4	14nm	1 603 682	1.53260	15	8.04%	6.79%
L-14-C-1	14nm	332 538	0.34351	10	6.74%	5.26%
L-14-C-2	14nm	442 007	0.77175	13	3.90%	4.31%
L-14-C-3	14nm	456 862	0.39919	13	14.18%	7.59%
L-7-1	7nm	354 195	0.10331	12	8.54%	13.13%
L-7-2	7nm	761 252	0.31134	16	8.41%	7.68%
L-7-3	7nm	251 436	0.08152	10	12.36%	11.11%
L-7-4	7nm	281 655	0.09316	14	13.02%	12.84%
L-7-5	7nm	2 089 586	1.20538	16	10.42%	14.14%
RLM-7-A-1	7nm	5 976	0.00932	14	12.22%	17.96%
RLM-7-A-2	7nm	8 584	0.00206	8	4.87%	2.55%
RLM-7-A-3	7nm	12 414	0.01031	10	7.55%	5.03%
RLM-7-A-4	7nm	20 584	0.01143	14	12.60%	11.42%
RLM-7-A-5	7nm	16 477	0.00370	8	5.74%	3.31%
RLM-7-A-6	7nm	17 797	0.00756	9	7.91%	9.44%
RLM-7-A-7	7nm	20 205	0.00482	10	11.38%	5.92%
RLM-7-A-8	7nm	54 302	0.01674	8	19.72%	9.52%
RLM-7-A-9	7nm	125 735	0.03024	12	8.46%	4.15%
RLM-7-A-10	7nm	136 485	0.06033	10	11.68%	6.21%
RLM-7-A-11	7nm	195 895	0.07505	10	9.18%	5.52%
RLM-7-A-12	7nm	259 180	0.17650	10	8.12%	5.15%
RLM-7-A-13	7nm	713	0.00031	8	37.48%	26.37%
RLM-7-A-14	7nm	260 924	0.09465	10	7.85%	8.51%
RLM-7-A-15	7nm	295 835	0.15734	15	11.53%	15.23%
RLM-7-A-16	7nm	375 467	0.08266	10	9.19%	10.31%

Table 6.1: Test instances.



(a) Global routing congestion map of L-14-A-2.



(b) Global routing congestion map of L-14-C-1.

Figure 6.1: Sample congestion maps. White and green areas are completely uncongested, yellow areas are denser, orange and red areas are congested and purple indicates over-congestion. L-14-A-2 has large regions with high congestion and even some over-congested hotspots whereas L-14-C-1 is completely uncongested.

Instance	Technology	Number of Nets	Chip Area (mm^2)	Number of Wiring Layers	Portion of Non-Default Wires	Portion of Nets Assigned to Wide Layers
L-7n-1	7nm	373 431	0.40578	16	11.46%	14.66%
L-7n-2	7nm	625 111	0.47298	16	12.73%	13.17%
L-7n-3	7nm	646 633	0.36118	16	12.50%	12.03%
L-7n-4	7nm	805 244	0.34399	16	11.36%	13.19%
L-7n-5	7nm	1 044 976	0.36182	16	12.14%	14.26%
L-7n-6	7nm	1 170 967	0.46358	16	8.86%	12.55%
L-7n-7	7nm	1 206 790	0.63733	16	9.63%	14.95%
L-7n-8	7nm	1 735 036	1.20538	16	12.34%	15.82%
RLM-7n-A-1	7nm	733	0.00031	8	20.79%	10.50%
RLM-7n-A-2	7nm	6 076	0.00967	14	7.31%	13.35%
RLM-7n-A-3	7nm	11 456	0.01031	10	4.87%	2.90%
RLM-7n-A-4	7nm	15 411	0.00430	8	3.36%	2.80%
RLM-7n-A-5	7nm	18 556	0.00555	8	13.93%	8.20%
RLM-7n-A-6	7nm	20 808	0.00876	9	7.35%	7.17%
RLM-7n-A-7	7nm	56 721	0.01674	8	17.21%	5.77%
RLM-7n-A-8	7nm	115 652	0.03024	12	4.13%	3.22%
RLM-7n-A-9	7nm	136 479	0.06033	10	11.46%	5.51%
RLM-7n-A-10	7nm	199 445	0.07505	10	9.37%	4.28%
RLM-7n-A-11	7nm	236 442	0.17650	10	6.15%	5.20%
RLM-7n-A-12	7nm	247 082	0.09465	10	7.34%	8.83%
RLM-7n-A-13	7nm	152 363	0.13733	10	7.96%	8.80%
RLM-7n-A-14	7nm	382 578	0.08266	10	5.50%	6.43%

Table 6.2: Newer test instances used to verify results.

machines with sufficient memory available. We also measure wire length and the number of vias because these metrics are a good indicator of routing efficiency, power consumption and timing behavior. Typically, high wire length and many vias lead to high congestion, high power consumption and bad timing. Furthermore, because we can not measure timing characteristics directly due to technical reasons, we measure the occurrence of a number of timing-wire undesirable configurations. Most importantly, so-called scenic nets or scenics [4]. We call a net scenic, if its wiring is at least 25 µm long and at least 25%, 50% or 100% respectively longer than the length of an approximately minimum Steiner tree. See Figure 6.2 for an example of a scenic net. Scenic nets have a high probability to show bad timing behavior because additional wire length often introduces higher delays. Furthermore we measure the number of layer and taper fuses in the computed routing (see Section 3.12 for a definition of layer and taper fuses).

We measure the number of remaining design rule violations after detailed routing. We briefly explain the most important classes of design rule violations. The most important design rule violation is a short, two shapes of different nets that touch each other, leading to an electrical short (Figure 6.3(a)). Similarly important are diff-net spacing violations, which occur if shapes from different nets do not touch but are too close together (Figure 6.3(b)). Such shapes have a high probability to be connected electrically by the inaccuracies of the physical production process. The next class of design rule violations are so-called same-net spacing violations (Figure 6.3(c)). For technical reasons, also shapes of the same net have to abide certain distance rules in many cases. These errors are not as important as diff-net spacing violations, because they usually can be fixed more easily in a post-processing step. Another very important class of design rules are so-called rectangular shapes constraints (Figure 6.3(d)). In certain technologies, on some layers

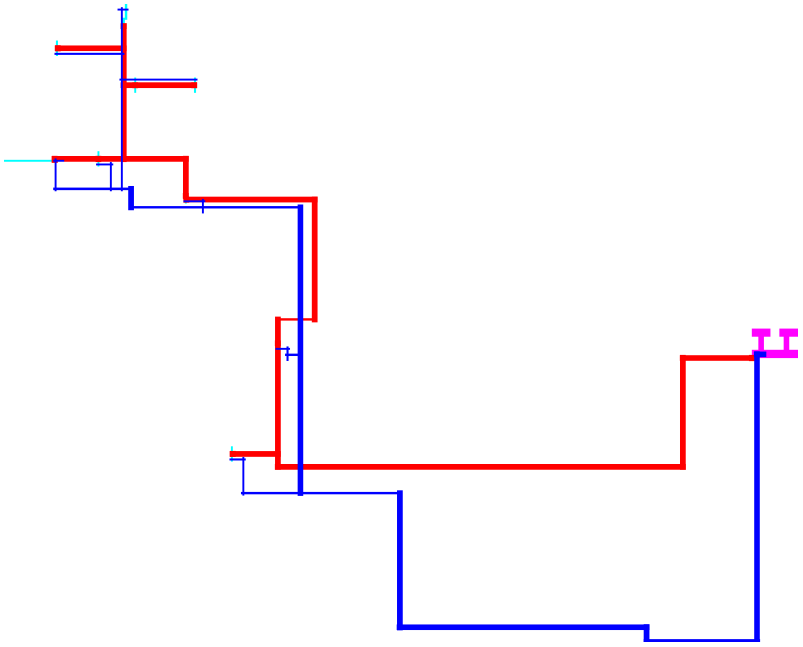
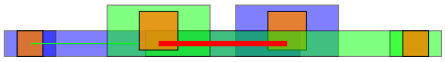


Figure 6.2: Scenic net. Detailed wires are drawn in blue and global wires in red. The sink pins are drawn in cyan and the source pin is marked in magenta.

only rectangular shapes can be produced and each non-rectangular shape constitutes a design rule violation which is often hard to fix later on. The last very important class of design rule violations are so-called minimum area violations (Figure 6.3(e)). Shapes on each routing layer need to have at least a certain area. These errors by definition require additional routing space to be fixed and thus are hard to fix late in the flow. Furthermore, we report two less important but very common error classes individually, namely minimum edge length and via extension errors. In most technologies, there are certain restrictions on short metal edges (Figure 6.3(f)) and via middle shapes require certain metal areas on the neighboring wiring layers (Figure 6.3(g)). Violations of these rules are called minimum edge length errors and via extension errors. Such errors are usually comparatively easy to fix after routing because they only require very local modifications in a single net. We summarize all remaining design rule violations under other.

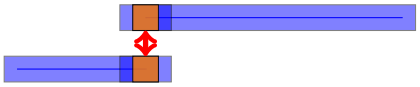
6.2 Computing Track Patterns

In this section, we present experimental results for the techniques developed in Chapter 4. We tested four different versions. All tested versions obey the same hard track patterns for some wire models and only differ in the usage of different soft track patterns for wire models that do not need to obey a hard track pattern. The first version, our baseline, is the version that was previously used by BonnRouteDetailed in practice. It contained optimal track patterns for the default wire model on each layer plus some hand-crafted track patterns for some frequent wire models. All other wire models were allowed to use the union of all defined track patterns. In the tables we denote this version by “old”. The



(a) Shorts. Shapes of different nets intersect.

(b) Diff-net spacing violations. Shapes of different nets are too close to each other but do not intersect.

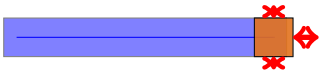


(c) Same-net spacing violations. Different shapes of the same net are too close to each other but do not intersect. In this case the via middle shapes are too close.

(d) Rectangular shape constraints. Sometimes non-rectangular metal components are forbidden.

(e) Minimum area violation. All metal components must have at least some given area.

(f) Minimum edge length errors. Two adjacent short edges are forbidden.



(g) Via extension errors. The metal on a routing layer needs to extend a certain distance over the projection of the middle shape in each direction.

Figure 6.3: Important classes of design rule violations. Blue and green indicate metal shapes on a routing layer (of different nets). Orange indicates via shapes. Red marks indicate the violations.

second version is the most simple version we could come up with, computing an optimal track pattern (TP3) for the default wire model on each layer and using that track pattern for each wire model (denoted by "default tp"). The third version computes an optimal track pattern (TP3) for each wire model individually ("opt sep"). The fourth version uses our algorithm described in Section 4.3 including the modifications discussed in Section 4.4 ("opt").

When analyzing the results in this section one should be aware of some facts. First, on many layers, especially on the lower ones, there are hard track patterns given that need to be obeyed by all versions of BonnRouteDetailed. Further, especially on smaller instances, but also on some large ones, only a very limited set of different wire models is used and for some of them there is a unique clearly optimal track pattern that is computed by all four versions. This is partially due to the fact that the choice of wire models and the design of the instances (e.g. power rails) was tuned for the existing tools including BonnRouteDetailed, often discarding wire models that were not handled well by the routing tools. Therefore, especially on smaller instances, there is little freedom to optimize track patterns at all. Due to the reasons stated above, one can expect even greater benefits once instances are tuned to the new optimized track patterns. Unfortunately, the design even of a single very large scale integrated circuit is so complex that the effort to execute the whole physical design process twice with different track patterns is too large to be feasible. Therefore we restrict ourselves to routing results on existing designs with the different soft track pattern versions, even if our new optimized track patterns could give even better results if the choice of wire models and other features were retuned to the new behavior of BonnRouteDetailed. Furthermore, we need to mention that the 7nm instances in our main testbed are mostly in a very early design stage, thus having comparatively many design problems such as overcongested areas or illegal pins. Additionally, the 7nm instances have comparatively few non-default wire models (the large 7nm designs all have less than 14% of non-default wire models whereas some of the large 14nm designs have over 25%) and on 7nm there are more layers with hard track patterns than on 14nm. Therefore the 14nm instances have a higher potential for optimizing the soft track patterns whereas the 7nm instances have more design rule violations.

First, we discuss the run time of our algorithm itself, which is negligible. The longest run time occurs on instance L-14-A-2 where it runs slightly over one minute. On all other instances it runs faster than one minute and on most instances it runs in under ten seconds. These run times are negligible in comparison to the total run time of detailed routing which even with the best version takes almost ten hours on some instances.

Now, we discuss routing results with the four versions of soft track patterns described above. Tables 6.3 and 6.4 show overall routing results on the 14nm and 7nm instances of our main testbed respectively.

Both for the 14nm as well as for the 7nm instances, all major metrics improve in the "opt" version. Run time reduces by 40% and 33% respectively. Memory consumption also shows a minor decrease. The number of vias decreases by over 1% and over 0.5% respectively and also wire length decreases by 0.3% and 0.2%. The number of scenics decreases by roughly 10% and 4% respectively. On 14nm, the number of layer fuses is reduced by almost 50%, on 7nm they decrease by around 15%. The number of taper fuses decreases by 4% on 14nm and by 14% on 7nm. As expected, the improvements are larger

Version	Run Time (hh:mm:ss)	Memory (GB)	Vias (10^6)	Wire Length (m)	Scenics 25	Scenics 50	Scenics 100	Layer Fuses	Taper Fuses
				sum RLM-14-A					
old	5:39:19	136.4	13.4	30.7	12 605	2 343	173	79	723
default tp	5:13:09	132.8	13.4	30.6	12 348	2 313	178	79	435
opt sep	5:21:29	135.2	13.4	30.6	12 390	2 317	182	68	1 378
opt	5:20:59	134.5	13.4	30.6	12 394	2 322	174	59	551
				sum RLM-14-B					
old	1:23:28	67.7	4.7	7.5	3 585	509	79	22	254
default tp	1:33:14	66.5	4.7	7.5	3 554	552	102	60	267
opt sep	1:29:26	65.9	4.7	7.5	3 593	527	81	30	285
opt	1:28:00	66.3	4.7	7.5	3 563	509	79	30	242
				L-14-A-1					
old	3:30:56	32.7	3.6	11.8	2 136	544	172	147	2 713
default tp	2:27:52	29.3	3.6	11.8	2 201	562	175	146	2 953
opt sep	2:32:21	31.9	3.6	11.8	2 072	509	157	75	3 048
opt	1:35:55	29.5	3.6	11.8	2 127	513	165	66	2 919
				L-14-A-2					
old	19:56:18	92.2	16.9	41.4	13 883	2 100	232	677	4 961
default tp	14:54:25	88.2	16.8	41.4	13 954	2 154	253	781	5 398
opt sep	10:52:27	93.1	16.7	41.2	11 559	1 692	166	284	4 961
opt	7:57:49	89.3	16.6	41.2	10 773	1 610	156	179	4 992
				L-14-A-3					
old	6:04:31	77.1	13.4	25.3	7 842	949	110	519	2 173
default tp	4:24:36	74.0	13.4	25.3	8 350	979	120	442	2 228
opt sep	4:57:42	76.8	13.3	25.3	7 871	913	126	399	2 739
opt	4:36:51	74.5	13.3	25.3	7 741	891	122	391	2 304
				L-14-B-1					
old	9:37:00	67.0	12.4	25.7	8 464	859	31	361	2 354
default tp	6:52:10	64.1	12.3	25.6	8 138	886	54	529	2 186
opt sep	6:19:34	67.4	12.2	25.6	7 672	768	29	187	2 675
opt	5:32:42	64.5	12.2	25.6	7 374	699	34	165	2 203
				L-14-B-2					
old	7:40:51	72.9	13.6	31.5	8 676	859	45	62	2 797
default tp	5:25:13	69.7	13.4	31.5	8 326	838	42	59	2 571
opt sep	5:43:27	73.5	13.4	31.4	7 888	765	38	67	2 806
opt	5:18:10	70.2	13.4	31.4	7 685	742	35	39	2 604
				L-14-B-3					
old	18:06:29	82.7	17.0	32.0	19 019	2 620	199	375	5 727
default tp	12:12:30	80.4	16.8	31.9	18 574	2 570	188	323	5 616
opt sep	12:55:05	85.8	16.8	31.9	17 418	2 283	161	197	6 121
opt	9:54:15	80.4	16.7	31.8	16 713	2 148	141	173	5 525
				L-14-B-4					
old	14:07:26	83.7	16.9	29.9	17 205	1 905	105	348	4 030
default tp	11:07:55	80.2	16.7	29.8	16 622	1 870	105	364	3 793
opt sep	11:00:29	84.1	16.7	29.8	15 760	1 697	84	219	4 524
opt	9:31:57	80.8	16.7	29.8	15 323	1 590	89	189	3 700
				L-14-C-1					
old	1:04:22	27.2	3.1	6.3	1 285	136	1	3	131
default tp	1:02:24	26.2	3.0	6.3	1 230	131	2	0	120
opt sep	58:15	27.0	3.0	6.3	1 248	123	3	2	136
opt	1:04:19	26.0	3.0	6.3	1 232	133	7	6	126
				L-14-C-2					
old	3:15:00	33.0	4.6	10.6	1 800	306	4	23	499
default tp	2:01:16	32.1	4.6	10.6	1 666	266	1	30	472
opt sep	2:24:13	33.1	4.6	10.6	1 691	284	2	36	543
opt	2:13:02	30.9	4.6	10.6	1 697	281	0	31	478
				L-14-C-3					
old	4:23:48	33.0	4.7	9.9	3 899	753	190	187	1 586
default tp	2:49:21	32.2	4.6	9.9	3 741	691	188	115	1 156
opt sep	3:39:50	32.4	4.6	9.9	3 944	765	195	139	1 676
opt	2:38:36	32.4	4.6	9.9	3 656	689	182	124	1 197
				total sum					
old	**94:49:28**	**805.6**	**124.2**	**262.6**	**100 399**	**13 883**	**1 341**	**2 803**	**27 948**
default tp	**70:04:05**	**775.7**	**123.3**	**262.3**	**98 704**	**13 812**	**1 408**	**2 928**	**27 195**
	-26.11%	-3.71%	-0.74%	-0.10%	-1.69%	-0.51%	5.00%	4.46%	-2.69%
opt sep	**68:14:18**	**806.2**	**123.0**	**261.9**	**93 106**	**12 643**	**1 224**	**1 703**	**30 892**
	-28.04%	0.07%	-0.96%	-0.27%	-7.26%	-8.93%	-8.72%	-39.24%	10.53%
opt	**57:12:35**	**779.2**	**122.8**	**261.7**	**90 278**	**12 127**	**1 184**	**1 452**	**26 841**
	-39.67%	-3.27%	-1.16%	-0.34%	-10.08%	-12.65%	-11.71%	-48.20%	-3.96%

Table 6.3: Routing results for different soft track patterns used in BonnRouteDetailed for 14nm instances.

Version	Run Time (hh:mm:ss)	Memory (GB)	Vias (10^6)	Wire Length (m)	Scenics 25	Scenics 50	Scenics 100	Layer Fuses	Taper Fuses
				L-7-1					
old	1:03:48	20.1	3.9	3.3	1 332	87	0	47	549
default tp	49:38	20.0	3.9	3.3	1 269	81	0	45	550
opt sep	49:06	20.0	3.9	3.3	1 264	73	1	45	607
opt	51:28	20.0	3.9	3.3	1 301	78	1	42	329
				L-7-2					
old	2:19:57	40.4	7.4	6.3	1 524	332	2	510	653
default tp	1:36:58	39.0	7.4	6.3	1 534	327	0	513	641
opt sep	1:31:22	39.7	7.4	6.3	1 529	326	0	501	638
opt	1:28:43	39.3	7.4	6.3	1 556	337	1	489	528
				L-7-3					
old	8:17:32	17.5	3.3	2.8	5 698	1 607	241	6 538	1 450
default tp	7:35:03	16.9	3.3	2.8	5 701	1 596	233	6 801	1 471
opt sep	6:39:14	17.0	3.3	2.8	5 541	1 477	221	6 194	1 444
opt	6:19:30	17.0	3.3	2.8	5 413	1 480	200	5 864	1 456
				L-7-4					
old	19:42:27	22.6	4.1	3.6	13 169	4 015	577	9 170	2 887
default tp	12:49:30	21.1	4.1	3.6	12 957	3 830	545	8 187	2 821
opt sep	12:43:36	21.3	4.1	3.6	12 754	3 864	540	7 712	2 860
opt	11:06:34	21.1	4.0	3.5	12 602	3 682	518	7 286	2 823
				L-7-5					
old	13:38:55	123.2	24.8	29.9	31 179	8 959	1 698	1 136	5 464
default tp	10:38:46	121.0	24.6	29.8	30 675	8 888	1 710	999	5 366
opt sep	9:53:14	124.5	24.6	29.8	30 434	8 816	1 705	897	5 808
opt	9:09:18	122.0	24.6	29.8	30 221	8 784	1 677	979	3 703
				sum RLM-7-A					
old	6:14:30	134.6	19.9	17.9	10 939	2 192	186	1 936	4 466
default tp	5:06:15	130.5	19.9	17.9	10 815	2 099	180	1 797	4 489
opt sep	5:15:10	132.7	19.9	17.9	10 769	2 135	178	1 745	4 451
opt	5:17:51	130.4	19.9	17.9	10 723	2 106	177	1 721	4 459
				total sum					
old	**51:17:09**	**358.4**	**63.3**	**63.7**	**63 841**	**17 192**	**2 704**	**19 337**	**15 469**
default tp	**38:36:10**	**348.4**	**63.0**	**63.6**	**62 951**	**16 821**	**2 668**	**18 342**	**15 338**
	-24.73%	-2.79%	-0.44%	-0.15%	-1.39%	-2.16%	-1.33%	-5.15%	-0.85%
opt sep	**36:51:42**	**355.2**	**63.0**	**63.6**	**62 291**	**16 691**	**2 645**	**17 094**	**15 808**
	-28.13%	-0.89%	-0.48%	-0.19%	-2.43%	-2.91%	-2.18%	-11.60%	2.19%
opt	**34:13:24**	**349.9**	**63.0**	**63.6**	**61 816**	**16 467**	**2 574**	**16 381**	**13 298**
	-33.27%	-2.37%	-0.53%	-0.22%	-3.17%	-4.22%	-4.81%	-15.29%	-14.03%

Table 6.4: Routing results for different soft track patterns used in BonnRouteDetailed for 7nm instances.

for 14nm instances than for 7nm instances with the exception of the number of taper fuses. Overall, experimental results show a very significant improvement in run time as well as some improvements in the metrics relevant for timing behavior. It is surprising that already the most simple version "default tp" is better than the baseline in most metrics. Even this version improves run time by 26% and 24% respectively. It also shows some benefits in number of vias and wire length but increases the number of scenics 100 and the number of layer fuses on 14nm instances slightly. The version "opt sep" is a little bit better than "default tp" but not as good as "opt". It decreases run time by 28% on both technologies and shows improvements in most metrics, in particular it already decreases the number of layer fuses on 14nm instances by almost 40%. However, it increases the number of taper fuses by more than 10% and 2% respectively.

If we look at individual instances in more detail, it becomes clear that the improvements differ very much on different instances. On the small 14nm instances RLM-14-A and RLM-14-B, as well as on L-14-C-1 run time stays roughly the same with all tested versions. On the other hand, on some of the large 14nm designs we see drastic decreases in run time, for example on L-14-A-2 run time reduces from 20 hours to 8 hours, a reduction of 60%. On 7nm, the effect is similar; improvements are much higher on some of the larger designs. This is due to the fact that on larger designs there is usually a higher fraction of wire length on layers without hard track patterns (because larger designs include higher layers which have no hard track patterns) and thus more room for optimization by our algorithm. Especially L-14-A-2 has an unusually diverse mixture of non-default wire models on some layers.

Next, we look at the remaining design rule violations after BonnRouteDetailed. Table 6.5 and 6.6 show the number of the most important design rule violations remaining after BonnRouteDetailed.

We can see that the number of design rule violations remaining after BonnRouteDetailed decreases significantly in the "opt" version both on 14nm and on 7nm. Design rule violations of each category apart from diff-net spacing violations on 14nm and rectangular shape constraint violations on 7nm are significantly reduced. On 14nm, there are 9% less shorts, 26% less same-net spacing violations, 35% less min area errors and almost 10% less rectangular shape constraint violations. Minimum edge length errors are reduced by almost 50% and via extension errors are reduced by 16%. On 7nm, due to the fact that the designs are less clean, there are many more shorts and diff-net spacing violations to begin with. Consequently there is also much more room for improvements and shorts are reduced by 24% and diff-net spacing violations by 20%. Same net spacing violations are reduced by 18% and min area violations by 9%. Furthermore, minimum edge length violations are reduced by 26% and via extension errors by 15%. Overall, these are very significant reductions in many important error classes greatly improving the quality of BonnRouteDetailed.

If we look at the other versions, on 7nm all versions consistently improve all error classes except for rectangular shape constraints. In each error class, the improvement is smallest with "default" tp and largest with "opt". On 14nm, "default tp" shows 24% more shorts and 4% more diff-net spacing violations than the baseline run and "opt sep" produces 16% more diff-net spacing violations and 3% more rectangular shape constraint violations than the baseline run. Overall the "opt" run is clearly superior. Therefore we

Version	Shorts	Diff Net Spacing	Same Net Spacing	Rectangular Shape Constraint	Minimum Area	Minimum Edge Length	Via Extension	Other
				sum RLM-14-A				
old	120	1 532	2 388	958	24	690	1 399	4 752
default tp	103	1 382	1 983	863	23	529	1 181	4 457
opt sep	140	1 755	2 018	1 142	20	566	1 686	3 277
opt	116	1 533	1 950	804	25	440	1 175	4 021
				sum RLM-14-B				
old	84	341	670	202	5	194	262	258
default tp	185	446	651	204	4	296	350	274
opt sep	89	364	618	208	2	152	322	277
opt	73	329	608	204	5	234	269	262
				L-14-A-1				
old	357	1 478	1 635	294	20	1 237	1 932	824
default tp	460	1 604	1 370	292	12	795	1 849	668
opt sep	53	1 004	1 303	319	13	849	1 700	819
opt	131	1 185	1 134	308	4	799	1 657	665
				L-14-A-2				
old	278	3 236	7 340	508	49	7 283	4 009	2 178
default tp	627	3 125	4 455	407	37	3 898	2 636	1 432
opt sep	153	2 501	3 853	393	28	3 185	2 231	1 510
opt	131	2 297	3 558	388	19	3 716	1 936	1 297
				L-14-A-3				
old	100	1 519	1 717	208	13	2 279	1 005	282
default tp	62	1 186	1 326	180	5	1 434	722	369
opt sep	15	1 412	1 499	186	14	1 544	639	359
opt	36	1 517	1 172	180	15	1 379	704	300
				L-14-B-1				
old	1 294	3 141	3 382	203	190	3 567	2 010	1 304
default tp	1 484	3 210	3 166	202	144	1 781	2 173	983
opt sep	1 143	3 047	3 610	216	158	1 878	1 836	1 123
opt	1 119	2 548	2 916	200	150	1 687	1 655	1 019
				L-14-B-2				
old	80	8 698	2 959	232	22	3 132	590	369
default tp	164	8 175	2 172	246	8	1 417	515	275
opt sep	167	8 713	2 318	248	12	1 498	601	374
opt	132	8 123	1 840	244	11	1 333	496	286
				L-14-B-3				
old	1 030	6 798	6 526	291	90	6 292	11 629	1 283
default tp	1 099	8 975	5 922	253	63	3 142	10 838	778
opt sep	1 034	12 150	6 514	268	72	3 497	11 212	880
opt	909	9 474	5 247	267	52	2 958	10 351	662
				L-14-B-4				
old	1 891	4 063	6 125	289	108	5 792	11 597	1 118
default tp	2 413	4 170	5 710	298	63	3 026	11 195	794
opt sep	2 010	4 624	6 571	298	75	3 367	11 090	876
opt	2 133	3 972	5 403	285	56	3 032	10 680	717
				L-14-C-1				
old	19	470	414	63	5	53	75	26
default tp	28	500	417	63	4	64	77	36
opt sep	19	462	415	61	6	106	150	45
opt	45	505	387	56	2	56	66	29
				L-14-C-2				
old	185	548	693	29	12	312	297	60
default tp	130	434	485	21	14	111	194	37
opt sep	133	517	518	26	11	112	266	46
opt	179	560	556	22	8	126	221	47
				L-14-C-3				
old	117	767	1 515	169	12	1 149	550	740
default tp	114	689	1 130	166	6	604	451	695
opt sep	175	1 170	1 323	170	14	890	583	614
opt	60	339	1 189	164	10	704	413	608
				total sum				
old	**5 555**	**32 591**	**35 364**	**3 446**	**550**	**31 980**	**35 355**	**13 194**
default tp	**6 869**	**33 896**	**28 787**	**3 195**	**383**	**17 097**	**32 181**	**10 798**
	23.65%	4.00%	-18.60%	-7.28%	-30.36%	-46.54%	-8.98%	-18.16%
opt sep	**5 131**	**37 719**	**30 560**	**3 535**	**425**	**17 644**	**32 316**	**10 200**
	-7.63%	15.73%	-13.58%	2.58%	-22.73%	-44.83%	-8.60%	-22.69%
opt	**5 064**	**32 382**	**25 960**	**3 122**	**357**	**16 464**	**29 623**	**9 913**
	-8.84%	-0.64%	-26.59%	-9.40%	-35.09%	-48.52%	-16.21%	-24.87%

Table 6.5: Remaining design rule violations for different soft track patterns used in BonnRouteDetailed for 14nm instances.

Version	Shorts	Diff Net Spacing	Same Net Spacing	Rectangular Shape Constraint	Minimum Area	Minimum Edge Length	Via Extension	Other
				L-7-1				
old	2 904	3 225	1 063	901	110	1 884	3 317	1 031
default tp	2 796	3 134	809	899	106	1 530	3 294	975
opt sep	2 974	3 357	833	909	123	1 413	3 099	942
opt	2 810	3 358	776	904	122	1 176	3 072	893
				L-7-2				
old	1 477	2 063	1 001	21	32	1 771	1 606	987
default tp	1 407	1 972	732	21	32	1 393	1 502	955
opt sep	1 346	1 899	816	23	19	1 212	1 293	786
opt	1 465	1 965	793	21	25	1 114	1 250	815
				L-7-3				
old	23 213	43 977	3 940	11	346	4 719	6 698	3 521
default tp	21 569	42 295	3 805	5	369	4 444	6 285	3 588
opt sep	18 521	36 141	3 524	19	294	3 814	5 494	3 236
opt	18 117	35 281	3 440	6	273	3 872	5 439	3 255
				L-7-4				
old	19 346	46 792	5 644	20	446	7 163	7 189	5 010
default tp	15 086	38 795	5 138	17	401	6 073	6 345	4 870
opt sep	15 105	37 651	4 816	26	393	5 750	6 205	4 659
opt	13 337	34 441	4 475	23	337	5 404	5 797	4 417
				L-7-5				
old	2 657	12 577	7 532	209	402	13 028	11 780	6 225
default tp	1 800	10 654	6 218	211	362	10 713	11 545	6 401
opt sep	1 953	10 483	6 394	192	378	10 019	10 768	5 821
opt	1 863	11 004	5 759	204	422	8 969	10 264	5 597
				sum RLM-7-A				
old	917	4 518	4 390	25	766	5 728	2 970	3 618
default tp	736	4 009	4 287	25	745	4 991	2 815	3 874
opt sep	768	4 232	4 194	28	755	4 914	2 898	3 698
opt	662	4 000	4 203	36	744	4 960	2 811	3 688
				total sum				
old	**50 514**	**113 152**	**23 570**	**1 187**	**2 102**	**34 293**	**33 560**	**20 392**
default tp	**43 394**	**100 859**	**20 989**	**1 178**	**2 015**	**29 144**	**31 786**	**20 663**
	-14.10%	-10.86%	-10.95%	-0.76%	-4.14%	-15.01%	-5.29%	1.33%
opt sep	**40 667**	**93 763**	**20 577**	**1 197**	**1 962**	**27 122**	**29 757**	**19 142**
	-19.49%	-17.14%	-12.70%	0.84%	-6.66%	-20.91%	-11.33%	-6.13%
opt	**38 254**	**90 049**	**19 446**	**1 194**	**1 923**	**25 495**	**28 633**	**18 665**
	-24.27%	-20.42%	-17.50%	0.59%	-8.52%	-25.66%	-14.68%	-8.47%

Table 6.6: Remaining design rule violations for different soft track patterns used in BonnRouteDetailed for 7nm instances.

use the "opt" version in practice and for all other tests.

Due to the fact that the 7nm instances in our main testbed are relatively unclean and due to the fact that the results vary very much between different instances, we reran all four versions with a newer version of other parts of BonnRouteDetailed on a set of newer and largely clean 7nm instances. This way we can verify that our results are not biased by overfitting to the test instances we used both for tuning and evaluation. Furthermore, it is interesting to evaluate more mature 7nm designs. Results can be found in Table 6.7 and Table 6.8. Note that the newer 7nm instances have a lot fewer remaining design rule violations which is both due to the fact that the instances are cleaner and that the abilities of BonnRouteDetailed to obey design rules have been significantly improved.

Overall routing results for the newer 7nm instances look similar to the results for the older 7nm instances. Runtime decreases similarly, the decrease in number of vias and wire length is slightly smaller, improvements in scenics are comparable. Layer fuses decrease more significantly by 23% instead of by 15% but taper fuses do not decrease. On both 7nm testbeds there are significant improvements, but less than on the 14nm instances. This is likely due to the fact that there are more layers in 14nm where track patterns can be optimized freely than in 7nm.

Looking at the remaining design rule violations, one first notes that the total number of remaining design rule violations is significantly smaller on the newer 7nm instances than both on the older 7nm and the 14nm instances. This is due to the newer version of design rule violation avoiding routines in BonnRouteDetailed and to the cleaner state of the instances. When looking at the relative numbers, improvements in design rule violations are better on the newer 7nm instances than both on the older 7nm instances and on the 14nm instances. Shorts and diff-net spacing violations decrease by roughly 30%, same-net spacing violations even decrease by 80%. Minimum edge length violations decrease by 64% and via extension errors decrease by over 40%. The other error classes decrease to a lesser extent.

Concluding, these results confirm the results on our main testbed. Improvements in run time and overall routing results are smaller in 7nm because there are less layers to optimize but still significant. Improvements in remaining design rule violations are larger on cleaner instances. There are still larger benefits, including in timing, to be expected once all other components of the chip design flow (such as the choice of the used wire models) are fully adapted to our changes.

6.3 Checking Diff-Net Rules

In this section, we present experimental results for the techniques described in Chapter 5. We have tested four different versions. The first version does not precompute and store legality information in the fast grid. Instead, it recomputes legality information on the fly each time it is needed. The fast grid stores and reports legality information for intervals rather than single points. Therefore, we have tested whether it is also beneficial if the version without fast grid calculates legality information for a whole interval at a time. Results show that the lowest run times were achieved when computing legality information for single points rather than intervals. For short intervals (ten points of interest around the query location), small increases in run time were observed (+2%) and for longer intervals

Version	Run Time (hh:mm:ss)	Memory (GB)	Vias (10^6)	Wire Length (m)	Scenics 25	Scenics 50	Scenics 100	Layer Fuses	Taper Fuses
				L-7n-1					
old	1:54:17	28.0	4.1	6.5	3 707	1 017	73	131	1 925
default tp	1:18:39	26.8	4.1	6.5	3 731	1 013	69	75	1 940
opt sep	1:17:26	27.7	4.1	6.5	3 613	1 000	69	99	1 895
opt	1:16:08	27.0	4.1	6.5	3 579	1 000	67	119	1 911
				L-7n-2					
old	2:54:49	36.5	6.6	9.9	4 110	1 007	44	153	2 900
default tp	2:01:56	35.4	6.6	9.9	3 985	1 009	37	122	2 952
opt sep	2:18:24	37.2	6.6	9.9	3 981	999	37	120	2 849
opt	1:59:57	35.7	6.6	9.9	3 941	997	38	129	2 898
				L-7n-3					
old	3:31:54	35.6	6.9	10.0	3 719	438	23	125	2 058
default tp	2:45:01	34.5	6.9	10.0	3 690	430	23	130	2 084
opt sep	2:41:00	35.7	6.9	10.0	3 682	432	24	134	1 982
opt	2:33:12	34.2	6.9	10.0	3 657	425	23	83	2 015
				L-7n-4					
old	4:39:17	41.0	9.4	12.0	8 910	1 220	47	1 067	3 003
default tp	3:19:05	39.7	9.4	12.0	8 681	1 219	50	914	3 012
opt sep	3:22:39	40.9	9.4	12.0	8 558	1 211	48	836	2 944
opt	3:02:08	39.5	9.4	12.0	8 561	1 191	46	848	2 938
				L-7n-5					
old	6:23:14	50.6	11.0	14.7	10 661	1 692	50	1 572	3 872
default tp	3:54:38	47.2	10.9	14.6	10 338	1 632	55	1 313	3 901
opt sep	3:54:03	48.1	10.9	14.6	10 119	1 619	51	1 157	3 831
opt	3:36:24	47.2	10.9	14.6	10 164	1 572	39	1 179	3 885
				L-7n-6					
old	5:26:09	54.5	13.6	15.2	16 248	2 823	147	1 708	3 254
default tp	4:05:38	52.8	13.6	15.2	15 972	2 770	150	1 359	3 301
opt sep	4:01:46	53.8	13.6	15.1	15 670	2 708	149	1 349	3 199
opt	3:45:32	52.9	13.6	15.1	15 655	2 681	140	1 353	3 225
				L-7n-7					
old	8:27:47	62.5	13.7	20.9	13 305	3 023	347	855	6 012
default tp	5:43:15	60.5	13.7	20.9	13 055	2 983	357	744	6 030
opt sep	5:38:13	61.8	13.7	20.9	13 018	2 942	343	694	5 931
opt	4:59:01	59.7	13.7	20.9	12 932	2 933	325	595	5 965
				L-7n-8					
old	9:24:31	87.1	18.9	28.7	25 864	7 209	896	1 091	9 705
default tp	7:22:56	84.9	18.8	28.7	25 744	7 236	869	1 032	9 686
opt sep	7:09:36	87.4	18.8	28.6	25 288	7 103	848	857	9 563
opt	6:36:58	83.8	18.8	28.6	25 306	7 026	842	869	9 648
				sum RLM-7n-A					
old	3:35:39	97.2	15.5	15.5	7 781	1 659	206	71	995
default tp	3:09:18	96.5	15.4	15.4	7 726	1 635	191	58	1 000
opt sep	3:14:07	98.5	15.4	15.4	7 703	1 605	189	51	959
opt	3:08:44	97.0	15.4	15.4	7 703	1 630	194	41	968
				total sum					
old	**46:17:37**	**493.0**	**99.7**	**133.3**	**94 305**	**20 088**	**1 833**	**6 773**	**33 724**
default tp	**33:40:26**	**478.4**	**99.4**	**133.1**	**92 922**	**19 927**	**1 801**	**5 747**	**33 906**
	-27.26%	-2.97%	-0.22%	-0.09%	-1.47%	-0.80%	-1.75%	-15.15%	0.54%
opt sep	**33:37:14**	**491.1**	**99.3**	**133.0**	**91 632**	**19 619**	**1 758**	**5 297**	**33 153**
	-27.38%	-0.38%	-0.31%	-0.16%	-2.83%	-2.33%	-4.09%	-21.79%	-1.69%
opt	**30:58:04**	**476.9**	**99.3**	**133.0**	**91 498**	**19 455**	**1 714**	**5 216**	**33 453**
	-33.11%	-3.26%	-0.36%	-0.20%	-2.98%	-3.15%	-6.49%	-22.99%	-0.80%

Table 6.7: Routing results for different soft track patterns used in BonnRouteDetailed for newer 7nm instances.

Version	Shorts	Diff Net Spacing	Same Net Spacing	Rectangular Shape Constraint	Minimum Area	Minimum Edge Length	Via Extension	Other
				L-7n-1				
old	8	40	213	5	1	38	24	16
default tp	18	49	43	2	2	13	4	26
opt sep	1	19	35	4	1	18	6	22
opt	0	28	31	5	2	11	7	21
				L-7n-2				
old	8	40	208	5	9	35	35	55
default tp	36	83	76	7	12	26	31	56
opt sep	12	63	55	10	7	43	22	48
opt	6	35	68	6	2	25	27	43
				L-7n-3				
old	23	103	186	11	3	36	13	65
default tp	27	118	84	17	8	54	16	64
opt sep	14	87	70	13	6	115	11	47
opt	24	102	72	21	10	39	10	47
				L-7n-4				
old	62	231	261	19	14	78	62	42
default tp	68	164	77	14	19	42	39	54
opt sep	64	192	82	9	12	78	41	36
opt	40	100	68	14	15	31	35	33
				L-7n-5				
old	20	111	700	5	16	118	66	51
default tp	25	77	88	9	16	35	43	59
opt sep	24	104	81	4	7	53	51	45
opt	32	101	79	4	12	26	47	43
				L-7n-6				
old	14	92	379	13	9	66	47	30
default tp	32	187	89	19	13	27	40	49
opt sep	17	48	29	8	9	20	22	28
opt	14	61	32	9	9	18	25	30
				L-7n-7				
old	84	301	426	17	15	118	55	81
default tp	83	325	108	18	9	48	59	82
opt sep	53	224	77	27	15	63	37	82
opt	30	104	59	16	13	33	28	78
				L-7n-8				
old	112	280	872	25	30	112	98	48
default tp	73	280	277	28	37	33	58	66
opt sep	41	125	234	22	28	56	48	53
opt	75	181	225	29	25	33	47	56
				sum RLM-7n-A				
old	86	339	252	30	15	49	18	32
default tp	71	342	61	29	14	23	17	36
opt sep	94	330	63	20	16	19	43	22
opt	80	341	51	24	13	18	20	27
				total sum				
old	**417**	**1 537**	**3 497**	**130**	**112**	**650**	**418**	**420**
default tp	**433**	**1 625**	**903**	**143**	**130**	**301**	**307**	**492**
	3.84%	5.73%	-74.18%	10.00%	16.07%	-53.69%	-26.56%	17.14%
opt sep	**320**	**1 192**	**726**	**117**	**101**	**465**	**281**	**383**
	-23.26%	-22.45%	-79.24%	-10.00%	-9.82%	-28.46%	-32.78%	-8.81%
opt	**301**	**1 053**	**685**	**128**	**101**	**234**	**246**	**378**
	-27.82%	-31.49%	-80.41%	-1.54%	-9.82%	-64.00%	-41.15%	-10.00%

Table 6.8: Remaining design rule violations for different soft track patterns used in BonnRouteDetailed for newer 7nm instances.

Instance	Total Run Time of BonnRouteDetailed (hh:mm:ss (Speedup))			
	No Fast Grid	Fast Grid (Two States)	Fast Grid (Three States)	Full Fast Grid
sum RLM-14-A	13:58:04	6:01:17 (2.3)	5:36:57 (2.5)	5:33:46 (2.5)
sum RLM-14-B	3:28:12	1:29:55 (2.3)	1:24:19 (2.5)	1:24:48 (2.5)
L-14-A-1	4:13:01	2:05:22 (2.0)	1:53:05 (2.2)	1:53:45 (2.2)
L-14-A-2	18:40:07	8:30:11 (2.2)	7:48:55 (2.4)	8:25:05 (2.2)
L-14-A-3	9:07:59	4:35:53 (2.0)	4:07:43 (2.2)	4:34:45 (2.0)
L-14-B-1	11:22:31	5:21:57 (2.1)	5:08:42 (2.2)	5:30:52 (2.1)
L-14-B-2	10:53:19	5:42:04 (1.9)	5:39:58 (1.9)	5:27:54 (2.0)
L-14-B-3	24:27:07	12:28:02 (2.0)	11:14:37 (2.2)	11:03:43 (2.2)
L-14-B-4	21:07:31	11:32:56 (1.8)	10:25:03 (2.0)	10:09:10 (2.1)
L-14-C-1	2:21:07	1:06:48 (2.1)	58:32 (2.4)	1:01:49 (2.3)
L-14-C-2	4:43:25	2:32:22 (1.9)	2:07:28 (2.2)	2:13:57 (2.1)
L-14-C-3	6:37:02	3:26:20 (1.9)	2:54:12 (2.3)	2:50:00 (2.3)
sum 14nm	**130:59:25**	**64:53:07 (2.0)**	**59:19:31 (2.2)**	**60:09:34 (2.2)**
L-7-1	2:06:14	58:01 (2.2)	49:51 (2.5)	48:54 (2.6)
L-7-2	3:03:05	1:36:07 (1.9)	1:28:57 (2.1)	1:34:50 (1.9)
L-7-3	12:59:30	7:15:53 (1.8)	6:16:30 (2.1)	6:35:51 (2.0)
L-7-4	28:47:25	13:44:39 (2.1)	13:02:33 (2.2)	12:21:30 (2.3)
L-7-5	19:18:35	10:41:32 (1.8)	10:59:11 (1.8)	9:27:19 (2.0)
sum RLM-7-A	11:34:12	5:34:12 (2.1)	5:19:23 (2.2)	5:16:07 (2.2)
sum 7nm	**77:49:01**	**39:50:24 (2.0)**	**37:56:25 (2.1)**	**36:04:31 (2.2)**
total sum	**208:48:26**	**104:43:31 (2.0)**	**97:15:56 (2.1)**	**96:14:05 (2.2)**

Table 6.9: Total run time of BonnRouteDetailed and speedup in relation to the version without fast grid.

run time increased more drastically (for 20 points of interest in both directions, +25% and for 50 points of interests run time roughly doubled). Therefore we use a version without fast grid and with computing checking information for single points rather than intervals as baseline.

Similar tests were done for the fast grid, showing that cutting-off the interval around the query location at 20 points of interest in both directions yields the best results, although differences were much smaller in this case. Other cut-off values between 10 and 50 yield only slightly larger run times and even computing legality information for single points results in a moderate increase in run time of roughly 20%. Therefore we choose 20 as the best value for the parameter *max_length* introduced in Section 5.3.

We tested three different versions of the fast grid. The full version which stores one of four different states as described in Chapter 5 as well as two intermediate versions storing two and three states respectively. The version storing two states can only distinguish if some wire model at some location is currently allowed or potentially forbidden. It does not store if something is forbidden by a blockage or forbidden by something that might be ripped-up or might be currently ignored. Therefore, only one bit of memory is needed to store the state but more additional queries to the grid need to be made. The version storing three states distinguishes if something is allowed, always forbidden (because it is illegal due to a blockage) or potentially forbidden. It does not distinguish between objects that can be ripped-up and objects that can only be ignored but not ripped-up. This version uses two bits of memory for the state of each interval but potentially has to store less intervals thus still saving some memory. On the other hand, some additional queries to the grid need to be made.

Table 6.9 shows the total run time of BonnRouteDetailed with the four versions de-

Instance	Run Time of Main Detailed Routing Step (hh:mm:ss (Speedup))			
	No Fast Grid	Fast Grid (Two States)	Fast Grid (Three States)	Full Fast Grid
sum RLM-14-A	12:53:19	4:46:53 (2.7)	4:23:42 (2.9)	4:23:20 (2.9)
sum RLM-14-B	3:04:40	1:06:02 (2.8)	1:00:04 (3.1)	1:00:04 (3.1)
L-14-A-1	3:57:19	1:46:46 (2.2)	1:34:50 (2.5)	1:34:07 (2.5)
L-14-A-2	17:24:46	7:09:09 (2.4)	6:30:31 (2.7)	7:01:05 (2.5)
L-14-A-3	8:07:27	3:32:41 (2.3)	3:09:33 (2.6)	3:31:10 (2.3)
L-14-B-1	10:33:07	4:28:59 (2.4)	4:16:45 (2.5)	4:35:36 (2.3)
L-14-B-2	9:57:43	4:39:28 (2.1)	4:38:04 (2.1)	4:28:04 (2.2)
L-14-B-3	22:58:51	10:43:49 (2.1)	9:32:18 (2.4)	9:25:23 (2.4)
L-14-B-4	19:40:15	9:41:18 (2.0)	8:48:18 (2.2)	8:34:41 (2.3)
L-14-C-1	2:00:17	46:53 (2.6)	37:51 (3.2)	39:39 (3.0)
L-14-C-2	4:16:10	1:59:46 (2.1)	1:40:38 (2.5)	1:43:56 (2.5)
L-14-C-3	6:10:42	2:57:27 (2.1)	2:27:42 (2.5)	2:23:16 (2.6)
sum 14nm	**121:04:36**	**53:39:11 (2.3)**	**48:40:16 (2.5)**	**49:20:21 (2.5)**
L-7-1	1:56:18	47:06 (2.5)	39:31 (2.9)	38:31 (3.0)
L-7-2	2:42:58	1:15:03 (2.2)	1:07:20 (2.4)	1:13:13 (2.2)
L-7-3	12:52:15	7:07:44 (1.8)	6:08:51 (2.1)	6:28:04 (2.0)
L-7-4	28:38:19	13:35:11 (2.1)	12:52:58 (2.2)	12:12:14 (2.3)
L-7-5	18:14:25	9:30:19 (1.9)	9:46:05 (1.9)	8:17:33 (2.2)
sum RLM-7-A	10:42:38	4:39:06 (2.3)	4:26:02 (2.4)	4:22:35 (2.4)
sum 7nm	**75:06:53**	**36:54:29 (2.0)**	**35:00:47 (2.1)**	**33:12:10 (2.3)**
total sum	**196:11:29**	**90:33:40 (2.2)**	**83:41:03 (2.3)**	**82:32:31 (2.4)**

Table 6.10: Run time of main detailed routing step and speedup in relation to the version without fast grid.

scribed above and Table 6.10 shows the run time of the main detailed routing step of BonnRouteDetailed which is the only step that uses the precomputed legality information stored in the fast grid. The total run time of BonnRouteDetailed decreases by a factor of 2.0, 2.1 and 2.2 respectively with the versions with two, three and four states stored in the fast grid. For the larger 14nm instances, the decrease in run time is similar, it is between factor 2.0 and 2.3 for each instance. On the smaller 14nm instances, the decrease is a little bit higher (factor 2.5 on average). For the larger 7nm instances, the decrease is mostly between factor 1.9 and 2.3 with one exception, instance L-7-1 for which run time reduces by a factor of 2.6.

The versions with only two or three states show similar but slightly smaller reductions in run time. If only the main detailed routing step is considered, the speedup is slightly larger, with the full fast grid it then ranges between factor 2.0 and 3.1 with an average of 2.4. The two versions with only two and three states achieve an average run time improvement of factor 2.2 and 2.3 respectively.

Table 6.11 shows the memory consumption of BonnRouteDetailed. Total memory of BonnRouteDetailed only increases very moderately, by about 8% on average with the full fast grid version and slightly less with the other versions. This is due to the fact that the implementation of the fast grid is very efficient and a large fraction of the total memory used by BonnRouteDetailed is used by other parts such as the net data structure and the path search.

All other important metrics like wire length, number of vias, timing behavior and number of remaining design rule violations are not affected by these changes, because this technique only reduces run time but does not change the routing result at all. In total, we gain more than a factor of two in run time for very little additional memory consumption

Instance	Total Memory Usage of BonnRouteDetailed (GB)			
	No Fast Grid	Fast Grid (Two States)	Fast Grid (Three States)	Full Fast Grid
sum RLM-14-A	127.7	132.6 (+4%)	133.4 (+4%)	134.5 (+5%)
sum RLM-14-B	65.6	65.4 (-0%)	67.6 (+3%)	66.6 (+1%)
L-14-A-1	27.5	30.1 (+9%)	29.5 (+7%)	30.3 (+10%)
L-14-A-2	79.0	87.9 (+11%)	89.2 (+13%)	89.4 (+13%)
L-14-A-3	66.6	73.7 (+11%)	74.4 (+12%)	75.0 (+13%)
L-14-B-1	57.4	63.4 (+10%)	64.2 (+12%)	64.6 (+12%)
L-14-B-2	62.8	69.4 (+11%)	70.0 (+12%)	70.5 (+12%)
L-14-B-3	72.8	79.3 (+9%)	80.9 (+11%)	80.9 (+11%)
L-14-B-4	73.2	79.2 (+8%)	80.5 (+10%)	80.5 (+10%)
L-14-C-1	25.1	27.0 (+8%)	26.6 (+6%)	26.6 (+6%)
L-14-C-2	31.5	31.1 (-1%)	31.5 (-0%)	31.4 (-0%)
L-14-C-3	30.4	31.5 (+4%)	31.8 (+4%)	31.4 (+3%)
sum 14nm	**719.8**	**770.8 (+7%)**	**779.6 (+8%)**	**781.7 (+9%)**
L-7-1	18.9	19.7 (+4%)	19.8 (+5%)	19.8 (+5%)
L-7-2	36.5	39.0 (+7%)	39.4 (+8%)	39.4 (+8%)
L-7-3	16.5	16.6 (+1%)	17.0 (+3%)	16.9 (+2%)
L-7-4	20.7	20.9 (+1%)	21.2 (+3%)	21.1 (+2%)
L-7-5	109.8	120.6 (+10%)	121.7 (+11%)	121.8 (+11%)
sum RLM-7-A	125.5	132.1 (+5%)	131.1 (+5%)	131.1 (+5%)
sum 7nm	**327.8**	**349.1 (+6%)**	**350.2 (+7%)**	**350.1 (+7%)**
total sum	**1047.7**	**1119.8 (+7%)**	**1129.8 (+8%)**	**1131.8 (+8%)**

Table 6.11: Total memory usage of BonnRouteDetailed with different version of the fast grid.

and identical results in all other metrics. Therefore we use the full fast grid in practice and for all other tests.

Note that in [34], a similar approach was proposed. [34] reports better speedups on their testbed than we do on ours. However there are some fundamental differences. First, the approach in [34] relies on some assumptions regarding the legality of jogs simplifying the problem whereas we solve it exactly. These assumptions have largely been fulfilled in former technologies but can not be assumed anymore on instances manufactured in recent technologies. Second, we completely re-implemented the simple diff-net rule checking used inside BonnRouteDetailed and largely improved its run time, thus speeding up the baseline. Third, we use a completely different testbed than [34] and also other parts of BonnRouteDetailed have substantially changed in the meantime. More run time is spent in parts that are independent of the fast grid. Thus the possible speedup by using the fast grid is smaller.

We did not rerun these experiments on the newer 7nm instances, as these results are very stable across different instances and as it would be technically non-trivial to port the implementation of the intermediate versions to the newest version of other parts of BonnRouteDetailed needed for the newer instances.

6.4 Parallelization

In this section, we present results of BonnRouteDetailed running with different numbers of threads. Total run time of these experiments is very high, so we only present results of BonnRouteDetailed using the best versions of the algorithms described in Chapter 4 and

Chapter 5. Our algorithms run within the parallelization framework initially developed in [27]. Note that unlike in [27], we run BonnRouteDetailed with each number of threads on the machine while nothing else is running. Therefore, our speedups are more reliable, but should on average also be smaller compared to [27]. Further note that since [27] additional steps of BonnRouteDetailed have been parallelized and that instances tested in [27] and our testbed can not be compared.

First, we briefly discuss our testbed. Due to the high run times, we chose a few representative instances from each technology for our parallelization experiments. We further reran with a newer version of other parts of BonnRouteDetailed on some newer 7nm instances to assess if parallelization speedup is influenced by technology or by the state of the instances. We only use medium size or large instances for parallelization experiments, because run times on small instances are small and thus less interesting. We chose RLM-14-A-11, L-14-A-1, L-14-A-2 and L-14-A-3 from the 14nm testcases. This includes the largest 14nm instance that we have (L-14-A-2) and the instances L-14-A are the cleanest large 14nm instances available. We added RLM-14-A-11 as an example of an easy and clean instance. From the main 7nm testbed, we chose L-7-1, L-7-2 and L-7-5. These are the three largest instances of the set L-7, again inlcuding the largest 7nm instance available. From the newer 7nm instances, we chose L-7n-2, L-7n-6 and L-7n-7. We did not include the largest instance here due to run time reasons, but included the second and third largest instances available as well as a smaller one. Most of these instances are congested, but not over-congested. L-7n-2, L-14-A-1, RLM-14-A-11 and L-7-2 however are generally uncongested. L-14-A-1 and L-14-A-2 have some smaller hotspots with over-congestion.

We run the same code multi-threaded and single-threaded, in particular locking and collision detection is enabled also single-threaded. This simplifies the implementation and makes debugging easier because the same code is used in a single-threaded debug run as in a multi-threaded production run. In practice, BonnRouteDetailed is almost always run multi-threaded anyway. Further, we made a single-threaded test run on our parallelization testbed, disabling locking and collision detection. Run time improved only very slightly by less than four percent.

Total run times of BonnRouteDetailed with up to 64 threads can be found in Table 6.12(a). Up to 16 threads, multi-threaded run times are very good. With 4 threads, we achieve a speedup of factor 3.6 on average, with 16 threads of 12.6. With 32 threads there is still a speedup of 19.1 and of 24.1 with 64 threads. The 14nm instances consistently achieve higher speedups than 7nm instances. For example, with 64 threads, the average speedup for 14nm instances is 31.1 but only 18.0 for 7nm instances. Less congested instances parallelize worse than congested instances. When looking only at the main detailed routing step, the results are even more impressive. Table 6.12(b) shows run times of the main detailed routing step of BonnRouteDetailed. With 64 threads, we achieve an average speedup of factor 31.2. Again, 14nm instances achieve higher speedups (factor 40.0 on average) than 7nm instances (factor 23.8 on average). Especially for 14nm instances, speedups with up to 16 threads are very close to optimal (14nm instances achieve an average speedup of 15.8 with 16 threads). The speedup of L-14-A-1 with 16 threads is slightly higher than the number of threads. This is due to the nondeterminism involved.

Note that the parallelization in BonnRouteDetailed does not depend on the number

Instance	Total Run Time of BonnRouteDetailed (hh:mm:ss (Speedup))						
	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads	64 Threads
RLM-14-A-11	3:45:01	1:59:17 (1.9)	1:01:18 (3.7)	34:36 (6.5)	17:42 (12.7)	12:06 (18.6)	11:26 (19.7)
L-14-A-1	20:49:01	11:34:15 (1.8)	5:39:13 (3.7)	3:24:54 (6.1)	1:26:21 (14.5)	59:53 (20.9)	56:22 (22.2)
L-14-A-2	92:22:49	48:27:54 (1.9)	24:36:41 (3.8)	13:01:41 (7.1)	6:12:40 (14.9)	4:02:17 (22.9)	2:32:34 (36.3)
L-14-A-3	49:39:52	26:24:32 (1.9)	13:11:12 (3.8)	7:21:57 (6.7)	3:36:44 (13.7)	2:19:23 (21.4)	1:40:34 (29.6)
sum 14nm	**166:36:43**	**88:25:58 (1.9)**	**44:28:24 (3.7)**	**24:23:08 (6.8)**	**11:33:27 (14.4)**	**7:33:39 (22.0)**	**5:20:56 (31.1)**
L-7-1	8:25:29	4:24:45 (1.9)	2:21:19 (3.6)	1:21:21 (6.2)	46:39 (10.8)	30:45 (16.4)	24:30 (20.6)
L-7-2	14:03:56	7:26:45 (1.9)	4:29:28 (3.1)	2:49:48 (5.0)	1:28:35 (9.5)	1:01:10 (13.8)	1:01:05 (13.8)
L-7-5	87:24:48	46:04:47 (1.9)	25:05:28 (3.5)	15:58:56 (5.5)	8:11:13 (10.7)	5:24:55 (16.1)	4:40:43 (18.7)
sum 7nm	**109:54:13**	**57:56:17 (1.9)**	**31:56:15 (3.4)**	**20:10:05 (5.4)**	**10:26:27 (10.5)**	**6:56:50 (15.8)**	**6:06:18 (18.0)**
total sum	**276:30:56**	**146:22:15 (1.9)**	**76:24:39 (3.6)**	**44:33:13 (6.2)**	**21:59:54 (12.6)**	**14:30:29 (19.1)**	**11:27:14 (24.1)**

(a) Total run time of BonnRouteDetailed and speedup in relation to one thread.

Instance	Run Time of Main Detailed Routing Step (hh:mm:ss (Speedup))						
	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads	64 Threads
RLM-14-A-11	2:31:47	1:21:31 (1.9)	40:41 (3.7)	22:04 (6.9)	10:12 (14.9)	6:37 (22.9)	5:58 (25.4)
L-14-A-1	18:30:17	10:18:23 (1.8)	4:57:34 (3.7)	2:58:12 (6.2)	1:09:03 (16.1)	45:44 (24.3)	43:22 (25.6)
L-14-A-2	79:39:00	42:24:11 (1.9)	21:27:04 (3.7)	11:02:07 (7.2)	4:59:25 (16.0)	3:04:23 (25.9)	1:43:15 (46.3)
L-14-A-3	41:03:54	21:41:33 (1.9)	10:41:11 (3.8)	5:54:02 (7.0)	2:38:07 (15.6)	1:33:12 (26.4)	1:00:14 (40.9)
sum 14nm	**141:44:58**	**75:45:38 (1.9)**	**37:46:30 (3.8)**	**20:16:25 (7.0)**	**8:56:47 (15.8)**	**5:29:56 (25.8)**	**3:32:49 (40.0)**
L-7-1	7:21:47	3:48:46 (1.9)	2:00:04 (3.7)	1:07:10 (6.6)	36:25 (12.1)	21:33 (20.5)	13:04 (33.8)
L-7-2	11:49:09	6:12:00 (1.9)	3:44:28 (3.2)	2:20:11 (5.1)	1:07:33 (10.5)	41:27 (17.1)	35:59 (19.7)
L-7-5	80:51:47	42:20:06 (1.9)	22:52:37 (3.5)	14:28:10 (5.6)	7:03:51 (11.4)	4:18:41 (18.8)	3:22:59 (23.9)
sum 7nm	**100:02:43**	**52:20:52 (1.9)**	**28:37:09 (3.5)**	**17:55:31 (5.6)**	**8:47:49 (11.4)**	**5:21:41 (18.7)**	**4:12:02 (23.8)**
total sum	**241:47:41**	**128:06:30 (1.9)**	**66:23:39 (3.6)**	**38:11:56 (6.3)**	**17:44:36 (13.6)**	**10:51:37 (22.3)**	**7:44:51 (31.2)**

(b) Run time of main detailed routing step of BonnRouteDetailed and speedup in relation to one thread.

Table 6.12: Run time with different numbers of threads.

of threads to be a power of two in any way. With 63 threads we achieve roughly the same speedup of the total BonnRouteDetailed run time than with 64 threads (factor 24.3 on average on all instances, 31.8 and 17.9 on 14nm and 7nm instances respectively) and with 48 threads we achieve a factor 22.8 on average (28.5 and 17.5 on 14nm and 7nm instances respectively).

Memory usage naturally increases with the number of threads, but only moderately. With 8 threads, BonnRouteDetailed uses 9% additional memory, with 16 threads 14%. With 32 threads, 22% of additional memory are needed and even with 64 threads, only 41% of additional memory are required.

Table 6.13 shows routing results for different number of threads. Table 6.13(a) shows key statistics such as number of vias, wire length and some timing related key figures. There are always some fluctuations in these numbers due to non-determinism, but there are no significant changes in any of these numbers. Some of them tend to become even slightly better with more threads, and certainly not worse. Table 6.13(b) shows remaining design rule violations with different number of threads. There is a slight increase in diff-net spacing violations and shorts with more threads. With 64 threads, shorts increase by almost 5% and diff-net spacing violations by about 4% compared to the single-threaded run. The other error classes show no clear trend, there are some fluctuations but nothing significant. Overall, there is only a very slight degradation of results with 64 threads compared to one thread (and even less with fewer threads).

Due to the fact that the 7nm instances in our main testbed parallelize so much worse than the more mature 14nm instances, we also reran these experiments on some of the newer 7nm instances. The results can be found in Table 6.14(a) for the total BonnRouteDetailed run time and in Table 6.14(b) for the main detailed routing step. Both for the total run time and for the main detailed routing step, the newer 7nm instances parallelize better than the older 7nm instances but worse than the 14nm instances. This to some extent supports our expectation that more mature, less unclean instances parallelize better. However, there is still a gap between the mostly clean 7nm and the 14nm instances which might be the subject of future research.

Overall, BonnRouteDetailed achieves very good parallel speedups with our algorithms with almost identical results even with 64 threads.

Number of Threads	Vias (10^6)	Wire Length (m)	Scenics 25	Scenics 50	Scenics 100	Layer Fuses	Taper Fuses
1	70.5	120.0	54 281	12 324	2 153	2 116	14 889
2	70.5	120.0	54 295	12 322	2 131	2 173	14 859
	-0.00%	-0.00%	0.03%	-0.02%	-1.02%	2.69%	-0.20%
4	70.5	120.0	54 203	12 259	2 109	2 131	14 746
	-0.00%	-0.00%	-0.14%	-0.53%	-2.04%	0.71%	-0.96%
8	70.5	120.0	54 340	12 333	2 137	2 157	14 792
	-0.00%	-0.00%	0.11%	0.07%	-0.74%	1.94%	-0.65%
16	70.5	120.0	54 331	12 283	2 137	2 071	14 826
	-0.02%	-0.00%	0.09%	-0.33%	-0.74%	-2.13%	-0.42%
32	70.5	120.0	54 197	12 284	2 115	2 082	14 683
	-0.01%	-0.01%	-0.15%	-0.32%	-1.76%	-1.61%	-1.38%
48	70.5	120.0	54 067	12 246	2 159	2 118	14 870
	-0.01%	-0.01%	-0.39%	-0.63%	0.28%	0.09%	-0.13%
63	70.5	120.0	54 104	12 332	2 119	2 113	14 825
	-0.02%	-0.01%	-0.33%	0.06%	-1.58%	-0.14%	-0.43%
64	70.5	120.0	54 182	12 263	2 130	2 064	14 758
	-0.01%	-0.01%	-0.18%	-0.49%	-1.07%	-2.46%	-0.88%

(a) General results.

Number of Threads	Shorts	Diff Net Spacing	Same Net Spacing	Rectangular Shape Constraint	Minimum Area	Minimum Edge Length	Via Extension	Other
1	6 419	20 995	13 706	2 027	595	17 099	18 917	10 085
2	6 545	21 144	13 679	2 030	611	17 099	18 963	10 054
	1.96%	0.71%	-0.20%	0.15%	2.69%	0.00%	0.24%	-0.31%
4	6 331	21 075	13 546	2 038	581	17 071	18 957	10 081
	-1.37%	0.38%	-1.17%	0.54%	-2.35%	-0.16%	0.21%	-0.04%
8	6 564	21 573	13 663	2 006	595	17 118	18 998	10 138
	2.26%	2.75%	-0.31%	-1.04%	0.00%	0.11%	0.43%	0.53%
16	6 473	21 462	13 705	2 064	628	17 317	18 930	10 049
	0.84%	2.22%	-0.01%	1.83%	5.55%	1.27%	0.07%	-0.36%
32	6 691	21 845	13 598	2 014	601	17 192	19 039	10 118
	4.24%	4.05%	-0.79%	-0.64%	1.01%	0.54%	0.64%	0.33%
48	6 401	21 423	13 479	2 025	606	16 968	18 878	10 115
	-0.28%	2.04%	-1.66%	-0.10%	1.85%	-0.77%	-0.21%	0.30%
63	6 795	21 496	13 513	1 999	589	17 183	18 886	10 134
	5.86%	2.39%	-1.41%	-1.38%	-1.01%	0.49%	-0.16%	0.49%
64	6 726	21 785	13 632	1 999	565	17 293	18 949	10 223
	4.78%	3.76%	-0.54%	-1.38%	-5.04%	1.13%	0.17%	1.37%

(b) Remaining design rule violations.

Table 6.13: Routing results with different numbers of threads (summed over all instances).

Instance	Total Run Time of BonnRouteDetailed (hh:mm:ss (Speedup))						
	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads	64 Threads
L-7n-2	20:07:01	11:05:58 (1.8)	5:50:58 (3.4)	3:03:57 (6.6)	1:45:07 (11.5)	1:16:38 (15.8)	1:03:33 (19.0)
L-7n-6	40:10:11	20:34:21 (2.0)	10:53:33 (3.7)	5:46:33 (7.0)	3:09:03 (12.7)	2:15:43 (17.8)	1:49:04 (22.1)
L-7n-7	52:40:53	28:49:03 (1.8)	14:56:34 (3.5)	7:54:01 (6.7)	4:29:35 (11.7)	3:05:22 (17.1)	2:26:20 (21.6)
total sum	**112:58:05**	**60:29:22 (1.9)**	**31:41:05 (3.6)**	**16:44:31 (6.7)**	**9:23:45 (12.0)**	**6:37:43 (17.0)**	**5:18:57 (21.3)**

(a) Total run time of BonnRouteDetailed and speedup in relation to one thread.

Instance	Run Time of Main Detailed Routing Step (hh:mm:ss (Speedup))						
	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads	64 Threads
L-7n-2	17:21:51	9:30:08 (1.8)	4:54:15 (3.5)	2:28:05 (7.0)	1:18:27 (13.3)	52:04 (20.0)	36:15 (28.7)
L-7n-6	35:09:45	17:47:25 (2.0)	9:15:48 (3.8)	4:44:37 (7.4)	2:24:31 (14.6)	1:32:41 (22.8)	57:02 (37.0)
L-7n-7	47:35:17	25:51:59 (1.8)	13:10:19 (3.6)	6:45:28 (7.0)	3:37:03 (13.2)	2:18:30 (20.6)	1:31:56 (31.1)
total sum	**100:06:53**	**53:09:32 (1.9)**	**27:20:22 (3.7)**	**13:58:10 (7.2)**	**7:20:01 (13.7)**	**4:43:15 (21.2)**	**3:05:13 (32.4)**

(b) Run time of main detailed routing step of BonnRouteDetailed and speedup in relation to one thread.

Table 6.14: Run time with different numbers of threads on newer 7nm instances.

Summary

The subject of this thesis is detailed routing of very large scale integrated (VLSI) circuits. This task is very challenging due to the enormous instance sizes as well as the theoretical hardness of the problem. Large instances can include several million nets which are to be packed into a graph with hundreds of billions of nodes. Even much simpler sub-problems like finding a minimum length Steiner tree or the vertex disjoint path problem on grid graphs are NP-hard. Furthermore, complex design rules and other side constraints need to be obeyed.

We improve main components of BonnRouteDetailed, the detailed routing tool developed at the Research Institute for Discrete Mathematics at the University of Bonn in a cooperation with IBM. It has been and is used by IBM to successfully develop some of the most complex and powerful processor chips of the world. We give a comprehensive overview of BonnRouteDetailed and briefly describe the most important parts of this state-of-the-art detailed routing tool.

In this thesis we describe two key contributions in detail. First, we discuss the problem of generating soft track patterns automatically. We discuss different objectives and combine them into a simple and effective objective function. We present an efficient algorithm computing optimal solutions for up to three track patterns and describe how it can be used to compute high-quality soft track patterns for BonnRouteDetailed.

Second, we develop an axiomatic description of diff-net rules and derive important properties. We use this theoretical foundation to describe diff-net rules that can be effectively handled by our routing tool. We develop an efficient algorithm for checking such diff-net rules and present a highly optimized implementation inside the parallelization framework of BonnRouteDetailed.

Last but not least, we present excellent experimental results with both our novel automatic soft track pattern generation as well as our fast implementation of diff-net rules inside BonnRouteDetailed. Our automatic soft track patterns generation does not only save the time to manually provide hard-coded soft track pattern and make the detailed routing flow much more robust, they also lead to clearly superior results. Run time is reduced by 40% and 33% on 14nm and 7nm instances respectively. Almost all metrics to measure the quality of results improve significantly. Wire length and number of vias decrease, there are fewer scenics and fewer layer and taper fuses. Also, the number of remaining design rule violations decreases substantially.

Our efficient diff-net rule checking implementation speeds up the total run time of BonnRouteDetailed by more than a factor 2. Furthermore, unlike the previous approach, BonnRouteDetailed is now able to check many diff-net rules exactly. Parallelization of

this implementation is very good, the main routing step of BonnRouteDetailed achieves a speedup of factor 40 with 64 threads on 14nm instances and a speedup of factor 32 with 64 threads on newer 7nm instances.

Together, our contributions substantially improve the run time of BonnRouteDetailed as well as the quality of routing solutions computed by BonnRouteDetailed and in particular improve its ability do deal with complex diff-net rules exactly as well as pack wires of different widths efficiently.

Glossary

A

$\mathcal{A}$ The chip area. (Page 6.)

$applies(s_1, s_2, l, sc_1, sc_2, cd, sp_1, sp_2)$ A helper function returning if the two given shapes s_1 and s_2 have the given layer l, shape classes sc_1 and sc_2, color dependency cd and shape purposes sp_1 and sp_2. (Page 94.)

$\mathcal{AS}_t$ Data structure to store temporary additional shapes for thread t. (Pages 31, 98.)

$attr(s)$ The attributes of a shape s. These include the layer, shape class, color and the shape purpose. (Page 7.)

B

$\bar{b}_{rel}(cand)$ Average relative number of blocked tracks of a track pattern candidate $cand$. (Page 55.)

$\bar{b}_{rel}(wm_1, T_1, wm_2, T_2)$ Average relative number of tracks of wire model wm_2 with track pattern T_2 that are blocked by a track of wire model wm_1 with track pattern T_1. (Page 55.)

$bbox(R)$ The bounding box of a set of rectangles R. (Page 6.)

$b(DR)$ The maximum of $b(dr)$ over all distance rules dr in DR. (Page 88.)

$b(dr)$ The locality constant of a diff-net rule dr is the smallest integer b such that for any two shapes s_1, s_2 with $d_{max}(s_1, s_2) > b$ it returns $true$. (Page 88.)

$b_{min}(wm_1, wm_2)$ Minimum number of tracks of wire model wm_2 that a track of wire model wm_1 can block if the tracks of wire model wm_2 are packed densest possible (without regard of the power rails). (Page 52.)

$\bar{bp}_{rel}(cand)$ Modified version of $\bar{b}_{rel}(cand)$ for partial candidates, dividing by the desired total numbers of tracks instead of the current numbers. (Page 62.)

$b_{sc_1,sc_2,l}(DR)$ The maximum of $b_{sc_1,sc_2,l}(dr)$ over all distance rules dr in DR. (Page 88.)

$b_{sc_1,sc_2,l}(dr)$ The same as $b(dr)$ but only shapes on layer l and for the first and second shape only shapes with shape class sc_1 and sc_2 respectively are considered. (Page 88.)

$b_{sc,l}(DR)$ The maximum of $b_{sc,l}(dr)$ over all distance rules dr in DR. (Page 88.)

$b_{sc,l}(dr)$ The same as $b(dr)$ but only shapes on layer l and for the first shape only shapes with shape class sc are considered. (Page 88.)

$\tilde{b}_{min}(wm_1, wm_2)$ Modified version of $b_{min}(wm_1, wm_2)$ which is strictly greater than zero. (Page 54.)

$b(wm_1, T_1, wm_2, T_2)$ Sum of the number of tracks that are blocked by all tracks in T_1 of wire model wm_1 with regard to wire model wm_2 and track pattern T_2. (Page 54.)

$b(wm_1, t, wm_2, T_2)$ Number of tracks that track t of wire model wm_1 blocks with regard to wire model wm_2 and track pattern T_2. (Page 54.)

C

$c2c_d^p(s_1, s_2)$ The center to center diff-net rule requiring the centers of two shapes to be at least d apart measured in the p-norm. This diff-net rule is not monotone. (Page 96.)

$CAND(c)$ Set of (partial) track pattern candidates considered at coordinate c. The contained track pattern candidates contain tracks up to c. (Page 56.)

$\mathcal{CD}$ The set of all color dependencies. Diff-net rules can depend on the fact that both shapes have the same, a different or arbitrary colors. (Page 94.)

$\chi_{wm_1,wm_2}(t_1, t_2)$ Function indicating if two tracks t_1 and t_2 of two wire models wm_1 and wm_2 block each other. If they do, returns 1, and 0 otherwise. (Page 54.)

$\mathcal{COL}$ The set of all colors. (Page 6.)

consistent A diff-net rule is consistent if whenever some shape is illegal with respect to another shape and it can be split into parts then at least one part is also illegal. (Page 89.)

D

diff-net rule A function taking two shapes and returning a Boolean value such that the function is symmetric, local and invariant under translation. (Page 89.)

$dist_d^p(s_1, s_2)$ The diff-net rule requiring that shapes have at least distance d measured by the p-norm. (Page 92.)

DR The set of all diff-net rules on the given chip. (Page 100.)

$\mathcal{DR}$ The set of all diff-net rules. (Page 89.)

E

$\mathcal{ELS}$ The set of all extended legality states. (Page 106.)

$els(s)$ The extended legality state of shape s. s can be legal, blocked by something that is removable, blocked by something that is ignorable or unconditionally illegal. (Page 106.)

$els(st)$ The extended legality state of stick st. st can be legal, blocked by something that is removable, blocked by something that is ignorable or unconditionally illegal. (Page 106.)

E^v All via edges of the track graph. (Page 104.)

E^w All wire edges (in x- or y-direction) of the track graph. (Page 104.)

F

fast grid Data structure inside BonnRouteDetailed caching legality information for most frequently used wire and via models. (Pages 10, 18.)

$fgc(e_w, wm)$ The color used by the fast grid for the wire edge e_w with wire model wm. (Page 105.)

$fgc_b(e_v, vm)$ The color used by the fast grid for the bottom shape of the via edge e_v with via model vm. (Page 105.)

$fgc_m(e_v, vm)$ The color used by the fast grid for the middle shape of the via edge e_v with via model vm. (Page 105.)

$fgc_t(e_v, vm)$ The color used by the fast grid for the top shape of the via edge e_v with via model vm. (Page 105.)

$fgi(e_v, vm)$ The extended legality information stored in the fast grid for via edge e_v and via model vm. (Page 107.)

$fgi(e_w, wm)$ The extended legality information stored in the fast grid for wire edge e_w and wire model wm. (Page 107.)

$fgvm(l)$ The set of via models for which fast grid information is stored on layer l. (Page 107.)

$fgwm(l)$ The set of wire models for which fast grid information is stored on layer l. (Page 107.)

FINAL Set of final track pattern candidates (no track can be added anymore to any wire model). (Page 56.)

$f(wm)$ Estimated relative frequency of wire model wm. (Page 43.)

G

grid Data structure to store all shapes of a chip in BonnRouteDetailed. Allows efficient parallel geometric queries and parallel updates. (Pages 9, 16.)

$G = (V, E)$ The track graph. (Page 103.)

H

hard track pattern Track pattern that is mandatory. (Pages 9, 41.)

homogeneous A set of shapes is called homogeneous if all shapes have the same attributes. (Page 7.)

$hor_{a,d}(s_1, s_2)$ The horizontal diff-net rule requiring that shapes have distance at least d in x-direction if they have run-length at least a in y-direction. (Page 93.)

I

invariant under representation A diff-net rule is invariant under representation if changing the representation of some metal area by shapes does not influence the result with respect to other shapes. (Page 89.)

invariant under translation A function taking two shapes and returning a Boolean value is invariant under translation if moving both shapes equally does not change the result. (Page 88.)

$\mathcal{IS}_t$ Data structure to store shapes temporarily ignored by thread t. (Pages 31, 98.)

L

$\mathcal{L}$ The set of all layers. (Page 5.)

layer fuse Timing-wise undesirable configuration when an intermediate piece of wire is on a low layer. (Page 38.)

local A function taking two shapes and returning a Boolean value is local if for far enough shapes it always returns *true*. (Page 88.)

$lt(T)$ Last track of a track pattern candidate T for some wire model. (Page 54.)

$\mathcal{L}_{via}$ The set of all via layers. (Page 5.)

$\mathcal{L}_{wiring}$ The set of all wiring layers. (Page 5.)

M

monotone A diff-net rule is monotone if whenever a shape is legal with regard to another shape then so is any subshape. (Page 89.)

msp A function mapping a shape purpose to the resulting extended legality state if something is blocked by a shape with the given shape purpose. (Page 106.)

N

$n(T)$ Number of tracks of a track pattern candidate T for some wire model. (Page 54.)

$n_t(wm)$ Number of tracks to be computed for wire model wm. (Page 62.)

$n(wm)$ Number of tracks of wire model wm that fit between two consecutive power rails. (Page 45.)

O

$obj(cand)$ Objective function of a track pattern candidate *cand*. (Page 55.)

$o_n(cand)$ Term of the objective function regarding number of tracks of a track pattern candidate *cand*. (Page 55.)

P

p Power rail pitch. (Page 42.)

pin fuse Timing-wise undesirable configuration using another pin as a part of the connection of some pin to the electrical source. (Page 38.)

$pref(l)$ The preferred direction of layer l. (Page 5.)

R

$\mathcal{R}$ The set of all closed, two-dimensional, axis-parallel rectangles. (Page 6.)

$\mathcal{R}_0$ The set of all closed, two-dimensional, axis-parallel rectangles containing $(0, 0)$. (Page 6.)

$\mathcal{R}_{stick}$ The set of all zero- or one-dimensional rectangles. (Page 6.)

$run\_length(s_1, s_2)$ The maximum of the run length in x and y direction. (Page 92.)

$run\_length_x(s_1, s_2)$ The length of the intersection of the projections of the shapes s_1 and s_2 onto the x-axis or -1 times the distance between the projections. (Page 92.)

$run\_length_y(s_1, s_2)$ The length of the intersection of the projections of the shapes s_1 and s_2 onto the y-axis or -1 times the distance between the projections. (Page 92.)

S

$\mathcal{S}$ The set of all shapes. (Page 7.)

$\mathcal{S}_b$ The set of all blockage shapes. (Page 7.)

$\bar{s}_p(wm)$ Simplified spacing between the stick of a wire running in preferred direction of wire model wm and a power rail. (Page 44.)

$\bar{s}(wm_1, wm_2)$ Simplified spacing between the sticks of two wires running in preferred direction of wire models wm_1 and wm_2 used for soft track pattern calculation. (Page 44.)

$\mathcal{SC}$ The set of all shape classes. (Page 6.)

$\mathcal{SCAS}_t^l$ The set of shape classes of all additional shapes on layer l of thread t. (Page 98.)

$\mathcal{SCIS}_t^l$ The set of shape classes of all ignored shapes on layer l of thread t. (Page 98.)

$\mathcal{SC}_o$ The set of shape classes which are used for shapes on wiring layers. (Page 6.)

$\mathcal{SC}_v$ The set of all shape classes which are exclusively used for via middle shapes. (Page 6.)

$\mathcal{SG}$ The set of shapes that are currently stored in the grid. (Page 97.)

shape class Inside BonnRouteDetailed, every shape is assigned a shape class encoding its distance rule requirements. (Pages 6, 10, 23.)

$slack(T_{wm}, c)$ Slack of a (partial) track pattern candidate T_{wm} for some wire model wm and some current coordinate c. (Page 57.)

$slack(wm)$ Additional space between two consecutive power rails that can not be used to place wires of wire model wm. (Page 46.)

s **legal** Shape s is legal if it fulfills all diff-net rules to all shapes in the grid. (Page 98.)

s **legal**$_t$ Shape s is legal with respect to thread t if it fulfills all diff-net rules to all shapes in the grid, without shapes ignored by thread t and also to all additional shapes of thread t. (Page 98.)

s **legal**$_t$ **with rip-up** Shape s is legal with rip-up with respect to thread t if it is legal with respect to thread t if all wire shapes are ignored. (Page 98.)

s **legal with rip-up** Shape s is legal with rip-up if it is legal if all wire shapes are ignored. (Page 98.)

soft track pattern Track pattern which is not mandatory but used as a recommendation for efficient detailed routing. (Pages 9, 41.)

$\mathcal{SP}$ The set of shape purposes. Shapes can be used as wires, blockages or pins. (Page 6.)

$\mathcal{S}_p$ The set of all pin shapes. (Page 7.)

$s_p(wm)$ Simplified spacing between the shape of a wire running in preferred direction of wire model wm and a power rail. (Page 43.)

$\mathcal{ST}$ The set of all stick figures or short sticks. (Page 8.)

$stick(e_v, vm)$ The stick corresponding to via edge e_v with via model vm and the corresponding fast grid colors. (Page 105.)

$stick(e_w, wm)$ The stick corresponding to wire edge e_w with wire model wm and the corresponding fast grid color. (Page 105.)

st **legal** Stick st is legal if all its shapes are. (Page 98.)

st **legal**$_t$ Stick st is legal with respect to thread t if all its shapes are. (Page 98.)

st **legal**$_t$ **with rip-up** Stick st is legal with rip-up with respect to thread t if all its shapes are. (Page 99.)

st **legal with rip-up** Stick st is legal with rip-up if all its shapes are. (Page 99.)

$\mathcal{S}_w$ The set of all wire shapes. (Page 7.)

$s(wm_1, wm_2)$ Simplified spacing between the shapes of two wires running in preferred direction of wire models wm_1 and wm_2 used for soft track pattern calculation. (Page 43.)

symmetric A function taking two shapes and returning a Boolean value is symmetric if the order of its arguments does not matter. (Page 88.)

T

$\mathcal{T}$ The set of all threads. (Page 8.)

taper fuse Timing-wise undesirable configuration switching back and forth between thin and wide wire models. (Page 38.)

t_f First possible position to place a track for any wire model. (Page 56.)

$t_f(wm)$ First possible position to place a track for wire model wm. (Page 52.)

$t_i(T)$ i-th track of a track pattern candidate T for some wire model. (Page 54.)

t_l Last possible position to place a track for any wire model. (Page 56.)

$t_l(wm)$ Last possible position to place a track for wire model wm. (Page 52.)

$\mathcal{TPC}$ Set of all track pattern candidates (each track pattern candidate consists of a set of tracks for each wire model). (Page 54.)

$\mathcal{TPC}_1$ Set of all track pattern candidates that lose at most one track for each wire model. (Page 56.)

$\mathcal{TPC}_{wm}$ Set of all track pattern candidates for wire model wm. (Page 54.)

$\mathcal{TR}$ The set of all sets of tracks. (Page 8.)

TR_l Tracks of the given chip on layer l. (Page 103.)

T_{wm} Given track pattern for wire model $wm \in \mathcal{TPC}_{wm}$. (Page 54.)

$T_{wm}(cand)$ Tracks for wire model wm of track pattern candidate $cand$. (Page 54.)

V

$ver_{a,d}(s_1, s_2)$ The vertical diff-net rule requiring that shapes have distance at least d in y-direction if they have run-length at least a in x-direction. (Page 93.)

via stick figure A via stick figure or short via stick represents a via connecting two neighboring wiring layers. (Page 8.)

$\mathcal{VM}$ The set of all via models. (Page 7.)

$\mathcal{VS}$ The set of all via stick figures. (Page 8.)

W

wire stick figure A wire stick figure or short wire stick represents an axis parallel segment of a wire on a given layer. (Page 7.)

WM Set of wire models relevant for track pattern calculation. (Page 42.)

WM_i Set of wire models with given track pattern to be considered in objective function for track pattern calculation. (Page 42.)

$\mathcal{WM}$ The set of all wire models. (Page 7.)

WM_o Set of wire models to be optimized for track pattern calculation. (Page 42.)

w_n A parameter governing the tradeoff between maximizing the number of tracks for each wire model and minimizing the dependency between different tracks. (Page 55.)

w_p Power rail width. (Page 42.)

$\mathcal{WS}$ The set of all wire stick figures. (Page 7.)

$w(wm)$ Width of the wire model wm. (Page 43.)

Bibliography

[1] M. Ahrens. Efficient Algorithms for Routing a Net Subject to VLSI Design Rules. PhD Thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2020 (cit. on pp. 11, 29, 34, 36).

[2] M. Ahrens. Pin Access in VLSI Routing. Master's Thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2014 (cit. on pp. 10, 25, 26).

[3] M. Ahrens, M. Gester, N. Klewinghaus, D. Müller, S. Peyer, C. Schulte, and G. Téllez. Detailed Routing Algorithms for Advanced Technology Nodes. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.4 (2015), pp. 563–576 (cit. on pp. 9, 34, 36).

[4] C. J. Alpert, Z. Li, M. D. Moffitt, G.-J. Nam, J. A. Roy, and G. Téllez. What Makes a Design Difficult to Route. In: *Proceedings of the 19th International Symposium on Physical Design*. Association for Computing Machinery, 2010, pp. 7–12 (cit. on p. 125).

[5] C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, eds. Handbook of Algorithms for Physical Design Automation. Auerbach Publications, 2008 (cit. on p. 2).

[6] S. Batterywala, N. Shenoy, W. Nicholls, and H. Zhou. Track Assignment: A Desirable Intermediate Step between Global Routing and Detailed Routing. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. Association for Computing Machinery, 2002, pp. 59–66 (cit. on p. 2).

[7] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. Computational Geometry: Algorithms and Applications. Third revised Edition. Springer, 2008 (cit. on p. 20).

[8] D. R. Butenhof. Programming With Posix Threads. Addison-Wesley, 1997 (cit. on p. 20).

[9] D. Buttlar, J. Farrell, and B. Nichols. PThreads Programming. O'Reilly, 1996 (cit. on p. 20).

[10] C.-C. Chang and J. Cong. Pseudo Pin Assignment with Crosstalk Noise Control. In: *Proceedings of the 2000 International Symposium on Physical Design*. Association for Computing Machinery, 2000, pp. 41–47 (cit. on p. 2).

[11] T.-C. Chen and Y.-W. Chang. Multilevel Full-Chip Gridless Routing With Applications to Optical-Proximity Correction. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.6 (2007), pp. 1041–1053 (cit. on p. 2).

[12] J. Chuzhoy, D. H. K. Kim, and R. Nimavat. Almost Polynomial Hardness of Node-Disjoint Paths in Grids. In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. Association for Computing Machinery, 2018, pp. 1220–1233 (cit. on p. 3).

[13] E. G. Coffman, M. J. Elphick, and A. Shoshani. System Deadlocks. In: *ACM Computing Surveys* 3.2 (1971), pp. 67–78 (cit. on p. 21).

[14] J. Cong, J. Fang, and K.-Y. Khoo. An Implicit Connection Graph Maze Routing Algorithm for ECO Routing. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 1999, pp. 163–167 (cit. on p. 2).

[15] J. Cong, J. Fang, and Y. Zhang. Multilevel Approach to Full-Chip Gridless Routing. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2001, pp. 396–403 (cit. on p. 2).

[16] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. In: *Numerische Mathematik* 1.1 (1959), pp. 269–271 (cit. on p. 29).

[17] M. R. Garey and D. S Johnson. The Rectilinear Steiner Tree Problem is NP-Complete. In: *SIAM Journal on Applied Mathematics* 32.4 (1977), pp. 826–834 (cit. on p. 3).

[18] M. Gester. VLSI Routing for Advanced Technology. PhD Thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2015 (cit. on pp. 10, 17, 34, 100).

[19] M. Gester, D. Müller, T. Nieberg, C. Panten, C. Schulte, and J. Vygen. BonnRoute: Algorithms and Data Structures for Fast and Good VLSI Routing. In: *ACM Transactions on Design Automation of Electronic Systems* 18.2 (2013), 32:1–32:24 (cit. on pp. 9, 32, 36).

[20] S. Held, B. Korte, D. Rautenbach, and J. Vygen. Combinatorial Optimization in VLSI design. In: *Combinatorial Optimization: Methods and Applications.* Ed. by V. Chvatal. IOS Press, 2011 (cit. on p. 1).

[21] S. Held, D. Müller, D. Rotter, R. Scheifele, V. Traub, and J. Vygen. Global Routing with Timing Constraints. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.2 (2018), pp. 406–419 (cit. on p. 12).

[22] S. Held, D. Müller, D. Rotter, V. Traub, and J. Vygen. Global Routing with Inherent Static Timing Constraints. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 102–109 (cit. on p. 12).

[23] D. Henke. Pfadsuche im Detailed Routing. Bachelor's Thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2016 (cit. on p. 29).

[24] T. Hoefler and R. Belli. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, 2015, 73:1–73:12 (cit. on p. 121).

[25] R. M. Karp. On the Computational Complexity of Combinatorial Problems. In: *Networks* 5.1 (1975), pp. 45–68 (cit. on p. 3).

[26] N. Klewinghaus. Efficient Detailed Routing. Diplomarbeit. Rheinische Friedrich-Wilhelms-Universität Bonn, 2013 (cit. on pp. 12, 14, 16, 36).

[27] N. Klewinghaus. Fast Parallelisation for Detailed Routing in VLSI Design. Diplomarbeit. Rheinische Friedrich-Wilhelms-Universität Bonn, 2013 (cit. on pp. 12, 18, 20, 32, 97, 103, 140).

[28] B. Korte, D. Rautenbach, and J. Vygen. BonnTools: Mathematical Innovation for Layout and Timing Closure of Systems on a Chip. In: *Proceedings of the IEEE* 95.3 (2007), pp. 555–572 (cit. on p. 9).

[29] B. Korte and J. Vygen. Combinatorial Optimization: Theory and Algorithms. Sixth Edition. Vol. 21. Springer, 2018 (cit. on p. 5).

[30] B. Korte and J. Vygen. Combinatorial Problems in Chip Design. In: *Building Bridges: Between Mathematics and Computer Science*. Ed. by M. Grötschel, G. O. H. Katona, and G. Sági. Springer, 2008, pp. 333–368 (cit. on p. 1).

[31] M. R. Kramer and J. van Leeuwen. The Complexity of Wire-Routing and Finding Minimum Area Layouts for Arbitrary VLSI Circuits. In: *of Advances in Computing Research*. JAI Press, 1984, pp. 129–146 (cit. on p. 3).

[32] L. Lavagno, L. Scheffer, and G. Martin. EDA for IC Implementation, Circuit Design, and ProcessTechnology. Electronic Design Automation for Integrated Circuits Handbook. CRC Press, 2006 (cit. on p. 2).

[33] H. Li, G. Chen, B. Jiang, J. Chen, and E. F. Y. Young. Dr. CU 2.0: A Scalable Detailed Routing Framework with Correct-by-Construction Design Rule Satisfaction. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. Association for Computing Machinery, 2019, pp. 1–7 (cit. on p. 2).

[34] D. Müller. Fast Resource Sharing in VLSI Routing. PhD Thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2009 (cit. on pp. 18, 100, 103, 139).

[35] M. Neuwohner. Trackless TrackAssignment. Bachelor's Thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2018 (cit. on pp. 2, 8).

[36] F. Nohn. Detailed Routing im VLSI-Design unter Berücksichtigung von Multiple-Patterning. Diplomarbeit. Rheinische Friedrich-Wilhelms-Universität Bonn, 2012 (cit. on p. 34).

[37] C. Panten. Paralleles Verdrahten von VLSI-Chips auf Shared-Memory-Basis. Diplomarbeit. Rheinische Friedrich-Wilhelms-Universität Bonn, 2005 (cit. on pp. 12, 18).

[38] R. Scheifele. RC-aware Global Routing. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. Association for Computing Machinery, 2016, 21:1–21:8 (cit. on p. 12).

[39] R. Scheifele. Timing-Constrained Global Routing with RC-Aware Steiner Trees and Routing Based Optimization. PhD Thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2019 (cit. on p. 12).

[40] C. Schulte. Design Rules in VLSI Routing. PhD Thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2012 (cit. on pp. 9, 13, 18, 23, 24, 36, 87, 97, 100).

[41] A. Sterin. Postprocessing von Pfaden im Detailed Routing. Bachelor's Thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2015 (cit. on pp. 11, 34).

[42] F. Sun, H. Chen, C. Chen, C. Hsu, and Y. Chang. A Multithreaded Initial Detailed Routing Algorithm Considering Global Routing Guides. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. Association for Computing Machinery, 2018, pp. 1–7 (cit. on p. 2).

[43] J. Zühlke. Verfahren zur Beschleunigung der Chipverdrahtung. Diplomarbeit. Rheinische Friedrich-Wilhelms-Universität Bonn, 2008 (cit. on p. 100).