VLSI Routing for Advanced Technology

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von
Michael Gester
aus
Freudenberg

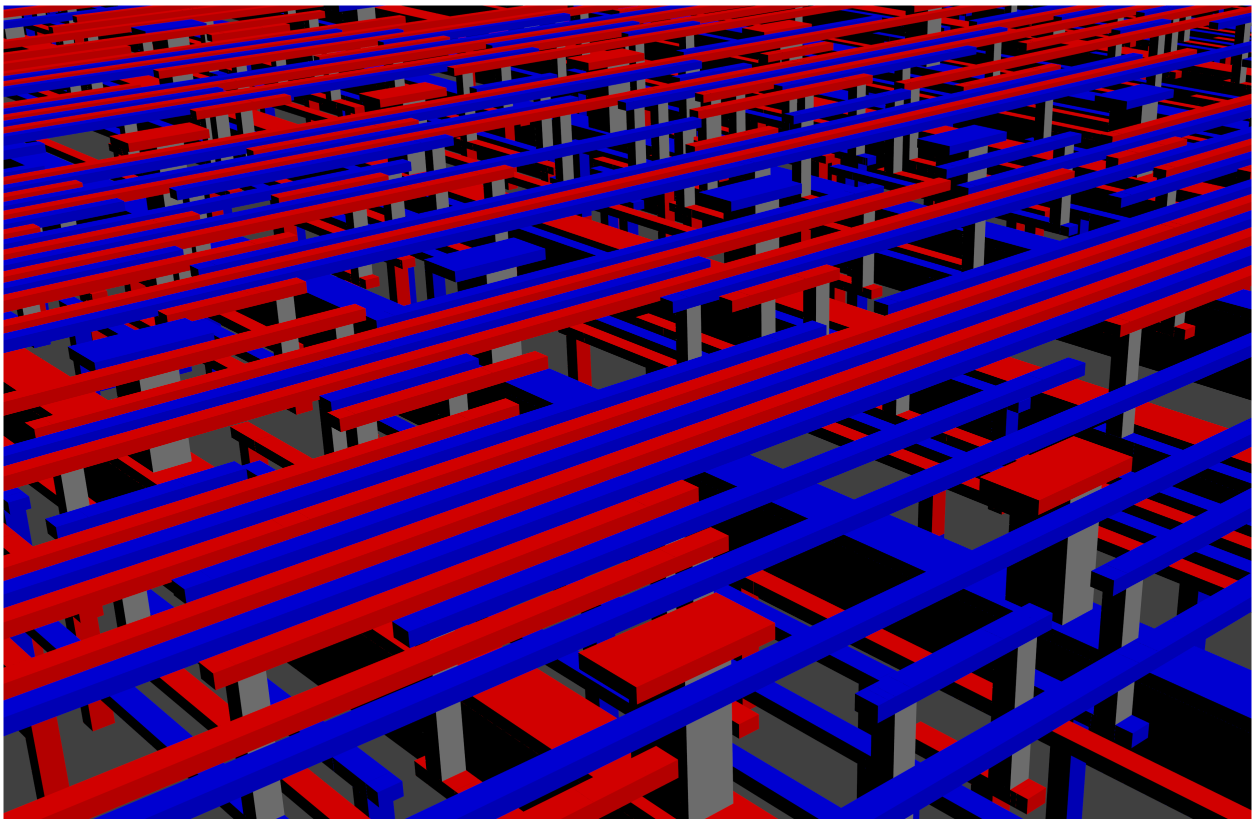
Bonn, Januar 2015

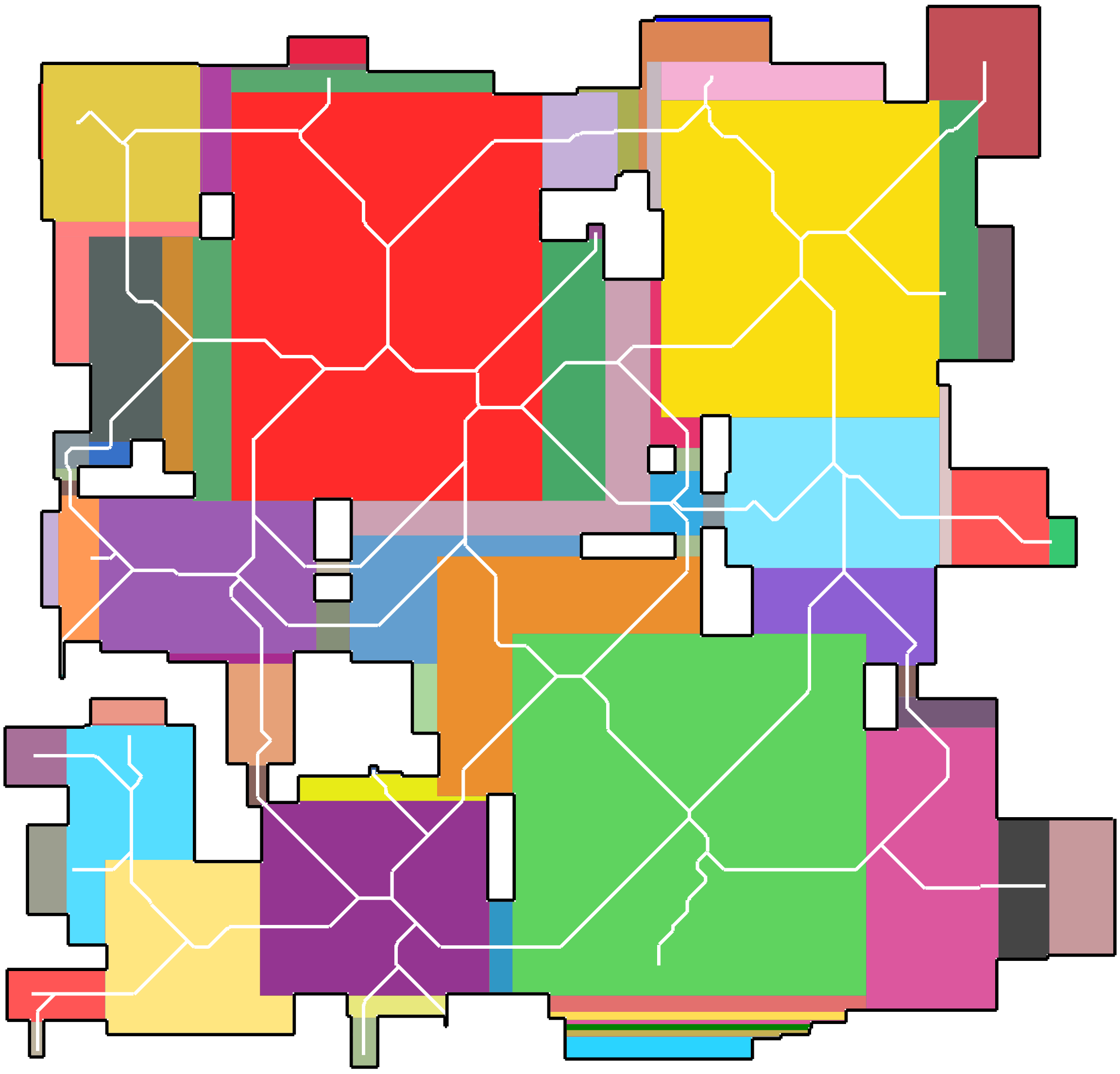
Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Gutachter: Prof. Dr. Jens Vygen
2. Gutachter: Prof. Dr. Stephan Held

Tag der Promotion: 12. März 2015

Erscheinungsjahr: 2015





Acknowledgments

At this place I would like to express my gratitude to my supervisor Professor Dr. Jens Vygen for his excellent support over the past years. This work has benefited substantially from his valuable ideas und experience.

Special thanks go to Professor Dr. Dr. h.c. Bernhard Korte for the exceptional working conditions at the Research Institute for Discrete Mathematics which laid the foundations for this thesis.

I would like to thank my former and present colleagues in the BonnRoute team: Markus Ahrens, Niko Klewinghaus, Dr. Dirk Müller, Dr. Tim Nieberg, Christian Panten, Rudi Scheifele, and Dr. Christian Schulte. They have all contributed extensively to the success of BonnRoute, and it was a great pleasure to share ideas and develop solutions for VLSI routing in cooperation with them. Further thanks go to Professor Dr. Stephan Held, Professor Dr. Stefan Hougardy, and Jannik Silvanus for very helpful discussions on several topics, and to Markus Ahrens, Niko Klewinghaus, Dr. Dirk Müller, and Dr. Jan Schneider for proof-reading parts of this thesis. Moreover, I would like to thank Dr. Nicolai Hähnle and Dr. Jan Schneider for fruitful discussions and contributions to polygon width decompositions. I also want to thank all other colleagues for the friendly and encouraging atmosphere at the institute.

I am very grateful for the excellent cooperation with IBM that has made the success of BonnRoute possible. At this place I would especially like to thank Karsten Muuss, Dr. Sven Peyer, Dr. Christian Schulte, and Dr. Gustavo Tellez for the great collaboration.

Finally, my personal thanks go to my parents for the greatest possible assistance, and to Blanka for her continuous encouraging support and her patience while I was writing this thesis.

Contents

1 Introduction

VLSI[1] *design* is the process of creating construction plans for complex *integrated circuits*, commonly known as *chips*, which contain up to billions of *transistors*. Due to its high complexity, this process is divided into several steps, each of them comprising hard mathematical problems from fields such as *combinatorial optimization* and *computational geometry*. VLSI design has been and still is a driving force for many research areas in these fields. It poses great challenges due to enormous instance sizes, complex constraints, and competing objective functions. In this thesis we focus on problems arising from advanced technology *design rules* in *routing*, a major step in the VLSI design process. The main contributions are as follows: In Section 4.2.2 we describe a new linear time algorithm for computing a *width-preserving* decomposition of a simple rectilinear polygon into rectangles in order to check width-dependent design rules in routing efficiently. In Section 5.4 we present a *detailed routing* approach which incorporates *multiple patterning technology* design rules, including experimental results which confirm that this approach is superior compared to an industry standard router (see Table 5.4).

In order to motivate the routing step we give a brief overview of the VLSI design process. The process starts with an algorithmic specification of the desired chip functionality which is first translated to a *register-transfer level* description where the functionality is modeled as a flow of digital signals (*high-level synthesis*, see Coussy and Morawiec [2010]). This description is further converted into a *netlist* which contains a list of *circuits* and their intended connectivity (*logic synthesis*, see Devadas et al. [1994]). These circuits are physical realizations of comparatively simple logical functions consisting of interconnected transistors. The circuits are predesigned and collected in a *library* which is reused for the design of different chips. Connectivity information is encoded by *nets*, where each net contains a set of *pins* that define inputs or outputs of the circuits which are to be connected.

The netlist serves as input for the subsequent *physical design* step (Alpert et al. [2008], Held et al. [2011]) where the circuits and connections between them are located within a given three-dimensional *chip area*. The chip area consists of several stacked *layers* which are manufactured separately. One major step in physical design is *routing* where pins of each net are connected by *wires*. The wires either run horizontally or vertically on a layer or connect two layers. See the upper title picture for a section of a routed real-world chip. Wires, as well as other objects like pins or blockages, have to respect *design rules* such as minimum distances, minimum

[1]Very Large Scale Integration

segment lengths, and minimum areas which reflect manufacturing requirements. Minimum distances usually depend on the width of the involved objects which are representable as rectilinear polygons on each layer. However, for simplicity reasons these polygons are usually decomposed into rectangles in routing. This motivates the question how to efficiently compute a decomposition which allows for checking of width-dependent minimum distance rules on the resulting rectangles. We answer this question for different width notions in Chapter 4.

After physical design all objects (also called *features*) have fixed positions within the chip area, and they are checked with respect to manufacturing requirements (encoded by design rules) and transformed in a way such that expected manufacturing variations are compensated (Liebmann [2003]). The outcome of these steps is the final construction plan for the chip which is then manufactured by using *lithographic techniques* (see Jaeger [2002] for details). The lithographic manufacturing process of VLSI chips becomes increasingly difficult with the continuous shrinking of feature sizes. The wavelength of the light source used for creating the features (currently 193 nm) has not changed for more than a decade, but in spite of this new techniques have been developed to allow for increased feature density. One such technique is *multiple patterning*. The idea is to assign features on one chip layer to different manufacturing steps which are commonly abstracted as *colors*. In its simplest form, the features are assigned two different masks and the final layout on the chip is produced by two subsequent exposure steps using these masks. Here the positions of the masks are chosen such that the desired spacings between features are realized. See the upper title picture for an example of such an assignment on a real-world chip. Computing and maintaining the color assignment complicates routing substantially since design rules now depend not only on the geometry of the involved objects but also on their colors. In Chapter 5 we investigate the main challenges in routing which arise from multiple patterning and advanced technology design rules, and we present efficient solutions for these challenges.

The thesis is organized as follows. After compiling the basic notation and definitions (Chapter 2) we give a short introduction to VLSI routing (Chapter 3). In Chapter 4 we investigate several polygon decomposition problems, most of them related to design rule checking in VLSI design. As a main result of this thesis, in Section 4.2.2 we give an $O(n)$ time algorithm for decomposing a simple rectilinear polygon with n vertices into regions having uniform width. For rectilinear polygons with holes the runtime increases to $O(n \log n)$. Here the width at a point of the polygon is defined as the edge length of the largest square that contains the point and is contained in the polygon. See the lower title picture for an example of such a decomposition. Our algorithm makes use of the *Voronoi core*, a subset of the L_∞ *Voronoi diagram* of the polygon edges, which is given by white lines in the lower title picture. This algorithm can be used to preprocess polygonal shapes on a chip into rectangles such that width-dependent design rules can be checked on these rectangles instead of polygons.

In Chapter 5 we first define multiple patterning formally and collect several related theoretical results in connection with color assignment problems. Subsequently, we present an efficient approach for managing multiple patterning and advanced technology design rules in VLSI routing (Section 5.4). This approach contains *multi-label shortest paths* (Section 5.4.2.2) as a key algorithmic concept. Computing shortest paths is one of the basic problems to be solved in routing. However, the paths computed by standard shortest path algorithms cannot cope with complicated design rules. The basic idea of multi-labeling is to encode certain path properties (which model design rules) as *labels*, and to allow only certain *label transitions* for the paths computed. In this way we are able to compute nearly clean (w.r.t. design rules) connections for nets, in contrast to the popular approach to first compute unclean connections and try to fix design rule violations afterwards by a post-processing step. In advanced chip technologies, such post-processing often fails (see the results in Section 5.4.4), for example due to missing space for fixing the violations locally. Further aspects of our multiple patterning approach are an efficient color management (Section 5.4.1) and a framework that uses multi-labeling only when necessary instead of always (Section 5.4.2.4). This even improves the quality of results (Section 5.4.4).

The presented multiple patterning approach has been implemented in *BonnRoute* (Gester et al. [2013]), the routing tool contained in the *BonnTools* software package which covers all major physical design steps. The BonnTools (Korte et al. [2007]) are being developed at the Research Institute for Discrete Mathematics, University of Bonn, in cooperation with IBM and have been used successfully by IBM and its customers for the design of more than thousand highly complex chips, for more than two decades. In Section 5.4.4 we present results confirming that BonnRoute produces high-quality routing solutions on real-world multiple patterning designs fast. We also compare a combined flow (BonnRoute followed by an external cleanup step) against an industry standard router used by IBM and obtain far superior results with our combined flow (see Table 5.4 on page 102). This result is one of the main contributions of this thesis. We finally note that our proposed combined BonnRoute flow is the default tool for *signal routing* at IBM, used for designing all their *ASIC chips* as well as their *server chips* for both single patterning and multiple patterning technologies.

2 Notation and Definitions

We compile some basic notation and definitions used in this thesis. For a set A we denote the set of all k-element subsets of A as $\binom{A}{k}$. Let $A \subseteq \mathbb{R}^n$ for some $n \in \mathbb{N}_{>0}$. We denote the *border* of A as ∂A, the *interior* of A as A°, and the *closure* of A as $\bar{A}$. All these terms are to be understood relatively, that means with respect to the affine hull of A. For example, the interior of a line segment in $\mathbb{R}^2$ contains all of its points except the endpoints.

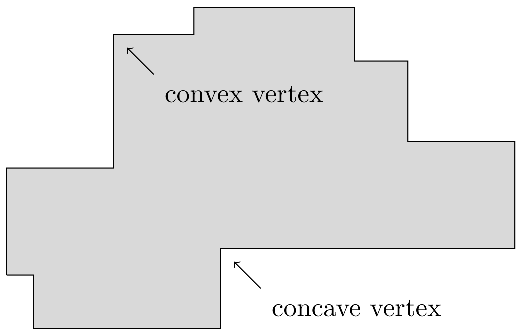
Two sets $A, B \subseteq \mathbb{R}^n$ are called *interior-disjoint* if $A^\circ \cap B^\circ = \emptyset$. A set $\mathcal{A} := \{A_1, A_2, \ldots, A_k\}$ with $A_i \subseteq \mathbb{R}^n$ for all $i \in \{1, 2, \ldots, k\}$ is called interior-disjoint if A_i and A_j are interior-disjoint for all $i \neq j$. The *Minkowski sum* of two sets $A, B \subseteq \mathbb{R}^n$ is defined as $A \oplus B := \{a + b \,|\, a \in A, b \in B\}$.

For two points $p, q \in \mathbb{R}^n$ we denote the line segment connecting p and q, that is the set $\{x \in \mathbb{R}^n \,|\, x = p + c \cdot (q - p), 0 \leq c \leq 1\}$, as $\overline{pq}$. For a finite sequence of points $p_1, p_2, \ldots, p_k \in \mathbb{R}^2$ the set $C = \overline{p_1p_2} \cup \overline{p_2p_3} \ldots \cup \overline{p_{k-1}p_k}$ is called a *polygonal chain*. It is called *closed* if $p_1 = p_k$, and *simple* if each point in $\{p_1, p_2, \ldots, p_k\}$ is contained in at most two line segments and the inner points of each line segment are not contained in any other line segment.

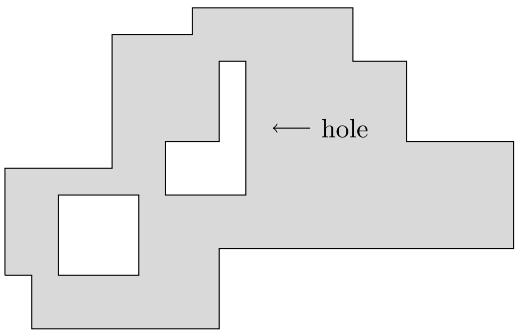
The closure of the area enclosed by a simple closed polygonal chain is a *simple polygon*. A *polygon* is a set $P := P_1 \setminus ({P_2}^\circ \cup \ldots \cup {P_k}^\circ)$, where $P_1, \ldots, P_k$ are simple polygons and $P_2, \ldots, P_k$, the *holes* of P, do not intersect pairwise and are all contained in ${P_1}^\circ$. The line segments of the polygonal chains defining a polygon are the *edges* and their endpoints are the *vertices* of the polygon. As usual, we assume that two neighboring edges are not contained in a common line. A vertex is called *convex* if the angle formed by the two adjacent edges inside of the polygon is less than 180° and it is called *concave* otherwise. A *rectilinear polygon* is a polygon whose edges are all parallel to the x- or y-axis. See Figures 2.1a and 2.1b for examples.

When we consider simple geometric figures like rectangles or trapezoids we often identify them with their enclosed area (including the border, if not stated differently).

For measuring lengths and distances in $\mathbb{R}^3$ we use the following functions.



(a) A simple rectilinear polygon.



(b) A rectilinear polygon with two holes.

Figure 2.1

Definition 2.1. *For two points $p = (x_1, y_1, z_1), q = (x_2, y_2, z_2) \in \mathbb{R}^3$ we define*

$$\begin{aligned}
d_1(p,q) &:= \|p-q\|_1 := |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2| && (L_1 - \textit{distance}) \\
d_2(p,q) &:= \|p-q\|_2 := \sqrt{|x_1 - x_2|^2 + |y_1 - y_2|^2 + |z_1 - z_2|^2} && (L_2 - \textit{distance}) \\
d_\infty(p,q) &:= \|p-q\|_\infty := \max(|x_1 - x_2|, |y_1 - y_2|, |z_1 - z_2|) && (L_\infty - \textit{distance})
\end{aligned}$$

For two closed and bounded sets $A, B \subseteq \mathbb{R}^3$ we define

$$\begin{aligned}
d_1(A,B) &:= \min_{p \in A, q \in B} d_1(p,q) \\
d_2(A,B) &:= \min_{p \in A, q \in B} d_2(p,q) \\
d_\infty(A,B) &:= \min_{p \in A, q \in B} d_\infty(p,q)
\end{aligned}$$

For basic notions in graph theory we refer to Korte and Vygen [2012].

3 VLSI Routing Overview

We now give an overview and formalization of VLSI routing with a focus on detailed routing. The description is not meant to be universal, but depicts how the routing flow works in BonnRoute.

The basic task in VLSI routing is to connect certain predetermined *pins* on a chip with *wires*. A set of pins which have to be connected is called a *net*. The pins are either inputs or outputs of small integrated *circuits* on the chip or serve as a connection to the exterior. The term circuit is ambiguous since the chip itself is also a large complex circuit, but here we consider circuits as black boxes realizing comparatively simple logical functions. Such circuits (e.g. inverter or NANDs) are usually *predesigned* and reused on different chips of the same technology. See Schneider [2014] for details on the design of such circuits which is done on a *transistor level*.

A chip consists of several stacked and parallel *layers* which are manufactured separately. The circuits and its involved transistors are placed on lower layers, and wires connecting nets are placed on *routing layers* above. We now define the search space and the basic geometric objects for routing.

Definition 3.1. *The* chip area *is a nonempty rectangular cuboid*

$$A_{\text{chip}} := [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \times [z_{\min}, z_{\max}] \subseteq \mathbb{R}^3$$

with $x_{\min}, x_{\max}, y_{\min}, y_{\max}, z_{\min}, z_{\max} \in \mathbb{Z}$ *and* $z_{\min}, z_{\max}$ *even. We further define*

$$\begin{aligned}
Z_{\text{all}} &:= \{z_{\min}, \dots, z_{\max}\} && \textit{(the set of } \text{layers}\textit{)} \\
Z_{\text{wiring}} &:= \{z \in Z_{\text{all}} : z \textit{ even}\} && \textit{(the set of } \text{wiring layers}\textit{)} \\
Z_{\text{via}} &:= \{z \in Z_{\text{all}} : z \textit{ odd}\} && \textit{(the set of } \text{via layers}\textit{)} \\
A_z &:= \{(x, y, z) \in A_{\text{chip}}\} && \textit{(the } \text{chip area on layer } z\textit{)}
\end{aligned}$$

We will see later how wiring and via layers relate and why they alternate. We restrict ourselves to *Manhattan routing* meaning that all wires on a chip run parallel to the x-, y- or z-axis. This restriction, allowing an easier representation of the routing solution space and the involved objects, is still common practice, although there have been attempts to soften this restriction by also allowing diagonal wires, called *X architecture* (Teig [2002], Ho et al. [2005], Chen et al. [2003]).

Definition 3.2. *Let* $A := [x_1, x_2] \times [y_1, y_2] \times [z_1, z_2] \subseteq \mathbb{R}^3$. *We denote A as*

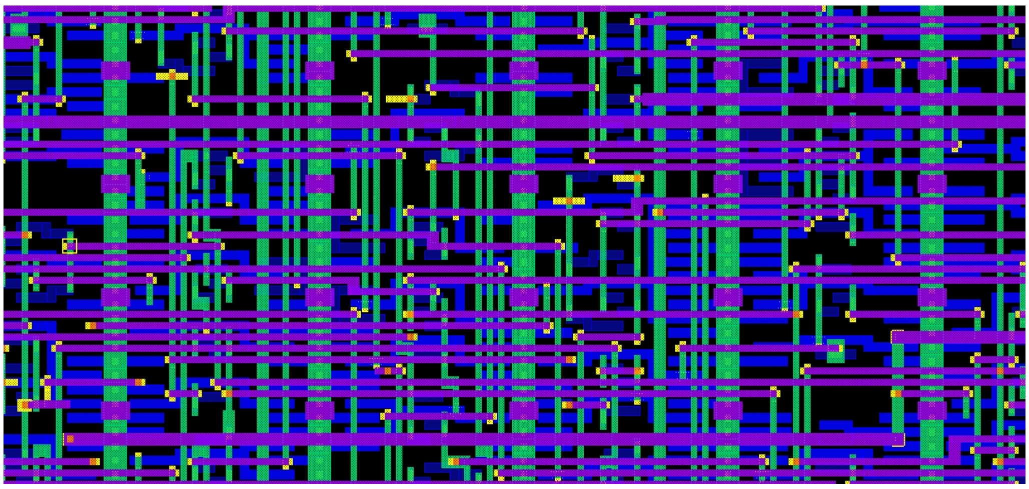


Figure 3.1: Routing snapshot showing less than $\frac{1}{5.000.000.000}$ of a chip. Blue shapes are pins on layer 0, green shapes are plane wires on layer 2, purple shapes are plane wires on layer 4, and yellow shapes are (parts of) vias connecting two layers.

(i) shape, *if* $x_1 < x_2$, $y_1 < y_2$ *and* $z_1 = z_2$,

(ii) stick, *if exactly one of* $|x_1 - x_2|$, $|y_1 - y_2|$, *or* $|z_1 - z_2|$ *is nonzero.*

Hence geometrically a shape is a rectangle with an assigned z-coordinate and a stick is a line segment parallel to the x-, y-, or z-axis. We denote the layer on which a shape A lies by $z(A)$.

There are three basic types of objects on a chip which are relevant for routing: *wires*, *pins*, and *blockages*. For wires, we differentiate between *plane wires* running in x- or y-dimension and *vias* running in z-dimension. See Figure 3.1 for a routing snapshot viewed from above.

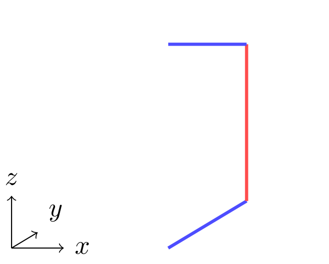
Definition 3.3. *(i) A* plane wire w *is a pair* (r, s) *where* r *is a stick,* s *is a shape and* $r \subseteq s \subseteq A_z$ *for some* $z \in Z_{\text{wiring}}$. *We call* $\text{stick}(w) := r$ *the* stick *and* $\text{shape}(w) := s$ *the* shape of the plane wire.

(ii) A via v *is a quadruple* $(r, s_{\text{bot}}, s_{\text{cut}}, s_{\text{top}})$ *with* $r = \{x\} \times \{y\} \times [z, z+2]$ *for some* $z \in Z_{\text{wiring}}$ *such that* $(x, y, z) \in s_{\text{bot}} \subseteq A_z$, $(x, y, z+1) \in s_{\text{cut}} \subseteq A_{z+1}$, $(x, y, z+2) \in s_{\text{top}} \subseteq A_{z+2}$ *and* s_{bot}, s_{cut}, s_{top} *are shapes. We call* $\text{stick}(v) := r$ *the* stick *and* $\text{bot}(v) := s_{\text{bot}}$, $\text{cut}(v) := s_{\text{cut}}$, *and* $\text{top}(v) := s_{\text{top}}$ *the* bottom, cut, *and* top shape of the via, *respectively.*

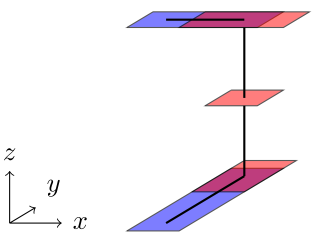
(iii) A wire *is a plane wire or a via.*

(iv) A pin *is a finite set of shapes.*

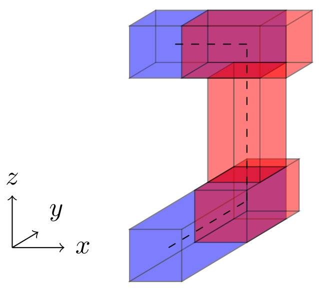
(v) A blockage *is a finite set of shapes.*



(a) Wire sticks as seen in BonnRoute.



(b) Wire shapes as seen in BonnRoute.



(c) Three-dimensional abstraction of shapes on the chip.

Figure 3.2: Different wire representations for two plane wires (blue) and a via (red). Overlapping parts of plane wires and via are purple.

(vi) A net *is a finite set of pins.*

The stick of a plane wire or via serves as one-dimensional abstraction of the two-dimensional shape(s) which is much easier to handle for example in shortest path algorithms. See Figure 3.2 for an illustration of typical wires in different representations. The shape(s) of an object shall correspond to the metalized area the object will fill out on the manufactured chip layer. We note that due to manufacturing inaccuracies and correction steps modifying the objects for the manufacturing process (such as *optical proximity correction*) the real area on a chip will not be exactly the same as the set of shapes seen in routing.

In addition to the geometry of these objects, we also manage other properties like the net a wire belongs to or the layers on which a net should be routed. To simplify notation we will use such other properties only when needed.

The basic task in routing is to compute connections between the pins of a net which will carry electrical signals on a working chip later on. For electrical connectivity it is not sufficient that two wire shapes touch in one point, but they rather have to share a common area whose minimum size is specified by an *overlap rule*. It would be quite cumbersome to always respect and check this rule in routing algorithms, so one uses the following simpler model for connectivity using the corresponding sticks which is sufficient in practice.

Definition 3.4. *Two wires are called* connected *if their sticks have at least one point in common. A* connection for a net $\mathcal{N}$ *is a finite set of wires C such that*

(i) $\{\text{stick}(w) \mid w \in C\}$ *is a connected set*

(ii) for all pins $P \in \mathcal{N}$ there exists a pin shape $s \in P$ and a wire $w \in C$ with $s \cap \text{stick}(w) \neq \emptyset$

(iii) for all wires $w_1, w_2 \in C, w_1 \neq w_2$: $|\text{stick}(w_1) \cap \text{stick}(w_2)| \leq 1$

In fact the wire sticks are defined exactly in such a way that they allow for this easy model for connectivity. At first sight the above model for connectivity is not sufficient to model all situations occuring in practice. For example, a pin shape may be used as part of the connection of a net which in our model is not incorporated. However, we do not go into technical details here and just mention that these and similar situations can also be modeled e.g. by inserting appropriate dummy wires.

For each wiring layer $z \in Z_{\text{wiring}}$ we define a *preferred dimension* $\text{pdim}(z) \in \{x, y\}$ in which most wires should be directed to improve space utilization (see Figure 3.1). In routing algorithms this is realized by higher costs for the usage of wires directed in the other dimension, called the *non-preferred dimension* ($\text{ndim}(z) \in \{x, y\}$). We call a plane wire whose stick runs in the preferred or non-preferred dimension *pref wire* or *jog*, respectively.

We assume that neighboring wiring layers have orthogonal preferred dimensions which is common practice for several reasons: First, *crosstalk* (electrical interference) between long parallel wires on neighboring layers is avoided. Second, long jogs making efficient space utilization hard are less likely because each pair of neighboring layers covers both x- and y-dimension as preferred dimension.

Due to limitations in manufacturing a wire must have at least a certain width. While theoretically one could often use wires with arbitrary width greater than that minimum width, in practice it is reasonable to allow only a small number of different *wire types*.

Definition 3.5. *A* wire type *is given by one shape* $s_{z,\text{cut}} \subset \mathbb{R}^2 \times \{z\}$ *for each* $z \in Z_{\text{via}}$ *and by four shapes* $s_{z,\text{pref}}, s_{z,jog}, s_{z,\text{bot}}, s_{z,\text{top}} \subset \mathbb{R}^2 \times \{z\}$ *for each* $z \in Z_{\text{wiring}}$*. Let* W *be a wire type. We say that a plane wire* w *on layer* z has wire type W *if* $\text{shape}(w) = \text{stick}(w) \oplus s$*, where*

$$s := \begin{cases} s_{z,\text{pref}} & \textit{if } w \textit{ is a pref wire} \\ s_{z,jog} & \textit{if } w \textit{ is a jog} \end{cases}$$

We say that a via v *ranging from layer* z *to layer* $z+2$ has wire type W *if the following conditions hold:*

$$\begin{aligned} \text{bot}(v) &= (\text{stick}(v) \cap A_z) \oplus s_{z,\text{bot}} \\ \text{cut}(v) &= (\text{stick}(v) \cap A_{z+1}) \oplus s_{z+1,\text{cut}} \\ \text{top}(v) &= (\text{stick}(v) \cap A_{z+2}) \oplus s_{z+2,\text{top}} \end{aligned}$$

Typically, most nets on a chip use the same wire type which we call *standard wire type* and which has minimum allowed width on each layer. Wires using this wire type are denoted as *standard wires* or *1x wires*. Further wire types are typically chosen in a way such that they fit well to the 1x wires with respect to width and distance requirements. Their widths are often multiples of the minimum width, and wires using such wire types are denoted as 2x wires, 3x wires and so on. Wider wires are primarily used to speed up signals for long connections because of their lower resistance.

Since the main detailed routing step runs in a sequential fashion (see Algorithm 1 on page 14), allowing arbitrary positions for wires would lead to bad space utilization even if only standard wires are used. Therefore, we define a set of coordinates in dimension $\mathrm{ndim}(z)$ for each wiring layer z where (almost) all wire stick endpoints should lie on. These coordinates are typically chosen such that standard wires lying on them can be packed as dense as possible.

Definition 3.6. *For each wiring layer* $z \in Z_{\mathrm{wiring}}$ *with* $\mathrm{pdim}(z) = x$ *we have* track coordinates $T_z = \{t_z^1, \dots, t_z^{|T_z|}\}$, $y_{\min} \le t_z^1 < \dots < t_z^{|T_z|} \le y_{\max}$. *We call* $\mathrm{tracks}_z := \{[x_{\min}, x_{\max}] \times \{t\} : t \in T_z\}$ *the* set of tracks *on layer* z*.*

Analogously, for each wiring layer $z \in Z_{\mathrm{wiring}}$ *with* $\mathrm{pdim}(z) = y$ *we have* track coordinates $T_z = \{t_z^1, \dots, t_z^{|T_z|}\}$, $x_{\min} \le t_z^1 < \dots < t_z^{|T_z|} \le x_{\max}$. *We call* $\mathrm{tracks}_z := \{\{t\} \times [y_{\min}, y_{\max}] : t \in T_z\}$ *the* set of tracks *on layer* z*.*

A point $q = (x, y, z)$ *with* $\mathrm{pdim}(z) = x$ *(*$\mathrm{pdim}(z) = y$*) is called* point of interest *if* $x \in T_{z-2} \cup T_{z+2}$ *(*$y \in T_{z-2} \cup T_{z+2}$*) and* $y \in T_z$ *(*$x \in T_z$*). Here we use* $T_z := \emptyset$ *for* $z \notin Z_{\mathrm{all}}$*.*

For details how tracks are computed in BonnRoute see [Müller, 2009, Section 2.4]. We are now able to define the search space for the connections to be computed, called the *track graph.*

Definition 3.7. *The* track graph G_T *contains all points of interest as vertices and contains an edge between two vertices* $v_1 = (x_1, y_1, z_1)$ *and* $v_2 = (x_2, y_2, z_2)$ *if and only if they differ in exactly one coordinate and no other vertex is contained in the line segment* $\overline{v_1 v_2}$*.*

We identify vertices and edges of G_T with their corresponding points and line segments in $\mathbb{R}^3$. See Figure 3.3 for an example of a track graph.

Definition 3.8. *A wire* w *is called* on-track *if* $\mathrm{stick}(w)$ *is the union of edges in* $E(G_T)$*, and it is called* off-track *else.*

On a high level finding connections for nets using only on-track wires corresponds to packing Steiner trees in the track graph. See Figure 3.4 for an illustration of the theoretical view and the real-world view. However, the track graph may contain more than 300 billion vertices on real-world instances which makes near-optimal Steiner tree packing computation hopeless.

A natural way to overcome the huge instance size is to first compute rough connections in a coarsened version of G_T, a step called *global routing.* Here one first defines a two-dimensional grid and then contracts vertices of G_T lying on the same layer and within the same grid region (called *tile*).

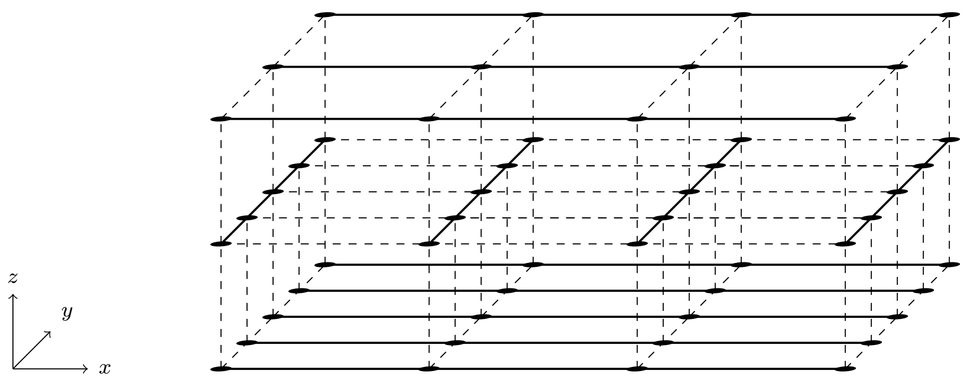


Figure 3.3: Example for a track graph. Edges are depicted by solid and dashed line segments between two neighboring vertices, and tracks are drawn as solid lines.

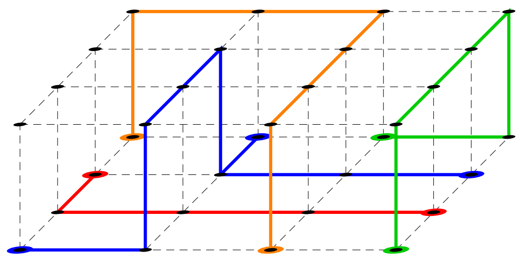
In this much smaller graph one computes Steiner trees for all nets which lead to *routing corridors* (small sections of G_T) by undoing the contraction for each tree. The actual connection for a net is then realized later in *detailed routing* by using only the routing corridor as search space. See Müller et al. [2011], Müller [2009] for the global routing approach used in BonnRoute which is based on an algorithm for the Min-Max Resource Sharing Problem and is able to take various objectives and constraints into account. Most important, a global router has to keep *congestion* low, meaning that not too many nets use the same routing corridors which will make it hard or even impossible to realize detailed connections within the corridors.

Some routing tools also use intermediate steps between global and detailed routing such as *switchbox routing* (Hitchcock [1969]) or *track assignment* (Batterywala et al. [2002], Chang and Cong [2001]) to fix long distance wires previous to detailed routing in order to further reduce detailed routing runtime and optimize objectives (e.g. *coupling* and *noise* reduction) for these long wires globally. BonnRoute does not use such a step, here the detailed connections are computed by an extremely fast *path search* algorithm (see Section 5.4.2) which can deal also with long connections fast enough in practice.

Since we focus on detailed routing in this thesis, we assume routing corridors for all nets as given from now on. We also assume that these corridors contain information on wire type usage for detailed routing.

Definition 3.9. *A* wire type region *is a pair* (W, R)*, where* W *is a wire type and* $R \subseteq A_{\text{chip}}$ *is a union of finitely many axis-parallel cuboids. A* routing corridor *is a finite set of wire type regions with pairwise distinct wire types.*

We assume that each net N has an assigned routing corridor $\text{corr}(N)$ based on global routing. A connection for N is only allowed to use a wire w if there exists a wire type region (W, R) in $\text{corr}(N)$ such that w has wire type W and $\mathit{stick}(w) \subseteq R$.


(a) Theory: Four node-disjoint Steiner trees in a three-dimensional grid graph.

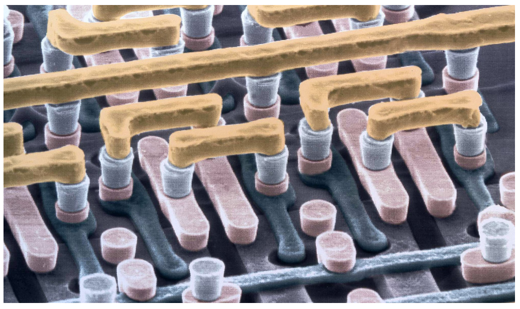

(b) Real world: VLSI routing on a real chip viewed by an electron microscope. (Picture adapted from Peyer [2007])

Figure 3.4

In BonnRoute this restriction is relaxed by enlarging wire type regions gradually if no feasible connection is found otherwise.

Since computing shortest Steiner trees within the given corridors would still be too slow in practice, BonnRoute computes shortest paths between connected components of a net sequentially. Here a connected component is a maximal connected set consisting of wire sticks and pin shapes of the net. We start with choosing two connected components S and T of a net N. Then we compute a *restricted routing corridor* $\mathrm{rcorr}(N, S, T)$ which contains only those regions of $\mathrm{corr}(N)$ which are relevant for finding an S-T-connection, basically following the edges of the global routing connection for N from S to T. Searching for an S-T-connection only within $\mathrm{rcorr}(N, S, T)$ speeds up the shortest path computation significantly since many redundant label steps can be saved this way.

We compute a path consisting of wire sticks inside of $\mathrm{rcorr}(N, S, T)$ connecting S and T and proceed with new connected components S' and T'. We iterate this procedure until N is connected. How such an S-T-connection is computed and what is done when no connection is found is explained in Section 5.4.2.

BonnRoute uses a separate routine to precompute short access paths to pins and connections between nearby pins (*short connections*), the *pin access* (see Ahrens et al. [2015], Ahrens [2014]). Accessing the pins is becoming more challenging and critical in advanced technology nodes. Our algorithm is suitable for high pin density routing of small standard circuits, and is capable of handling advanced technology design rules and various objectives. The pin access routine is important for three reasons: First, not all pins can be connected by using only on-track wires, i.e. within the track graph. Second, the sequential routing procedure sketched above may block pins when connecting another pin, leading to *convergence* issues, that means not all nets can be connected in the end. Third, the pin access can globally optimize objectives over all access paths and short connections, while the sequential routing step cannot.

Algorithm 1: High-level overview of BonnRoute

```
1 Compute short connections and pin access paths
2 Compute global routing
3 Sort nets w.r.t. to criticality
4 while Not all nets connected do
5     Choose a not connected net $N$
6     Choose two connected components $S$ and $T$ of $N$
7     Compute rcorr$(N, S, T)$ based on global routing for $N$
8     Compute connection between $S$ and $T$ in rcorr$(N, S, T)$
9     Post-process found connection
```

Algorithm 1 shows a simplified high-level overview of BonnRoute. It is important to point out that the first and second step of Algorithm 1 operate on all nets *simultaneously* and are thus able to globally optimize certain objective functions. In contrast to that, long connections are computed in a sequential manner until all nets are connected (lines 4 to 9). Therefore, for long connections it is of special importance to provide some guidance such that dense wire packings can be obtained, although the connections are not globally optimized. One such guidance are routing tracks, but it turns out that they are not sufficient for routing in advanced multiple patterning technologies. We will return to this important topic in Section 5.4. All routing steps in BonnRoute are efficiently parallelized, see Ahrens [2014] (pin access), Müller et al. [2011] (global routing), and Klewinghaus [2013] (long connections) for details.

One remaining question is what we do in cases where in line 8 no connection can be found. Then we allow to use wires which are only legal if some present wires are ripped out, at some high cost. If we found a connection this way, then we try to reroute (parts of) all connections where wires have been ripped out before. For these connections we may have to rip out other wires again, and so forth. If this process (called *rip-up and reroute*, see Salowe [2008]) does not converge after a certain number of iterations or we do not even find a connection when rip-up is allowed, then we allow wires to leave the restricted routing corridor by some specified margin. If still no connection is found this way (e.g. if a pin is covered by a blockage), then we compute a connection which is allowed to cross arbitrary shapes, at some very high cost. Therefore, in any case some connection for a net is computed and we finish the main loop in Algorithm 1 at some time. We use location based rip-up costs which increase over time to avoid cyclic rip-up and reroute sequences. See Hetzel [1998] for more details.

The complicated lithographic manufacturing process of a chip requires numerous *design rules* which have to be satisfied before the chip can be produced. These design rules tend to become more complicated with each new technology generation. We list the most important types of design rules here, special rules arising from multiple patterning technology are discussed in Chapter 5:

- **diff-net-mindist:** Two shapes on the same layer and not part of the same net must have at least a certain L_2-distance.
- **same-net-mindist:** Two non-intersecting shapes on the same layer and of the same net must have at least a certain L_2-distance.
- **minarea:** Each connected set of shapes on a layer must have at least a certain area.
- **minedge:** Each edge of a rectilinear polygon representing a connected set of shapes must have at least a certain length.
- **minwidth:** Each shape must have a certain minimum width.
- **minenclosure:** The projection of a via cut shape to a neighboring wiring layer must be enclosed by shapes of this wiring layer, with some specified minimum margin.
- **interlayer via mindist:** The projections of two via cut shapes on neighboring via layers which are not part of the same net must have at least a certain L_2-distance.

The first and last rule are *diff-net-rules*, all others are *same-net-rules*. We simplified some rules for convenience. For more details and a formal definition of design rules see Schulte [2012]. Technically, mindist rules operate on connected components of shapes (representable as rectilinear polygons), and the required L_2-distance depends on various geometric properties of the involved polygons such as area, width, or edge lengths at those points of the polygon where distance is measured. However, in practice it would be cumbersome to permanently maintain connected components as rectilinear polygons. One is rather interested in storing objects as easy as possible, e.g. as rectangles. Therefore, an interesting question is if the polygons can be decomposed into rectangles in such a way that certain design rules can be checked on the rectangles, giving the same results as if checked on the polygons. We give a positive answer to this question in Chapter 4 for one of the most important classes of design rules, width-dependent mindist rules, and present efficient algorithms computing this decomposition. From now on we will thus assume mindist rules based on shapes as defined above. See Schulte [2012] for a justification why also for many other design rules it is sufficiently accurate to decompose polygons and check rules between rectangles later on.

In the following we assume a *checking oracle* which is able to decide if a set of shapes violates any design rule or not. This oracle is used by routing algorithms to query for legal wire locations.

Definition 3.10. *A* checking oracle *is a function ψ which returns for a given set of shapes $\mathcal{S}$ if these shapes violate any design rule ($\psi(\mathcal{S}) = 0$) or not ($\psi(\mathcal{S}) = 1$). If $\psi(\mathcal{S}) = 1$, then we call $\mathcal{S}$* DRC-clean.

Here *DRC* stands for *design rule check*. Following the above definition, a chip is manufacturable if the set of all shapes on the chip is DRC-clean. We will give some details on the usage of the checking oracle in Section 5.4.3. We assume that the checking oracle knows to which net a shape belongs which is necessary to decide whether same-net or diff-net rules apply.

For routing algorithms same-net rules are usually much harder to obey than diff-net rules. The main reason is that with respect to diff-net rules a partial illegal connection stays illegal when wires are added (with some exceptions), but with respect to same-net rules a partial illegal connection may become legal by adding a small piece of wire (e.g. for minarea or minedge rules). Therefore, the approach to discard any illegal partial solution does not work for same-net rules. In Section 5.4.2 we describe an approach how to respect same-net rules while computing connections.

4 Polygon Decompositions in VLSI Design

Decompositions of polygons into simpler geometric objects have a long history in computational geometry, see Keil [2000] for a survey. These simpler objects can be for example special types of polygons such as monotone, star-shaped or convex polygons or fixed geometric shapes such as trapezoids, rectangles, squares or triangles. When using simpler objects, data structures can be kept simple and efficient which is of special importance in VLSI design where rectangles (instead of rectilinear polygons) are used as main geometric data type and occur millionfoldly.

Another main motivation for such decompositions is that many geometric problems can be solved much easier on the simpler objects than on the original polygon, so it makes sense to first decompose the polygon and then solve the problem on the simpler objects. To use this approach, one has to ensure that the problem to solve translates somehow from the polygon to the simpler objects. For example, if we are given a rectilinear polygon and a decomposition into rectangles and we want to check if a given point is contained in the polygon, then we could answer this question by just checking if the point is contained in any of the rectangles. So in this case the problem translates easily. However, if we want to determine the maximum horizontal width of the polygon, i.e. the length of a longest horizontal line segment contained in the polygon, we can not just take the maximum of the horizontal widths of all rectangles because this may be smaller. However, we can build the decomposition in such a way that it is guaranteed that some rectangle attains the maximum horizontal width of the polygon. Generally speaking, the decomposition has to obey certain contraints depending on the problem we want to solve.

In the following we focus on decomposing a rectilinear polygon into rectangles, which is of particular interest in several parts of VLSI design (Keil [2000]). Section 4.1 summarizes the most important known results for such decompositions without additional constraints. In Section 4.2 we consider constrained decomposition problems arising in VLSI design in connection with design rule checking. The main result of this chapter is a new efficient algorithm computing a *two-dimensional width decomposition* of a rectilinear polygon (Section 4.2.2).

Subsequently, in Section 4.3 we give an efficient algorithm for decomposing the union of expanded polygons, solving an important subproblem in *clock network design*. This result is not restricted to rectilinear polygons.

Throughout this chapter P always denotes a polygon and n is the number of border segments of P. We assume a polygon given as a set of boundaries (one outer boundary and one for each hole), each of which is stored as a doubly linked list of points in the plane, the polygon vertices.

4.1 Unconstrained Polygon Decomposition

We first consider the decomposition of a rectilinear polygon into rectangles without any additional constraints, using as few rectangles as possible. This classical problem in computational geometry occurs for example when translating connected metal components on a chip layer to rectangle sets in VLSI design, minimizing the storage amount for the rectangles, and in VLSI mask generation (see Keil [2000]). Depending on the application, we may or may not allow proper intersections of distinct rectangles, leading to the following two problems.

Polygon Coverage Problem

Instance: A rectilinear polygon P.

Task: Compute a minimum rectangle set covering the same area as P.

Polygon Partitioning Problem

Instance: A rectilinear polygon P.

Task: Compute a minimum interior-disjoint rectangle set covering the same area as P.

While both problems look quite similar, the difficulty of solving them differs substantially. The Polygon Coverage Problem is known to be NP-hard even for simple rectilinear polygons (Culberson and Reckhow [1988]) and MAXSNP-hard for arbitrary rectilinear polygons (Berman and DasGupta [1992]), implying that no polynomial-time approximation scheme exists, unless P=NP.

Franzblau [1989] gave a simple sweepline heuristic running in $O(n \log n)$ time which computes a solution where the number of rectangles is at most $2m$ for simple polygons and at most $O(m \log m)$ for polygons with holes, where m is the minimum number of rectangles covering the polygon. Kumar and Ramesh [2003] described the first polynomial time approximation algorithm with an $o(\log n)$ approximation factor for polygons with holes, computing a solution with at most $O(m\sqrt{\log n})$ rectangles.

In contrast, the Polygon Partitioning Problem can be solved optimally in linear time for simple rectilinear polygons. This result (Keil [2000]) is obtained by first applying Chazelle's algorithm (Chazelle [1991]) to obtain a triangulation of the simple polygon in linear time and then applying the partitioning algorithm of Liou et al. [1989]. For arbitrary rectilinear polygons which may also contain degenerated holes (single points not contained in the polygon but surrounded by polygon points only) the best known algorithm runs in $O(n^{\frac{3}{2}} \log n)$ time (Soltan and Gorpinevich [1993]) while the best known lower bound for runtime is $\Omega(n \log n)$ (Liou et al. [1989]). Note that the Polygon Partitioning Problem with degenerated holes was claimed to be NP-hard by Lingas [1982] until disproven (unless P=NP) by the algorithm of Soltan and Gorpinevich [1993].

We now want to give some insight into the basic techniques used in the best known algorithms for the Polygon Partitioning Problem. Let P be an arbitrary rectilinear polygon in the following.

Definition 4.1. *A* chord *of P is a line segment whose interior is contained in P° and whose endpoints are contained in ∂P. A horizontal or vertical chord where both endpoints are concave vertices of P is called* special.

Now let m be the number of concave vertices, h the number of holes, c the maximum number of non-intersecting special chords of P, and r_{opt} the number of rectangles in an optimal solution for the Polygon Partitioning Problem. The following theorem gives a nice characterization for all optimal solutions.

Theorem 4.2. *Lipski et al. [1979], Ohtsuki [1982], Ferrari et al. [1984]*
$r_{\text{opt}} = m - c - h + 1$

This theorem can be seen as the basic module for most algorithms solving the Polygon Partitioning Problem optimally. The key step is to find a maximum set of non-intersecting special chords. This can be modeled as a maximum stable set in the bipartite graph $G = (A \,\dot{\cup}\, B, E)$, where A and B contain one vertex for each horizontal or vertical special chord, respectively, and E contains an edge between two vertices if their corresponding special chords intersect.

Using the fact that G is bipartite, it is sufficient to compute a maximum matching in G from which a minimum vertex cover and finally a maximum stable set can be easily obtained, as is well known. This approach was first used by Lipski et al. [1979] who applied the previously fastest known algorithm for finding maximum matchings in bipartite graphs by Hopcroft and Karp [1973], resulting in an $O(n^{\frac{5}{2}} \log n)$ time algorithm.

The best known algorithms for the Polygon Partitioning Problem cited above all use the same approach, but are based on techniques to solve the matching problem faster by exploiting the special structure of the bipartite graph.

4.2 Polygon Decomposition and Design Rule Checking

We now consider a problem already touched in Chapter 3: How to decompose polygons into rectangle sets such that certain design rules can be checked on the

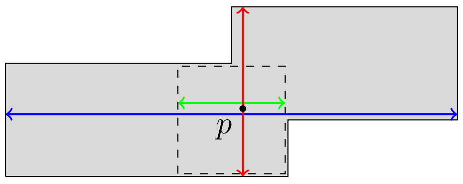


Figure 4.1: Three different possiblilities to measure the width of the polygon at p: In blue the horizontal extension of the polygon at p, in red the vertical extension at p, and in green the edge length of a largest inscribed square containing p (see Definitions 4.5 and 4.12).

resulting rectangles, giving the same results as if checked on the polygons? A related problem is to preprocess polygons in a way such that later design rule checking queries can be processed faster. We will show that these problems can be solved efficiently for one of the most important classes of design rules, width-dependent mindist rules.

We first have to specify how the width of a polygon at a certain point is defined. There are different possible measures as illustrated in Figure 4.1. We will consider all depicted width measures (being the most important used in the context of VLSI design rules), the one-dimensional width (x-width in blue, y-width in red in Figure 4.1) in Section 4.2.1 and the two-dimensional width (green in Figure 4.1) in Section 4.2.2.

Now let us assume that we are given one of these width measures as an oracle function w, such that $\mathrm{w}(p, P)$ is the width at point $p \in P$ for a rectilinear polygon P.

Definition 4.3. *Given a width measure* w*, the* width class of size s *is the set* $C_s := \{p \in P \mid \mathrm{w}(p, P) = s\}$. *A decomposition of P into interior-disjoint rectangles with the property that $\mathrm{w}(\cdot, P)$ is constant within the interior of each rectangle is called a* width decomposition of P with respect to w.

Definition 4.4. *A (width-dependent)* mindist rule *is a function* $\delta : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. *For two rectilinear polygons P and Q and two points $p \in P$ and $q \in Q$, we say that p and q* satisfy δ *if* $d_2(p, q) \geq \delta(\mathrm{w}(p, P), \mathrm{w}(q, Q))$. *We say that P and Q* satisfy δ *if* $d_2(p, q) \geq \delta(\mathrm{w}(p, P), \mathrm{w}(q, Q))$ *for all* $p \in P, q \in Q$.

The following important subproblems in design rule checking (see Schulte [2012]) have to be solved millions of times in VLSI routing.

POINT DISTANCE RULE CHECKING PROBLEM	
Instance:	Two rectilinear polygons P and Q, two points $p \in P$ and $q \in Q$, and a mindist rule δ.
Task:	Report if p and q satisfy δ.

Polygon Distance Rule Checking Problem	
Instance:	Two rectilinear polygons P and Q, and a mindist rule δ.
Task:	Report if P and Q satisfy δ.

Since polygons representing pins or blockages on a chip are fixed during VLSI routing, it makes sense to spend some preprocessing time on these structures to speed up the frequent distance rule checking queries later on. The Point Distance Rule Checking Problem essentially reduces to querying $\mathrm{w}(p, P)$, so we get to the following problem.

Polygon Width Query Problem	
Instance:	A rectilinear polygon P.
Task:	Build a data structure which can report for any point $p \in P$ the width $\mathrm{w}(p, P)$ fast.

Given a width decomposition of P, the Polygon Distance Rule Checking Problem reduces to checking distances between rectangles only (which is actually done in BonnRoute, see Schulte [2012] for more details), motivating the following problem.

Polygon Width Decomposition Problem	
Instance:	A rectilinear polygon P.
Task:	Compute a width decomposition of P with respect to w.

We now show how to solve the Polygon Width Query Problem and the Polygon Width Decomposition Problem efficiently for both one-dimensional and two-dimensional width measures. Typically the polygons occuring in the above applications are simple in practice, but we also cover polygons with holes.

4.2.1 One-Dimensional Width Decomposition

Definition 4.5. *The x-*width *(y-*width*) of a rectilinear polygon P at a point $p \in P$, denoted as $\mathrm{w_x}(p, P)$ ($\mathrm{w_y}(p, P)$), is the maximum horizontal (vertical) length of a rectangle R with $p \in R \subseteq P$, where R is called x-*width (y-width) representative for *p.*

In contrast to the examples in Figure 4.1 we here use the maximum length of a rectangle instead of a line segment to simplify degenerated situations. Our goal is to decompose a given rectilinear polygon P into an interior-disjoint rectangle set $\mathcal{S}$ containing x-width (or y-width) representatives for all $p \in P$. Clearly such an $\mathcal{S}$ also serves as a width decomposition, solving the Polygon Width Decomposition Problem with respect to x-width (or y-width).

Intuitively such a decomposition forces its rectangles to be maximally extended (see Figure 4.2) which is formalized in the following definition.

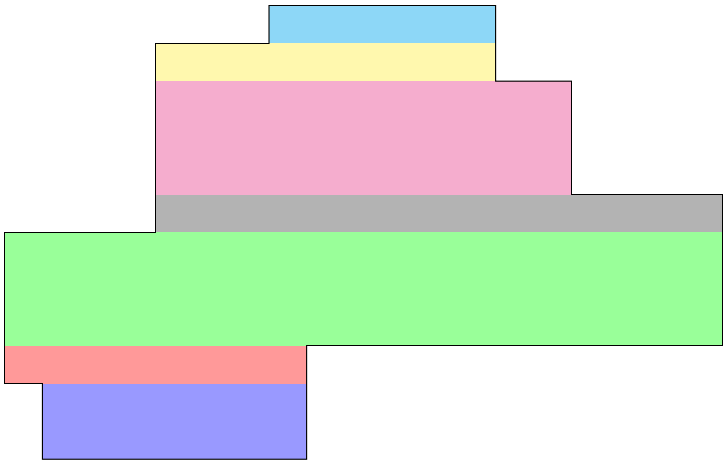


Figure 4.2: A width decomposition of a simple polygon with respect to $\mathrm{w_x}$.

Definition 4.6. *We call a rectangle set $\mathcal{S}$ x-*maximized *(y-*maximized*) if the left and right edge (the bottom and top edge) of each rectangle in $\mathcal{S}$ are completely contained in $\partial(\bigcup \mathcal{S})$.*

See Figure 4.3 for examples. The following lemma shows that our desired decomposition has to be an x-maximized (or y-maximized) rectangle set.

Lemma 4.7. *Given a polygon P and an interior-disjoint rectangle set $\mathcal{S}$ covering the same area as P, then $\mathcal{S}$ is x-maximized (y-maximized) if and only if for each point $p \in P$ there is an x-width (y-width) representative in $\mathcal{S}$.*

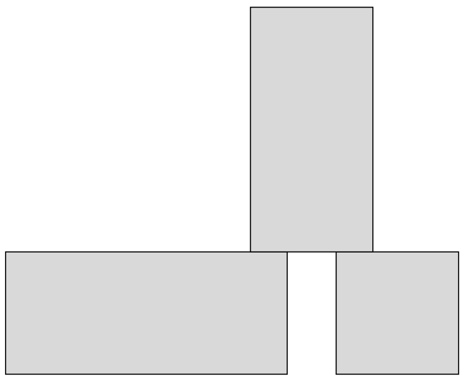
Proof. W.l.o.g. we consider the horizontal case. First suppose that $\mathcal{S}$ is not x-maximized, then we choose a rectangle R in $\mathcal{S}$ with a segment $s = \{x\} \times [y_1, y_2]$ in its left or right border not intersecting ∂P. Now we choose a point $p \in R^\circ$ with $y_1 < y(p) < y_2$. We have $\mathrm{w_x}(p, R) < \mathrm{w_x}(p, P)$ and the only rectangle of $\mathcal{S}$ covering p is R (note that $\mathcal{S}$ is interior-disjoint), so p has no x-width representative in $\mathcal{S}$.

On the other hand, an x-maximized rectangle set $\mathcal{S}$ covering P clearly contains x-width representatives for any point $p \in P$ since every rectangle in $\mathcal{S}$ has maximal extension in x-dimension. $\square$

Now we can reformulate our decomposition problem as follows, adding the objective to use as few rectangles as possible.

Polygon Stripe Partitioning Problem	
Instance:	A rectilinear polygon P and a dimension d (x or y).
Task:	Compute a minimum d-maximized and interior-disjoint rectangle set covering the same area as P.

We now characterize optimal solutions for this problem. Let r_{hor} and r_{ver} be optimal solutions for the Polygon Stripe Partitioning Problem for dimension


(a) This rectangle set is x-maximized, but not y-maximized.

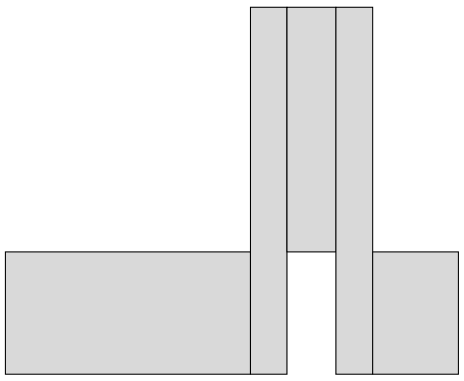

(b) This rectangle set is y-maximized, but not x-maximized.

Figure 4.3

x and y, respectively, and let c_{hor} and c_{ver} be the number of horizontal and vertical special chords of P, respectively. We need the following lemma which is similar to Theorem 4.2 (again m is the number of concave vertices and h is the number of holes in P).

Lemma 4.8. $r_{\text{hor}} = m - c_{\text{hor}} - h + 1$ *and* $r_{\text{ver}} = m - c_{\text{ver}} - h + 1$.

Proof. W.l.o.g. we consider the horizontal case. Our proof is constructive, more precisely we construct the unique optimal solution for the Polygon Stripe Partitioning Problem in dimension x and deduce the claimed equality from it.

Let C be the set of horizontal chords in P which contain at least one concave vertex of P. Let k be the number of connected components into which $P \setminus \bigcup C$ decomposes. The boundaries of these connected components define an x-maximized and interior-disjoint rectangle set $\mathcal{S}$ which covers P, where $|\mathcal{S}| = k$. We claim that $k = m - c_{\text{hor}} - h + 1$.

We first assume $h = 0$. Then P is simply-connected, and successively subtracting all chords in C from P increases the number of connected components by $|C|$ (one in each step), thus we obtain $k = |C| + 1$. We have $m = |C| + c_{\text{hor}}$ because the elements in C cover each concave vertex of P, but we have to count special chords of C twice. The combination of both equalities yields $k = m - c_{\text{hor}} + 1$.

If $h > 0$, then we choose an arbitrary hole H and a leftmost vertical segment s in ∂H. We choose a rectangle $R \subseteq P$ whose right border is contained in s° and whose left border is contained in the interior of another vertical segment in $\partial P \setminus \partial H$, see Figure 4.4. The closure of $P \setminus R$ defines a new polygon which we denote as P'. We have $C' = C$, $h' = h - 1$, $m' = m$, $c'_{\text{hor}} = c_{\text{hor}}$, and $k' = k + 1$, where C', h', m', c'_{hor}, and k' are defined analogously. By induction we obtain $k = m - c_{\text{hor}} - h + 1$ as claimed.

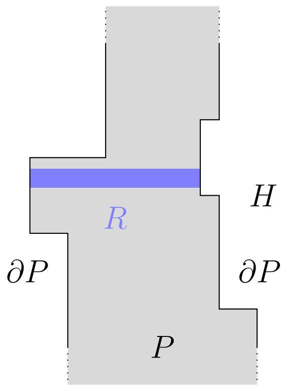


Figure 4.4

Since $\mathcal{S}$ is a valid solution for the Polygon Stripe Partitioning Problem in dimension x, we have $r_{\text{hor}} \leq |\mathcal{S}| = m - c_{\text{hor}} - h + 1$. On the other hand, rectangles of an arbitrary x-maximized and interior-disjoint rectangle set cannot cross any chord of C, implying that $\mathcal{S}$ is the unique optimal solution and $r_{\text{hor}} = k = m - c_{\text{hor}} - h + 1$. □

The unique optimal solution for the horizontal Polygon Stripe Partitioning Problem given in the above proof can be constructed in optimal linear time for simple polygons. Fournier and Montuno [1984] described a decomposition of a general simple polygon into trapezoids which gives exactly this optimal solution when applied to a rectilinear simple polygon. They also showed that this decomposition can be obtained from a polygon triangulation in $O(n)$ time, thus for simple polygons we can apply Chazelle's famous linear time triangulation algorithm (Chazelle [1991]) and achieve a linear total runtime for the Polygon Stripe Partitioning Problem.

Despite the theoretically optimal runtime, the above approach is not very practicable since the linear time polygon triangulation method is quite involved. In the BonnTools software collection we implemented a standard sweep line approach, where (for the x-maximized case) the sweep line moves from bottom to top and keeps track of all intersecting vertical polygon segments at each time in a balanced binary search tree. Along the way all rectangles of the partition can be easily built. The worst-case runtime of this algorithm is $O(n \log n)$ which can be easily proved by standard arguments for sweep line algorithms. Because there are only simple data structures involved, it is very fast in practice, and it also generalizes to polygons with holes. See [Boissonnat et al., 1998, Chapter 12] for more details on a similar sweep approach.

We obtain the following results for the Polygon Width Decomposition Problem and Polygon Width Query Problem with respect to one-dimensional width.

Theorem 4.9. *Given a rectilinear polygon P and a dimension d (x or y), a width* *decomposition of P with respect to* $\mathrm{w_d}$ *containing a minimum number of rectangles ($m - c_{\text{hor}} - h + 1$ many) can be computed in $O(n \log n)$ time. If P is simple, then the runtime reduces to $O(n)$.*

Proof. By Lemma 4.7, solving the Polygon Stripe Partitioning Problem does the job, producing $m - c_{\text{hor}} - h + 1$ rectangles by Lemma 4.8. The runtimes are obtained from the results above. □

Corollary 4.10. *Given a rectilinear polygon P and a dimension d (x or y), we can build a data structure in $O(n \log n)$ time using $O(n)$ space which is able to report a d-width representative for any given point $p \in P$ in $O(\log n)$ time. If P is simple, then the preprocessing time reduces to $O(n)$.*

Proof. We compute a width decomposition of P, build a point location data structure in time $O(n)$ (see e.g. Kirkpatrick [1983]) which reports for any $p \in P$ its (at most two) covering rectangles in time $O(\log n)$, and we report the larger one (w.r.t. d-width) of these rectangles. □

Since optimal stripe partitioning (in x or y dimension) serves as a nice unique representation for rectilinear polygons as rectangle sets, it is an interesting question how far from optimal (in terms of the number of rectangles used for the representation) this representation is in the worst case. Next we prove some sharp bounds, justifying that stripe partitioning is used as the standard decomposition method for rectilinear polygons in the BonnTools. For this the sweepline approach sketched above is used. In the following theorem we use the same notations as in Lemma 4.8.

Theorem 4.11. *Let P be a rectilinear polygon with h holes. We have*

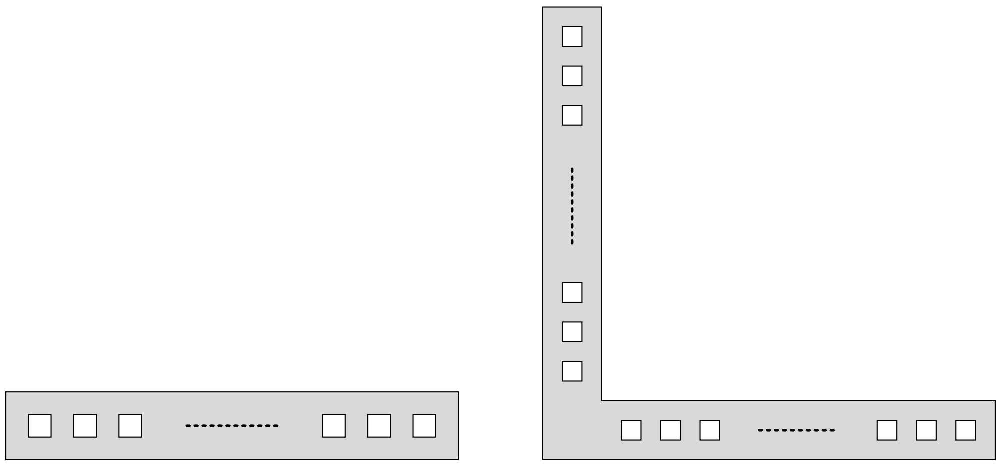
$$\frac{r_{\text{hor}}}{r_{\text{opt}}} \leq \begin{cases} 2 & \textit{if } h = 0 \\ 3 & \textit{else} \end{cases}, \qquad \frac{r_{\text{ver}}}{r_{\text{opt}}} \leq \begin{cases} 2 & \textit{if } h = 0 \\ 3 & \textit{else} \end{cases}, \qquad \frac{\min(r_{\text{hor}}, r_{\text{ver}})}{r_{\text{opt}}} \leq \begin{cases} \frac{3}{2} & \textit{if } h = 0 \\ 2 & \textit{else} \end{cases}$$

The above inequalities are best possible. In particular, running an algorithm for the Polygon Stripe Partitioning Problem *for both dimensions and taking the solution with less rectangles is a 2-approximation algorithm for the* Polygon Partitioning Problem. *If the polygon is simple we have an approximation guarantee of $\frac{3}{2}$.*

Proof. The first estimation follows by

$$\frac{r_{\text{hor}}}{r_{\text{opt}}} = \frac{m - c_{\text{hor}} - h + 1}{m - c - h + 1} \leq \frac{m - h + 1}{\frac{m}{2} - h + 1} \leq \begin{cases} 2 & \text{if } h = 0 \\ \frac{\frac{3}{4}m + 1}{\frac{1}{4}m + 1} \leq 3 & \text{else} \end{cases}$$

Here we use Theorem 4.2 and Lemma 4.8 for the first equality, $c \leq \frac{m}{2}$ for the first inequality, and $h \leq \frac{m}{4}$ for the else case. The proof for $\frac{r_{\text{ver}}}{r_{\text{opt}}}$ is analogous.



(a) Polygon with $h > 0$ holes, $r_{\text{ver}} = 3h + 1$, $r_{\text{opt}} = r_{\text{hor}} = h + 3$, $\frac{r_{\text{ver}}}{r_{\text{opt}}} \overset{h\to\infty}{\to} 3$.

(b) Polygon with $h > 0$ holes, $r_{\text{hor}} = r_{\text{ver}} = 4h + 4$, $r_{\text{opt}} = 2h + 6$, $\frac{\min(r_{\text{hor}}, r_{\text{ver}})}{r_{\text{opt}}} \overset{h\to\infty}{\to} 2$.

Figure 4.5

The third estimation follows by

$$\begin{aligned}\frac{\min(r_{\text{hor}}, r_{\text{ver}})}{r_{\text{opt}}} &= \frac{m - \max(c_{\text{hor}}, c_{\text{ver}}) - h + 1}{m - c - h + 1} \le \frac{m - \frac{c}{2} - h + 1}{m - c - h + 1} \\ &\le \frac{\frac{3}{4}m - h + 1}{\frac{m}{2} - h + 1} \le \begin{cases} \frac{3}{2} & \text{if } h = 0 \\ \frac{\frac{2}{4}m+1}{\frac{1}{4}m+1} \le 2 & \text{else} \end{cases}\end{aligned}$$

We use Theorem 4.2 and Lemma 4.8 for the first equality, $c_{\text{hor}} + c_{\text{ver}} \ge c$ for the first inequality, $c \le \frac{m}{2}$ for the second inequality (note that the fraction attains its maximum value for $c = \frac{m}{2}$), and $h \le \frac{m}{4}$ for the else case.

The examples in Figure 4.5 and Figure 4.6 show that the bounds are best possible for all cases.

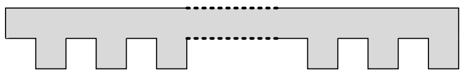
□

In practice, stripe partitioning is typically much closer to the optimal solution as expressed by these worst-case bounds.

4.2.2 Two-Dimensional Width Decomposition

We now consider a width measure for polygons taking both dimensions into account.

Definition 4.12. *The (two-dimensional)* width *of a rectilinear polygon P at a point $p \in P$, denoted as* $\mathrm{w}_{2\mathrm{D}}(p, P)$*, is the edge length of a largest square Q with $p \in Q \subseteq P$. We call Q* width representative *for p.*


(a) Polygon with k spikes, $r_{\text{ver}} = 2k$, $r_{\text{opt}} = r_{\text{hor}} = k+1$, $\frac{r_{\text{ver}}}{r_{\text{opt}}} \overset{k\to\infty}{\to} 2$.

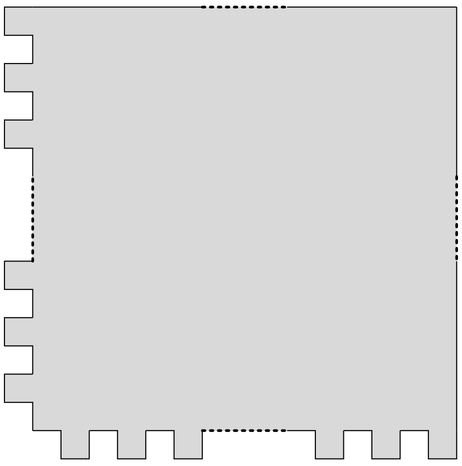

(b) Polygon with k spikes, $r_{\text{hor}} = r_{\text{ver}} = \frac{3}{2}k$, $r_{\text{opt}} = k+1$, $\frac{\min(r_{\text{hor}}, r_{\text{ver}})}{r_{\text{opt}}} \overset{k\to\infty}{\to} \frac{3}{2}$.

Figure 4.6

See the green line in Figure 4.1 on page 20 for an example. In the following we will always use w_{2D} as width measure. We consider the following two problems.

Polygon 2D Width Decomposition Problem	
Instance:	A rectilinear polygon P.
Task:	Compute a width decomposition of P with respect to w_{2D}.

Polygon 2D Width Query Problem	
Instance:	A rectilinear polygon P.
Task:	Build a data structure which can report for any point $p \in P$ a width representative for p with respect to w_{2D} fast.

See Figure 4.7 for an example of a width decomposition with respect to w_{2D}. We first want to mention some work related to these problems.

If we are only interested in a largest width representative, that means a largest inscribed square in a polygon, we are dealing with *largest empty circle* problems which have been well-known in computational geometry for a long time. In its original form (Shamos and Hoey [1975]) a finite set of points in the plane is given and one is searching for the largest circle centered within the convex hull of the points and not containing any of the points in its interior. Shamos and Hoey [1975] showed how to solve this problem in $O(n \log n)$ time using Voronoi diagrams. The problem of finding a *largest empty square* can be solved similarly in $O(n \log n)$ time using L_∞ Voronoi diagrams (Hwang [1979], Lee and Wong [1980]). If we are given pairwise non-intersecting (but possibly touching) line segments instead of points,

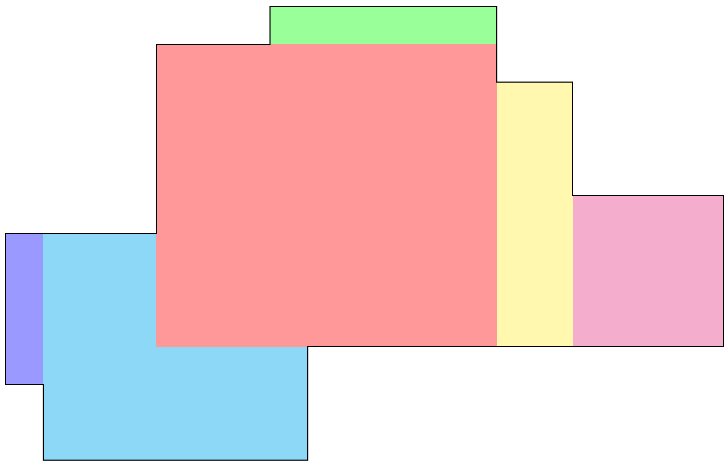


Figure 4.7: A width decomposition of a simple polygon with respect to $\mathrm{w}_{2\mathrm{D}}$.

then a largest empty square can still be obtained in $O(n \log n)$ time by using the L_∞ Voronoi diagrams of line segments (Papadopoulou and Lee [2001]). This can also be used to obtain a largest inscribed square in a (not necessarily rectilinear) polygon by taking border edges of the polygon as segments.

Kaplan and Sharir [2012] considered the problem where the width measure $\mathrm{w}(p, P)$ is defined in terms of the radius of the largest disk that contains p but whose interior avoids P, where P is a set of n points in the plane. They showed that $\mathrm{w}(p, P)$ can be queried in time $O(\log^2 n)$ after a preprocessing step taking $O(n \log^2 n)$ time and $O(n \log n)$ space.

A similar problem, where P is a simple polygon and $\mathrm{w}(p, P)$ is the radius of the largest disk inside of P containing p, was studied by Augustine et al. [2013]. They provide a solution where $\mathrm{w}(p, P)$ can be queried in time $O(\log n)$ using $O(n \log^2 n)$ preprocessing time and $O(n \log n)$ space.

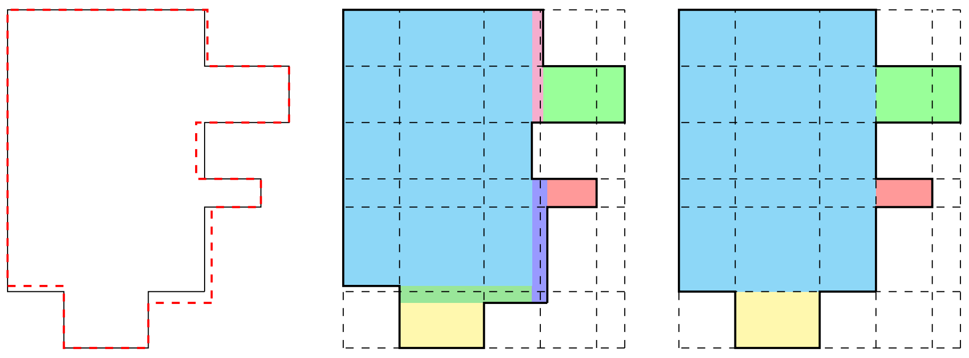
Boissonnat et al. [2001] showed that it is possible to preprocess a convex polygon P in $O(n)$ time and space, such that given as a query a set of k points, the largest disk inside of P enclosing all k points may be computed in $O(k \log n)$ time and $O(n + k)$ space.

Augustine et al. [2010] proved that the problem of finding a rectangle of maximum area that avoids a given point set can be solved in $O(\log n)$ query time, given $O(n^2 \log n)$ time and space preprocessing.

The latter problem was also solved by Kaplan et al. [2012], using $O(n\alpha(n) \log^4 n)$ preprocessing time, $O(n\alpha(n) \log^3 n)$ space, and $O(\log^4 n)$ query time, where $\alpha(n)$ is the inverse Ackermann function.

Dumitrescu and Jiang [2012] studied the number of maximal empty axis-parallel rectangles (or boxes, in higher dimension) among a randomly chosen point set.

In the following we present a new algorithm solving the Polygon 2D Width Decomposition Problem in optimal linear time for simple rectilinear polygons,



(a) Degenerated polygon (black) and slightly perturbed polygon satisfying our general position assumptions (red).

(b) Width decomposition for the perturbed polygon. The Hanan grid of the original polygon is drawn as dashed lines.

(c) A width decomposition for the original polygon is obtained by rounding corners of width classes to the next vertex on the Hanan grid.

Figure 4.8: Illustration of degenerated case.

reporting a linear number of rectangular width regions. Using this result, we are able to build a data structure in linear time and size which is able to answer width queries in $O(\log n)$ time. For rectilinear polygons with holes the preprocessing time increases to $O(n \log n)$. Note that for our applications in VLSI design polygons are usually simple.

To simplify our proofs, we assume that P is in general position in the following sense. We assume that no two horizontal segments of ∂P have the same y-coordinate and no two vertical segments of ∂P have the same x-coordinate. Let $c_1, c_2, \dots, c_n$ be the x-coordinates of vertical segments and the y-coordinates of horizontal segments of ∂P (ordered arbitrarily). We assume that for any four distinct indices $i, j, k, l \in \{1, 2, \dots, n\}$ we have $c_i - c_j \neq c_k - c_l$. In other words, no difference between two coordinates occurs more than once.

Of course, polygons occuring in VLSI design do not fulfill these assumptions in general. Typically there are many segments that have equal x- or y-coordinates. However, we can achieve all conditions by slightly perturbing the input data. After applying our algorithm to the perturbed polygon, we can round the borders of the width classes to the next x- and y-coordinates occuring in vertices from the original polygon. This works because the shapes of width classes vary continuously with changes of the coordinates in P. This procedure is illustrated in Figure 4.8. For more details on standard perturbation techniques for geometric algorithms see Seidel [1998]. Instead of perturbing the data, the algorithm can also be adapted such that the general position assumption is not necessary.

Our algorithm makes use of the L_∞ *Voronoi diagram* of P.

Definition 4.13. *The L_∞* bisector *of two line segments s_1, s_2 is the set $\{x \in \mathbb{R}^2 \mid \mathrm{d}_\infty(s_1, x) = \mathrm{d}_\infty(s_2, x)\}$. The L_∞* Voronoi diagram *of P is the set of all points $p \in P$ for which there exist at least two different border segments s_1, s_2 of P with*

$$\mathrm{d}_\infty(p, s_1) = \mathrm{d}_\infty(p, s_2) = \min_{q \in \partial P} \mathrm{d}_\infty(p, q).$$

The L_∞ Voronoi diagram can be computed in time $O(n \log n)$ for arbitrary polygons (see Papadopoulou and Lee [2001]) and in time $O(n)$ for simple polygons (see Chin et al. [1999]).

We will skip the term L_∞ in the following, when there is no risk for confusion. It is easy to see that the Voronoi diagram is a subset of the union of all bisectors between line segments of the polygon. Therefore, bisectors define the structure of Voronoi diagrams and can be viewed as their basic modules.

Bisectors of line segments may contain two-dimensional parts, see Figure 4.9a. To avoid this, we use lines with slope ± 1 as bisectors for touching segments instead of original L_∞ bisectors (see again Figure 4.9a). We call them *refined bisectors*. We are only interested in the parts of the bisectors within P. By our general position assumption, no border segment of an inscribed square in P can have proper intersection with two non-touching segments of ∂P. This implies that two-dimensional bisector parts between non-touching segments do not appear within P. For more details see [Aurenhammer et al., 2013, Chapter 7] and Papadopoulou and Lee [2001].

Using the refined bisectors to define the Voronoi diagram (more exactly, intersecting the Voronoi diagram with the union of all refined bisectors between any two border segments) results in a *refined Voronoi diagram* which we denote as $\mathcal{V}_\infty(P)$. In the following, we only consider this Voronoi diagram which consists of line segments only. We call points on $\mathcal{V}_\infty(P)$ which lie on ∂P or have at least three incident segments in $\mathcal{V}_\infty(P)$ *Voronoi vertices* and the parts between these vertices *Voronoi edges*. For more information on Voronoi diagrams in general and its applications see Aurenhammer et al. [2013] and Okabe et al. [2009].

We associate with every point p lying on $\mathcal{V}_\infty(P)$ the unique square $Q(p)$ centered at p and touching the nearest segments of ∂P. The following simple but important lemma is the key relation between the Polygon 2D Width Decomposition Problem and the Voronoi diagram which we use for our algorithm.

Lemma 4.14. *All width representatives for points in P are of the form $Q(q)$ for some q lying on $\mathcal{V}_\infty(P)$.*

Proof. Each width representative Q for some point p touches ∂P in at least two non-touching segments, otherwise Q could be enlarged within P while still containing p, a contradiction to the definition of width representatives. Therefore, the center of Q lies on $\mathcal{V}_\infty(P)$. □

Before we show how a width decomposition can be obtained efficiently by using this fact we need to prove some structural properties of $\mathcal{V}_\infty(P)$.

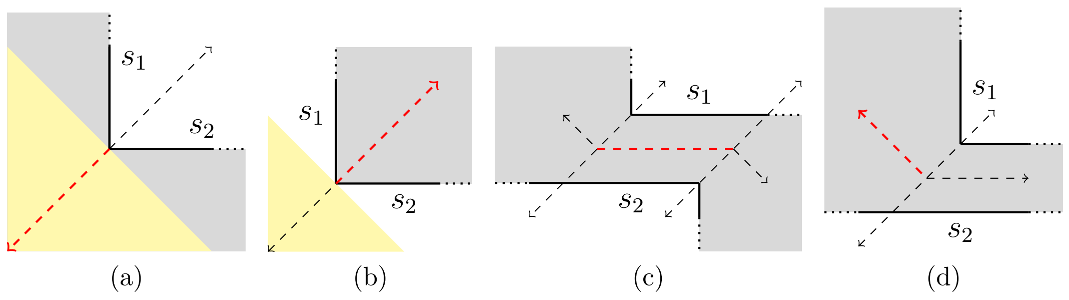


Figure 4.9: Each figure shows a section of a rectilinear polygon (gray) and refined bisectors of its segments in dashed lines. In Figures (a) and (b) the yellow areas are two-dimensional parts belonging to the original, not refined bisectors. Parts contributing to the Voronoi edge of s_1 and s_2 are red.

Lemma 4.15. *All Voronoi edges of $\mathcal{V}_\infty(P)$ are horizontal, vertical or diagonal segments.*

Proof. First note that for each Voronoi edge e there exist two segments s_1, s_2 of ∂P, such that each point on e has the same d_∞-distance to s_1 and s_2, and there is no segment of ∂P with smaller distance (just by definition of the Voronoi diagram, note that this property still holds with our redefined bisectors).

If s_1 and s_2 are touching, then the bisector of the segments is a diagonal line, see Figures 4.9a and 4.9b. Thus the Voronoi edge, being a connected subset of the bisector, is a diagonal segment.

If s_1 and s_2 are non-touching and parallel (say both horizontal, w.l.o.g.), then by our general position assumptions the segments have different y-coordinates. The part of the bisector inside of P consists of at most three pieces, and both endpoints of the horizontal piece intersect bisectors induced by vertices incident to s_1 and s_2 (see Figure 4.9c). The Voronoi edge of s_1 and s_2 clearly cannot cross these bisectors and thus consists of the horizontal segment only.

If s_1 is horizontal and s_2 is vertical (or vice versa), then the part of the bisector inside of P again consists of at most three pieces (see Figure 4.9d). Here only one diagonal part of the bisector contributes to the Voronoi edge, because the Voronoi edge cannot cross the bisector induced by the polygon vertex which causes the break in the bisector of s_1 and s_2. □

Lemma 4.15 implies that $\mathcal{V}_\infty(P)$ can be interpreted as a planar straight-line graph $G = (V, E_{\text{orth}} \dot\cup E_{\text{diag}})$, where E_{orth} is the set of horizontal and vertical edges and E_{diag} is the set of diagonal edges. In the following we identify vertices and edges with their embeddings in the plane. We now collect some statements about the structure of G for later use.

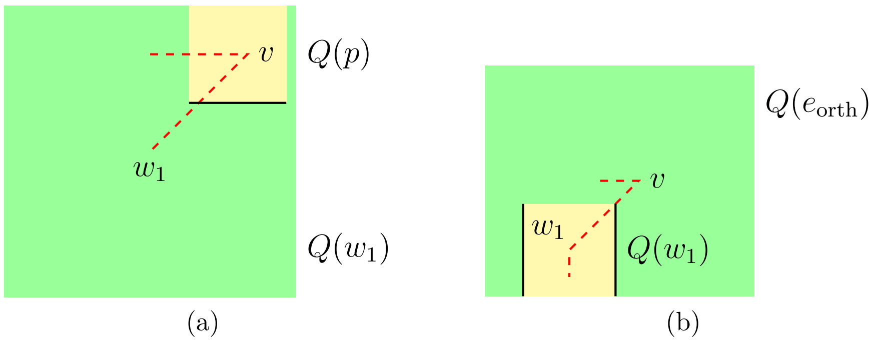


Figure 4.10: Illustration of contradictions in Lemma 4.19 (G in red).

The next two lemmas follow easily from the definition of $\mathcal{V}_\infty(P)$ and our general position assumption, see also the proof of Lemma 4.15 and Figure 4.9.

Lemma 4.16. *For any edge $e = \{v, w\} \in E_{orth}$ and any point q on e, ∂P intersects both border segments of $Q(q)$ which are parallel to e, at least one of them in its interior. If $q \notin \{v, w\}$, then both are intersected in their interiors.*

Lemma 4.17. *For any edge $e \in E_{diag}$ and any distinct $v', w' \in e$ we have either $Q(v') \subsetneq Q(w')$ or $Q(w') \subsetneq Q(v')$.*

Lemma 4.18. *For all $v \in V$ we have* $|\delta(v) \cap E_{orth}| = \begin{cases} 0, & \text{if } v \in \partial P \\ 1, & \text{if } v \in P \setminus \partial P \end{cases}$.

Proof. If $v \in \partial P$, then v is a vertex of P and the only Voronoi edge containing v is diagonal (see Figures 4.9a and 4.9b), proving $|\delta(v) \cap E_{\text{orth}}| = 0$.

If v is a Voronoi vertex in $P \setminus \partial P$, then by our general position assumptions there is exactly one border segment of $Q(v)$ whose interior does not intersect ∂P, thus we can move $Q(v)$ in direction of this segment while touching two parallel border segments of ∂P, implying the existance of an incident horizontal or vertical Voronoi edge. By the general position assumptions we cannot have two such incident edges, so $|\delta(v) \cap E_{\text{orth}}| = 1$. □

Lemma 4.19. *Let $e_{\text{orth}}, e_{\text{diag}} \in E(G)$ be incident edges forming a $45°$ angle, where $e_{\text{diag}} = \{v, w_1\}$ is the diagonal and $e_{\text{orth}} = \{v, w_2\}$ the horizontal or vertical edge. Then $w_1 \in \partial P$.*

Proof. Suppose e_{orth} is horizontal and e_{diag} leaves the right endpoint of e_{orth} to the lower left, all other cases are symmetric.

By Lemma 4.17, we have either $Q(v) \subsetneq Q(w_1)$ or $Q(w_1) \subsetneq Q(v)$. If $Q(w_1)$ had greater width than $Q(v)$ (see Figure 4.10a), then we could choose a point p in the interior of e_{orth} such that the lower border segment of $Q(p)$ (the black line in

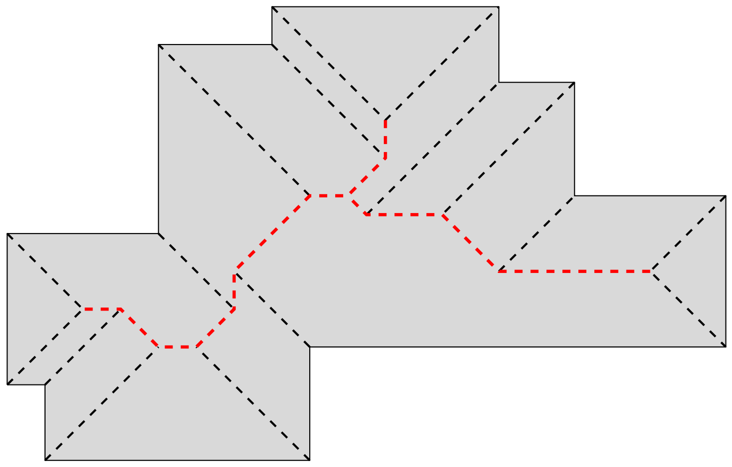


Figure 4.11: Rectilinear polygon P, refined Voronoi diagram $\mathcal{V}_\infty(P)$ (all dashed lines), and Voronoi core G_{VC} (red dashed lines).

Figure 4.10a) is contained in the interior of $Q(w_1)$, but by Lemma 4.16 this segment also intersects ∂P, a contradiction.

So we have $Q(e_{\mathrm{orth}}) \supsetneq Q(w_1)$ as illustrated in Figure 4.10b. Suppose $w_1 \notin \partial P$, then by Lemma 4.18 w_1 is incident to an edge $e'_{\mathrm{orth}} \in E_{\mathrm{orth}}$. By Lemma 4.16, ∂P intersects both segments of $Q(w_1)$ parallel to e'_{orth} (the black lines in Figure 4.10b), at least one in its interior, so ∂P intersects the interior of $Q(e_{\mathrm{orth}})$, a contradiction.

Therefore, we have $w_1 \in \partial P$ as claimed. □

We now define the *Voronoi core* G_{VC} as the embedded planar graph that is obtained from G by deleting all vertices lying on ∂P (the leaves of G) and their incident edges (see Figure 4.11). We note that this graph is a subset of the *medial axis* introduced by Blum et al. [1967] which for rectilinear polygons coincides with the *straight skeleton* introduced by Aichholzer et al. [1995]. The medial axis has applications in pattern recognition (Duda and Hart [1973], Rosenfeld [1986]), solid modeling (Vermeer [1993]), and mesh generation (Gürsoy and Patrikalakis [1992])). The straight skeleton is used in computer graphics (Tanase and Veltkamp [2003]), graph drawing (Bagheri and Razzazi [2012]), and for roof construction (Aichholzer et al. [1995], Ahn et al. [2013]). The roots of straight skeletons used for roof construction actually go back to the 19th century ([Peschka, 1877, p. 86-122]) as pointed out by Aichholzer et al. [2012].

Corollary 4.20. *For each edge $e \in E_{orth}$, all incident edges in G_{VC} are diagonal and form $135°$ angles with e.*

Proof. Incident edges must be diagonal by Lemma 4.18 and cannot form $45°$ angles with e because G_{VC} does not contain edges ending in ∂P (see Lemma 4.19), thus they must form $135°$ angles as claimed. □

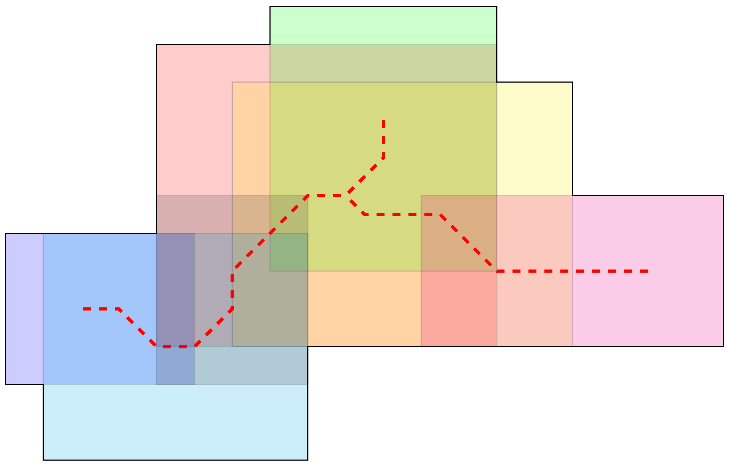


Figure 4.12: Rectilinear polygon with Voronoi core G_{VC} (red dashed lines) and overlapping edge rectangles (different colors).

By Corollary 4.20, we can provide E_{orth} with a natural notion of diagonal neighborhood. For an edge $e \in E_{\mathrm{orth}}$ we define $n_{\nearrow}(e)$, the *upper right neighbor* of e, as the edge $f \in E_{\mathrm{orth}}$ that is reached from the top or right end of e, respectively, when following the diagonal edge in the top right direction. If no such diagonal edge exists, we set $n_{\nearrow}(e) := \emptyset$. In the same sense the *upper left neighbor* $n_{\nwarrow}(e)$, the *lower right neighbor* $n_{\searrow}(e)$, and the *lower left neighbor* $n_{\swarrow}(e)$ are defined.

We need the following stronger version of Lemma 4.14.

Lemma 4.21. *All width representatives for points in P are of the form $Q(q)$ for some point q lying on an edge of E_{orth}.*

Proof. By Lemma 4.14, we already have $Q(q)$ for q lying on $\mathcal{V}_\infty(P)$. Suppose q lies on the interior of a diagonal edge e, then by Lemma 4.17 moving q along e in one direction gives an empty square larger than $Q(q)$ and containing $Q(q)$, thus $Q(q)$ cannot be a width representative for any point in P. Clearly q cannot lie on ∂P, thus by Lemma 4.18 it must lie on an edge of E_{orth}. □

For $e \in E_{\mathrm{orth}}$ the union of all $Q(q)$ for points q on e is a rectangle which we denote as $Q(e)$ and which we call *edge rectangle* (see Figure 4.12). We define $\mathrm{w}(e)$ as the width (the smaller edge length) of $Q(e)$ and $C_e := C_{\mathrm{w}(e)}$ (see Definition 4.3 on page 20) for $e \in E_{\mathrm{orth}}$. We can now extend Lemma 4.21 as follows.

Lemma 4.22. *For all $p \in P$ we have*

$$\mathrm{w}_{2\mathrm{D}}(p, P) = \max\{\mathrm{w}(e) \mid e \in E_{orth} \wedge p \in Q(e)\}$$

and each width class can be written as

$$C_e = Q(e) \setminus \bigcup_{\substack{f \in E_{orth} \\ \mathrm{w}(f) > \mathrm{w}(e)}} Q(f).$$

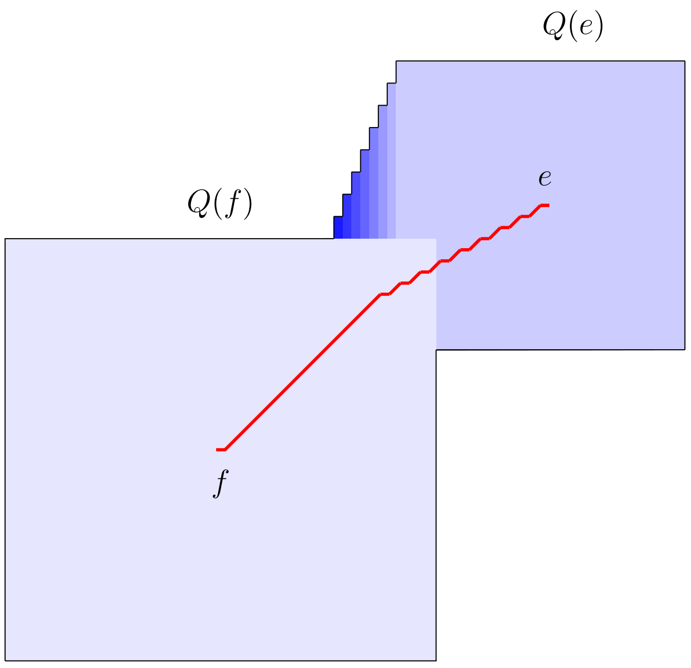


Figure 4.13: Polygon with Voronoi core (red) and width decomposition (the lighter the blue, the greater the width class). To obtain C_e we must subtract $Q(f)$ from $Q(e)$, and e and f are far apart in the Voronoi core.

for some $e \in E_{orth}$.

Proof. The first part follows directly from Lemma 4.21. For the second part, note that by our general position assumption the widths of all edge rectangles are pairwise distinct, so for each width class C_s we have exactly one edge e with $\mathrm{w}(e) = s$. Subtracting all edge rectangles with greater width from $Q(e)$ results in $C_s = C_e$. $\square$

The lemma implies that, given the Voronoi core for a polygon, its width classes can be computed by only building differences of edge rectangles. However, using the formula for C_e from Lemma 4.22 may result in quadratic total runtime, so it is an interesting question whether subtracting a *constant* number of edge rectangles is sufficient to obtain a single width class. The main difficulty here is that edges whose edge rectangles are needed for the difference may be arbitrary far away from e with respect to the Voronoi core.

Lemma 4.23. *In general, there is no $k \in o(n)$ such that for all $e \in E_{orth}$*

$$C_e = Q(e) \setminus \bigcup_{\substack{f \in E_k(e) \\ \mathrm{w}(f) > \mathrm{w}(e)}} Q(f)$$

holds, where $E_k(e)$ is the set of edges in E_{orth} having distance at most k to e in the graph G_{VC}.

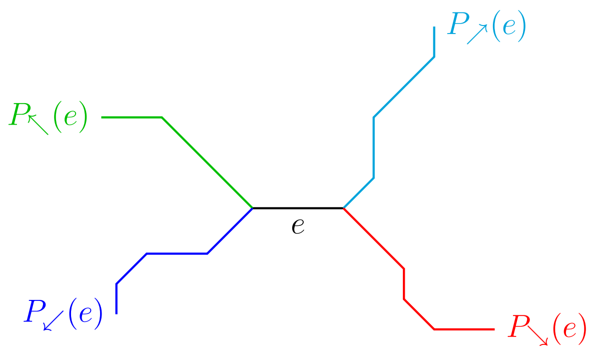


Figure 4.14: Example for an edge $e \in E_{\text{orth}}$ and its diagonal paths.

Proof. The instance in Figure 4.13 can be clearly extended to an instance with n arbitrary large, showing that for each $k \in o(n)$ we have $C_e \subsetneq Q(e) \setminus \bigcup_{\substack{e \in E_k(e) \\ \mathrm{w}(e) > w}} Q(e)$ as claimed. $\square$

We now show that each width class can be obtained by subtracting at most four edge rectangles and how to find these rectangles efficiently.

For any $e \in E_{\text{orth}}$ we define $P_{\nearrow}(e)$ to be the undirected path in G_{VC} that contains all edges of the sequence $(n_{\nearrow}(e), n_{\nearrow}(n_{\nearrow}(e)), (n_{\nearrow}(n_{\nearrow}(n_{\nearrow}(e))), \ldots)$, all connecting diagonal edges between them, and the diagonal edge connecting e and $n_{\nearrow}(e)$. We define $P_{\nwarrow}(e)$, $P_{\searrow}(e)$ and $P_{\swarrow}(e)$ analogously, and we call them *the diagonal paths of* e in the following. See Figure 4.14 for an example.

We further define $e_{\nearrow}$ to be the first edge f on $P_{\nearrow}(e)$ with $\mathrm{w}(f) > \mathrm{w}(e)$, or $e_{\nearrow} := \emptyset$ if no such edge exists. Let $e_{\nwarrow}$, $e_{\searrow}$ and $e_{\swarrow}$ be defined analogously. We call them the *the diagonal edge pointers of* e.

We are now ready to prove the following description of width classes which involves only a constant number of edge rectangles.

Theorem 4.24. *For each $e \in E_{orth}$ we have*

$$C_e = Q(e) \setminus (Q(e_{\nearrow}) \cup Q(e_{\searrow}) \cup Q(e_{\nwarrow}) \cup Q(e_{\swarrow})).$$

Proof. Let $f \in E_{\text{orth}}$ be arbitrary with $\mathrm{w}(f) > \mathrm{w}(e)$ and $Q(e) \cap Q(f) \neq \emptyset$ maximal, i.e. there is no $g \in E_{\text{orth}}$ with $\mathrm{w}(g) > \mathrm{w}(e)$ and $Q(e) \cap Q(g) \supsetneq Q(e) \cap Q(f)$. If there is no such f, then we clearly have $C_e = Q(e)$, finishing the proof.

We show that $f \in \{e_{\nearrow}, e_{\searrow}, e_{\nwarrow}, e_{\swarrow}\}$, proving the theorem by using Lemma 4.22. We first claim that $Q(f)$ contains at least one vertex of $Q(e)$. Suppose not, then by $\mathrm{w}(f) > \mathrm{w}(e)$ and $Q(e) \cap Q(f) \neq \emptyset$ the interior of $Q(e) \cup Q(f)$ contains a line segment $s \in \partial Q(e)$ of length greater than $\mathrm{w}(e)$ and parallel to e. By Lemma 4.16 applied to e, ∂P intersects s which contradicts $s \in (Q(e) \cup Q(f))^{\circ}$ and proves the

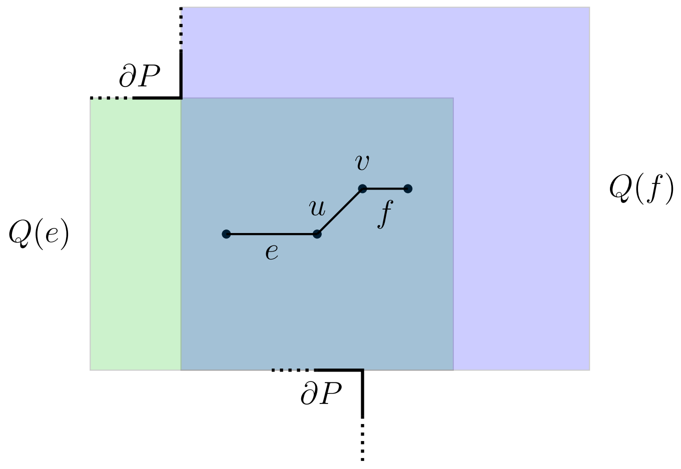


Figure 4.15: If ∂P does not intersect the right border of $Q(e)$, then the only edge f with $\mathrm{w}(f) > \mathrm{w}(e)$ and $Q(e) \cap Q(f) \neq \emptyset$ maximal is either the upper right (as shown) or lower right neighbor of e.

claim. We assume that $Q(f)$ contains the upper right corner of $Q(e)$, other cases are symmetric.

Let v be the left or lower vertex of f and u the right or upper vertex of e. We will slide a point q starting from v along the Voronoi core until reaching u by using only edges in lower or left direction, implying that $f \in E(P_{\nearrow}(e))$.

We first consider the case that ∂P does not intersect the right border of $Q(u)$. Then by Lemma 4.16 applied to e and u, e must be horizontal and for either $\bar{f} := n_{\nearrow}(e)$ or $\bar{f} := n_{\searrow}(e)$ we must have $\mathrm{w}(\bar{f}) > \mathrm{w}(e)$ (see Figure 4.15). We also have $Q(e) \cap Q(f) \subseteq Q(e) \cap Q(\bar{f})$, otherwise the interior of $Q(f)$ would intersect ∂P, a contradiction. By the maximality assumption for f we must have $Q(e) \cap Q(f) = Q(e) \cap Q(\bar{f})$ which implies $f = \bar{f} \in \{e_{\nearrow}, e_{\searrow}\}$ as claimed, by our general position assumption. Similarly, if ∂P does not intersect the upper border of $Q(u)$, then e must be vertical and $f \in \{e_{\nearrow}, e_{\nwarrow}\}$ as claimed. So in the following we may assume that ∂P intersects both right and upper border of $Q(u)$. This implies that for each p on G_{VC} with $Q(p)$ containing the upper right corner of $Q(u)$ the interior of $Q(p)$ cannot intersect one of the blue lines in Figure 4.16, thus we have $Q(e) \cap Q(p) = Q(u) \cap Q(p)$.

By the maximality assumption for f, the interiors of the left and lower border parts of $Q(v)$ not contained in $Q(u)$ (the red lines in Figure 4.16) intersect a vertical and horizontal segment of ∂P, respectively, and the bisector of these two segments induces a diagonal edge in G_{VC} from v to the lower left. We move q to the other vertex w of this edge. Note that $Q(u) \cap Q(q)$ stays equal for this movement. See Figure 4.17 for an example.

By Lemma 4.18, w is incident to an edge $f' \in E_{\mathrm{orth}}$, and by Corollary 4.20 f' proceeds in lower or left direction. Suppose $f' \neq e$ and f' is horizontal as in Figure 4.17 (the vertical case is symmetric), then we can move q along f' until

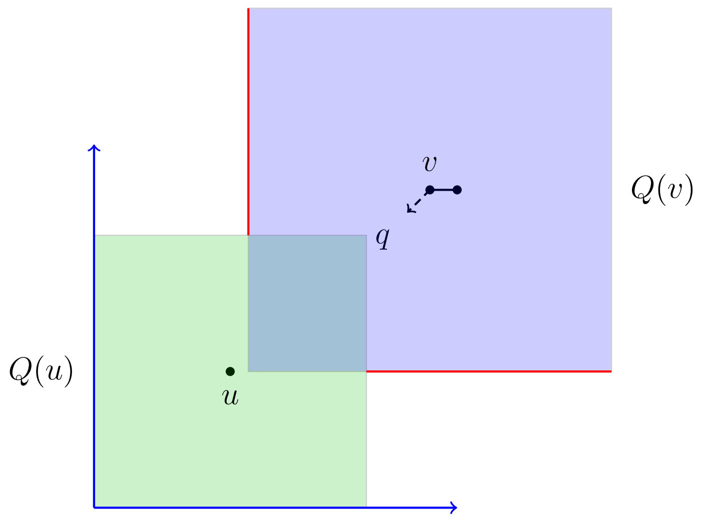


Figure 4.16

either the left border of $Q(q)$ intersects ∂P (which happens at the latest at the blue line) or the lower right corner of $Q(q)$ meets the polygon vertex denoted as p in Figure 4.17. In both cases, we arrive at a Voronoi vertex v' incident to a diagonal edge in lower left direction. Note that $Q(e) \cap Q(q) = Q(u) \cap Q(q)$ increases for this movement, thus we must have $\mathrm{w}(f') \leq \mathrm{w}(e)$ by our maximality assumption for f.

Now we can iterate this procedure to follow edges in lower or left direction. Note that at any time $Q(q)$ contains the upper right corner of $Q(u)$ and the interior of $Q(q)$ cannot intersect the blue lines in Figures 4.16 to 4.18, thus q is in upper right direction of u until, after a finite number of steps, we must have $q = u$. Also note that $Q(e) \cap Q(q) = Q(u) \cap Q(q)$ never shrinks, so for all traversed horizontal or vertical edges $g \neq f$ we have $\mathrm{w}(g) \leq \mathrm{w}(e)$, implying that $f = e_{\nearrow}$ as claimed.

□

By Theorem 4.24, the computation of all width classes can be reduced to determining diagonal edge pointers for each edge in E_{orth} and subtracting four rectangles from one other rectangle, where the latter part can be trivially done in constant time.

The determination of all edge pointers on a maximal diagonal path (that means, a diagonal path of some edge which is not a proper subset of a diagonal path of some other edge) in one direction can be abstracted to the following problem.

All Next Greater Numbers Problem	
Instance:	A sequence of real numbers $x_1, x_2, \dots, x_n$.
Task:	For each index $i \in \{1, 2, \dots, n\}$ find the smallest index $j > i$ with $x_j > x_i$, if existing.

Here the indices correspond to the edges on the diagonal path (ordered in direction

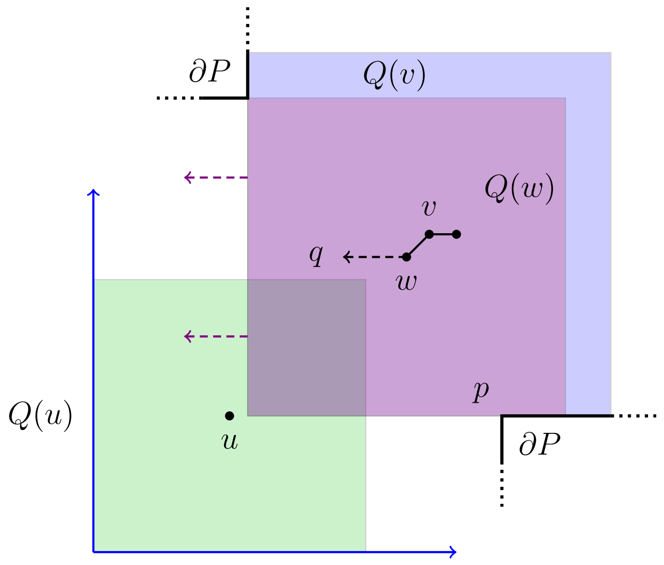


Figure 4.17

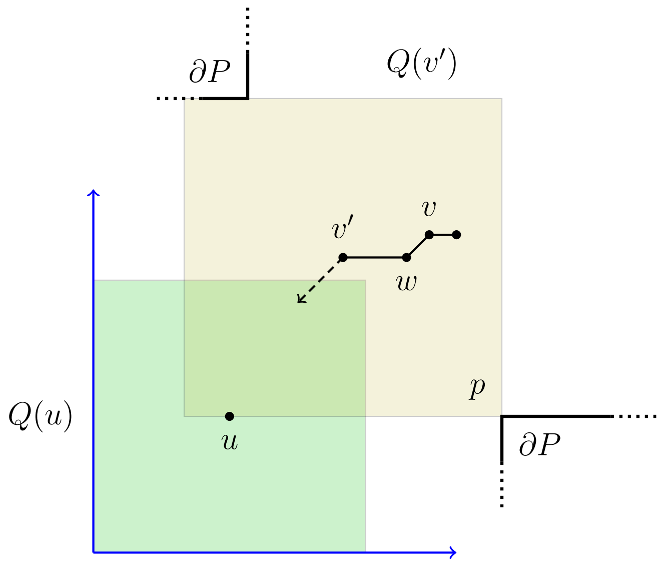


Figure 4.18

of the pointers to be determined) and the real numbers correspond to the widths of the edge rectangles. This problem can be solved in $O(n)$ time with the All Next Greater Numbers Algorithm (see Algorithm 2).

Theorem 4.25. *The* All Next Greater Numbers Algorithm *works correctly and runs in $O(n)$ time.*

Proof. To prove the correctness we show that the following conditions hold after executing line 8 of the algorithm:

Algorithm 2: All Next Greater Numbers Algorithm

Input : A sequence of real numbers $x_1, x_2, \dots, x_n$.

Output: A function next : $\{1, 2, \dots, n\} \to \{1, 2, \dots, n\}$ such that next$[i]$ is the smallest index greater than i with $x_{\text{next}[i]} > x_i$, if existing, and next$[i] = -1$ else.

```
1 prev[$i$] ← $-1$, next[$i$] ← $-1$   $\forall i \in \{1, 2, \dots, n\}$
2 for $i \leftarrow 2$ to $n$ do
3     $j \leftarrow i - 1$
4     while $j \geq 1$ and $x_j < x_i$ do
5         next[$j$] ← $i$
6         $j$ ← prev[$j$]
7     if $j \geq 1$ and $x_j \geq x_i$ then
8         prev[$i$] ← $j$
9 return next
```

(i) $\forall j \leq i$: prev$[j]$ is the largest index $m < j$ with $x_m \geq x_j$ (or -1 if no such index exists)

(ii) $\forall j \leq i$: next$[j]$ is the smallest index m with $j < m \leq i$ and $x_m > x_j$ (or -1 if no such index exists)

For $i = 2$ there are two possible cases: If $x_1 < x_2$ we enter the while-loop and set next$[1] \leftarrow 2$ in line 5, otherwise we set prev$[2] \leftarrow 1$ in line 8, thus for the first iteration ($i = 2$) both conditions are clearly satisfied after line 8.

Now suppose $i > 2$ and the conditions hold for $i-1$. We show that they also hold for i. If $x_{i-1} \geq x_i$, then in iteration i we only have to set prev$[i] \leftarrow i - 1$ which is done correctly in line 8 of iteration i.

Otherwise, if $x_{i-1} < x_i$, we have to set next$[j] \leftarrow i$ for each index j with $x_j \geq x'_j \ \forall j < j' < i$ and $x_j < x_i$. Because condition (i) holds for $i-1$, we traverse exactly those indices in the while-loop and set next$[j] \leftarrow i$ correctly. After finishing the while-loop, j is either -1 or the largest index smaller than i such that $x_j \geq x_i$, in which case we correctly set prev$[i] \leftarrow j$ in line 8. In summary, both conditions are satisfied after line 8 of iteration i. Condition (ii) for $i = n$ proves the correctness of the algorithm.

Next we analyze the runtime. When arriving at the body of the while-loop for some j we have $x_j < x_i$, thus in any later iteration $i' > i$ we have prev$[i'] \neq j$ by condition (i). Therefore, we never reach the body of the loop for the same j again, bounding the total number of iterations of the while-loop by n. So the total runtime of the algorithm is also bounded by $O(n)$. □

Now we have all ingredients to solve the Polygon 2D Width Decomposition Problem efficiently, see Algorithm 3.

Algorithm 3: Width Decomposition Algorithm

```
Input  : A rectilinear polygon $P$ with $n$ vertices.
Output: A width decomposition $W$ of $P$, containing $O(n)$ rectangles.
1  $W \leftarrow \emptyset$
2  Compute Voronoi core $\mathcal{V}_\infty(P)$
3  foreach $e \in E_{orth}$ do
4      if $e_\swarrow$ or $e_\nearrow$ not set then
5          Set $e'_\swarrow$ and $e'_\nearrow$ for all edges $e' \in E_{\text{orth}} \cap (E(P_\swarrow(e)) \cup \{e\} \cup E(P_\nearrow(e)))$
6      if $e_\nwarrow$ or $e_\searrow$ not set then
7          Set $e'_\nwarrow$ or $e'_\searrow$ for all edges $e' \in E_{\text{orth}} \cap (E(P_\nwarrow(e)) \cup \{e\} \cup E(P_\searrow(e)))$
8  foreach $e \in E_{orth}$ do
9      Add constant number of interior-disjoint rectangles covering the closure of
       $Q(e) \setminus (Q(e_\nearrow) \cup Q(e_\searrow) \cup Q(e_\nwarrow) \cup Q(e_\swarrow))$ to $W$
10 return $W$
```

We first compute the Voronoi core (line 2) which takes $O(n \log n)$ time for polygons with holes (Papadopoulou and Lee [2001]) and $O(n)$ time for simple polygons (Chin et al. [1999]). For polygons with holes, this is the only step requiring super-linear time. We assume the Voronoi core given in an appropriate data structure for planar straight-line graphs such as *doubly connected edge list* (Muller and Preparata [1978]) or *quad edge data structure* (Guibas and Stolfi [1985]). In fact, the only thing we need is that for each edge we can access its incident edges in constant time.

Subsequently, for each edge in E_{orth} we correctly set all diagonal edge pointers on maximal diagonal paths containing the edge (see lines 4 to 7). Note that we set the pointers only once for each maximal diagonal path, therefore this step takes linear time in total by using the All Next Greater Numbers Algorithm (see Theorem 4.25).

Finally, we traverse all edges in E_{orth} again and build the width decomposition by using Theorem 4.24 (line 9). We summarize the results in the following theorem.

Theorem 4.26. *Given a rectilinear polygon P, a width decomposition of P containing $O(n)$ rectangles can be computed in $O(n \log n)$ time. If P is simple, then the runtime reduces to $O(n)$.*

Corollary 4.27. *Given a rectilinear polygon P, we can build a data structure in $O(n \log n)$ time using $O(n)$ space which is able to report a width representative for any given point $p \in P$ in $O(\log n)$ time. If P is simple, then the preprocessing time reduces to $O(n)$.*

Proof. We first compute a width decomposition using Theorem 4.26 resulting in $O(n)$ rectangles. For each such rectangle we store its corresponding horizontal or vertical Voronoi edge. Then we build a point location data structure in time $O(n)$

(see e.g. Kirkpatrick [1983]) which reports for any $p \in P$ its at most four covering rectangles in time $O(\log n)$. For each such rectangle, we easily find a largest empty square covering p and centered at the Voronoi edge stored for the rectangle in constant time, and report a largest such square. □

See Figure 4.19 for an example of a width decomposition. For practical implementations the complicated linear time algorithm for computing the Voronoi core of a simple polygon may be replaced by a much simpler randomized algorithm running in $O(n \log^* n)$ expected time (see Chin et al. [1999]).

From a theoretical point of view it is interesting to consider generalizations of the Width Query Problem in terms of the involved geometric objects, see the following very general (and rough) problem description.

Generalized Width Query Problem

Instance: A set of two-dimensional geometric objects S, a two-dimensional geometric object C.

Task: A data structure which can report for any point $p \in \mathbb{R}^2$ (the width of) a largest scaled copy of C containing p that is interior-disjoint to S.

Note that much of the related work mentioned in the beginning of this section fits into this problem definition. In the following we assume that C is compact, convex, symmetric with respect to the origin, and that $(0,0) \in C^\circ$. We define d_C as the *symmetric convex distance function* induced by C which is specified as follows. To measure the distance $d_C(p,q)$ the set C is translated by the vector p (the red vector in Figure 4.20), resulting in a set C'. The ray from p through q (the blue ray in Figure 4.20) intersects the boundary of C' at a unique point q'. We set $d_C(p,q) := \frac{d_2(p,q)}{d_2(p,q')}$. It is easy to verify that $q \mapsto d_C((0,0),q)$ is a norm in the plane. See [Aurenhammer et al., 2013, Chapter 7] for more details. We also assume that the objects in S are simply-connected compact sets. The width is no longer measured by the edge length of a largest empty square but by the diameter of a largest scaled copy of C that is interior-disjoint to S, so we set

$$\mathrm{w}_C(p,S) := 2 \cdot \max\{w \mid \exists q \in \mathbb{R}^2 : d_C(q,p) \le w \le d_C(q,s)\ \forall s \in S\}.$$

Each q that maximizes w in this formula is the center of a width representative for p.

We now sketch a very general approach to solve the Generalized Width Query Problem for a large class of geometric objects. For this, we first compute the Voronoi diagram of S with respect to d_C. For details on Voronoi diagrams for symmetric convex distance functions see [Aurenhammer et al., 2013, Chapter 12]. For each point p on the Voronoi diagram we define $Q(p)$ as the unique maximal scaled copy of C centered at p and interior-disjoint to S. For a

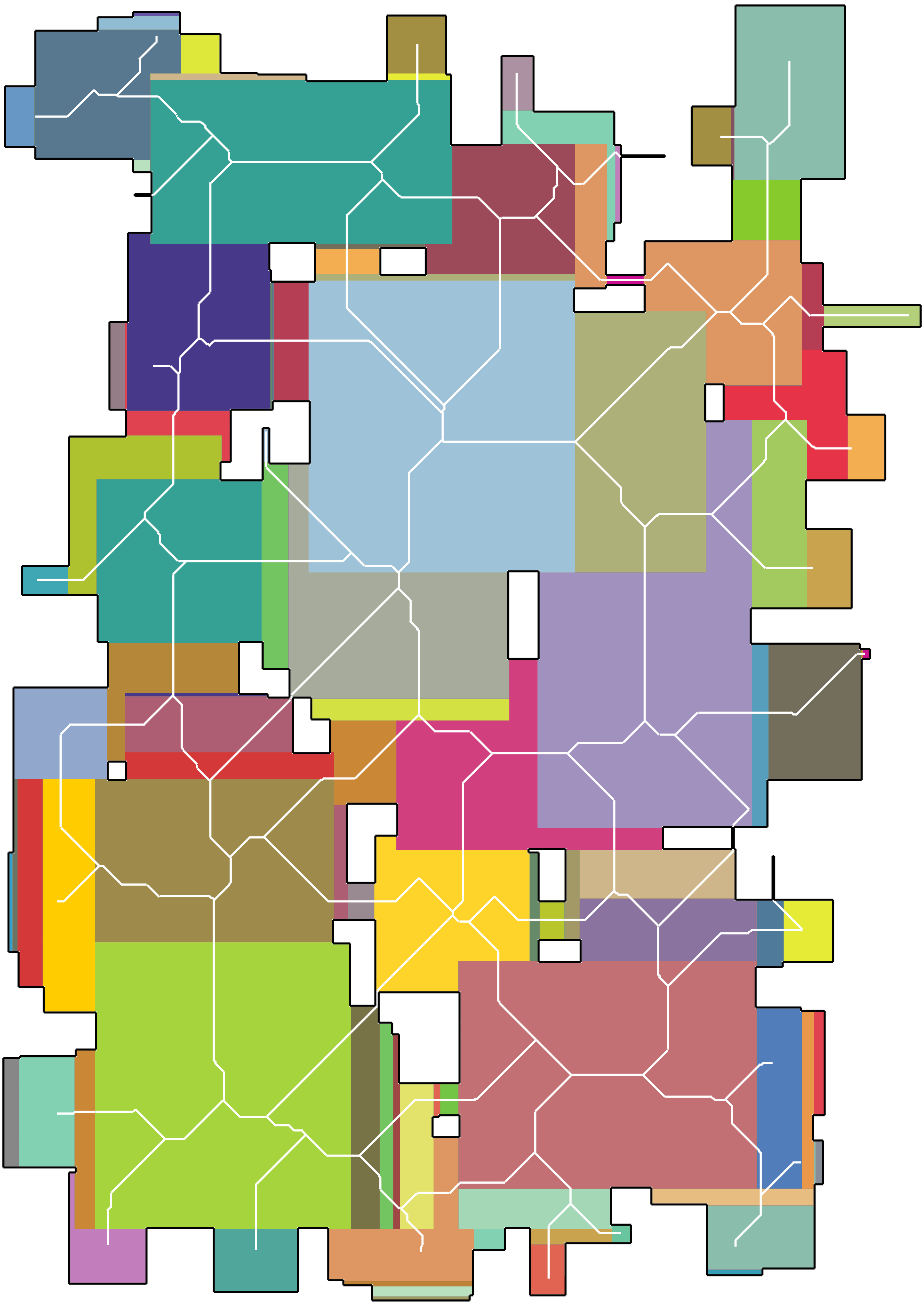


Figure 4.19: Width decomposition of a polygon with holes, Voronoi core in white and different width classes in random colors.

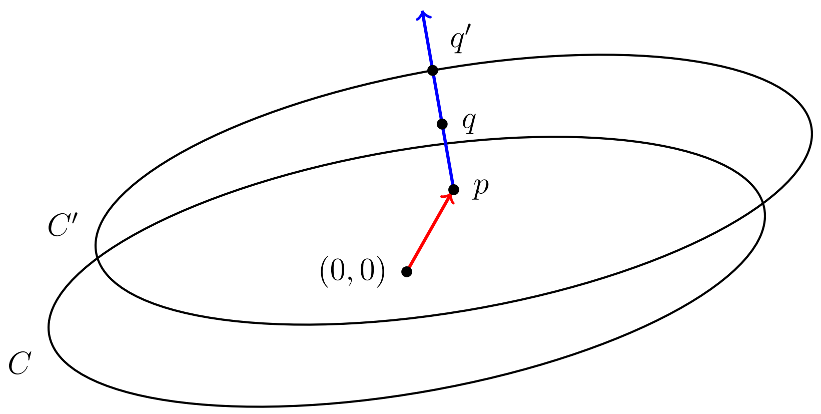


Figure 4.20

Voronoi edge e we define γ_e as the set containing all points belonging to e and we set $Q(e) := \bigcup_{p\in\gamma_e} Q(p)$. We further define the *width lifting function* of e as $l_e : Q(e) \to \mathbb{R}$, $l_e(p) := \max\{\mathrm{w}_C(q, S) \,|\, q \in \gamma_e, p \in Q(q)\}$ for $p \in Q(e)$. This function simply assigns each point $p \in Q(e)$ the maximum width of a scaled copy of C containing p and centered at γ_e.

We clearly have $\mathrm{w}_C(p, S) := \max_e l_e(p)$ since for each p there is a largest scaled copy of C containing p which is centered at the Voronoi diagram and thus centered at some Voronoi edge. Our goal is to partition the plane into maximal connected regions such that for all points p in the interior of one region $\max_e l_e(p)$ can be defined by a *single* width lifting function. Such a partition is known as *maximization diagram* or *upper envelope* of a set of functions, see Sharir and Agarwal [1995] where this topic is described in great detail. Given such a partition, the last step is now to build an appropriate point location data structure on top of the partition. In summary we have the following preprocessing steps:

a) Compute the Voronoi diagram of S with respect to d_C.

b) Compute the width lifting function l_e for each Voronoi edge e.

c) Compute the upper envelope $\mathcal{M}$ of all l_e.

d) Build a point location data structure on $\mathcal{M}$.

Based on this, a width query for a point p can be answered as follows:

(i) Find a region containing p by querying the point location structure.

(ii) Evaluate the width lifting function corresponding to the found region at p yielding $\mathrm{w}_C(p, S)$.

Of course, the crucial question is for which types of geometric objects a Voronoi diagram, an upper envelope, or a point location data structure can be computed in reasonable time, and what the complexity of the involved structures, particularly of the width lifting functions, is. We do not go into details how to obtain a width representative instead of the width here since this very much depends on the type of the geometric objects.

Let us now reconsider the Polygon 2D Width Query Problem. Here the width lifting function is constant for each $e \in E_{\text{orth}}$, namely $l_e(p) = \mathrm{w}(e)$ for $p \in Q(e)$, and Theorem 4.24 implies that each single region of the maximization diagram of all width lifting functions can be computed in constant time, given the diagonal edge pointers. Note that our solution for the Polygon 2D Width Query Problem follows exactly the proposed approach for the Generalized Width Query Problem above.

As another example, we now analyze this approach for the case where S is a set of line segments ($n := |S|$) and C is a symmetric convex polygon with constant complexity. The Voronoi diagram of S with respect to d_C can be computed in time $O(n \log n)$ and has complexity $O(n)$ in this case (Leven and Sharir [1987]). For simplicity we assume that no segment of C is parallel to any line segment in S, then the Voronoi diagram does not contain any two-dimensional parts (see [Aurenhammer et al., 2013, Chapter 7]). It is easy to see that the functions l_e are piecewise linear functions of constant complexity (see Figure 4.21 for an example), and by the results in [Sharir and Agarwal, 1995, Chapter 7] the maximization diagram of them has combinatorial complexity at most $O(n^2\alpha(n))$ and can be computed in time $O(n^{2+\epsilon})$ for any $\epsilon > 0$. Finally, building a point location data structure on top of the maximization diagram can be done in time linear in the combinatorial complexity of the maximization diagram, namely $O(n^2\alpha(n))$, allowing location queries in time $O(\log(n^2\alpha(n))) = O(\log n)$ (Kirkpatrick [1983]). The corresponding width lifting function can be evaluated in constant time. In summary, we can build a data structure in $O(n^{2+\epsilon})$ time (for any $\epsilon > 0$) using $O(n^2\alpha(n))$ space which is able to report $\mathrm{w}_C(p, S)$ for a given point $p \in \mathbb{R}^2$ in $O(\log n)$ time. The same result can be obtained for the case where S is a set of interior-disjoint simple polygons with n edges in total (see again Leven and Sharir [1987]).

We do believe that this result is not best possible. The interesting question is if the complexity of the maximization diagram and its computation can be better bounded by utilizing the structure of the Voronoi diagram, where the width lifting functions essentially arise from. This bounding was exactly what we needed to obtain the efficient solution for the Polygon 2D Width Decomposition Problem. We feel that obtaining similar results for other types of geometric objects (also for curved objects) serves as an interesting future research topic in computational geometry. For many types of geometric objects the computation and complexity of the maximization diagram seems to be the bottleneck, so again the key seems to be utilizing the structure of the Voronoi diagram.

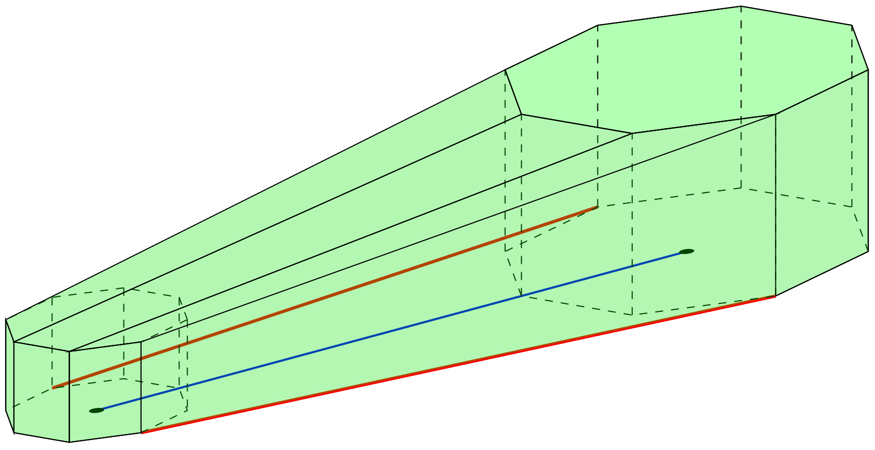


Figure 4.21: Part of a Voronoi edge (blue) of two line segments (red) with respect to a convex distance function defined by a regular octagon, and the width lifting function corresponding to this part of the edge (upper envelope of green polyhedron which is the convex hull of the two prisms).

We finally note the interesting fact that the Voronoi diagram itself can be viewn as the minimization diagram of certain distance functions, as first observed by Edelsbrunner and Seidel [1986]. So our above approach contains minimization or maximization diagrams twice, first in the Voronoi diagram and second to obtain the data structure for width queries, giving a hint to the power of this concept. These diagrams are also closely related to *Davenport-Schinzel sequences*, another well-known tool in computational geometry and combinatorics (see Sharir and Agarwal [1995]).

4.3 Decomposing the Union of Expanded Polygons

We now consider a geometric problem arising from *clock network design*, a step in VLSI design dedicated to the construction of a network transmitting the *clock signals* to storage elements (*latches*) on a chip. In Maßberg [2009] the tool *BonnClock* (part of the BonnTools software collection) is described which builds such a network (called *clocktree*) successively in the following way.

First, latches which are allowed to receive the clock signal roughly at the same time and which are not too far apart are combined to *clusters*. The involved latches of one cluster are called *sinks* and can receive the clock signal from a common *source*. The position of the source and the device realizing the source are determined later.

We are now interested in a set M of feasible positions where the source can be placed later on. Such positions must not be too far apart from any sink and must not intersect with any region on the chip that is already blocked by other devices.

The maximal distance from the source to any sink (which we denote by r in the following) is measured in d_1-distance because the routes connecting source and sinks later on may only run in horizontal or vertical direction.

Modeling the set of sinks as a point set $\{p_1, p_2, \ldots, p_n\}$ and the blockages as a rectangle set $\{R_1, R_2, \ldots, R_m\}$ in the plane we can describe the set of feasible positions as follows:

$$\begin{aligned} M &= \{\, x \in \mathbb{R}^2 \,|\, d_1(x, p_i) \le r \ \forall i \in \{1, 2, \ldots, n\} \text{ and } x \notin R_j \ \forall j \in \{1, 2, \ldots, m\} \,\} \\ &= \bigcap_{i \in \{1,2,\ldots,n\}} A_i \ \setminus \bigcup_{j \in \{1,2,\ldots,m\}} R_j \end{aligned}$$

where $A_i := \{x \in \mathbb{R}^2 \,|\, d_1(x, p_i) \le r\}$, so each A_i is a scaled and translated copy of the L_1 unit square.

So alltogether we have a set of clusters, and for each cluster C we have a set M_C where a source for C may be placed later on. Now all sinks and sources are added as nodes to the clocktree, together with one edge between each sink and its assigned source.

In the next step the previous sinks are not considered anymore, but the previous sources are considered as sinks now. These new sinks are again combined to clusters, and feasible positions for new sources are determined. This process is iterated until one arrives at a common source for all current sinks, the *root* of the clocktree. Note that in general each A_i is the union of convex polygons with horizontal, vertical and diagonal segments only (called *octagons* because of at most eight vertices).

Finally, for all nodes of the clocktree suitable positions and devices realizing these nodes are chosen, and the connections corresponding to the edges of the clocktree are routed. For these choices objectives such as power usage and area usage are taken into account.

See Maßberg [2009] for more details on clock network design which we only sketched briefly here. A key problem in the whole process can be stated as follows: Given an interior-disjoint octagon set $\mathcal{S}$, compute an interior-disjoint octagon set $\mathcal{S}'$ covering

$$\cup_{S \in \mathcal{S}} \{x \in \mathbb{R}^2 \,|\, d_1(x, S) \le d\} \ = \ \bigcup \mathcal{S} \oplus A$$

for $A := \{x \in \mathbb{R}^2 \,|\, \|x\|_1 \le d\}$ being a scaled copy of the L_1 unit square. In Gester [2009] an $O(n \log n)$ algorithm for this problem is given, using the L_∞ Voronoi diagram of octagons.

We study a more general variant of this problem and give a much simpler $O(n \log n)$ algorithm solving this problem.

Definition 4.28. *A set $A \subset \mathbb{R}^2$ is called* zonogon *if it is the finite Minkowski sum of line segments.*

A zonogon is either a line segment or a simple polygon. If it is a line segment, then we consider the two endpoints as *vertices* of the zonogon. See [Ziegler, 1995, Chapter 7] for more details on zonogons. A scaled copy of the L_1 unit square can be represented as the following Minkowski sum of two line segments and thus is a zonogon:

$$\{x \in \mathbb{R}^2 \mid \|x\|_1 \leq d\} \quad = \quad \overline{\left(-\frac{d}{2}, -\frac{d}{2}\right)\left(\frac{d}{2}, \frac{d}{2}\right)} \oplus \overline{\left(-\frac{d}{2}, \frac{d}{2}\right)\left(\frac{d}{2}, -\frac{d}{2}\right)}.$$

We consider the following problem.

Polygon Expansion Problem	
Instance:	An interior-disjoint polygon set $\mathcal{S}$, a zonogon A with a fixed number of vertices.
Task:	Compute an interior-disjoint polygon set $\mathcal{S}'$ covering $\bigcup \mathcal{S} \oplus A$.

We note that for a zonogon A with k vertices we can compute a set of line segments whose Minkowski sum equals A in time $O(k)$ (see Boltyanskii and Yaglom [1961]), so we may assume the line segments as given in the input w.l.o.g. As objective, we want the output set $\mathcal{S}'$ to contain preferably simple and few polygonal objects.

A natural approach to solve the Polygon Expansion Problem is to first expand all polygons of $\mathcal{S}$ by A, that means computing the polygon set $\mathcal{P} := \{S \oplus A \mid S \in \mathcal{S}\}$, and represent this set as interior-disjoint polygon set afterwards. However, one difficulty here is that $\mathcal{P}$ might contain $\Omega(|\mathcal{S}|^2)$ many pairs of intersecting polygons, even if the contour of $\mathcal{P}$ has low complexity, see Figure 4.22. If $\mathcal{P}$ only contains axis-aligned rectangles, then this difficulty can be resolved: We can compute the contour of n rectangles in optimal time $O(n \log n + m)$, where m is the number of contour edges (see Cheng and Janardan [1991]). However, for more general polygons in $\mathcal{P}$, even for the case that all polygons are octagons as in our original application, dealing with $\Omega(|\mathcal{S}|^2)$ many intersecting polygons is much harder and it is not clear how to compute the contour of $\mathcal{P}$ (or a decomposition into an interior-disjoint polygon set of size $O(m)$) efficiently.

Therefore, it might be a better idea to use an approach which avoids intersecting objects at all. In Gester [2009] the special case where $\mathcal{S}$ is an octagon set and A is a square is considered, and the L_∞ Voronoi diagram of $\mathcal{S}$ is used to avoid intersections. There it is proved that the interior-disjoint polygon set $\mathcal{P}' := \{(S \oplus A) \cap \mathrm{VR}(S) \mid S \in \mathcal{S}\}$ covers $\bigcup \mathcal{S} \oplus A$ and can be computed in $O(n \log n)$ time, where $\mathrm{VR}(S)$ denotes the Voronoi region of S.

In the following we describe a much simpler $O(n \log n)$ time algorithm yielding $O(n)$ interior-disjoint trapezoids as output (Algorithm 4). The key idea is that expanding $\mathcal{S}$ by $A = s_1 \oplus s_2 \oplus \ldots \oplus s_k$ can be reduced to a sequence of one-dimensional expansion steps, using the associativity of the Minkowski sum. More detailed, the algorithm successively computes the sets $\bigcup \mathcal{S} \oplus s_1$, $\bigcup \mathcal{S} \oplus s_1 \oplus s_2$, $\ldots$,$\bigcup \mathcal{S} \oplus s_1 \oplus$

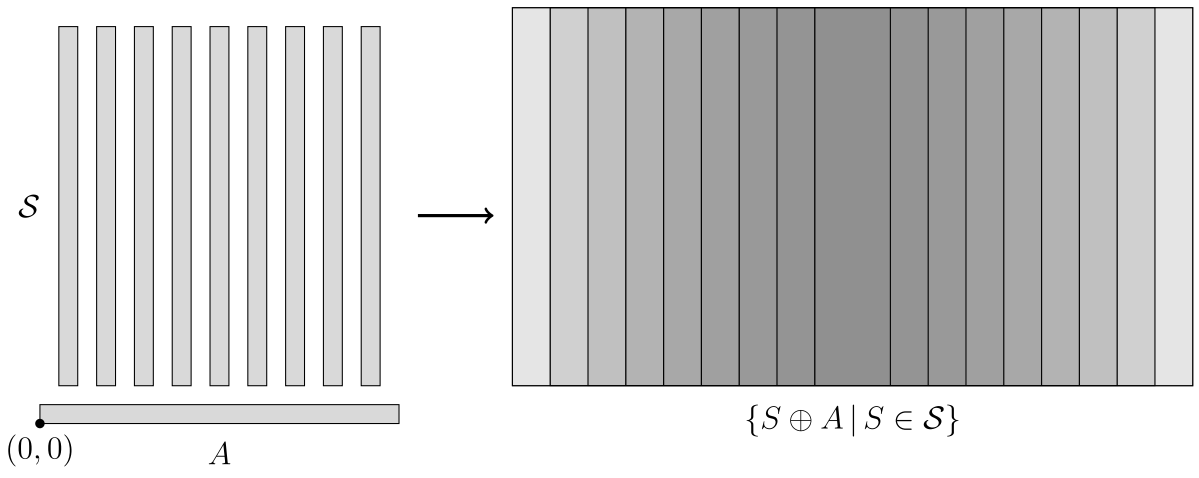


Figure 4.22: Instance with $\Omega(|\mathcal{S}^2|)$ many pairs of intersecting expanded polygons. The darker the gray, the more expanded polygons overlap.

$s_2 \oplus \ldots \oplus s_k$. In any step we are given an interior-disjoint polygon set $\mathcal{S}'$ and a line segment s', and compute $\mathcal{S}' \oplus s'$ which can be done fast by using a *trapezoidal map*, a well-known data structure in computational geometry which has applications in the context of point location problems (Berg et al. [2008]) and polygon triangulation (Seidel [1991], Fournier and Montuno [1984]). We describe this data structure in the following.

Let L be a set of non-crossing line segments and let P be the set of endpoints of segments in L. Let B be the bounding box of P and L' be the union of L and of the four border segments of B. For each point $p \in P$, let p_{up} be the maximal vertical segment whose inner part does not intersect any segment of L', and whose lower endpoint is p. If no such segment exists, we set $p_{\text{up}} = \emptyset$. Similarly, let p_{down} be the maximal vertical segment whose inner part does not intersect any segment of L', and whose upper endpoint is p. If no such segment exists, we set $p_{\text{down}} = \emptyset$.

It is easy to see that the set of line segments $L' \cup \bigcup_{p \in P}(p_{\text{down}} \cup p_{\text{up}})$ subdivides B into trapezoidal regions whose left and right borders are vertical segments. In other words, the connected components of $B \setminus (L' \cup \bigcup_{p \in P}(p_{\text{down}} \cup p_{\text{up}})$ are open trapezoids, and we define $\mathcal{T}(L)$ to be the set containing all these open trapezoids.

The set $\mathcal{T}(L)$ corresponds to a trapezoidal map as defined in Berg et al. [2008] and can be computed in time $O(n \log n)$ with a simple plane sweep algorithm (cf. Chan [1994]). There also exist simple algorithms using random sampling, running in $O(n \log n)$ expected time and using $O(n)$ expected storage, well suited for building a point location data structure (see Berg et al. [2008]). The following result can also be found in Berg et al. [2008].

Lemma 4.29. $|\mathcal{T}(L)| \leq 3 \cdot |L| + 1$.

For an interior-disjoint polygon set $\mathcal{S}$ we define $\mathcal{T}(\mathcal{S}) = \mathcal{T}(L)$, where L is the set

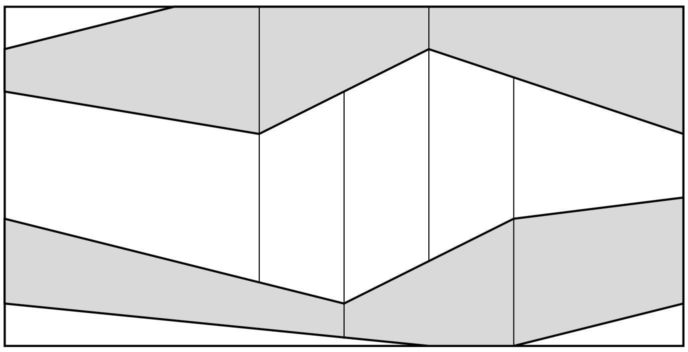


Figure 4.23: A trapezoidal map of two polygons (gray), given by black lines.

of line segments of all polygons in $\mathcal{S}$. In this case, a trapezoid in $\mathcal{T}(\mathcal{S})$ is either fully covered by a polygon in $\mathcal{S}$ or it is disjoint to all polygons in $\mathcal{S}$ which we will use for our algorithm. See Figure 4.23 for an example of a trapezoidal map. Given $\mathcal{T}(\mathcal{S})$ for a polygon set $\mathcal{S}$ and given a line segment $s = \overline{(0,0)(0,d)}$ for some $d \in \mathbb{R}$, computing the Minkowski sum $\bigcup \mathcal{S} \oplus s$ is easy as we will see in the following.

We are now ready to describe and analyze Algorithm 4 which solves the Polygon Expansion Problem. The input is an interior-disjoint polygon set $\mathcal{S}$ and a set $A = s_1 \oplus s_2 \oplus \ldots \oplus s_k$ where all s_i are line segments. For the runtime analysis, note that we assume k to be fixed. The output is an interior-disjoint trapezoid set $\mathcal{S}'$ covering $\bigcup \mathcal{S} \oplus A$. In the following let n be the number of vertices of all polygons in $\mathcal{S}$. Throughout the algorithm $\mathcal{S}'$ is an interior-disjoint polygon or trapezoid set. In each call of Procedure `expand_1d` $\mathcal{S}'$ is expanded in one direction given by the current s_i.

We first prove the correctness of the algorithm. It is easy to see that the output set $\mathcal{S}'$ consists of interior-disjoint trapezoids because all trapezoids inserted into $\mathcal{S}''$ in Procedure `expand_1d` are interior-disjoint by definition of $\mathcal{T}(\mathcal{S}')$ and by the choice of $\mathcal{T}'$ in line 14 (see Figure 4.24 for a visualization of line 14). It remains to show that these trapezoids cover exactly the set $\bigcup \mathcal{S} \oplus A$. We show by induction that in the i-th call of Procedure `expand_1d` the returned set $\mathcal{S}''$ covers exactly $\mathcal{S}_i := \bigcup \mathcal{S} \oplus s_1 \oplus s_2 \oplus \ldots \oplus s_i$, proving the above claim by associativity of the Minkowski sum. We further set $\mathcal{S}_0 := \bigcup \mathcal{S}$.

Suppose that in the i-th iteration $\mathcal{S}'$ covers S_{i-1} at the beginning of Procedure `expand_1d` (which is trivial for $i = 1$). W.l.o.g. we may assume $s_i = \overline{(0,0)(0,d)}$, otherwise the whole instance is rotated in line 6 and rotated back in line 16. Now observe that when $\mathcal{S}''$ is returned, it contains two types of trapezoids: first trapezoids covering all points in S_{i-1} (inserted in line 11), and second trapezoids covering all points in $(S_{i-1} \oplus s_i) \setminus S^{\circ}_{i-1}$ (inserted in line 15). Therefore, $\mathcal{S}''$ covers exactly $\mathcal{S}_i$ as claimed.

The runtime of Procedure `expand_1d` is clearly dominated by computing $\mathcal{T}(\mathcal{S}')$ which can be done in time $O(n \log n)$ as noted above. Since k is fixed, also the total

Algorithm 4: Polygon Expansion Algorithm

```
Input  : An interior-disjoint polygon set $\mathcal{S}$, a zonogon $A = s_1 \oplus s_2 \oplus \ldots \oplus s_k$.
Output: An interior-disjoint trapezoid set $\mathcal{S}'$ covering $\bigcup \mathcal{S} \oplus A$.

1  $\mathcal{S}' \leftarrow \mathcal{S}$
2  for $i \leftarrow 1$ to $k$ do
3      $\mathcal{S}' \leftarrow$ expand_1d($\mathcal{S}', s_i$)
4  return $\mathcal{S}'$

5  Procedure expand_1d($\mathcal{S}', s_i$)
6      Rotate and translate $\mathcal{S}'$ and $s_i$ such that $s_i = \overline{(0,0)(0,d)}$ for some $d \in \mathbb{R}$
7      Compute $\mathcal{T}(\mathcal{S}')$
8      $\mathcal{S}'' \leftarrow \emptyset$
9      foreach trapezoid $T$ in $\mathcal{T}(\mathcal{S}')$ do
10         if $T$ is covered by $\mathcal{S}'$ then
               // add covered trapezoids
11             $\mathcal{S}'' \leftarrow \mathcal{S}'' \cup \{\bar{T}\}$
12         else
               // add filled empty trapezoids
13             $s_{\text{low}} \leftarrow$ lower segment of $T$
14             $\mathcal{T}' \leftarrow$ set of at most two interior-disjoint trapezoids covering
               $(s_{\text{low}} \oplus s_i) \cap \bar{T}$
15             $\mathcal{S}'' \leftarrow \mathcal{S}'' \cup \mathcal{T}'$
16     Rotate and translate $\mathcal{S}''$ back
17     return $\mathcal{S}''$
```

runtime of the algorithm is $O(n \log n)$. By applying Lemma 4.29 once for each call of Procedure `expand_1d` the output set $\mathcal{S}'$ contains $O(n)$ trapezoids. We summarize our results in the following theorem.

Theorem 4.30. *Given an interior-disjoint polygon set $\mathcal{S}$ with n vertices in total and a zonogon A with a fixed number of vertices, a set of $O(n)$ interior-disjoint trapezoids $\mathcal{S}'$ covering $\bigcup \mathcal{S} \oplus A$ can be computed in time $O(n \log n)$.*

See Figure 4.25 for a visualization of Procedure `expand_1d` of Algorithm 4. Note that in line 7 we may use any algorithm for computing a trapezoidal map, for example random sampling algorithms which seem to be easiest to implement, but provide only a $O(n \log n)$ expected runtime bound. In this case there is no sweep line process involved in our algorithm. Also note that the algorithm can be easily adapted to output the boundary of $\bigcup \mathcal{S} \oplus A$ instead of a covering trapezoid set. For this we replace the foreach-loop by a sweep line procedure (in vertical direction) which does not insert trapezoids directly into $\mathcal{S}''$ as in lines 11 and 15, but maintains

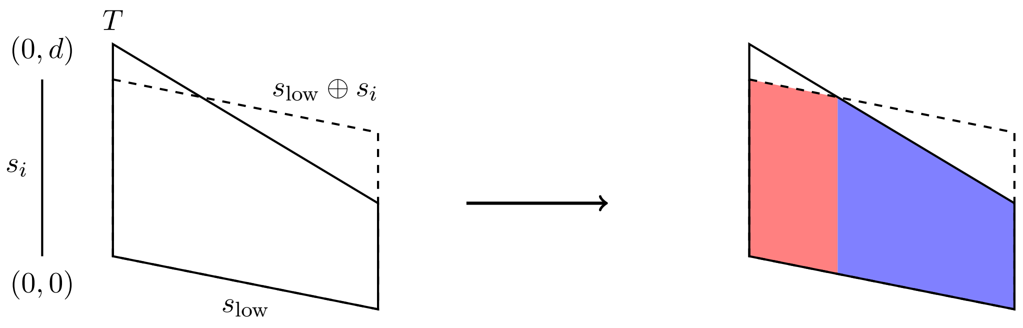


Figure 4.24: Visualization of line 14 of Algorithm 4: decomposition of $(s_{\text{low}} \oplus s_i) \cap T$ (left side) into two trapezoids (in red and blue, right side).

the union of neighboring trapezoids as unfinished polygons and inserts them later into $\mathcal{S}''$.

We implemented Algorithm 4 as part of the BonnTools where it is used for the application in clock network design described in the beginning of this section. See Figure 4.26 for an example instance solved by this algorithm.

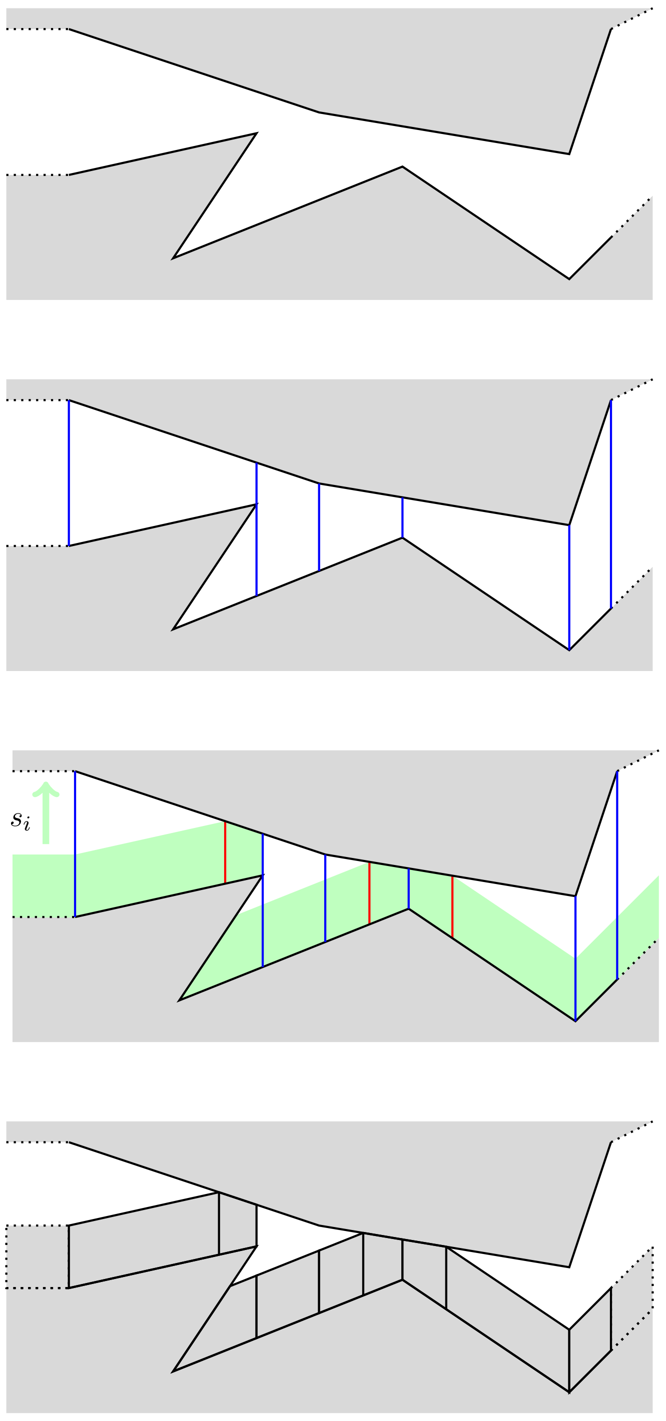
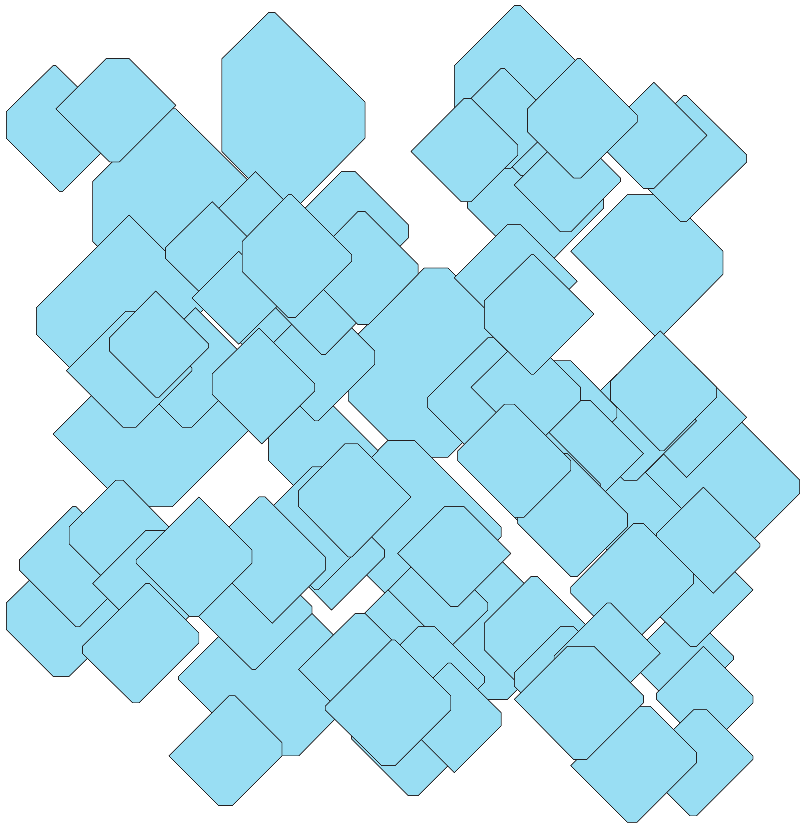
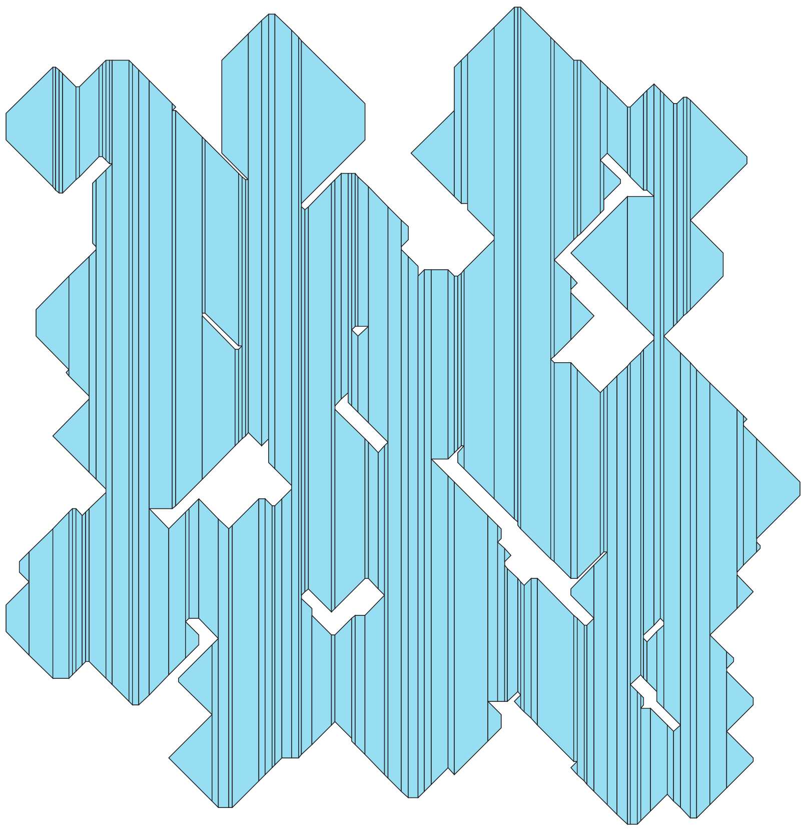


Figure 4.25: **Visualization of Procedure `expand_1d`. The first picture shows sections of two polygons, in the second picture blue lines giving a trapezoidal map of the empty space between the polygons are added (for simplicity, the polygons itself are not decomposed into trapezoids here). In the third picture the empty trapezoids are filled with respect to s_i. New left and right borders of resulting trapezoids are drawn as red lines. The last picture shows the resulting decomposition into trapezoids.**



(a) A set of octagons, resulting from interior-disjoint octagons expanded by a certain L_1 distance.



(b) A set of interior-disjoint trapezoids covering the same area.

Figure 4.26

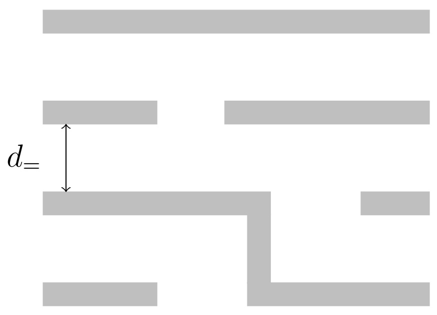
5 VLSI Routing for Multiple Patterning Technology

In this chapter we focus on new challenges in VLSI routing arising from *multiple patterning*. Multiple patterning is a technique for increasing feature density on a chip layer by assigning objects on this layer to different manufacturing steps. These steps are typically abstracted as *colors*. The main difficulty for a routing tool is that design rules now depend not only on the geometry of the involved objects but also on their colors modelling the different manufacturing steps. Basically the minimum allowed distances between same-colored shapes are considerably larger than distances between shapes colored differently. The routing tool has to choose and maintain these colors and produce a routing solution which is clean with respect to the color-dependent design rules.

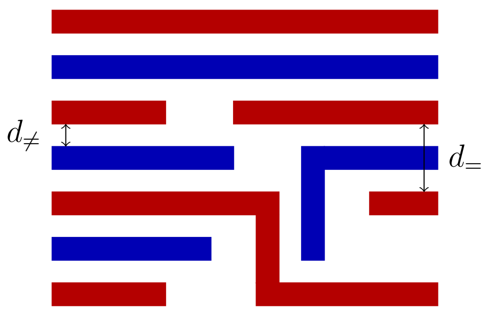
The most important manufacturing technologies for *double patterning*, where two different manufacturing steps for objects on one layer are used, are *LELE (litho-etch-litho-etch)* and *SADP (self-aligned double patterning)*. In LELE the objects on a layer are assigned to two different masks and the final layout on the chip is produced by two exposure steps using these masks. Here one needs a very high accuracy in mask positioning to create the appropriate spacings between objects printed by different masks. See Figure 5.1 for an illustration. Here the color dependency of the shapes is symmetric, that means permuting colors does not influence the feasibility of shapes, and it is possible to overlay shapes of different colors. Such an overlay is called *stitch* and has to obey complex design rules to ensure electrical connectivity between the involved shapes manufactured by different masks. Also possible manufacturing variations are particularly critical for stitches and impinge on the *yield rate* (the amount of manufactured chips that work as intended). Therefore it is desirable to minimize the use of stitches.

In SADP technology chemical spacers, called *sidewalls*, are created around objects printed by the first mask (called *mandrel mask*). The second type of objects is built by gaps between these sidewalls. To allow that not all such gaps result in metalized objects, a *block mask* is used to prohibit certain areas from being metalized. For SADP color dependency is not symmetric and stitches are technically not possible.

For more details on the lithographic process see Finders et al. [February 2008] for LELE and Kim et al. [2006], Maenhoudt et al. [2005] for SADP. Both LELE and SADP technologies can be extended to more than two colors (called triple patterning, quadruple patterning and so on) to further increase feature density or



(a) Layout without multiple patterning, all shapes are manufactured by the same mask. $d_=$ is the mindist between any two shapes.



(b) Layout with multiple patterning (LELE technology), one mask for blue and one mask for red shapes. $d_=$ is the mindist between any shapes on the same mask, $d_{\neq}$ is the mindist between any shapes on different masks.

Figure 5.1

to reduce coloring conflicts. In the following, the terms LELE and SADP include any of these extensions as well.

For economical reasons only few layers on a chip are manufactured in multiple patterning technology, typically *transistor layers* and lower wiring or via layers. Here it is also possible that a via layer is produced with single patterning technology, while both neighboring wiring layers are produced with multiple patterning technology. This poses tough challenges to the routing tool since minimum distances between vias are much larger than minimum distances between wires on the neighboring layers. We get back to this problem in Section 5.4.2 where we show how to avoid via (same-net-)mindist violations correct-by-construction.

In Section 5.2 we investigate the fundamental problem how to color a given layout with respect to color-dependent design rules. This problem occurs on multiple patterning transistor and routing layers, depending on the multiple patterning methodology used. In Section 5.3 we describe how layouts can be built such that they are guaranteed to be colorable afterwards.

In the main part of this chapter, Section 5.4, we explain how multiple patterning is managed in BonnRoute. The most important tools used for this are an *automatic coloring*, *track patterns*, and a *multi-label path search*. In Section 5.4.1 we describe how BonnRoute uses automatic coloring and track patterns to maintain an easy routing flow on the one hand and achieve high packing density of wires on the other hand. Section 5.4.2 describes how detailed routing for long connections is done in presence of multiple patterning. Our approach does not require stitches at all and hence avoids the yield and routability penalties associated with it. It is suited for both LELE and SADP technologies and is used for routing real-world multiple patterning instances (see Section 5.4.4). We keep color dependencies off the standard shortest path algorithm by using the automatic coloring described in Section 5.4.1. A core tool of our approach is a very general and powerful multi-label shortest path algorithm which is used to compute design rule clean paths and non-trivially colored paths in situations where standard shortest path algorithms do not find good solutions. In Section 5.4.4 we present results demonstrating that BonnRoute produces high-quality routings in short runtime on real-world multiple patterning designs. For these results we combined BonnRoute with an external procedure for cleaning up remaining design rule violations.

5.1 Multiple Patterning Setting

We now define our multiple patterning setting formally. For each layer $z \in Z_{\text{all}}$ we assume a set of multiple patterning colors $\{1, \dots, k_z\}$. On a single patterning layer we have $k_z = 1$. A *coloring for a shape set* $\mathcal{S}$ is a function $\alpha_{\mathcal{S}}$ mapping each shape $S \in \mathcal{S}$ to a color $c \in \{1, \dots, k_{z(S)}\}$. For a subset $\mathcal{T} \subseteq \mathcal{S}$ we denote $\alpha_{\mathcal{S}}$ restricted to the domain $\mathcal{T}$ as $\alpha_{\mathcal{T}}$.

For multiple patterning technology a checking oracle cannot decide legality of shapes just by their geometry, but it needs to take the colors of all involved shapes into account. Therefore, we assume a *colored checking oracle* ψ_c which is a function such that given a shape set $\mathcal{S}$ and a coloring $\alpha_{\mathcal{S}}$ for $\mathcal{S}$ we have

$$\psi_c(\mathcal{S}, \alpha_{\mathcal{S}}) \mapsto \begin{cases} 0, & \text{if the shapes in } \mathcal{S} \text{ colored by } \alpha_{\mathcal{S}} \text{ violate any design rule} \\ 1, & \text{else.} \end{cases}$$

We call a coloring $\alpha_{\mathcal{S}}$ with $\psi_c(\mathcal{S}, \alpha_{\mathcal{S}}) = 1$ a *feasible coloring* for $\mathcal{S}$. We further assume an *uncolored checking oracle* ψ_u which is a function such that given a shape set $\mathcal{S}$ we have

$$\psi_u(\mathcal{S}) \mapsto \begin{cases} 1, & \text{if for each } \mathcal{T} \in \binom{\mathcal{S}}{2} \text{ there is a coloring } \alpha'_{\mathcal{T}} \text{ with } \psi_c(\mathcal{T}, \alpha'_{\mathcal{T}}) = 1 \\ 0, & \text{else.} \end{cases}$$

An uncolored checking oracle always assumes the best case for the colors when checking two shapes. We may have $\psi_u(\mathcal{S}) = 1$ although there is no coloring $\alpha_{\mathcal{S}}$ such that $\psi_c(\mathcal{S}, \alpha_{\mathcal{S}}) = 1$. The advantage of an uncolored checking oracle is that it does not need a coloring of the shapes as input and thus allows a very simple multiple patterning methodology, taking coloring complexity completely off the routing algorithms and shifting it to a separate and independent coloring step, see Algorithm 5.

Algorithm 5: Uncolored Routing Methodology

```
1 Compute uncolored routing using the uncolored checking oracle $\psi_u$
2 Compute coloring of routing minimizing DRC-errors with respect to $\psi_c$
3 Resolve remaining DRC-errors by local post-processing and rip-up and reroute,
  using $\psi_c$ as checking oracle
```

Note that here the uncolored main routing step is not affected by multiple patterning at all. The coloring step raises the question how to (partially) color a given uncolored layout optimally which we discuss in Section 5.2. The main challenge of this methodology is to resolve the remaining errors in the third step. Here it is not clear how much of the routing has to be restructured and if routing convergence can be obtained at all. Also here we have to take color-dependent design rules into account, in contrast to the first step.

Another possible methodology is that routing algorithms know about colors *all the time* (and not only in the cleanup step as in Algorithm 5) and compute and maintain them for all created wires. However, if the main sequential routing step chooses colors for wires arbitrarily while satisfying the colored checking oracle, this may lead to bad space utilization caused by gaps which cannot be used by wires routed later on. See Section 5.4.1 and Figure 5.5a on page 73 for more details.

In Section 5.4 we describe a third methodology which is implemented and succesfully used in BonnRoute for routing real-world multiple patterning designs. Here wires are automatically colored according to a pattern, guiding the sequential routing step and producing dense wire packings. Only if a connection cannot be routed this way we allow deviations from the pattern by using a multi-label shortest path algorithm.

5.2 Coloring Given Layouts

In this section we consider the problem of coloring an existing uncolored chip layout from a theoretical perspective. In practice, design rules depending on the color operate only on shapes on the same layer. Therefore, we can solve the coloring problem for each layer seperately.

Layer Coloring Problem
Instance: A shape set $\mathcal{S}$ on layer $z \in Z_{\text{all}}$.
Task: Compute a coloring $\alpha_{\mathcal{S}}$ such that $\psi_c(\mathcal{S}, \alpha_{\mathcal{S}}) = 1$, or decide that there is no such coloring.

We assume from now on that the feasibility of a shape set can be determined by checking all two-element subsets of the set. That means, for a shape set $\mathcal{S}$ and a coloring $\alpha_{\mathcal{S}}$, we assume that $\psi_c(\mathcal{S}, \alpha_{\mathcal{S}}) = 1$ if and only if for all $\mathcal{T} \in \binom{\mathcal{S}}{2}$ we have $\psi_c(\mathcal{T}, \alpha_{\mathcal{T}}) = 1$. This assumption is realistic since we are only interested in color-dependent design rules here, and these are typically mindist rules involving only two shapes. Since minimum distances between same-colored shapes are larger than minimum distances between differently-colored shapes, we may also assume that if two non-intersecting shapes are legal when colored equally, then they are also legal when colored differently. Note that for intersecting shapes this is not true because then the different colors correspond to a stitch which has to obey complex design rules. We here restrict ourselves to the case where no stitches are allowed at all, that means intersecting shapes must be colored equally. Therefore, we assume that $\psi_c(\mathcal{S}, \alpha_\mathcal{S}) = 0$ if $\alpha_\mathcal{S}$ colors any two intersecting shapes of $\mathcal{S}$ with different colors. In Section 5.2.3 we discuss how stitches can be incorporated, if needed.

We now show how the Layer Coloring Problem can be transferred into well-known optimization problems on graphs for different design rule settings.

5.2.1 Color-Symmetric Design Rules

We first assume that we only have *color-symmetric design rules*, that means permuting the colors of shapes does not influence the feasibility of these shapes. Formally, for each shape set $\mathcal{S}$ on layer z, for each coloring $\alpha_\mathcal{S}$, and for each permutation $\pi : \{1, 2, \dots, k_z\} \to \{1, 2, \dots, k_z\}$ we have $\psi_c(\mathcal{S}, \alpha_\mathcal{S}) = \psi_c(\mathcal{S}, \pi \circ \alpha_\mathcal{S})$. This setting is realistic for LELE technology, but not for SADP technology.

In the following we use the *connected component partition* $\mathcal{C}_\mathcal{S}$ of $\mathcal{S}$, that is the set of maximal subsets $\mathcal{A} \subseteq \mathcal{S}$ with the property that $\bigcup \mathcal{A}$ is a connected set. Since we do not allow stitches, all shapes in one connected component must get the same color. We further assume $\psi_u(\mathcal{S}) = 1$ which is motivated by the application of layout coloring in Algorithm 5, leading to the following problem.

Symmetric Layer Coloring Problem	
Instance:	A shape set $\mathcal{S}$ on layer $z \in Z_{\text{all}}$ with $\psi_u(\mathcal{S}) = 1$, and color-symmetric design rules.
Task:	Compute a coloring $\alpha_\mathcal{S}$ such that $\psi_c(\mathcal{S}, \alpha_\mathcal{S}) = 1$, or decide that there is no such coloring.

We now translate this problem to a graph coloring problem. A *k-coloring of a graph* is a mapping of its vertices to numbers in $\{1, \dots, k\}$ such that vertices incident to the same edge are mapped to different numbers (see Korte and Vygen [2012]).

Definition 5.1. *The* color-symmetric conflict graph *for a shape set $\mathcal{S}$ is the undirected graph $G_\mathcal{S}$ with vertices*

$$V(G_\mathcal{S}) := \mathcal{C}_\mathcal{S}$$

and edges

$$E(G_\mathcal{S}) := \{\{C_1, C_2\} \in \binom{\mathcal{C}_\mathcal{S}}{2} \mid \exists S_1 \in C_1, S_2 \in C_2 : \psi_c(\{S_1, S_2\}, \alpha_{\{S_1,S_2\}}) = 0\},$$

where $\alpha_{\{S_1,S_2\}}$ is the coloring function which maps both S_1 and S_2 to color 1.

In the following we use $n := |V(G_\mathcal{S})|$ and $m := |E(G_\mathcal{S})|$. The graph $G_\mathcal{S}$ contains an edge between any two connected components of shapes which are not allowed to have the same color. Since design rules are color-symmetric and we have $\psi_u(\mathcal{S}) = 1$, each two components not connected by an edge do not produce any design rule

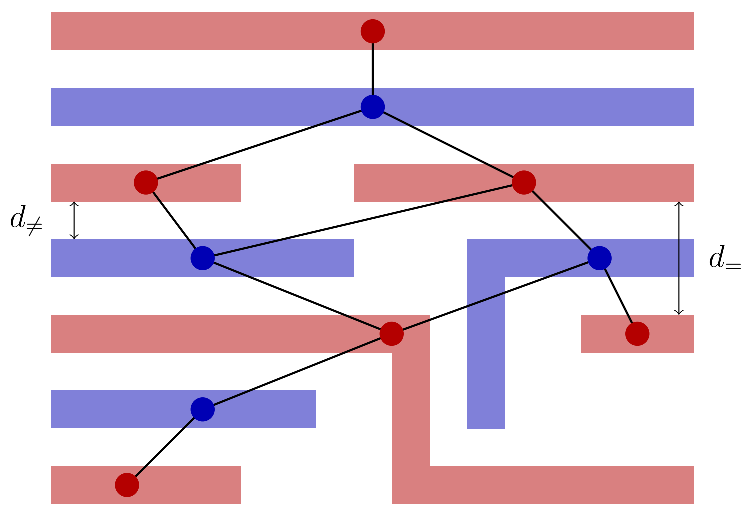


Figure 5.2: A 2-coloring of the graph $G_{\mathcal{S}}$ corresponding to the colored instance in Figure 5.1b.

violation when colored differently. Therefore, each coloring $\alpha_{\mathcal{S}}$ with $\psi_c(\mathcal{S}, \alpha_{\mathcal{S}}) = 1$ corresponds to a k_z-*coloring* in $G_{\mathcal{S}}$ and vice versa. It remains to solve the following problem.

GRAPH k-COLORING PROBLEM	
Instance:	An undirected graph G.
Task:	Compute a k-coloring of G, or decide that there is no such coloring.

See Figure 5.2 for an example. Before turning to this problem, we first discuss the runtime for converting an instance of the SYMMETRIC LAYER COLORING PROBLEM to the graph $G_{\mathcal{S}}$. The connected component partition $\mathcal{C}_{\mathcal{S}}$ of $\mathcal{S}$ can be computed in $O(|\mathcal{S}| \log |\mathcal{S}|)$ time (see Imai and Asano [1983]), yielding the vertices of $G_{\mathcal{S}}$. In the literature the connected components are often assumed to be given as rectilinear polygons in the input, but we use shapes because we assume the checking oracle to check shapes only which is more realistic than checking arbitrary rectilinear polygons. If simple rectilinear polygons are given, we can decompose them into shapes appropriate for the checking oracle in linear time by using the algorithm in Section 4.2.2.

The runtime for generating the edges of $G_{\mathcal{S}}$ very much depends on the complexity of the design rules. In the worst case we need $|\mathcal{S}|^2$ calls to the design rule oracle, but for realistic design rules it is sufficient to check for conflicts only locally within a certain distance, reducing the number of oracle calls substantially.

In the following we focus on the GRAPH k-COLORING PROBLEM and collect some theoretical results. For most results G needs to be planar which might not be the case for all possible design rules. Note that one can find a planar embedding of a given graph or decide that it is not planar in linear time (see Hopcroft and Tarjan [1974]).

A 2-coloring of a graph is also known as a *bipartition*. It is well known in graph theory that a graph is 2-colorable (*bipartite*) if and only if it has no odd cycles (i.e., cycles of odd length), and that one can decide if a graph is bipartite, and find a bipartition if there exists one, in linear time (see [Korte and Vygen, 2012, Chapter 2]; König [1916]). Therefore, we have the following proposition.

Proposition 5.2. *The* Graph 2-Coloring Problem *can be solved in linear time.*

For three colors we have the following negative result.

Theorem 5.3 (Dailey [1980])**.** *The* Graph 3-Coloring Problem *is NP-hard, even if G is planar and all vertices have degree* 4.

Even the problem of deciding whether such a graph is 3-colorable is NP-complete (Dailey [1980]). The fastest known algorithm for the Graph 3-Coloring Problem runs in $O(1.3289^n)$ time (Beigel and Eppstein [2005]).

The famous question if each planar graph is 4-colorable dates back to the year 1852 (see Fritsch and Fritsch [1994] for a history of this problem). The positive answer, known as the *four color theorem*, was first proved by Appel et al. [1977a,b] via a very complicated and computer-assisted proof. Later Robertson et al. [1997] gave a simpler, still computer-assisted proof which leads to an $O(n^2)$ algorithm for finding a 4-coloring for a given planar graph.

Theorem 5.4 (Appel et al. [1977a,b], Robertson et al. [1997])**.** *Each planar graph is* 4*-colorable, and a* 4*-coloring can be found in $O(n^2)$.*

For at least five colors the coloring problem becomes easy in planar graphs, see Frederickson [1984] for a description of several linear time algorithms.

Theorem 5.5. *A* 5*-coloring for a planar graph can be computed in $O(n)$ time.*

We point out the interesting fact that the planar Graph k-Coloring Problem is polynomially solvable for at most two or at least four colors while being NP-hard for three colors.

So far we only considered the problem of finding a coloring for all shapes or decide that none exists. However, when solving the layout coloring step in Algorithm 5, it is very unlikely that there exists a feasible coloring for *all* shapes. In such a case, solving the Symmetric Layer Coloring Problem just gives us the answer that there is no feasible coloring which does not give any hint how to resolve the conflicts. In practice, it is more convenient to find a maximum subset of connected components for which a feasible coloring exists, and to compute such a coloring. After that, we can focus on uncolored components only and try to get rid of the remaining conflicts by using rip-up and reroute techniques. We consider the following problem.

Partial Symmetric Layer Coloring Problem
Instance: A shape set $\mathcal{S}$ on layer $z \in Z_{\text{all}}$ with $\psi_u(\mathcal{S}) = 1$, and color-symmetric design rules.
Task: Compute a set of connected components $\mathcal{C}' \subseteq \mathcal{C}_{\mathcal{S}}$ containing the set of shapes $\mathcal{T} := \bigcup \mathcal{C}'$, and compute a coloring $\alpha_{\mathcal{T}}$ such that $\psi_c(\mathcal{T}, \alpha_{\mathcal{T}}) = 1$ with $|\mathcal{C}'|$ maximum.

We again use the conflict graph $G_{\mathcal{S}}$. We first consider the case $k_z = 2$, i.e. double patterning. Then the Partial Symmetric Layer Coloring Problem reduces to finding a maximum set $W \subseteq V(G_{\mathcal{S}})$ such that the induced subgraph $G_{\mathcal{S}}[W]$ is 2-colorable. This can be reformulated to finding a minimum set of vertices whose deletion makes $G_{\mathcal{S}}$ bipartite, known as the *graph bipartization problem* (Choi et al. [1989]).

Graph Bipartization Problem
Instance: An undirected graph G.
Task: Compute a minimum vertex set $U \subseteq V(G)$ such that $G[V \setminus U]$ is bipartite.

Such a vertex set U is also known as an *odd cycle transversal* since it hits all odd cycles. We have the following negative result.

Theorem 5.6 (Choi et al. [1989]). *The* Graph Bipartization Problem *is NP-hard, even when restricted to planar graphs whose maximum vertex degree exceeds three.*

On the positive side, the Graph Bipartization Problem is *fixed-parameter tractable*, leading to the following results. Let t be the cardinality of an optimal solution U of the Graph Bipartization Problem.

Theorem 5.7 (Reed et al. [2004],Hüffner [2005]). *The* Graph Bipartization Problem *can be solved in $O(3^t mn)$.*

Theorem 5.8 (Fiorini et al. [2008]). *The planar* Graph Bipartization Problem *can be solved in $O(n)$ time for fixed t.*

Recently, Kratsch and Wahlström [2012] gave a randomized polynomial *kernelization* for the Graph Bipartization Problem. A polynomial kernelization is a polynomial algorithm turning the graph instance into an equivalent instance whose size depends only polynomially on t. Their approach is based on matroid theory.

For $k_z \geq 4$ and $G_{\mathcal{S}}$ planar the Partial Symmetric Layer Coloring Problem is equivalent to the Symmetric Layer Coloring Problem since we always find a *complete* coloring by Theorem 5.4.

We now mention some results for the weighted version of the Partial Symmetric Layer Coloring Problem. Here each connected component in $\mathcal{S}$ is assigned a nonnegative weight and we want to find a set $|\mathcal{C}'|$ of maximum weight instead of maximum cardinality. These weights can be used to incorporate the estimated hardness of rerouting a component, for example. For $k_z = 2$ this problem reduces to the weighted Graph Bipartization Problem for which Baïou and Barahona [2014] gave an $O(n^{\frac{3}{2}} \log n)$ time algorithm if the graph is planar and all vertices have degree at most three. For graphs with maximum vertex degree at least four the problem is NP-hard, even for uniform weights. For non-planar graphs, the problem is already NP-hard for graphs with maximum vertex degree at least three, even for uniform weights. Both results are due to Choi et al. [1989] who also mentioned an interesting application of the planar graph bipartization problem for *via minimization* in VLSI routing. They consider a two-dimensional Manhattan routing and ask for an assignment of the wires to two layers minimizing the number of vias needed.

Since in practice the degree of a vertex in $G_{\mathcal{S}}$ corresponding to a large connected component can be much higher than three, the algorithm by Baïou and Barahona [2014] is not directly applicable to the Partial Symmetric Layer Coloring Problem. However, one could break up components artificially to fulfill the degree constraint, corresponding to possible stitch positions.

Goemans and Williamson [1998] described a $\frac{9}{4}$-factor approximation algorithm for the weighted Graph Bipartization Problem in planar graphs. Their algorithm runs in $O(n^3)$ time and extends to various other *hitting cycle problems*. See also Kahng et al. [2001] for experimental results on different heuristics for the planar weighted Graph Bipartization Problem in the context of double patterning. They obtain the best results by using the $\frac{9}{4}$-factor approximation algorithm of Goemans and Williamson [1998].

In the Partial Symmetric Layer Coloring Problem we were looking for a maximum partial coloring. Another reasonable approach is to compute a complete coloring while minimizing the number of remaining *conflicts pairs*. For a given coloring $\alpha_{\mathcal{S}}$ we denote each $\{C_1, C_2\} \in \binom{\mathcal{C}_{\mathcal{S}}}{2}$ for which there exist $S_1 \in C_1, S_2 \in C_2$ such that $\psi_c(\{\{S_1, S_2\}, \alpha_{\{S_1,S_2\}}\}) = 0\}$ as *conflict pair*.

Conflict Avoiding Symmetric Layer Coloring Problem
Instance: A shape set $\mathcal{S}$ on layer $z \in Z_{\text{all}}$ with $\psi_u(\mathcal{S}) = 1$, and color-symmetric design rules.
Task: Compute a coloring $\alpha_{\mathcal{S}}$ minimizing the number of conflict pairs.

The Conflict Avoiding Symmetric Layer Coloring Problem reduces to solving the following problem in $G_{\mathcal{S}}$ for $k = k_z$.

MAXIMUM k-CUT PROBLEM	
Instance:	An undirected graph G.
Task:	Compute a partition of $V(G)$ into k subsets $V_1, V_2, \dots, V_k$, maximizing the number of edges whose vertices are in different sets of the partition, i.e. $\lvert\{\{v,w\} \in E(G) \mid \exists i,j \in \{1,\dots,k\}, i \neq j :\ v \in V_i, w \in V_j\}\rvert$.

Here each set of the partition corresponds to one color, and since the number of edges whose vertices have different color is maximized, the number of edges whose vertices have the same color and thus the number of conflict pairs is minimized. We start with a negative result.

Theorem 5.9 (Kann et al. [1997])**.** *There is no polynomial time approximation algorithm for the* MAXIMUM k-CUT PROBLEM *with a relative error smaller than* $\frac{1}{34k}$*, unless P=NP.*

In the weighted version of the CONFLICT AVOIDING SYMMETRIC LAYER COLORING PROBLEM each pair $\{C_1, C_2\} \in \binom{\mathcal{C}_S}{2}$ is assigned a nonnegative weight and we want to find a coloring such that the total *weight* of conflict pairs is minimzed. Here weights can represent the hardness of conflicts.

Hadlock [1975] first proved that the weighted MAXIMUM 2-CUT PROBLEM in planar graphs is polynomially solvable by giving a nice reduction to the maximum weighted matching problem. A simple algorithm achieving a runtime of $O(n^{\frac{3}{2}} \log n)$ was proposed by Liers and Pardella [2012].

Note that approximation algorithms for the MAXIMUM k-CUT PROBLEM are of limited use for solving the CONFLICT AVOIDING SYMMETRIC LAYER COLORING PROBLEM because approximation ratios do not translate. For the sake of completeness, we nevertheless mention some important approximability results. For the weighted MAXIMUM 2-CUT PROBLEM (also denoted as *max-cut problem*) Goemans and Williamson [1995] described a polynomial time approximation algorithm producing a cut with weight at least $\frac{2}{\pi} \min_{0<\theta\leq\pi} \frac{\theta}{1-\cos\theta} \approx 0.878567$ times the optimal value. Their pioneering semidefinite programming approach was the first obtaining an approximation ratio significantly better than two which is achieved by a simple greedy algorithm already. Interestingly, the above approximation factor is best possible if the *unique games conjecture* (Khot [2002]) holds as shown by Khot et al. [2007]. For the general weighted MAXIMUM k-CUT PROBLEM the best known approximation algorithm for small fixed values of k was given by de Klerk et al. [2004]. Their algorithm produces a solution with weight at least 0.836008 for $k = 3$ and at least 0.857487 for $k = 4$. A naive randomized heuristic assigning each vertex one of the k subsets of the partition randomly yields a solution whose expected weight is at most a factor $\frac{1}{k}$ less than the weight of an optimal solution (see e.g. Kann et al. [1997]).

We conclude that, assuming color-symmetric rules and a given planar conflict graph, three colors for multiple patterning are most difficult to handle from a theoretical point of view. For two colors we have an efficient approximation algorithm for solving the planar Partial Symmetric Layer Coloring Problem, an efficient optimal algorithm for the Conflict Avoiding Symmetric Layer Coloring Problem, and a linear time algorithm for the Symmetric Layer Coloring Problem. For four colors all problems can be solved optimally in $O(n^2)$ time for planar graphs, but the corresponding complex algorithm which is based on the four color theorem is not suitable for practice. For at least five colors all problems can be solved in linear time for planar graphs.

5.2.2 General Design Rules

We now skip our assumption that design rules are color-symmetric. This setting is also well-suited for SADP technology. We directly focus on the partial coloring problem here. Again $\mathcal{C}_\mathcal{S}$ denotes the connected component partition of $\mathcal{S}$.

Partial Layer Coloring Problem

Instance: A shape set $\mathcal{S}$ on layer $z \in Z_{\text{all}}$.

Task: Compute a set of connected components $\mathcal{C}' \subseteq \mathcal{C}_\mathcal{S}$ containing the set of shapes $\mathcal{T} := \bigcup \mathcal{C}'$, and compute a coloring $\alpha_\mathcal{T}$ such that $\psi_c(\mathcal{T}, \alpha_\mathcal{T}) = 1$ with $|\mathcal{C}'|$ maximum.

This problem does not translate directly to a graph coloring problem anymore. We use a construction leading to a *stable set* problem.

Definition 5.10. *The* color-asymmetric conflict graph *for a shape set $\mathcal{S}$ is the undirected graph $H_\mathcal{S}$ with vertices*

$$V(H_\mathcal{S}) := \mathcal{C}_\mathcal{S} \times \{1, 2, \ldots, k_z\}$$

and edges

$$E(G_\mathcal{S}) := \{\{(C_1, c_1), (C_2, c_2)\} \in \binom{V(H_\mathcal{S})}{2} \mid (C_1 = C_2 \wedge c_1 \neq c_2) \vee \\ \exists S_1 \in C_1, S_2 \in C_2 : \psi_c(\{S_1, S_2\}, \alpha_{(S_1,c_1),(S_2,c_2)}) = 0\},$$

where $\alpha_{(C_1,c_1),(C_2,c_2)}$ is defined by $\alpha(C_1) = c_1$ and $\alpha(C_2) = c_2$.

The conversion of an instance of the Partial Layer Coloring Problem to the graph $H_\mathcal{S}$ can be done similarly to the color-symmetric case, see Section 5.2.1. We use $n := |V(H_\mathcal{S})|$ and $m := |E(H_\mathcal{S})|$ in the following. The graph $H_\mathcal{S}$ contains an edge between any two colored connected components of shapes which are not legal with respect to ψ_c, and between vertices representing the same component with different colors.

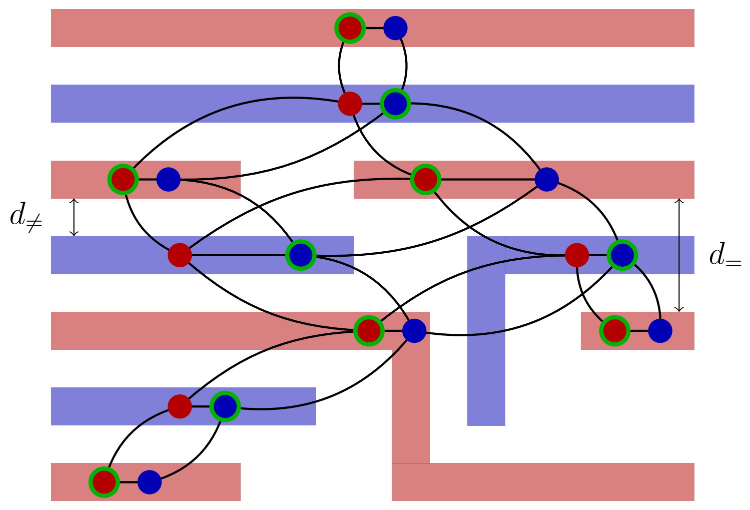


Figure 5.3: A stable set (green encircled vertices) in the graph $H_{\mathcal{S}}$ corresponding to the colored instance in Figure 5.1b.

By this construction, each coloring $\alpha_{\mathcal{T}}$ with $\mathcal{T} = \bigcup \mathcal{C}'$ for some $\mathcal{C}' \subseteq \mathcal{C}_{\mathcal{S}}$ and $\psi_c(\mathcal{T}, \alpha_{\mathcal{T}}) = 1$ corresponds to a *stable set* (a set of pairwise non-adjacent vertices, see Korte and Vygen [2012]) of size $|\mathcal{C}'|$ in $H_{\mathcal{S}}$ and vice versa. See Figure 5.3 for an example. Note that a stable set in $H_{\mathcal{S}}$ cannot have cardinality greater than $|\mathcal{C}_{\mathcal{S}}|$, because each connected component can contribute at most one vertex to a stable set. Therefore, finding an optimal solution for the Partial Layer Coloring Problem reduces to the following problem.

Maximum Stable Set Problem
Instance: An undirected graph G.
Task: Compute a stable set of maximum cardinality in G.

We have the following negative results.

Theorem 5.11 (Berman and Fujito [1995]). *The* Maximum Stable Set Problem *is MAXSNP-hard, that means there exists no polynomial-time approximation scheme unless P=NP, even for graphs with maximum vertex degree three.*

Theorem 5.12 (Garey and Johnson [1977]). *The* Maximum Stable Set Problem *is NP-hard for planar graphs with maximum vertex degree three.*

Halldórsson and Radhakrishnan [1997] showed that a simple greedy algorithm, which selects in each step a vertex v of minimum degree and deletes v and all its neighbors, is a $\frac{\Delta+2}{3}$-factor approximation algorithm for the Maximum Stable Set Problem in graphs with maximum vertex degree Δ. Chiba et al. [1982] described an $O(n \log n)$ time 2-factor approximation algorithm for the planar Maximum Stable Set Problem. For the same problem Baker [1994] developed an approximation scheme yielding a $\frac{k+1}{k}$-factor approximation algorithm for each fixed k, using

$O(8^k kn)$ time and $O(4^k n)$ space. See also Alekseev et al. [2008] for more hardness and approximability results on the Maximum Stable Set Problem in planar graphs. The best known approximation algorithm for the weighted Maximum Stable Set Problem in general graphs is due to Halldórsson [1999] and achieves a performance guarantee of $O(\frac{n}{\log^2 n})$. Held et al. [2012] described an efficient and numerically robust branch-and-bound implementation solving the weighted Maximum Stable Set Problem and presented a number of results on standard test instances, comparing their algorithm to other approaches.

Despite the theoretical hardness of the (weighted) Maximum Stable Set Problem, the formulation of the Partial Layer Coloring Problem as stable set problem in a conflict graph has some advantages. It offers a flexible and unified coloring approach, independent from the number of colors k_z and suited for LELE as well as SADP technologies.

5.2.3 Practical Approaches

The problem of coloring given layouts with respect to multiple patterning design rules is known as *layout decomposition* in the literature. Most of the proposed algorithms use *integer linear programming* or *semidefinite programming* approaches in combination with speed-up techniques such as conflict graph partitioning, see e.g. Kahng et al. [2008] for double patterning, Yu et al. [2011] for triple patterning, and Yu and Pan [2014] for quadruple patterning and beyond.

The main technique to speed up the runtime for solving layout decomposition problems is to partition the conflict graph into smaller subgraphs which can be colored independently and merged later on, following the *divide and conquer* paradigm. For example, different connected components of the conflict graph can be colored independently and vertices with degree less than k_z can be removed and colored afterwards. Furthermore, 1-cut, 2-cut and 3-cut removal has been proposed for double, triple and quadruple patterning, respectively. See (Kahng et al. [2008], Yu et al. [2011], Yu and Pan [2014]) for more details.

Many approaches also incorporate stitches as a means to minimize conflicts between same-colored shapes. For this, connected components (and their corresponding vertices in the conflict graph) are split, and the resulting new components are allowed to be colored differently. Kahng et al. [2010] described how to find all possible stitch locations and how to manipulate the conflict graph to incorporate these stitches for double patterning. Fang et al. [2012] showed that this method does not yield all possible stitch locations for triple patterning and proposed a *stitch-aware mask assignment* heuristic.

In practice, the layout to be colored often has a special structure which can be used to design polynomial time algorithms even for the general Partial Layer Coloring Problem. For example, often a partition of the chip area on a layer into stripes with small height (or width) is given by regular *power supply wires* which proceed over the whole chip width (or height). Each such stripe corresponds to a *circuit row* and contains only a small number of routing tracks (< 20), depending on the *circuit library*. Assuming that color conflicts do not cross power wires, this partition allows to solve the coloring problem for each stripe seperately and merge the obtained solutions afterwards. This also works if power wires are part of the coloring instance, see Ahrens [2012] for more details.

For BonnRoute we developed two coloring algorithms exploiting this stripe structure, one using a flexible integer programming formulation for the stable set problem (Nohn [2012]) and one polynomial time dynamic programming approach (Ahrens [2012]). In Ahrens [2012] experimental results for both algorithms are given and show that the dynamic programming approach is much faster. Both algorithms were used in the context of design-technology co-optimization in cooperation with our industry partner IBM, to quantify the colorability of layouts under preliminary double patterning design rules.

A similar approach was described in Tian et al. [2012] where a polynomial time coloring algorithm for triple patterning is given, using the circuit row structure. Here also stitches and *color balancing* are incorporated.

The theoretical reason why hard coloring problems can be solved fast in presence of seperating power wires is that the underlying conflict graphs have *path decompositions* (see Bodlaender [1994], Robertson and Seymour [1983]) of small width. Given such path decompositions, many NP-hard optimization problems can be solved in linear or at least polynomial time, see Bodlaender [1994]. We think that it is very promising to use this general concept of *linearizing* instances of bounded width to design fast algorithms for other optimization problems occuring in routing. The pin access algorithm of BonnRoute (Ahrens [2014]) is actually based on the same concept.

5.3 Creating Colorable Layouts

Next we consider the problem of creating (uncolored) layouts which are guaranteed to be colorable afterwards, making a conflict resolution step redundant. Note that it still might not be trivial to actually *compute* a coloring for such layouts. For example, Khanna et al. [2000] showed that even coloring a 3-colorable graph with four colors is NP-hard.

We now collect some sufficient conditions for the solvability of the Symmetric Layer Coloring Problem. Again, we assume that stitches are not available. Recall that the Symmetric Layer Coloring Problem can be formulated as a k_z-coloring problem in the color-symmetric conflict graph $G_{\mathcal{S}}$ (Definition 5.1).

For $k_z = 2$ the Symmetric Layer Coloring Problem has a feasible solution if and only if $G_{\mathcal{S}}$ has no odd cycles. For $k_z \geq 4$ we know that the Symmetric Layer Coloring Problem has always a feasible solution if $G_{\mathcal{S}}$ is planar (Theorem 5.4).

We now focus on sufficient conditions for the case where $G_{\mathcal{S}}$ is planar and $k_z = 3$. In the following we denote a cycle of length k in a graph as *k-cycle*.

Theorem 5.13. *(Grötzsch [1959]) A planar graph is* 3*-colorable if it contains no* 3*-cycles.*

See Thomassen [1994] for a short proof of this theorem. Grünbaum et al. [1963] tightened this result by showing that planar graphs with at most three 3-cycles are still 3-colorable while graphs with four 3-cycles are not in general.

Steinberg conjectured in 1975 that every planar graph without cycles of length four and five is also 3-colorable. This conjecture is still open, but there are some results into this direction.

Theorem 5.14. *(Borodin et al. [2005]) A planar graph is* 3*-colorable if it contains no cycles of length* 4-7*.*

Theorem 5.15. *(Wang and Chen [2007]) A planar graph is* 3*-colorable if it contains no cycles of length* 4,6,8*.*

Theorem 5.16. *(Borodin et al. [2009]) A planar graph is* 3*-colorable if it contains no cycles of length* 5,7 *and no adjacent* 3*-cycles.*

There are also results taking the minimum distance of distinct 3-cycles into account.

Theorem 5.17. *(Borodin and Glebov [2011]) Every planar graph without* 5*-cycles and with minimum distance between any* 3*-cycles at least two is* 3*-colorable.*

See Borodin [2013] for a survey on colorings of planar graphs and more sufficient conditions for 3-colorability.

The following problem related to the colorability of graphs was posed by Erdős and Rényi [1960]: What is the largest number Δ such that a random graph with average degree Δ is k-colorable with high probability? For example, it is known that almost all random graphs with average degree at most 4.03 are 3-colorable (Achlioptas and Moore [2003]), while almost all random graphs with average degree at least 4.99 are *not* 3-colorable (Kaporis et al. [2001]). See Coja-Oghlan and Vilenchik [2013] for more recent results on this topic.

Now that we know certain graphs allowing a feasible solution for the Symmetric Layer Coloring Problem, the next question is how to construct *artificial* (i.e. not manufacturing driven) design rules such that each layout satisfying an uncolored checking oracle based on these rules has a feasible coloring. This question is hard to answer in general, we give an example for the case $k_z = 2$. Here we can add artificial rules forbidding any jogs, allowing only standard wires running on routing tracks, and forcing same-color distances between wires on the same track. For realistic color-dependent design rules (see for example Figure 5.1) we can then color all shapes on odd tracks with one color and shapes on even tracks with the other color, without producing any conflict. However, these artificial rules making the coloring problem trivial are too restrictive in practice, since all wires must have the same width here.

Generally, designing such artificial rules involves a trade-off between simple over-restricted rules (such as in the example above) and complex less restrictive rules which mimic the sufficient conditions for conflict graph colorings on a shape basis. In this case, the complexity is shifted into the checking oracle and thus into routing algorithms which one normally tries to avoid. See also Liebmann et al. [2009] on this topic, where simple layouts through design-technology co-optimization are proposed, using artificial design rules. BonnRoute also uses simple artificial rules (*track patterns* and *preferred colors*) to manage multiple patterning, see Section 5.4.1.

5.4 Multiple Patterning in BonnRoute

In the following we describe the most important adaptions and innovations in BonnRoute related to multiple patterning technology. As pointed out in Chapter 3, the major step in BonnRoute where no global optimization takes place is computing *long connections* (see line 8 in Algorithm 1). This step is also affected most critically by the color dependency of shapes resulting from multiple patterning. Therefore, our main focus is on computing long connections in the following.

For details on how multiple patterning is incorporated into our pin access algorithm see Ahrens et al. [2015], Ahrens [2014]. In global routing, only few adaptions are needed for multiple patterning technology because it works on a much coarser view of the chip where the color dependency is not so critical.

We now review some work related to routing for multiple patterning technologies. Early detailed routing algorithms that have been proposed for multiple patterning (Cho et al. [2008], Yuan et al. [2009], Lin and Li [2010], Gao and Macchiarulo [2010]) have been targeted at LELE technology. These approaches have in common that they assume the availability of stitches and jogs which is not always realistic. Moreover, they perform coloring greedily during routing, without consideration of convergence issues, and do not consider multi-width routing and pin access. The primary goal of these approaches is to reduce stitch and coloring conflict counts which they achieve by various heuristics. Furthermore, triple patterning has been studied as a means to improve on these metrics Ma et al. [2012], Lin et al. [2012].

For SADP technology layer assignment has been proposed to resolve conflicts (Gao and Pan [2012]), but it is not clear if this is always possible under realistic via spacing constraints which exceed the track pitch considerably if via layers are produced with single patterning.

In contrast to the above approaches, BonnRoute does not require stitches and jogs, incorporates mixed-width wires while still obtaining high packing density, and handles real-world multiple patterning design rules.

Parts of this section will be published in Ahrens et al. [2015].

5.4.1 Routing Space and Automatic Coloring

We now describe how routing space is represented and used in presence of multiple patterning and how multiple patterning colors for routed wires are determined automatically in BonnRoute.

In previous technology nodes (up to 22 nm) BonnRoute precomputed for each layer routing tracks (Definition 3.6 on page 11) on which the majority of wires was routed. Here all different wire types (Definition 3.5 on page 10) used the same tracks, and only short connections and access paths were routed off-track (see Definition 3.8 on page 11) when necessary.

For advanced technology nodes this track concept is generalized in two ways: First, different track sets for different wire types (*track patterns*) are defined. Second, a *preferred color* for each track that should be used by the majority of wires is defined. Both concepts help to increase packing density by guiding the sequential detailed routing step used in BonnRoute (see Chapter 3 and Gester et al. [2013]) to pack wires as dense as possible.

The core idea of automatic coloring is to use these preferred colors as a *color pattern*, prescribing for each placed on-track wire a certain color. A routing algorithm computing a connection does not have to know anything about colors, but it just routes uncolored wires which adopt the color according to the pattern automatically, like a chameleon. A colored checking oracle is used to answer queries if wires are legal at certain positions. This concept is preferable to just creating a colorable layout (see Section 5.3) because here a color of a wire is available directly after it was routed, and not after *all* wires were routed. This avoids pessimistic artificial design rules and allows the flexibility to color wires violating the track or color pattern, if needed. We now formalize these concepts. Recall that tracks_z is the set of all tracks on layer z (see Definition 3.6 on page 11).

Definition 5.18. *A* track pattern *for a wire type W is a function* tp_W *assigning each layer $z \in Z_{\text{wiring}}$ a set of routing tracks such that* $\text{tp}_W(z) \subseteq \text{tracks}_z$*. We denote the set* $\bigcup_{z \in Z_{\text{wiring}}} \text{tp}_W(z)$ *as the* preferred tracks *for wire type W.*

A preferred coloring *for a wire type W is a function c_W assigning each preferred track of W a* preferred color *such that $c_W(t) \in \{1, \ldots, k_z\}$ for any $t \in \text{tp}_W(z)$ and $z \in Z_{\text{wiring}}$.*

We call a wire *on-track-pattern* if it is on-track and both endpoints of its stick lie on preferred tracks, and we call it *off-track-pattern* otherwise. For a plane wire, if these preferred tracks have the same preferred color (with respect to the wire type), then we call this color the *preferred color* of the wire. Note that jogs may not have a preferred color. For vias we have separate preferred colors for each via or wiring layer the via intersects. We call these colors the *layerwise preferred colors* of a via.

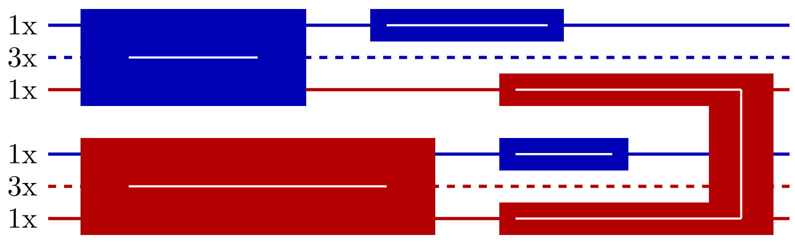


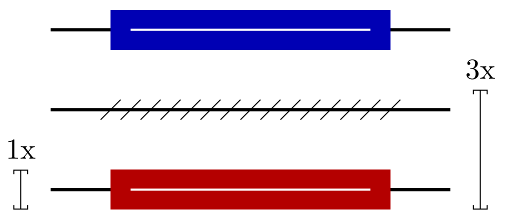
Figure 5.4: Example for track patterns and some wires placed and colored according to these track patterns. We have one track pattern for 1x wires (solid lines) and one for 3x wires (dashed lines). The colors correspond to multiple patterning colors. The color of a track represents its preferred color. The color of a wire shape is the preferred color of the wire.

Note again that different layers are completely independent in terms of coloring. We defined track patterns and preferred colors only for wiring layers because the track graph in which we compute long connections is based on wiring layers only (see Definition 3.7). Preferred colors for via cut shapes can be either aligned with colors for bottom or top shapes, or defined by seperate color schemes for via layers. Currently BonnRoute uses the first option.

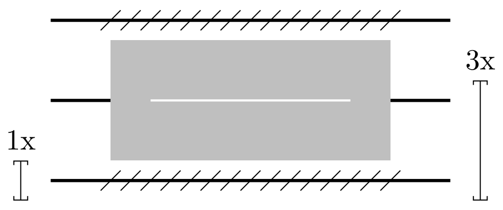
We want to use track patterns as a means of forcing an efficient usage of wires with respect to packing density. To this end, a track pattern for a wire type is typically defined in a way such that wires with this wire type can be packed as dense as possible on-track-pattern. For this also repeating blocked areas on a chip layer such as *power supply wires* are taken into account. As already mentioned in Chapter 3, typically most wires on a chip have the same standard wire type, so it is of special importance to allow dense packings for wires of this wire type. See Figure 5.4 for an example of track patterns for two different wire types.

BonnRoute is able to read in predefined track patterns, otherwise it computes them by using the algorithm described in [Müller, 2009, Section 2.4]. We do not go into details how these track patterns actually look like since this depends on the widths of all involved wire types on a chip and is related to routing methodology decisions of our industry partner IBM which we are not allowed to publish. However, we note that for mixed 1x, 3x and 5x wires with 1x mindist (these values are typical for our real-world multiple patterning designs) we obtain a high packing density for on-track-pattern wires. In general (for example for 1x, 2x and 4x wires with 1x mindist) a high packing density in case of mixed width wires is harder to obtain by defining track patterns only. If in such settings the packing density is not satisfying, then an additional track assignment step (which is not yet needed and integrated into BonnRoute) could help to obtain better packings (see Batterywala et al. [2002]).

For current chip technologies this track pattern concept is essential to obtain dense routing solutions. Without any hint how to place and color wires, the probability



(a) Two differently colored 1x wires making the crossed out part of the middle track unusable (minimum distance is 1x for diff-color and 3x for same-color). If both wires had the same color, then the middle track could be used.



(b) The 3x wire using the same tracks as 1x wires blocks three tracks, while shifting it by 1x distance below or above would only block two tracks for 1x wires (minimum distance is 1x between any wires).

Figure 5.5

of unusable gaps produced by the sequential routing step (see Figure 5.5a) would be much higher. Without separate tracks for different wire types, wider wire types block more tracks than necessary for other wire types (see Figure 5.5b).

BonnRoute uses track patterns in the following way. First, all wires are preferably routed on-track-pattern. Only if some connection cannot be found otherwise, then off-track pattern wires are allowed at some penalty cost. We will handle this case later. Second, the shapes of on-track-pattern wires are *automatically colored* with their (layerwise) preferred colors. The only problem occurs if a wire has no preferred color, which is only the case for jogs whose stick ends on tracks with different preferred colors. We have three options to handle such cases:

(i) We forbid such jogs without preferred color to be routed at all.

(ii) We add a stitch to the jog shape, such that the ends of the jog shape get the preferred colors of their tracks, see Figure 5.6a.

(iii) We color the jog shape completely in one of the preferred colors of the ending tracks. Here for the end of the jog with the wrong color we also need to insert a stitch because subsequent wires are assigned their preferred color, see Figure 5.6b. Here the color pattern is disarranged and a part of the red track above the upper wire gets blocked for wires in preferred color.

Currently BonnRoute uses the first option since on our real-world multiple patterning designs such jogs are not crucial for routing convergence. See Section 5.4.4 for more details.

For the vast majority of routed wires it is sufficient to stay on-track-pattern and use the preferred color, taking color dependency off the routing algorithms. Only if no connection can be found this way we allow to use other colors (see Section 5.4.2.4 for more details). This approach allows dense wire packings on the one hand since wires and colors are well aligned by track patterns, with only few exceptions (see

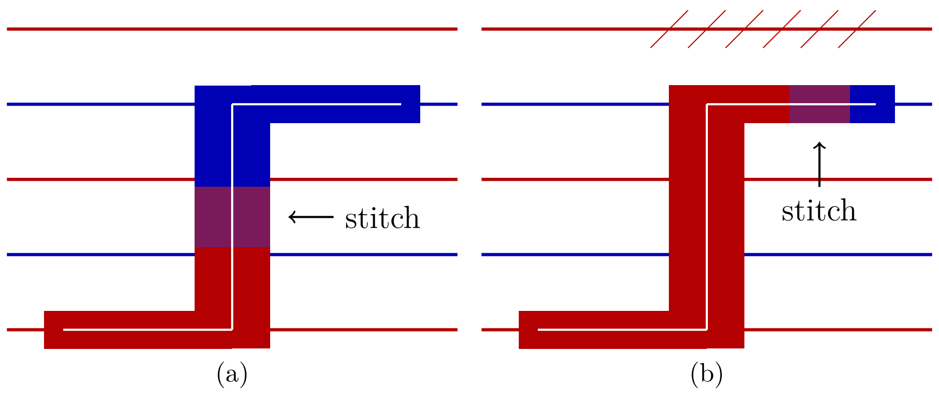


Figure 5.6: Two possibilities to color jogs having no preferred color. The overlay area corresponding to a stitch is purple.

Figure 5.4), and faster shortest path computations on the other hand since the involved algorithms do not need to try different color variants in presence of automatic coloring.

5.4.2 Computing Long Connections

5.4.2.1 Search Space and Problem Formulation

We now describe how the search space for the computation of long on-track-pattern connections is represented in BonnRoute. We use a directed version $\bar{G}$ of the track graph (see Definition 3.7 and Figure 3.3 on page 12) in the following, where each undirected edge is replaced by two oppositely directed edges. We assume that we are given a net N, two connected components $\bar{S}$ and $\bar{T}$ of N, and a *restricted routing corridor* $\mathrm{rcorr}(N, \bar{S}, \bar{T})$ (see Chapter 3), based on the global routing for net N. For simplicity we assume that this corridor consists of one wire type region (W, R) only, more wire types can be handled in a straightforward way. Let $A \subseteq V(\bar{G})$ be the set of vertices that are contained in R. Our goal is to connect the two components $\bar{S}$ and $\bar{T}$ by on-track wires within the induced subgraph $\bar{G}[A]$. If no connection can be found within $\bar{G}[A]$, then this restriction is relaxed gradually. However, note that we should aim at routing connections within the corridors given by global routing because otherwise the congestion estimation made by the global router is disregarded and we block routing space not foreseen by the global router.

The source and target components (consisting of pins or wires) are transformed to sets of *access objects* by an *access area oracle* (see [Schulte, 2012, Section 2.5] for details). Each such object contains one vertex of $\bar{G}[A]$ where the source or target component can be accessed (called the *access vertex*) together with at most one additional vertex of $\bar{G}[A]$ for each possible routing direction (called *direction vertex*). The access objects are build in such a way that when accessing an access vertex

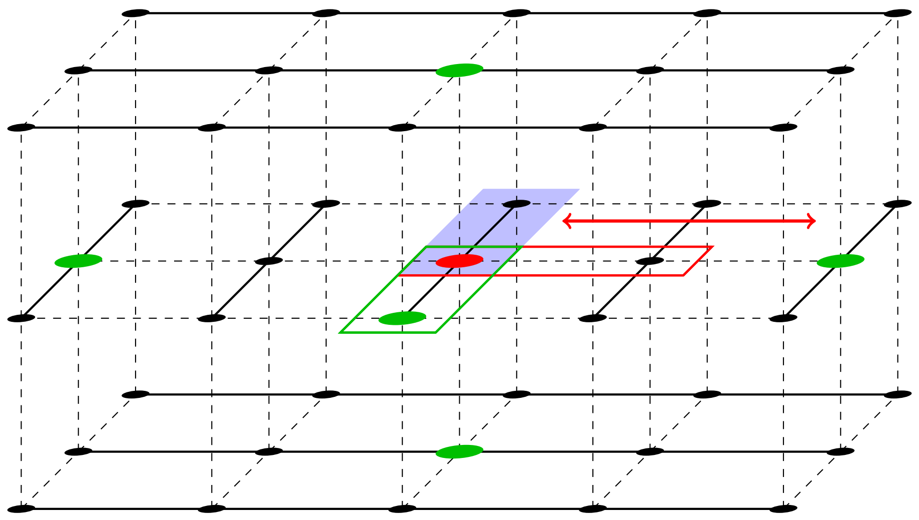


Figure 5.7: Section of a directed track graph (all edges are meant to be bidirectional) including an example for an access object in presence of one minedge rule whose minimum length is given by the red arrow. The blue shape is the pin to access, the red vertex one of its access vertices, and the green vertices the corresponding direction vertices. We assume that all shapes of the wire type to use have the same width as the pin, and that all tracks are preferred tracks with respect to the wire type. The example access wire with green border would be legal since it aligns with the pin, while the wire with red border would produce a too short edge. Therefore, accessing the access vertex from the right is only allowed when reaching the direction vertex on the right.

with a wire in a certain direction, this wire must run at least to the corresponding direction vertex to be legal with respect to same-net rules. If there is no direction vertex, then the access vertex cannot be legally accessed from this direction. See Figure 5.7 for an example.

Now let $S \subseteq V(\bar{G})$ be the set of access vertices of the source component $\bar{S}$ and $T \subseteq V(\bar{G})$ the set of access vertices of the target component $\bar{T}$. The decision which of the components becomes source and which target is arbitrary in principle, but can make a big difference in runtime. For example, precomputing estimated distances to the target for vertices in $\bar{G}$ (a speed-up feature called *future cost*, see Section 5.4.2.3) is faster if the target component is small.

We assume $A \cap S \neq \emptyset$ and $A \cap T \neq \emptyset$, otherwise we clearly cannot find a connection. In the following we identify each edge $e \in E(\bar{G}[A])$ with a *corresponding wire*, that is the unique wire with wire type W and e as its stick. Again, we identify edges of $\bar{G}[A]$ with their corresponding line segments here. Let G be the graph resulting from $\bar{G}[A]$ by

a) replacing each edge of the form (s, w) with $s \in S$ by an edge (s, w'), where w' is the direction vertex of access vertex s for the direction of edge (s, w), and deleting the opposite edge (w, s)

b) replacing each edge of the form (v, t) with $t \in T$ by an edge (v', t), where v' is the direction vertex of access vertex t for the direction of edge (v, t), and deleting the opposite edge (t, v)

c) removing each pref wire and via edge with at least one vertex contained in a non-preferred track of W

d) adding an edge (v_1, v_k) for each sequence of jog edges $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k)$ such that

   - all vertices $v_1, v_2, \dots, v_k$ are distinct
   - the outer vertices v_1 and v_k are contained in preferred tracks of W whose preferred colors are equal, say they have color c
   - the inner vertices $v_2, \dots, v_{k-1}$ are either not contained in preferred tracks, or the preferred colors of their preferred tracks differ from c,

   and afterwards removing the edges $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k)$ for each sequence as above

e) removing each edge whose corresponding wire colored with its preferred color introduces a diff-net-mindist violation to any other present colored shape on the chip, using the colored checking oracle (see Section 5.4.3 for implementation details).

Each modification step is done based on the result obtained from the preceding modification step. Note that the order of the steps is important. By construction, S-T-paths in G do not violate diff-net-mindist rules (see e)), and the two end segments of the path do not produce same-net errors in combination with the pins or wires they access (see a) and b)). Furthermore, all endpoints of segments of such paths lie on preferred tracks of W (see c) and d)), and jogs in such paths have a preferred color (see d)). The edges inserted in step d) are needed to jump from a preferred track to the next preferred track with the same preferred color by a jog. The main remaining potential DRC-errors are same-net errors caused by inner path segments which is precisely the main motivation for multi-label shortest paths (Section 5.4.2.2). The deletion of opposite edges in a) and b) is not necessary for standard shortest paths, but helpful for multi-label shortest paths. See Figure 5.8 for an example of the conversion from $\bar{G}[A]$ to G.

Note that G is not a three-dimensional grid graph in general, in contrast to $\bar{G}$, see again Figure 5.8. However, the algorithms we describe in the following are able to handle such graphs as well. We do not store G explicitly, but rather query the

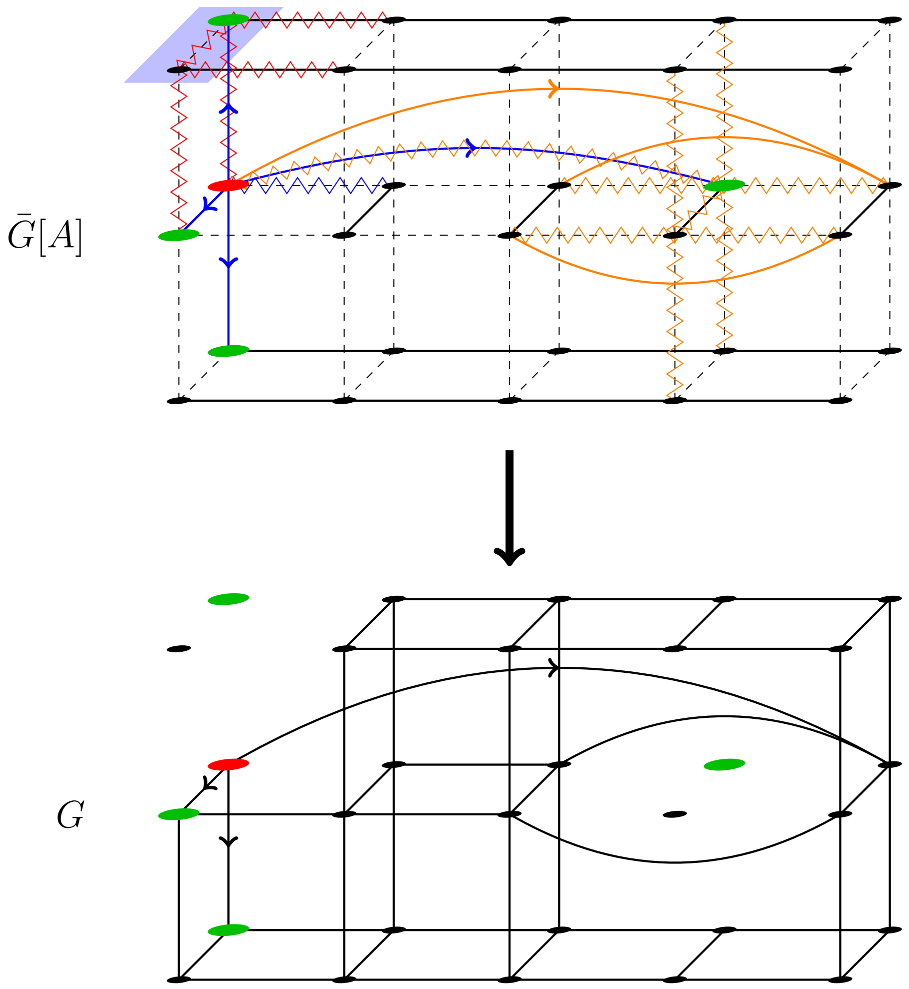


Figure 5.8: Example for the conversion from $\bar{G}[A]$ to G. The upper figure shows $\bar{G}[A]$, where solid black lines are tracks, all solid or dashed black lines between neighboring vertices are bidirectional edges in $\bar{G}[A]$, the red circle is a source access vertex, and the green vertices are the corresponding direction vertices. We assume that the fourth track on the middle layer is the only non-preferred track of the wire type to use, all preferred tracks have the same preferred color, and that the blue shape on the upper layer belongs to a different net. Each modification step corresponds to a color, where a zigzag line marks the deletion of an edge and a straight or curved line represents a new edge. First, according to modification step a) (blue), the outgoing and incoming edges of the access vertex are replaced by outgoing edges to direction vertices. Then according to step c) and d) (orange) all edges incident to the fourth track on the middle layer are deleted or replaced, respectively. Note that here the just created curved blue edge is deleted again. Finally, according to step e) (red), edges in conflict with the shape of a different net are deleted. The lower figure shows the resulting graph G, where solid lines with arrow are directed edges and solid lines without arrow are bidirectional edges.

colored checking oracle, the access area oracle, and the next preferred tracks for usable edges as needed. In this context, the modifications from $\bar{G}[A]$ to G can be understood as changed *label rules* for the used shortest path algorithm.

For a vertex $v \in V(G)$, we denote its x-, y- and z-coordinates by $x(v)$, $y(v)$ and $z(v)$, respectively. We further denote the set of possible directions for edges in G by $R := \{x_-, x_+, y_-, y_+, z_-, z_+\}$. For an edge $e = (v,w) \in E(G)$, we denote its direction by $r(e) \in R$ and its length by $l(e) := d_1(v,w)$.

We call a path P in G an r-path if $r(e) = r \in R$ for all $e \in E(P)$ and we call a path a *straight path* if it is an r-path for some $r \in R$. Further we denote the direction of a straight path P by $r(P)$. We define edge costs

$$c((v,w)) = \begin{cases} \gamma_{\{z(v),z(w)\}} & \text{if } (v,w) \text{ corresponds to a via} \\ \beta_{z(v)} \cdot l((v,w)) & \text{if } (v,w) \text{ corresponds to a jog} \\ l((v,w)) & \text{otherwise} \end{cases}$$

for two neighboring vertices $v, w \in V(G)$, where $\beta_{z(v)}, \gamma_{\{z(v),z(w)\}} \in \mathbb{N}_{>0}$ are layer-dependent parameters that encode penalty costs for wires running in non-preferred dimension and for vias, respectively. Now the core problem to be solved for computing on-track-pattern connections can be formulated as follows.

Path Search Problem

Instance: A search instance (G, c, S, T).

Task: Compute a shortest S-T-path in G with respect to cost function c, or decide that no such path exists.

5.4.2.2 Multi-Label Shortest Paths

In the following we present a generalization of the Path Search Problem taking additonal constraints for the resulting path into account. We discuss applications such as shortest paths with minimum segment lengths or shortest colored paths with stitch costs. In Section 5.4.2.3 we explain how this approach can be integrated into the interval-based on-track path search algorithm in BonnRoute. This integration was performed in joint work with Felix Nohn (Nohn [2012]).

There has been related work on finding shortest paths incorporating same-net rules such as minimum edge lengths, see for example Maßberg and Nieberg [2013] and Chang et al. [2013]. However, our approach is much more general and can also model non-geometric specifications such as colored design rules. It can be incorporated into any existing graph-based shortest path algorithm by only adapting the underlying graph (or the label rules of the algorithm, alternatively).

The key idea is to model path properties by assigning labels to the vertices of the path and allowing only certain label changes, at some specified cost. The following definition formalizes this idea.

Definition 5.19. *A* label system *is a triple* (L, t, d), *where* $L := \{l_1, l_2, \dots, l_k\}$ *is a finite set of* label types, $t : L \times L \times Z_{\text{wiring}} \times R \to \mathbb{N}_{>0} \cup \{\infty\}$ *is a* label transition function, *and* $d : L \times L \times Z_{\text{wiring}} \times R \to \mathbb{N}_0$ *is a* label cost function.

A label system is called well-defined *if it satisfies the following properties for all* $l_1, l_2 \in L, z \in Z_{\text{wiring}}, r \in R$:

(i) $t(l_1, l_1, z, r) \in \{1, \infty\}$

(ii) $t(l_1, l_2, z, r) \neq \infty \implies t(l_2, l_2, z, r) = 1$

(iii) $d(l_1, l_1, z, r) = 0$.

In the following we only use well-defined label systems since they are general enough for our purpose and we will utilize their properties for our graph construction. If we want to specify path properties (e.g. minimum segment lengths in certain directions) by label systems, we cannot just inspect the edges of a path in G since there may be straight paths which are long (and thus legal), but contain short edges only. Because of this we directly define multi-label sequences and paths based on straight paths instead of edges.

Definition 5.20. *A* multi-label sequence *in* G *is a pair* (P, ϕ), *where* P *is a sequence* $v_1, P_1, v_2, \dots, v_k, P_k, v_{k+1}$ *such that* P_i *is a straight path in* G *from* v_i *to* v_{i+1} *and* $\phi : \{v_1, v_2, \dots, v_{k+1}\} \to L$ *for a label system* $\mathcal{L} := (L, t, d)$. *If each vertex in* G *is traversed at most once* (P, ϕ) *is called* multi-label path. *A multi-label sequence is called* feasible *with respect to* $\mathcal{L}$ *if for each* $i \in \{1, 2, \dots, k\}$ *we have*

$$l(P_i) \geq t(\phi(v_i), \phi(v_{i+1}), z(v_i), r(P_i)).$$

The cost of a feasible multi-label sequence (P, ϕ) *is given by*

$$c_{\mathcal{L}}(P) := \sum_{i=1,2,\dots,k} (c(P_i) + d(\phi(v_i), \phi(v_{i+1}), z(v_i), r(P_i))).$$

As an example, we define a label system $\mathcal{L}_{\min} := (L, t, d)$ such that feasible multi-label S-T-paths in G are exactly those paths where all inner segments in x- or y-direction have length at least $l_{\min}$. Note that the end segments of a path are same-net clean by construction of G, see Section 5.4.2.1. For this let $L := \{\text{pref}, \text{jog}, \text{via}\}$, $d \equiv 0$ and the label transition function t given by the table in Figure 5.9.

Note that $\mathcal{L}_{\min}$ is a well-defined label system. Here the label types pref, jog and via at some vertex have the meaning that the incoming path segment must be directed in pref, jog or via direction, respectively. The label transition function ensures that at least distance $l_{\min}$ is covered between label type changes in x- or y-direction. Note that the end segments of the path may be shorter since one can start and end the path with an arbitrary label type. We consider the following problem.

			x_-/x_+	y_-/y_+	z_-/z_+
pref	$\to$	pref	1	∞	∞
jog	$\to$	pref	$l_{\min}$	∞	∞
via	$\to$	pref	$l_{\min}$	∞	∞
pref	$\to$	jog	∞	$l_{\min}$	∞
jog	$\to$	jog	∞	1	∞
via	$\to$	jog	∞	$l_{\min}$	∞
pref	$\to$	via	∞	∞	1
jog	$\to$	via	∞	∞	1
via	$\to$	via	∞	∞	1

Figure 5.9: Definition of label transition function for the label system $\mathcal{L}_{\min}$ for a layer z with preferred dimension x. The value of $t(l_1, l_2, r, z)$ is given by the entry in row $l_1 \to l_2$ and column r. For layers with preferred dimension y the roles of x and y are interchanged.

Multi-Label Path Search Problem

Instance: A search instance (G, c, S, T) and a well-defined label system $\mathcal{L}$.

Task: Compute a shortest feasible multi-label S-T-path in G with respect to cost function c and label system $\mathcal{L}$, or decide that no such path exists.

We cannot hope to solve this problem in polynomial time in general since it is NP-hard.

Theorem 5.21. *The* Multi-Label Path Search Problem *is NP-hard.*

Proof. We reduce the Hamiltonian s-t-Path Problem in a two-dimensional grid graph H (see Itai et al. [1982] for a proof of NP-hardness) to the Multi-Label Path Search Problem.

Let (H, s, t) be an instance of the Hamiltonian s-t-Path Problem, where $s := (s_x, s_y)$ and $t := (t_x, t_y)$. Let G be the three-dimensional grid graph containing a copy of H on layer 0, two additional vertices $s' := (s_x, s_y, 1)$ and $t' := (t_x, t_y, 1)$ and two additional edges $e_s := ((s_x, s_y, 1), (s_x, s_y, 0))$ and $e_t := ((t_x, t_y, 0), (t_x, t_y, 1))$.

We now define $n := |H|$ and $\mathcal{L} := (L, t, d)$, where $L := \{l_1, l_2, \ldots, l_n\}$,

$$t(l_i, l_j, z, r) := \begin{cases} 1, & \text{if } r \in \{x_-, x_+, y_-, y_+\} \text{ and } j \leq i + 1 \\ 1, & \text{if } r = z_- \text{ and } i = j = 1 \\ 1, & \text{if } r = z_+ \text{ and } i = j = n \\ \infty, & \text{else} \end{cases}$$

and $d \equiv 0$. Note that $\mathcal{L}$ is a well-defined label system. We further set $S = \{s'\}$ and $T = \{t'\}$.

Algorithm 6: Multi-label sequence algorithm

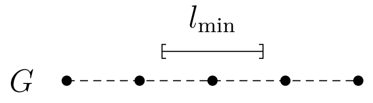
```
Input  : A search instance $(G, c, S, T)$ and a well-defined label system $\mathcal{L}$.
Output: A shortest feasible multi-label sequence $P$ in $(G, c, S, T)$ with respect
        to $\mathcal{L}$ (if existing).
1 Compute modified search instance $(G_\mathcal{L}, d_\mathcal{L}, S_\mathcal{L}, T_\mathcal{L})$
2 Compute shortest path $P'$ for instance $(G_\mathcal{L}, d_\mathcal{L}, S_\mathcal{L}, T_\mathcal{L})$
3 Map $P'$ to shortest feasible multi-label sequence $P$
4 return $P$
```

We claim that H contains a Hamiltonian s-t-path if and only if there is a feasible multi-label S-T-path in G with respect to $\mathcal{L}$, proving NP-hardness of the Multi-Label Path Search Problem. To see this, first note that leaving S is only possible with label type l_1 and entering T is only possible with label type l_n. Therefore, since the label type index increases by at most one at any label transition on layer 0, a feasible multi-label S-T-path must visit each vertex on layer 0 exactly once, yielding a Hamiltonian s-t-path in H. On the other hand, a Hamiltonian s-t-path P in H can be extended to a feasible multi-label S-T-path in G by just adding the two edges e_s and e_t and choosing the label types $l_1, l_1, l_2, l_3, \ldots, l_{n-1}, l_n, l_n$ for the $n + 2$ vertices of the path, starting at s'. □

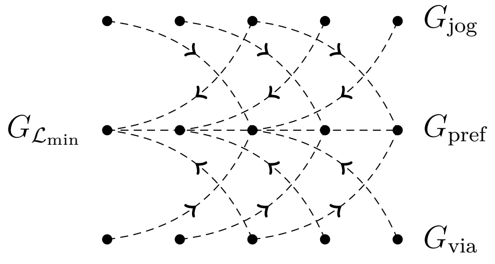
The constraint making the problem hard is that a path must not visit any edge or vertex multiple times. Therefore, we relax this constraint and search for a shortest feasible multi-label sequence as a start. This can be done in polynomial time as described in the following. We build a modified search instance $(G_\mathcal{L}, d_\mathcal{L}, S_\mathcal{L}, T_\mathcal{L})$ from (G, c, S, T) and a given well-defined label system $\mathcal{L}$, search a shortest path in $(G_\mathcal{L}, d_\mathcal{L}, S_\mathcal{L}, T_\mathcal{L})$ and then translate this path back to a shortest feasible multi-label sequence P in (G, c, S, T) (see Algorithm 6). If P is a path, then it is the desired shortest feasible multi-label path in G. We will discuss later how cycles in P are handled and how often they actually appear with reasonable label systems on practical instances.

We now describe how the modified search instance is built. Given a graph G and a label system $\mathcal{L} := (L, t, d)$, we define the graph $G_\mathcal{L}$ as follows. For each vertex in G we have one copy per label type in $G_\mathcal{L}$, that means $V(G_\mathcal{L}) := V(G) \times L$. Now let $v \in V(G_\mathcal{L})$, $l_1, l_2 \in L$ and $r \in R$ be fixed. Let w be the first vertex reachable in G by an r-path P from v whose length $l(P)$ is at least $t(l_1, l_2, z(v), r)$. If such a w exists, then we insert an edge $e = ((v, l_1), (w, l_2))$ into $G_\mathcal{L}$. The cost of e is defined by $d_\mathcal{L}(e) := c(P) + d(l_1, l_2, z(v), r)$.

We insert edges in this way for all vertices, label types and directions. The graph $G_\mathcal{L}$ contains exactly $|L|\,|V(G)|$ vertices and at most $|L|^2\,|V(G)|$ edges. Note that usually $|L|$ is a very small constant (≤ 5 for our applications). We denote the induced subgraph $G[V(G) \times \{l\}]$ as G_l for any label type $l \in L$. Edges between



(a) Simple grid graph G, dashed lines are bidirectional edges in x_- and x_+ direction, and $l_{\min}$ value needed for label system $\mathcal{L}_{\min}$.



(b) Graph $G_{\mathcal{L}_{\min}}$ corresponding to graph G in Figure (a) and label system $\mathcal{L}_{\min}$ based on $l_{\min}$ value in Figure (a). Curved dashed lines are directed and represent label changes, straight dashed lines are bidirectional.

Figure 5.10

different such subgraphs represent valid label changes.

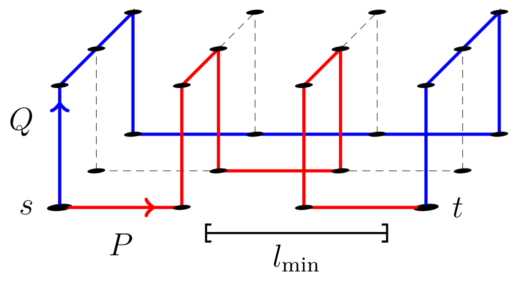
Building the graph $G_\mathcal{L}$ can be done in time $O(|L|^2 \cdot d_{max} \cdot |V(G)|)$, where d_{max} is the maximum number of edges on a straight path in G corresponding to an edge in $G_\mathcal{L}$ (the path denoted as P in the definition of the edges above). Assuming $\max_{l_1,l_2 \in L, r \in R} t(l_1,l_2,z,r) < \infty$ we have

$$d_{max} \leq \max_{z \in Z_{\text{wiring}}} \frac{\max_{l_1,l_2 \in L, r \in R} t(l_1,l_2,z,r)}{\min_{e=(v,w) \in E(G): z(v)=z} l(e)} + 1$$

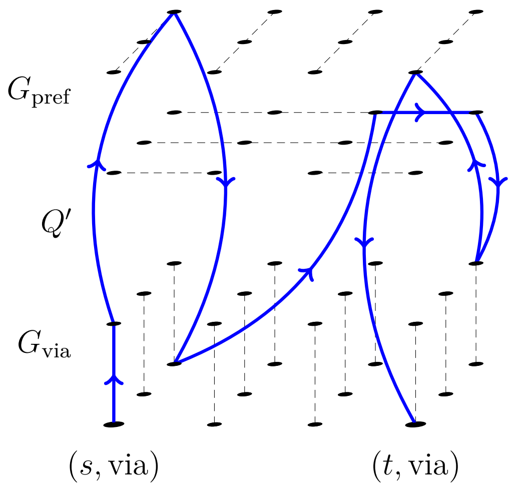
which is a small constant in practice since typically the numerator is a minimum segment length or a minimum distance originating from a design rule and the denominator is in the order of the minimum wire width. See Figure 5.10 for a small example of the graph construction for label system $\mathcal{L}_{\min}$. By defining $S_\mathcal{L} := \{(v,l) \in V(G_\mathcal{L}) \,|\, v \in S\}$ and $T_\mathcal{L} := \{(v,l) \in V(G_\mathcal{L}) \,|\, v \in T\}$ we obtain a new shortest path instance $(G_\mathcal{L}, d_\mathcal{L}, S_\mathcal{L}, T_\mathcal{L})$.

We now show that for a well-defined label system $\mathcal{L}$ every feasible multi-label path P in G corresponds to a path P' in $G_\mathcal{L}$ such that $c_\mathcal{L}(P) = d_\mathcal{L}(P')$. Let (P,ϕ) be a feasible multi-label path in G, where P is a sequence $v_1, P_1, v_2, \ldots, v_k, P_k, v_{k+1}$ of straight paths. Let $i \in \{1,2,\ldots,k\}$ be fixed. Then by definition $G_\mathcal{L}$ contains an edge $((v_i,\phi(v_i)),(w,\phi(v_{i+1})))$ such that w lies on the straight path P_i. We map P_i to this edge plus possible edges between $((w,\phi(v_{i+1})),(v_{i+1},\phi(v_{i+1})))$ which are part of $G_\mathcal{L}$ by definition (here we use that $\mathcal{L}$ is well-defined, otherwise these edges could be missing). Doing this mapping for each straight path of P yields a path P' in $G_\mathcal{L}$ with $c_\mathcal{L}(P) = d_\mathcal{L}(P')$ (note that keeping a label type does not produce label costs, since $\mathcal{L}$ is well-defined).

On the other hand, each path P' in $G_\mathcal{L}$ can be mapped to a feasible multi-label sequence P in G with $c_\mathcal{L}(P) = d_\mathcal{L}(P')$, just by mapping each edge of P'



(a) Three-dimensional grid graph G (edges are dashed and bidirectional) with a shortest s-t-path P (red) and a shortest feasible multi-label s-t-path Q (blue) with respect to label system $\mathcal{L}_{\min}$ and with the restriction that s and t are labeled with type via, i.e. $\phi(s) = \phi(t) = \text{via}$. While P contains five segments which are shorter than $l_{\min}$, potentially causing various same-net errors (minarea, minedge, via same-net-mindist), the path Q does not have any segment shorter than $l_{\min}$.



(b) The graph $G_{\mathcal{L}_{\min}}$ (edges are dashed and bidirectional) and the shortest (s, via)-(t, via)-path Q' corresponding to path Q from Figure (a) (blue). Each curved arc represents a label transition on path Q. Other label transition edges in $G_{\mathcal{L}_{\min}}$ which are not used by Q' are not drawn for the sake of clarity.

Figure 5.11

to a corresponding straight path in G which exists by construction of $G_{\mathcal{L}}$. For an example of a shortest feasible multi-label path with respect to $\mathcal{L}_{\min}$ in a graph G and its corresponding shortest path in $G_{\mathcal{L}_{\min}}$ see Figure 5.11.

For the shortest path search in $G_{\mathcal{L}}$ (step 2 in Algorithm 6) we can now use any existing graph-based shortest path algorithm. For the runtime analysis of this algorithm, note that the number of vertices and the number of edges increase at most by the factors $|L|$ and $|L|^2$, respectively, which are small in practice.

As mentioned earlier, we do store neither G nor $G_{\mathcal{L}}$ explicitly, but rather query if an edge is usable when we need it. See Section 5.4.3 for implementation details. Suppose we query if an edge $((v, l_1), (w, l_2))$ is usable, then it is very useful if the answer may depend on the involved label types l_1 and l_2. In this case label types can represent different wire types or wire colors, we just have to check if an edge is usable *when used with a certain wire type or color*. We illustrate this at the well-defined label system $\mathcal{L}_{\text{color}} := (L, t, d)$, where $L := \{\text{red}, \text{blue}\}$, $d \equiv 0$, and the label transition function t is given by the table in Figure 5.12. Here $s_{\min}$ can be

			x_-/x_+	y_-/y_+	z_-/z_+
red	$\to$	red	1	1	1
blue	$\to$	red	∞	$s_{\min}$	1
red	$\to$	blue	∞	$s_{\min}$	1
blue	$\to$	blue	1	1	1

Figure 5.12: Definition of label transition function for the label system $\mathcal{L}_{\text{color}}$ for layers with preferred dimension x. The value of $t(l_1, l_2, r, z)$ is given by the entry in row $l_1 \to l_2$ and column r. For layers with preferred dimension y the roles of x and y are interchanged.

used to control overlay rules for stitches at jogs, if desired. Now we say that edge $((v, l_1), (w, l_2))$ is usable if a wire running from v to w *and colored with* l_2 (not with the preferred color) does not introduce a diff-net-mindist violation.

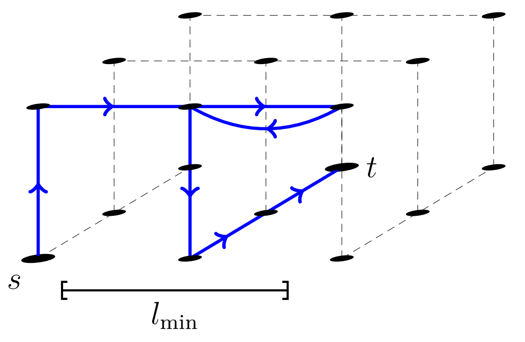
With this label system one can find shortest colored paths avoiding stitches in preferred dimension and allowing only safe stitches (controlled by $s_{\min}$) in non-preferred dimension. The label cost function can incorporate costs for such stitches. When using this label system, we may use a slightly different underlying graph G because we do not have to forbid jogs without preferred colors here since we do not use automatic coloring anyway. We do not go into more details here. We use label system $\mathcal{L}_{\text{color}}$ in cases where we did not find a path with automatic coloring, see Section 5.4.2.4.

We also have a similar label system where label types correspond to different wire types, and wire type changes are only allowed in certain directions and at some penalty cost. This is useful to avoid electrically bad configurations in wiring (for example a thin wire between two wide wires) and minedge errors caused by disadvantageous wire type changes.

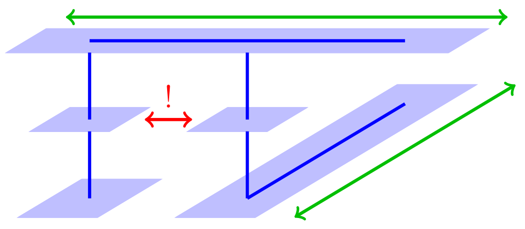
Next we describe how cycles in shortest feasible multi-label sequences are handled and how they can be avoided in advance.

Definition 5.22. *A* cycle *in a multi-label sequence* (P, ϕ) *in* G *is a cycle in* G *whose edges are all contained in straight paths of* P.

Note that the appearence of cycles very much depends on the involved label systems and cost functions. Cycles with length three or more occur very rarely in practical instances. On our testbed less than 0.4% of all multi-label path searches produce such cycles (using the 4STAGE implementation of the DRC-aware path search framework, see Section 5.4.4). We just remove such remaining cycles by deleting one or more of its edges afterwards, yielding a multi-label path which is feasible except for possible violations caused by these removals. These rare violations are not too problematic, and many of the corresponding same-net errors can be fixed by a post-processing, yielding paths which are almost clean in terms of those same-net errors respected by the label system.



(a) Shortest feasible multi-label sequence from s to t (blue) with respect to $\mathcal{L}_{\min}$ which contains a 2-cycle.



(b) Shapes corresponding to the multi-label sequence in the left figure. The 2-cycle does not lead to violated minedge rules (green lines), but it introduces a via same-net-mindist violation (red line).

Figure 5.13

Cycles of length two (2-cycles) appear more often, thus we treat them in a special way depending on the label system. If the label system does not model any distance rules, but rather minimum lengths for shapes (for example to satisfy minarea or minedge rules), then 2-cycles do not cause any trouble since they just provide additional shapes ensuring that minimum lengths are satisfied. In fact, 2-cycles are even desirable in such situations and allow to find legal solutions which shortest path approaches only controlling the lengths of *path segments* (e.g. Maßberg and Nieberg [2013]) do not find.

But if we have for example a label system forcing certain minimum distances between two vias on the path to compute, then 2-cycles may lead to violations of these distances. See Figure 5.13 for an example. One option would be to remove 2-cycles for such label systems and try to fix DRC-errors afterwards as it is done in the case of larger cycles. However, since the remaining errors are often hard to fix (if at all) we adapted our Dijkstra-based shortest path algorithm in the following way to avoid such 2-cycles. When labeling from a node v to an adjacent node w, we only allow this labeling operation if the currently shortest path to v does not arrive at v from the same direction as w. With this approach we avoid 2-cycles, but we may also forbid legal paths, thus the returned path may not be shortest possible anymore (or we may not even find a path if one exists). However, in practice this approach gives a good trade-off between length and the number of DRC-errors in the computed paths.

The prevention of 2-cycles can be also incorporated into the label system. However, this approach highly depends on the definition of the label system, hence we omit the details.

5.4.2.3 Multi-Label Interval-Based Path Search

In Section 5.4.2.2 we described a framework to compute shortest feasible multi-label sequences with existing graph-based shortest path algorithms (see Algorithm 6 on page 81). We now sketch how to integrate this framework into the *interval-based on-track path search* (simply denoted as path search in the following), the main shortest path algorithm in BonnRoute (Gester et al. [2013]) used for computing long connections. The path search uses a Dijkstra-based algorithm (Dijkstra [1959]) with two major speed-up features: a *future cost* to reduce the number of labeling steps, and merging vertices to so-called *intervals* to speed up sequences of labeling steps.

A *future cost* is a function $\pi : V(G) \to \mathbb{N}_0$ satisfying the following conditions:

$$\begin{aligned} c_\pi((v,w)) := c((v,w)) - \pi(v) + \pi(w) &\geq 0 \quad \text{for all } (v,w) \in E(G) \\ \pi(t) &= 0 \quad \text{for all } t \in T \end{aligned}$$

One easily observes that $\pi(v)$ is a lower bound on the distance (with respect to c) from v to T in G for each $v \in V(G)$. Let G' result from G by adding a vertex s and an edge (s,s') with $c_\pi((s,s')) := \pi(s')$ for each $s' \in S$. Then all shortest S-T-paths in G w.r.t. c are also shortest s-T-paths in G' w.r.t c_π and vice versa. However, computing them in G' with respect to c_π consumes substantially less label operations in practice (the better the lower bound π, the fewer label operations) since Dijkstra's algorithm operates *goal oriented* in this case, similarly to the A* heuristic proposed by Hart et al. [1968] which was applied to detailed routing by Rubin [1974]. Note that each future cost π for an instance (G,c,S,T) is also a valid (but potentially weak) future cost for $(G_\mathcal{L}, d_\mathcal{L}, S_\mathcal{L}, T_\mathcal{L})$. Therefore, we can use this future cost to compute a shortest path in step 2 of Algorithm 6.

BonnRoute uses two different future costs. The simpler one is given by

$$\pi_H((x,y,z)) := \mathrm{lb}_{\mathrm{wire}}(x,y) + \mathrm{lb}_{\mathrm{via}}(z)$$

where $\mathrm{lb}_{\mathrm{wire}}(x,y) := \min_{(x_t,y_t,z_t)\in T}(|x-x_t|+|y-y_t|)$ and $\mathrm{lb}_{\mathrm{via}}(z)$ is the minimum cost for a via connection from layer z to a layer containing a target location (see Hetzel [1998]). Let T_{rect} be a set of shapes covering exactly the vertices in T projected to one layer. Then $\mathrm{lb}_{\mathrm{wire}}(x,y)$ can be queried for each vertex (x,y,z) in time $O(\log |T_{\mathrm{rect}}|)$ by point location (Kirkpatrick [1983]) applied to the L_1 Voronoi diagram of the rectangles in T_{rect} by spending $O(|T_{\mathrm{rect}}| \log |T_{\mathrm{rect}}|)$ preprocessing time (Papadopoulou and Lee [2001], Gester [2009]).

The main advantage of π_H is its simplicity and fast computability since the runtime only depends on $|T_{\mathrm{rect}}|$ and not on the size of G. For runtime reasons, it here makes sense to check which of the two components to connect is representable by less rectangles and choose this as the target component. However, not considering the structure of G in π_H can lead to a big gap between $\pi_H(v)$ and the length of a shortest v-T-path.

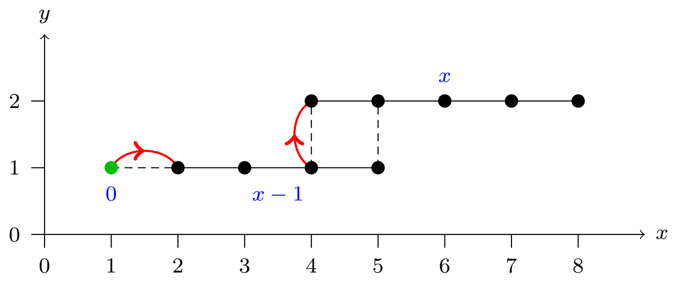


Figure 5.14: Small section of a path search instance, we assume unit costs in x- and y-dimension. Nine vertices can be labelled from the source (green) in only two label steps (red arcs) by merging vertices to intervals (solid lines) and storing label functions (blue) at intervals instead of label values at vertices.

To avoid this problem, Peyer et al. [2009] proposed a blockage-aware future cost π_P which computes shortest paths to T from all nodes in a supergraph G' of G that captures the structure of G quite well by ignoring small blockages and keeping larger blockages. In particular, G' is chosen such that $\pi_P \geq \pi_H$. This results in considerably fewer labeling steps in Dijkstra's algorithm, but the runtime for computing π_P is usually higher than for computing π_H. Therefore, we use π_P only if the global routing for the connection to route already contains a large detour, and π_H otherwise. The future cost used in the path search is denoted as π in the following.

The merging of congenerous and consecutive vertices to *intervals* was proposed in Hetzel [1998] for the special case of equidistant routing tracks which match in all layers with the same preferred dimension. The key idea is to group vertices that can be labeled at once to intervals and maintain *label functions* for these intervals. When the path search labels one vertex of an interval, then the label function of the whole interval is also updated at once, saving label operations on the other vertices of the interval later on. See Figure 5.14 for a small example. See Peyer et al. [2009] and Humpola [2009] for more details and further generalizations, and Gester et al. [2013] for a short description.

When computing a shortest path for our modified search instance $(G_{\mathcal{L}}, d_{\mathcal{L}}, S_{\mathcal{L}}, T_{\mathcal{L}})$ as given in Algorithm 6, we can use the same merging into intervals, but we may have to split intervals at label operations (such splits are not needed in the standard path search algorithm). We illustrate this in Figure 5.15.

In Nohn [2012] a new runtime bound for the path search applied to multi-label instances is given, including the additional split operations and the buildup of the modified search instance (step 1 in Algorithm 6 on page 81).

Theorem 5.23. *(*Nohn [2012]*) A shortest feasible multi-label sequence P with respect to a given label system $\mathcal{L} = (L, t, d)$ for the instance $(G,\ c,\ S,\ T)$ and future*

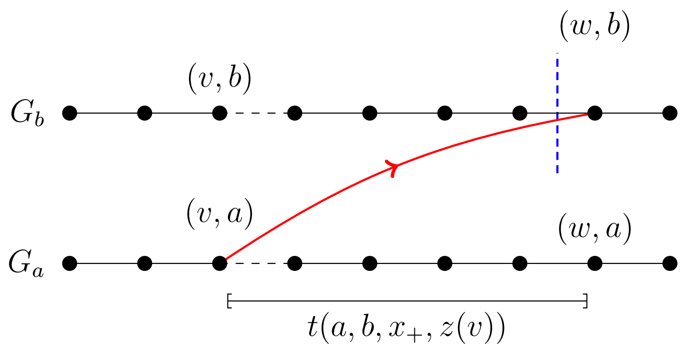


Figure 5.15: Example showing that intervals (solid lines) must be split for some label operations in graph $G_{\mathcal{L}}$. Labeling from (v, a) to (w, b) in direction x_+ (red arc) is allowed, but it is not allowed to any vertex left from (w, b). Therefore, the interval must be split at the blue dashed line to avoid implicit labels (given by the label function of the interval) in the left part.

cost $\pi : V(G) \to \mathbb{N}_0$ can be found in time

$$O(\min\{(\Lambda + 1)d_{\max} \cdot k^2 |\mathcal{I}|^2 \cdot \log(k |\mathcal{I}|),\ d_{\max} \cdot k^2 n \cdot \log(kn)\}),$$

where Λ is the cost of P w.r.t. c_π, $\mathcal{I}$ is the set of intervals representing the vertices of G, $d_{\max}$ is defined as in Section 5.4.2.2, $k = |L|$ and $n = |V(G)|$.

Note that k and $d_{\max}$ are very small constants in practice (typically both ≤ 5). Also note that an ordinary shortest path can be found in time

$$O(\min\{(\Lambda + 1) |\mathcal{I}| \cdot \log(|\mathcal{I}|),\ n \cdot \log(n)\})$$

for the same instance and notations as in the theorem above (see Hetzel [1998], Peyer et al. [2009], Humpola [2009] for details). We point out that the choice of design rules considered in the definition of the label system impacts its size k and hence offers a trade-off between accuracy (w.r.t. design rules) and runtime for multi-label paths.

In practical experiments multi-label path searches are also much slower than standard path searches, motivating that one should not always use multi-label path searches, but only when needed, which we explain in more detail in Section 5.4.2.4.

It is also an interesting question which speed-up the merging of vertices to intervals yields in the case of multi-labeling. Note that in this case the number of intervals contributes quadratically to the theoretical worst-case bound given in Theorem 5.23. Indeed, the speed-up gained by intervals is much lower for multi-labeling compared to the standard path search. For the most complex label system we use the speed-up is only 35%, while for the standard path search it is 65%. We did these experiments on a testbed similar to the one described in Section 5.4.4. Here the runs without merging vertices to intervals were also done with the interval-based path search where all intervals were split to singletons. Thus the actual benefit of intervals may be smaller since the path search could be implemented faster without supporting intervals, of course.

5.4.2.4 DRC-Aware Path Search Framework

A drawback of many shortest path algorithms used for detailed routing is that most same-net rules cannot be easily incorporated. See Figure 5.11a for an example where a shortest path may contain many same-net errors. In previous technology nodes (up to 22 nm with single patterning technology) BonnRoute computed long connections by simply calling a standard path search and then trying to fix same-net errors by a post-processing step. This approach gave excellent results in combination with an external DRC-fixing step as demonstrated in Gester et al. [2013].

However, in advanced technology nodes design rule dimensions do not scale well with feature miniaturization anymore (meaning that feature size decreases much more than for example minimum area values) which requires relatively more space for fixing same-net errors afterwards. Here a post-processing step often fails to make a path DRC-clean which motivates the need for a correct-by-construction path search mode.

In multiple patterning technologies, finding a connection with automatic coloring may fail, motivating the need for a path search mode which is able to choose arbitrary colors on its own.

However, the above desirable modes of the path search (using multi-labeling) are much slower than the standard path search combined with post-processing which is still sufficient in many situations. Therefore, we propose the framework sketched in Algorithm 7 incorporating same-net errors as well as non-trivial color choices (in addition to automatic coloring as described in Section 5.4.1). In the following we explain this framework which was developed in joint work with Markus Ahrens.

We denote different types of design rules such as minarea, minedge, or via same-net-mindist as *design rule types*. We denote the set of design rule types for which a path P has violations as $\mathrm{drt}(P)$, and we denote the set of design rule types a label system $\mathcal{L}$ respects as $\mathrm{drt}(\mathcal{L})$. We also need to combine different label systems to a new one which respects all rules and costs encoded in the original label systems.

Definition 5.24. *The* cross product *of two label systems* $\mathcal{L}_1 := (L_1, t_1, d_1)$ *and* $\mathcal{L}_2 := (L_2, t_2, d_2)$ *is the label system defined as* $\mathcal{L}_1 \times \mathcal{L}_2 := (L, t, d)$*, where*

$$\begin{aligned} L &:= L_1 \times L_2 \\ t((l_1, l_2), (l'_1, l'_2), z, r) &:= \max(t_1(l_1, l'_1, z, r), t_2(l_2, l'_2, z, r)) \\ d((l_1, l_2), (l'_1, l'_2), z, r) &:= d_1(l_1, l'_1, z, r) + d_2(l_2, l'_2, z, r) \end{aligned}$$

for all $(l_1, l_2), (l'_1, l'_2) \in L, z \in Z_{\text{wiring}}, r \in R$.

Corollary 5.25. *The cross product of two well-defined label systems is well-defined.*

Proof. For two well-defined label systems $\mathcal{L}_1 := (L_1, t_1, d_1)$ and $\mathcal{L}_2 := (L_2, t_2, d_2)$ the following properties hold for all $(l_1, l_2), (l'_1, l'_2) \in L_1 \times L_2, z \in Z_{\text{wiring}}, r \in R$:

(i) $\max(t_1(l_1, l_1, z, r), t_2(l_2, l_2, z, r)) \in \{1, \infty\}$

(ii) if $\max(t_1(l_1, l'_1, z, r), t_2(l_2, l'_2, z, r)) \neq \infty$,
then $\max(t_1(l'_1, l'_1, z, r), t_2(l'_2, l'_2, z, r)) = 1$

(iii) $d_1(l_1, l_1, z, r) + d_2(l_2, l_2, z, r) = 0.$

Therefore, $\mathcal{L}_1 \times \mathcal{L}_2$ is well-defined. □

As input for Algorithm 7 we use a set of well-defined label systems $\mathcal{M}$ which is ordered by the design rule types they respect. Formally, we define a partial order $\prec$ on $\mathcal{M}$ such that for $\mathcal{L}_1, \mathcal{L}_2 \in \mathcal{M}$ we have $\mathcal{L}_1 \prec \mathcal{L}_2$ if and only if $\text{drt}(\mathcal{L}_1) \subsetneq \text{drt}(\mathcal{L}_2)$, that means $\mathcal{L}_2$ respects all design rule types that $\mathcal{L}_1$ respects and at least one additional type. $\mathcal{M}$ always contains the *standard label system* which stands for a call of the standard path search without multi-labels, this label system is smaller than any other in $\mathcal{M}$ with respect to $\prec$.

The set $\mathcal{U}$ contains for each point in time the label systems which have not yet been used. Since it does not make sense to start a path search with the same label system twice, we always choose label systems from $\mathcal{U}$.

The path search is called in a loop, starting with as little restrictions as possible, the standard path search (see line 5 and line 11). Later the found path is post-processed (line 20), and if there are still DRC-errors left (for example due to missing space for local fixes), then we choose a label system in $\mathcal{U}$ which avoids as many remaining DRC-error types as possible and which is *least restrictive* with this property (that means smallest with respect to $\prec$), see line 24, and start a new path search using this label system. If P is DRC-clean, then we return P (line 26). We iterate this as long as there are DRC-errors that are fixable by any label system in $\mathcal{U}$ left. The set D collects all DRC-error types which have been left in post-processed paths over all previous iterations. The decisions when to change the path (line 22) and which label system to choose next (line 24) may be based on more complex criteria than just counting DRC-errors, of course. Also, in lines 23 and 25 one could take the path *before* post-processing into account instead of P. All these choices are only made heuristically and may be tuned in practice.

If we do not find any path at some point, we start a new search with $\mathsf{mode} = \mathsf{colored}$ which means that we now compute a shortest feasible multi-label sequence where we allow color deviations from the preferred track colors. This can be done by a modified label system which respects all design rules types in $\text{drt}(\mathcal{L})$ as well as all constraints and costs for possible color changes (stitches). This is exactly

Algorithm 7: DRC-aware path search framework.

Input : A search instance (G, c, S, T) and a partially ordered set of well-defined label systems $(\mathcal{M}, \prec)$ containing the standard label system as smallest element, as described.

Output: A path P in (G, c, S, T), preferably short and DRC-clean, or $\emptyset$ if no path exists.

```
1  $\mathcal{U} \leftarrow \mathcal{M}$
2  $P \leftarrow \emptyset$
3  $D \leftarrow \emptyset$
4  mode ← uncolored
5  $\mathcal{L} \leftarrow$ standard label system (the smallest in $\mathcal{U}$ w.r.t. $\prec$)
6  repeat
7      if mode = uncolored then
8          $\mathcal{L}' \leftarrow \mathcal{L}$
9      else
10         $\mathcal{L}' \leftarrow \mathcal{L} \times \mathcal{L}_{\text{color}}$
11     Compute a shortest feasible multi-label sequence $P'$ in $(G, c, S, T)$ w.r.t. $\mathcal{L}'$
12     if no such sequence found then
13         if mode = uncolored then
14             mode ← colored
15             goto line 7
16         else
17             return $P$
18     $\mathcal{U} \leftarrow \mathcal{U} \setminus \{\mathcal{L}\}$
19     Remove possible cycles in $P'$
20     Post-process $P'$
21     if $P'$ has fewer DRC-errors than $P$ or $P = \emptyset$ then
22         $P \leftarrow P'$
23         $D \leftarrow D \cup \mathrm{drt}(P)$
24     $\mathcal{L} \leftarrow \mathcal{L}'' \in \mathcal{U}$ with $|\mathrm{drt}(\mathcal{L}'') \cap D|$ maximal, and smallest w.r.t. $\prec$
25     if $\mathrm{drt}(\mathcal{L}) \cap \mathrm{drt}(P) = \emptyset$ then
26         return $P$
27 until false
```

what the label system $\mathcal{L} \times \mathcal{L}_{\text{color}}$ does which is set in line 10. Here $\mathcal{L}_{\text{color}}$ is a label system encoding allowed color changes and stitch costs, if needed, as described in Section 5.4.2.2. Note that by Corollary 5.25 $\mathcal{L} \times \mathcal{L}_{\text{color}}$ is well-defined, hence we can compute a shortest feasible multi-label sequence w.r.t. $\mathcal{L} \times \mathcal{L}_{\text{color}}$ efficiently. To keep packing density for later connections as high as possible, we include penalty costs for deviations from the preferred colors. These penalty costs are paid whenever we enter an interval with non-preferred color in the path search.

If in this mode still no path can be found, then we take the path from the last round, if there is any. Otherwise we know that there exists no path, even when using the least restrictive standard path search and allowing color deviations. In such a case we start the whole framework again, this time allowing rip-up of other nets and/or allowing wires to leave the restricted routing corridor by some specified margin. We may also start a rip-up loop if we already found a path which is bad with respect to DRC-errors, netlength, or other metrics, depending on the *criticality* of the path. We do not go into more details here.

How to choose the label systems in $\mathcal{M}$ very much depends on the design rules and the post-processing routines used in the framework. Defining all label systems used in BonnRoute in detail is out of the scope of this section. The most important DRC-error types avoided by label systems in BonnRoute are minarea, minedge, and via same-net-mindist errors. Here via same-net-mindist errors are of special interest for via layers where on neighboring wiring layers no jogs are allowed. Then the standard path search often finds paths where jogs are *simulated* by so-called *via bridges* (see Figure 5.13b on page 85 for an example) involving via same-net-mindist errors which are hard to fix afterwards. By multi-labeling, we are prepared to avoid these errors and are thus able to produce high-quality routings even if no jogs are allowed on certain layers, for example due to manufacturing reasons.

We also incorporate threshold values for *line-end* depending mindist rules, that means we avoid short edges forcing bigger spacings to other shapes. Moreover, we use a label system managing wire type changes within a path, avoiding electrically bad configurations and minedge errors caused by a wire type change. This label system is similar to $\mathcal{L}_{\text{color}}$ and can also be used in combination with other DRC-aware label systems by using the cross product.

It is often not the best choice to let $\mathcal{M}$ only contain the standard label system and one label system avoiding *all* above error types. In many cases only few error types are left in the post-processed path, and then the label system avoiding all errors might be too restrictive to find a path at all. See also Section 5.4.4 for experimental results confirming this. Also a less restrictive label system speeds up the multi-label path search call substantially.

We note that even with multi-labeling we do not model complex design rules exactly, but rather conservatively. For example, a label system avoiding minarea errors typically forces each single pref wire or jog to be long enough to cover the minimum required area, instead of tracking the area of a connected component under construction which would lead to a much more complex label system.

In Section 5.4.4 we present results for different variants of the DRC-aware path search framework described in this section.

5.4.3 Implementation Details

We now give some details on the implementation of the main data structures involved in computing long connections. One major data structure in BonnRoute is the *detailed grid*. It stores all shapes on the chip which are relevant for routing (wires, pins, and blockages) and allows layerwise rectangular intersection queries for these shapes. More exactly, it is able to report for any given shape S all currently present shapes on the chip intersecting S. The detailed grid is updated immediately when a new connection is routed. It is implemented similar to the *shape tree* as proposed by Schulte [2012]. The basic idea is to utilize that most shapes on a routing layer have only small extension in non-preferred routing dimension. Each layer is partitioned into stripes in preferred routing dimension, each stripe covering exactly one track and storing all shapes intersecting the stripe. Typically most shapes only intersect one stripe in practice. Schulte [2012] proposed a balanced search tree as the data structure managing the shapes within a stripe, and showed how rectangular intersection queries can be answered efficiently in practice. Recently, we achieved a considerable speed-up and memory saving by exchanging the balanced binary search tree inside of each stripe by an array.

To check if a new (colored) wire w to be routed is legal with respect to other present shapes, first an *influence region* R_w is computed, that is a (preferably small) three-dimenional area around the wire such that it is guaranteed that shapes not intersecting R_w do not influence the legality of w. Then all shapes intersecting R_w are collected by layerwise queries to the detailed grid, resulting in a shape set $\mathcal{S}$. Finally, the colored checking oracle reports if $\psi_c(\mathcal{S} \cup \{\text{shape}(w)\}, \alpha) = 1$, where α is the coloring for $\mathcal{S}$ and shape(w). The implementation of the checking oracle is completely seperated from the detailed grid, it operates only on shapes (which were usually collected using the detailed grid before). We omit implementation details of the checking oracle here.

The procedure to first collect shapes in an influence region and then check these shapes against the wire w can be speeded up drastically for the most common queries as Müller [2009] proposed. He developed the *fast grid* structure which stores for each track which positions are currently legal and which are not for the mostly used wire types. This information is stored as intervals in balanced binary search trees (one per track), including some special speed-up techniques, and can be queried very efficiently. The fast grid is updated continuously in synchronisation with the detailed grid. Hence the fast grid can be viewn as a cache for the most common and probable checking queries.

For rip-up and reroute, path search intervals store costs corresponding to wires which need to be ripped out when using the interval (see Hetzel [1998] for details). The costs shall reflect the criticality of connections to be ripped-up. For example, a connection where signals are allowed to run only a small detour to guarantee a certain *cycle time* on the chip is less likely to be ripped-up than a connection where a larger detour is allowed. There are many other criterions for the criticality in practice, we leave out the details here.

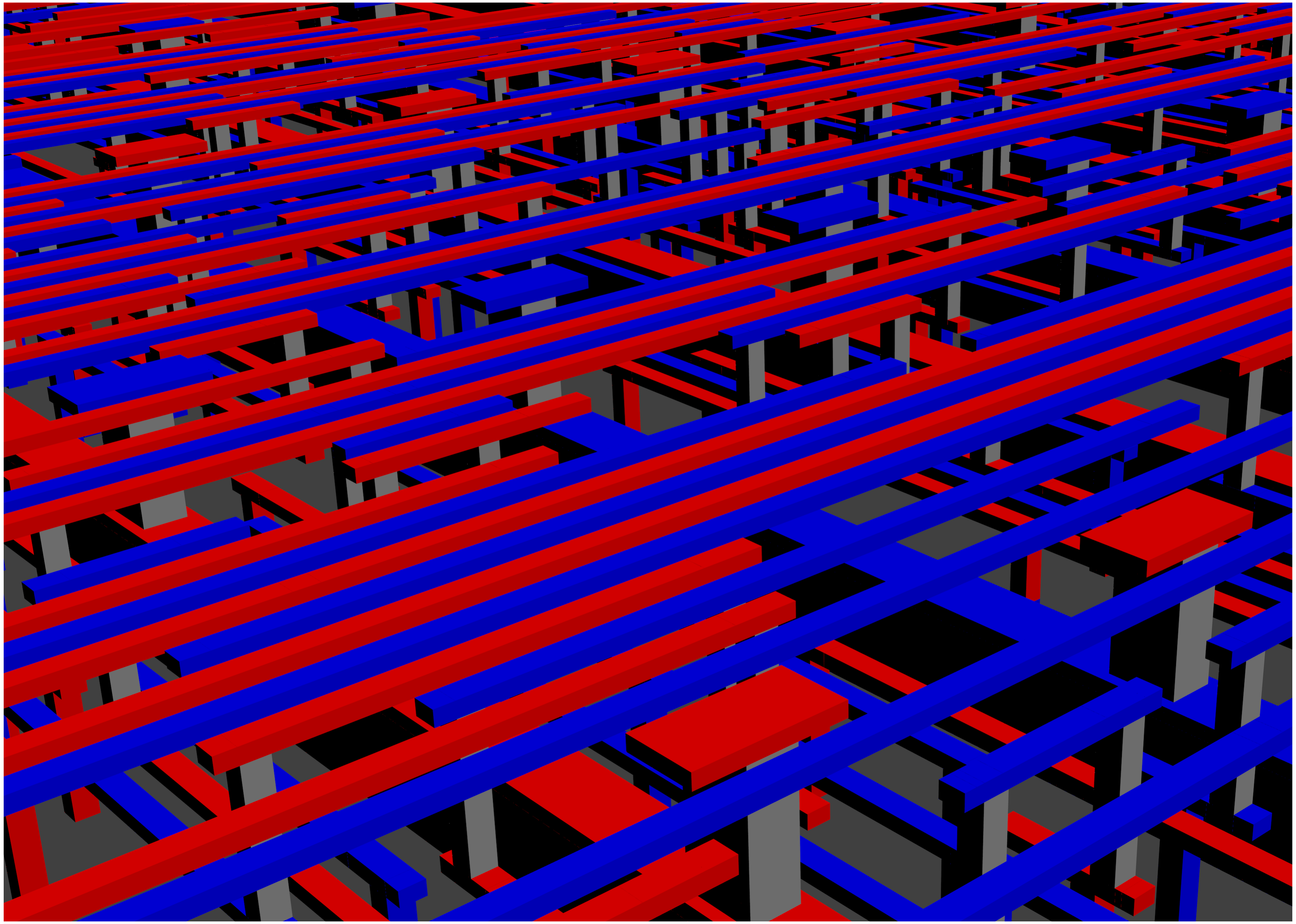


Figure 5.16: Section of a multiple patterning routing computed by BonnRoute on a real-world 14 nm design. All x- and y-coordinates are true to scale, z-extensions were adapted for the sake of clarity. Gray shapes are vias on a single patterning via layer.

5.4.4 Experimental Results

In this section we compare versions of BonnRoute with different variants for computing long connections. Furthermore, we present results comparing an industry standard router (ISR) with a combined BonnRoute flow (BR+ISR) and demonstrate that BR+ISR produces far superior routings in less runtime on real-world multiple patterning designs. We first compile the criteria which we use for evaluating routing quality.

- The *wiring length* is the sum over all lengths of plane wire sticks on the whole chip. Wiring length is one of the core routing metrics since it determines power consumption and signal delay to a great extent.

- The *number of vias* is important since vias have a high failing probability in manufacturing and a high resistance which impacts signal delays.

- Nets connected with a large detour are problematic because signals may take longer than the *cycle time* of the chip allows. We call a net *scenic* with respect to a percentage x if the sum over all plane wire sticks of the net is at least 25 µm and if this length is at least x% over the length of a minimum two-dimensional Steiner tree connecting all components of the net projected to one layer (ignoring blockages). Our results show the number of scenics w.r.t. 25%, 50%, and 100%.

- The number of *DRC-errors* is critical for the producibility of a chip. Our results show DRC-errors as reported from an external industry standard checking tool which uses *conservative* design rules which are manageable for routing. These rules are also used by BonnRoute and ISR. We also show detailed numbers for some selected error classes which are *shorts* (overlaps of connections of two different nets), *stitch errors* (overlaps of shapes colored differently), *diff-net mindist*, *same-net mindist*, *minedge*, and *minarea* errors (see Chapter 3). We note that on our testbed stitches are not allowed at all.

- An *open* is a missing connection between two components which are to be connected. Clearly it is essential for a properly working chip that there are no remaining opens.

- Last but not least, *runtime* is extremely important since routing is a major step in the overall physical design flow of a chip, and it is iterated together with other design steps. Our results also include the *memory consumption*.

The sum of all opens and DRC-errors is a very rough measure for the *quality of results* (in short *QOR*) in terms of logical and manufacturing errors which we also list in our result tables.

Chip	Nets	Image Size (mm × mm)	Wires (m)	Vias	DRC Errors	Opens
A	191	0.25 × 0.25	0.0003	106	6	228
B	2,212	0.08 × 0.08	0.001	1,163	1	2,630
C	2,432	0.03 × 0.05	0.0002	281	0	4,794
D	3,065	0.03 × 0.09	0.0003	282	0	6,431
E	3,241	0.04 × 0.08	0.0004	466	0	6,109
F	3,977	0.08 × 0.07	0.001	1,881	0	5,979
G	4,470	0.10 × 0.10	0.0003	343	0	8,600
H	5,950	0.10 × 0.06	0.002	3,168	186	10,490
I	10,801	0.05 × 0.27	0	0	0	21,951
J	10,986	0.15 × 0.08	0.003	3,874	0	20,168
K	12,798	0.17 × 0.09	0.003	4,554	0	22,744
L	13,472	0.35 × 0.22	0.003	3,233	0	23,559
M	14,715	0.10 × 0.14	0.002	2,005	1,552	27,818
N	16,423	0.10 × 0.16	0.01	8,996	0	26,204
O	17,049	0.27 × 0.21	0.01	7,561	0	29,844
P	37,360	0.23 × 0.30	0.01	12,857	0	63,360
Q	42,542	0.39 × 0.11	0.02	3,853	2	73,092
R	42,637	0.36 × 0.10	0.02	3,859	795	73,242
S	50,133	0.14 × 0.28	0.02	22,455	0	90,819
T	50,792	0.30 × 0.21	0.02	3,435	2	83,577
U	82,748	0.25 × 0.25	0.01	7,391	0	159,460
V	102,995	0.33 × 0.32	0.03	36,193	1,029	183,580
W	107,475	0.30 × 0.30	0.02	25,947	0	188,705
X	190,550	0.46 × 0.46	0.06	83,328	0	339,490
Y	338,092	0.53 × 0.65	0.14	187,254	31,205	634,375
Z	516,197	0.76 × 0.75	0.12	152,659	104	971,613
$\sum$	1,683,303		0.51	577,144	34,882	3,078,862

Table 5.1: Testbed consisting of 26 real-world 14 nm designs.

All following results were produced on a machine with 512 GB main memory and two Intel® Xeon® E5-2687W v3 CPUs, each having ten cores running at 3.10 GHz. Both tools, BonnRoute and ISR, were run using 20 threads. Our testbed consists of 26 real-world 14 nm designs, see Table 5.1. The instance size ranges from less than two hundred nets up to more than half a million nets. Most designs already contain some *prerouted* special nets, and some of these even contain scenics or severe DRC-errors such as shorts. It is typical in practice that a routing tool is also used for evaluating such *unclean* designs with respect to important metrics. The opens in Table 5.1 correspond to connections which have to be routed with individual, prescribed wire widths, via sizes, and routing layers based on timing requirements. This is the task of the routing tools we compare.

For the different versions for computing long connections we use the following basic label systems (sorted according to the partial order $\prec$ defined in Section 5.4.2.4):

- **STD:** A label system which just calls the standard path search without multi-labels.
- **LS1:** A label system which respects via same-net-mindist rules.
- **LS2:** A label system which respects the most important same-net rules (minarea, minedge, via same-net-mindist rules) and avoids short edges that force bigger same- and diff-net-distances (*line-end rules*) by choosing segment lengths large enough.
- **LS3:** Same as LS2, but using more conservative access area objects which make DRC-errors at the end segments of a connection less likely.

We compare the following versions for computing long connections (see Algorithm 7 on page 91):

- **NOML:** The DRC-aware path search framework is not run at all, instead the standard path search is run, followed by a post-processing step. This variant was default in BonnRoute up to 22 nm single patterning technology. Same-net rules are not respected at all in the standard path search.
- **1STAGE:** The DRC-aware path search framework is used with $\mathcal{M}$ containing only the label system LS3, i.e. here no standard label system is used. The most important same-net rules are respected correct-by-construction in this mode.
- **2STAGE** The DRC-aware path search framework is used with $\mathcal{M}$ containing the label systems STD and LS3.
- **4STAGE:** The DRC-aware path search framework is used with $\mathcal{M}$ containing the label systems STD, LS1, LS2, and LS3.

The versions 2STAGE and 4STAGE can be viewed as intermediate steps between the (nearly) correct-by-construction version 1STAGE and the NOML version not taking same-net errors into account at all in the path search. The motivation for these intermediate steps is that in most cases same-net errors can be fixed in the post-processing step, but in some cases sophisticated label systems are needed.

Note that 1STAGE, 2STAGE, and 4STAGE use the DRC-aware path search framework and are thus able to color paths non-trivially. This is of special importance if input pins or wires which are to be connected do not have the preferred color since such pins cannot be legally connected with the standard path search using automatic coloring.

Run	Time (hh:mm:ss)	Mem. (MB)	Wires (m)	Vias	Scenics 25%	QOR	DRC Errors	Opens
NOML	1:41:12	124,955	23.11	12,836,257	4,054	283,887	281,657	2,230
1STAGE	21:09:06	455,649	24.07	13,051,831	16,688	101,435	62,721	38,714
2STAGE	2:11:37	157,971	23.20	12,864,905	4,543	98,921	96,904	2,017
4STAGE	2:07:07	149,574	23.17	12,852,558	4,316	90,078	88,084	1,994

Table 5.2: Comparison of BonnRoute run with NOML, 1STAGE, 2STAGE, and 4STAGE.

We now compare results on our testbed produced by running BonnRoute with the four different versions for computing long connections. See Table 5.2 for a comparison table showing for each of the four versions one line containing results summed up over all 26 testcases. Detailed results for single testcases including selected DRC-error types can be found in Tables 5.5 to 5.8 on pages 103 to 106.

The run with NOML is fastest and consumes least memory, but it also has by far the most DRC-errors. The run with 1STAGE has fewest DRC-errors, but the runtime is unacceptable in practice. Also this run leaves by far the most opens which has one global and one local reason. The global reason is that always using the most conservative multi-label system wastes much routing space. This can also be seen at the wiring length and via and scenic numbers which are all considerably higher than in the other runs, although much fewer nets were connected. The local reason is the restricted flexibility for accessing pins or wires which may lead to situations where no legal access segment can be routed. The overall memory consumption is also drastically higher compared to the other runs. Alltogether, the results show that multi-labeling should be used only selectively. The runs using 2STAGE and 4STAGE follow this selective approach and provide a reasonable tradeoff between runtime and routing quality. The 4STAGE run is preferable since it has substantially fewer DRC-errors and is also slightly better in all other metrics. See Table 5.3 for statistics on the usage of all label systems in the 4STAGE run. Multi-label path searches consume only 40% of the total path search runtime. We note that among 9,141,233 shapes to be colored on the whole testbed 31,474 (0.34%) were colored with non-preferred color.

At first sight, it might be surprising that the run with 2STAGE has a higher DRC-error count than the run with 4STAGE. However, this is easily explainable by the way the DRC-aware path search framework works. We illustrate this at an example: Suppose we compute a path with the label system STD which contains DRC-errors after post-processing, and we cannot find a path using the label system LS3 for the same connection. Then in the case of 2STAGE this means that the best path found is the post-processed path containing DRC-errors, while in the case of 4STAGE we may find a path *without* any DRC-errors using the label system LS1 which is less

Label System	Time (hh:mm:ss)		Calls		Best Path	
STD	9:05:34	(59.74%)	2,625,145	(93.67%)	2,452,514	(93.87%)
LS1	40:07	(4.39%)	70,448	(2.51%)	65,213	(2.50%)
LS2	3:00:14	(19.74%)	41,810	(1.49%)	34,070	(1.30%)
LS3	5:52	(0.64%)	2,492	(0.09%)	1,666	(0.06%)
Col. STD	1:18:37	(8.61%)	57,516	(2.05%)	55,105	(2.11%)
Col. LS1	19:33	(2.14%)	2,505	(0.09%)	2,070	(0.08%)
Col. LS2	41:47	(4.58%)	2,472	(0.09%)	2,027	(0.08%)
Col. LS3	1:31	(0.17%)	276	(0.01%)	64	(0.00%)
$\sum$	15:13:15	(100.00%)	2,802,664	(100.00%)	2,612,729	(100.00%)

Table 5.3: Statistics for all (colored) label systems used in BonnRoute run with 4STAGE (see Table 5.8), summed up over all 26 testcases. In the second and third column the total runtime and the number of calls for a label system are given, respectively. In the fourth column the number of calls for which the found path was chosen as best path in the DRC-aware path search framework is given.

restrictive than LS3. This is actually one of the main ideas behind the framework, to be only as restrictive as necessary to avoid DRC-errors. Being conservative for all connections is bad (1STAGE), as is being optimistic for all connections (NOML).

We think that the best version 4STAGE (which is currently default for BonnRoute) can be further enhanced by adding additional intermediate label systems. As mentioned in Section 5.4.2.4, we also use a label system managing wire type changes which could be worth to be integrated into our default framework.

We now compare BonnRoute using 4STAGE with an industry standard router (ISR). The focus of BonnRoute is near optimum packing of wires with respect to wiring length, detours, power, timing, and manufacturing yield. Since the design rule complexity has continuously increased with each new technology, we aim at avoiding only the most important types of DRC-errors in BonnRoute. Therefore, we do not compare ISR to BonnRoute standalone, but to a combined BR+ISR tool where ISR is used as external post-processing step after BonnRoute to resolve remaining DRC-errors locally. For our experiments both tools, BR+ISR and ISR, were driven in default modes as used by designers at our industry partner IBM.

Table 5.4 shows results for BR+ISR and ISR run on selected chips of our testbed. Here we excluded the smallest chips and unclean chips containing many DRC-errors in the input (see Table 5.1). The results demonstrate that BR+ISR is far superior in every aspect. It runs more than twice as fast, consumes less memory, and considerably improves on all important routing metrics measured. The number of vias is reduced by more than 20% and the wiring length by more than 10%, positively affecting timing, power consumption, and manufacturing yield. BR+ISR produces significantly less scenics which also improves timing and shows that global routing and detailed routing are perfectly in tune with each other. Last but not least, the number of DRC-errors reduces by more than 60%, avoiding manual fixing and risk of manufacturing failures.

Interestingly, the main runtime of BR+ISR is consumed by the ISR step where only local DRC-error fixing is done. Hence we are confident to further improve the results of BR+ISR by reducing the need for fixing errors after BonnRoute by means of multi-labeling.

Alltogether, the results confirm that BR+ISR is able to route real-world multiple patterning designs *without stitches* almost DRC-clean, achieving high routing quality. We point out that the lead of BR+ISR over ISR in terms of runtime and routing quality has become even larger since our last similar comparison for single patterning technology (Gester et al. [2013]).

We finally note that BR+ISR is the default tool for *signal routing* at IBM, used for designing *ASIC chips* as well as *server chips* for several chip technologies.

Chip	Tool	Time (hh:mm:ss)		Mem.	Wires	Vias	Scenic Nets			QOR	DRC	Opens	Shorts	Stitch	Diff	Same	Min	Min
		BR	Total	(MB)	(m)		25%	50%	100%		Errors			Errors	Mindist	Mindist	Edge	Area
I	ISR		7:16	5,081	0.24	139,259	1,075	470	6	8	8	0	0	0	5	1	0	0
	BR+ISR	1:50	3:52	3,656	0.20	99,351	168	53	5	4	4	0	0	0	4	0	0	0
J	ISR		4:31	4,009	0.16	114,377	980	512	8	4	4	0	0	0	3	1	0	0
	BR+ISR	50	2:43	3,815	0.14	82,768	24	1	0	3	3	0	0	0	1	2	0	0
K	ISR		4:29	4,500	0.13	118,910	356	135	2	5	5	0	0	0	4	1	0	0
	BR+ISR	1:45	3:42	4,494	0.12	84,809	8	0	0	0	0	0	0	0	0	0	0	0
L	ISR		6:09	17,250	0.20	111,699	893	457	34	9	9	0	0	0	2	3	0	0
	BR+ISR	1:48	4:48	17,241	0.18	87,294	61	0	0	1	1	0	0	0	1	0	0	0
N	ISR		7:12	4,695	0.24	147,425	1,364	766	110	8	8	0	0	0	7	1	0	0
	BR+ISR	1:24	3:17	4,627	0.20	109,782	17	4	2	0	0	0	0	0	0	0	0	0
O	ISR		6:18	8,447	0.18	152,361	126	25	0	16	13	3	0	0	13	0	0	0
	BR+ISR	1:39	4:36	8,360	0.18	115,160	28	1	0	2	0	2	0	0	0	0	0	0
P	ISR		28:39	43,734	0.59	332,117	1,813	893	21	54	52	2	3	0	37	1	0	5
	BR+ISR	2:31	19:40	26,131	0.53	253,199	44	3	0	100	100	0	0	0	57	3	14	4
Q	ISR		25:49	11,442	0.79	481,519	5,476	3,265	699	12	12	0	0	0	9	1	0	0
	BR+ISR	4:16	9:32	11,096	0.61	332,020	415	106	7	13	13	0	0	0	1	0	0	0
S	ISR		19:50	11,561	0.70	515,523	2,309	972	38	11	11	0	0	0	4	7	0	0
	BR+ISR	3:26	9:10	11,135	0.62	389,290	81	3	1	10	10	0	0	0	0	9	0	0
T	ISR		14:35	16,012	0.52	456,011	1,348	438	7	7	7	0	0	0	4	1	0	0
	BR+ISR	2:29	8:04	16,022	0.47	324,304	149	4	0	4	4	0	0	0	2	0	0	0
U	ISR		41:20	17,447	1.15	799,298	3,600	1,335	49	72	72	0	0	0	59	6	0	4
	BR+ISR	8:10	18:58	17,361	1.04	598,830	46	0	0	18	18	0	0	0	16	0	0	1
W	ISR		39:30	24,532	1.45	874,683	6,679	3,910	927	50	45	5	0	0	29	4	0	2
	BR+ISR	4:18	13:00	24,437	1.20	714,891	522	122	10	6	6	0	0	0	4	2	0	0
X	ISR		1:58:44	51,112	3.91	1,763,759	12,768	6,659	786	45	45	0	0	0	30	11	0	1
	BR+ISR	11:17	29:40	51,546	3.29	1,518,763	251	6	1	15	15	0	0	0	5	10	0	0
Z	ISR		3:53:52	139,445	6.31	5,153,140	5,632	2,004	120	269	269	0	105	0	96	33	0	3
	BR+ISR	31:52	1:44:30	139,648	6.08	3,930,406	875	52	5	46	46	0	0	0	27	6	0	3
$\sum$			9:18:14	359,267	16.57	11,160,081	44,419	21,841	2,807	570	560	10	108	0	302	71	0	15
		1:17:35	3:55:32	339,569	14.86	8,640,867	2,689	355	31	222	220	2	0	0	118	32	14	8
			-57.81%	**-5.5%**	**-10.3%**	**-22.6%**	**-93.9%**	**-98.4%**	**-98.9%**	**-61.1%**	**-60.71%**	**-80%**	$-\infty$	**0%**	**-60.9%**	**-54.9%**	$+\infty$	**-46.7%**

Table 5.4: Comparison of ISR and BR+ISR on selected chips of our testbed.

Chip	Time	Mem.	Wires	Vias	Scenic Nets			QOR	DRC	Opens	Shorts	Stitch	Diff	Same	Min	Min
	(h:mm:ss)	(MB)	(m)		25%	50%	100%		Errors			Errors	Mindist	Mindist	Edge	Area
A	33	2,362	0.01	1,374	0	0	0	29	28	1	0	0	13	5	4	0
B	1:37	2,098	0.04	12,819	20	5	2	1,045	993	52	0	42	141	172	48	6
C	19	1,512	0.02	16,089	3	1	0	219	213	6	0	0	21	120	15	6
D	17	1,177	0.03	22,339	6	0	0	346	344	2	0	0	90	153	18	4
E	21	1,485	0.03	21,546	1	0	0	554	553	1	0	0	169	115	23	8
F	25	1,394	0.04	22,879	0	0	0	798	730	68	0	4	65	145	230	5
G	23	1,527	0.03	28,459	6	0	0	504	456	48	0	0	37	216	59	13
H	1:19	2,010	0.07	41,706	16	2	0	1,315	1,162	153	0	27	610	323	41	9
I	1:04	1,964	0.20	97,748	132	38	1	4,062	4,055	7	0	12	1,089	772	1,566	54
J	44	2,349	0.14	81,774	28	1	0	1,384	1,372	12	0	0	247	426	281	23
K	1:03	2,264	0.12	84,281	8	0	0	1,075	1,061	14	0	5	113	453	176	14
L	1:27	3,419	0.18	86,753	61	0	0	1,409	1,304	105	0	6	474	621	24	23
M	1:25	2,474	0.17	111,605	61	10	0	5,847	5,847	0	0	20	545	996	1,024	40
N	55	2,263	0.20	109,149	16	3	0	1,745	1,743	2	0	35	448	661	106	34
O	1:28	3,600	0.16	112,783	27	1	0	1,784	1,470	314	0	20	134	604	309	18
P	1:57	4,693	0.50	247,818	34	0	0	3,511	3,430	81	0	39	699	1,519	267	72
Q	2:19	3,833	0.61	328,683	398	96	4	5,733	5,723	10	0	464	1,087	1,845	802	55
R	2:53	3,675	0.55	327,368	235	39	1	7,635	7,631	4	0	410	1,622	1,808	1,332	88
S	2:46	6,282	0.61	386,427	69	1	0	6,179	6,145	34	0	0	907	1,970	1,314	117
T	2:18	4,122	0.47	322,199	146	4	0	4,499	4,474	25	0	803	597	1,796	436	75
U	5:57	6,648	1.04	594,130	40	0	0	8,846	8,597	249	0	19	824	4,202	766	232
V	5:21	6,535	1.28	762,918	132	3	0	14,391	14,344	47	0	56	2,849	3,818	3,744	125
W	3:44	5,890	1.20	711,843	515	115	2	8,103	8,054	49	0	63	708	4,812	370	171
X	8:45	11,296	3.28	1,510,787	236	1	0	23,114	23,063	51	0	134	4,101	7,244	5,132	306
Y	22:29	18,426	6.08	2,886,576	1,242	18	0	92,277	92,025	252	2,293	4,405	27,473	18,683	13,239	1,658
Z	29:23	21,657	6.05	3,906,204	622	36	3	87,483	86,840	643	104	94	8,855	21,202	35,436	881
$\sum$	1:41:12	124,955	23.11	12,836,257	4,054	374	13	283,887	281,657	2,230	2,397	6,658	53,918	74,681	66,762	4,037

Table 5.5: BonnRoute run using NOML.

Chip	Time	Mem.	Wires	Vias	Scenic Nets			QOR	DRC	Opens	Shorts	Stitch	Diff	Same	Min	Min
	(hh:mm:ss)	(MB)	(m)		25%	50%	100%		Errors			Errors	Mindist	Mindist	Edge	Area
A	38	3,215	0.01	1,413	0	0	0	11	9	2	0	0	0	2	0	0
B	2:53	7,111	0.04	12,201	17	0	0	674	513	161	0	8	12	14	9	1
C	36:52	5,930	0.02	16,722	19	2	0	78	39	39	0	0	7	7	0	3
D	1:37	3,163	0.03	23,039	28	1	0	82	50	32	0	0	4	6	0	1
E	2:07	5,392	0.04	22,104	20	1	0	260	210	50	0	0	13	9	5	4
F	2:19	4,453	0.04	23,414	7	0	0	554	468	86	0	0	27	5	142	2
G	4:15	6,884	0.03	29,674	28	1	0	197	127	70	0	11	11	20	31	4
H	2:45	5,492	0.07	42,487	66	4	1	587	340	247	0	0	204	18	24	8
I	16:01	12,800	0.21	98,441	393	60	0	1,448	1,098	350	0	0	34	37	831	22
J	8:31	9,742	0.14	83,389	93	4	0	307	186	121	0	6	12	25	14	11
K	5:10	12,138	0.12	86,431	89	8	0	234	131	103	0	3	16	12	6	5
L	35:32	21,225	0.18	88,427	103	2	0	380	157	223	0	3	36	13	18	8
M	9:49	9,203	0.18	116,491	316	25	1	4,078	3,938	140	0	5	582	264	527	24
N	5:04	8,334	0.21	111,874	59	4	0	390	256	134	0	0	29	20	32	12
O	8:17	14,651	0.17	113,981	83	2	0	584	136	448	0	8	9	42	7	9
P	35:34	18,182	0.52	256,890	215	21	1	875	428	447	0	5	22	90	10	29
Q	22:46	18,598	0.64	338,838	988	420	13	2,851	2,518	333	0	0	310	134	487	34
R	22:51	15,079	0.58	338,108	901	357	5	4,019	3,597	422	0	8	319	177	709	36
S	23:31	19,005	0.64	395,393	387	33	0	1,548	919	629	0	10	38	164	21	68
T	10:40	15,210	0.50	331,811	685	464	11	2,337	2,011	326	0	0	221	119	447	34
U	1:07:13	27,919	1.08	609,122	517	19	0	4,464	1,613	2,851	0	18	110	343	29	160
V	23:46	15,776	1.35	782,744	657	27	1	3,098	2,442	656	0	10	223	148	364	68
W	17:17	20,722	1.25	736,179	895	146	6	1,620	1,119	501	0	4	83	279	10	72
X	46:19	26,816	3.40	1,544,291	760	23	1	3,871	2,816	1,055	0	27	345	442	138	166
Y	9:27:02	85,922	6.22	2,866,708	4,007	239	2	49,858	29,769	20,089	2,292	71	10,627	1,891	2,351	826
Z	4:50:17	62,687	6.40	3,981,659	5,355	653	22	17,030	7,831	9,199	104	89	601	1,331	338	623
$\sum$	21:09:06	455,649	24.07	13,051,831	16,688	2,516	64	101,435	62,721	38,714	2,396	286	13,895	5,612	6,550	2,230

Table 5.6: BonnRoute run using 1STAGE.

Chip	Time	Mem.	Wires	Vias	Scenic Nets			QOR	DRC	Opens	Shorts	Stitch	Diff	Same	Min	Min
	(h:mm:ss)	(MB)	(m)		25%	50%	100%		Errors			Errors	Mindist	Mindist	Edge	Area
A	33	2,389	0.01	1,383	0	0	0	11	11	0	0	0	4	1	0	0
B	3:11	5,162	0.04	12,770	18	4	1	813	759	54	0	63	50	92	27	3
C	25	1,872	0.02	16,096	5	1	0	46	40	6	0	0	6	11	0	4
D	22	1,360	0.03	22,313	7	0	0	69	67	2	0	10	0	10	6	1
E	26	1,685	0.03	21,514	6	0	0	250	249	1	0	0	16	13	22	3
F	26	1,669	0.04	22,973	0	0	0	474	407	67	0	0	14	12	117	3
G	28	1,868	0.03	28,431	8	0	0	127	79	48	0	0	15	18	9	3
H	2:51	3,118	0.07	41,837	23	4	0	542	388	154	0	20	220	37	18	4
I	2:03	3,477	0.20	98,075	141	35	0	1,515	1,508	7	0	0	291	87	914	35
J	51	3,034	0.14	81,898	27	0	0	291	281	10	0	6	18	37	46	7
K	1:46	2,769	0.12	84,517	9	1	0	241	230	11	0	0	11	24	30	2
L	1:42	4,038	0.18	87,178	63	0	0	221	121	100	0	3	19	18	2	3
M	2:21	3,963	0.17	112,529	88	15	0	3,507	3,507	0	0	20	308	171	473	11
N	1:08	2,759	0.20	109,112	21	1	0	258	257	1	0	2	30	43	14	1
O	1:40	4,403	0.16	112,964	26	1	0	567	253	314	0	5	10	65	27	3
P	2:44	5,500	0.50	249,061	32	0	0	626	561	65	0	13	26	104	41	10
Q	3:10	4,780	0.61	329,158	418	100	3	971	964	7	0	4	64	124	125	14
R	3:08	4,588	0.55	328,234	252	43	1	2,023	2,021	2	0	9	102	128	383	17
S	3:46	7,626	0.62	387,025	86	0	0	1,386	1,355	31	0	10	61	237	116	49
T	2:22	5,004	0.47	322,969	148	5	0	592	580	12	0	0	42	111	90	17
U	6:31	7,195	1.04	596,460	73	2	0	2,204	1,958	246	0	11	90	535	174	63
V	7:06	7,581	1.29	765,403	140	3	0	2,692	2,678	14	0	10	296	160	562	8
W	4:16	6,159	1.20	713,303	542	107	3	1,303	1,278	25	0	39	67	319	64	22
X	12:09	13,382	3.29	1,513,965	271	0	0	3,358	3,354	4	0	54	346	489	399	56
Y	34:14	29,497	6.10	2,892,343	1,319	21	0	45,067	44,833	234	2,284	68	14,109	3,004	6,180	861
Z	31:58	23,093	6.09	3,913,394	820	44	4	29,767	29,165	602	104	53	727	2,809	16,855	224
$\sum$	2:11:37	157,971	23.20	12,864,905	4,543	387	12	98,921	96,904	2,017	2,388	400	16,942	8,659	26,694	1,424

Table 5.7: BonnRoute run using 2STAGE.

Chip	Time	Mem.	Wires	Vias	Scenic Nets			QOR	DRC	Opens	Shorts	Stitch	Diff	Same	Min	Min
	(h:mm:ss)	(MB)	(m)		25%	50%	100%		Errors			Errors	Mindist	Mindist	Edge	Area
A	29	2,427	0.01	1,381	0	0	0	11	11	0	0	0	4	1	0	0
B	3:09	5,531	0.04	12,840	16	2	0	660	600	60	0	19	33	47	11	1
C	31	1,710	0.02	16,032	6	1	0	34	28	6	0	0	0	5	0	4
D	29	1,372	0.03	22,346	7	0	0	49	46	3	0	0	0	12	4	0
E	22	1,669	0.03	21,524	3	1	0	225	224	1	0	0	17	12	9	4
F	26	1,658	0.04	22,938	0	0	0	467	400	67	0	0	17	14	112	3
G	25	1,916	0.03	28,416	7	1	0	118	70	48	0	0	14	13	9	3
H	3:09	2,843	0.07	41,773	16	6	0	463	307	156	0	12	199	12	12	4
I	1:55	3,094	0.20	97,919	145	37	0	1,369	1,362	7	0	0	212	66	866	48
J	48	2,855	0.14	81,939	34	1	0	241	231	10	0	0	14	20	46	7
K	49	2,560	0.12	84,512	8	0	0	190	182	8	0	0	5	17	17	2
L	1:40	3,920	0.18	87,033	63	0	0	247	146	101	0	0	22	25	5	3
M	1:55	3,491	0.17	111,989	72	4	0	3,669	3,669	0	0	20	330	162	587	16
N	1:02	2,692	0.20	109,216	12	2	0	231	231	0	0	0	21	31	8	1
O	1:36	4,192	0.16	112,913	28	1	0	524	210	314	0	5	7	50	14	3
P	2:24	5,631	0.50	248,611	33	2	0	508	443	65	0	2	31	58	46	8
Q	3:01	4,682	0.61	329,533	419	107	1	903	896	7	0	5	83	99	105	7
R	2:58	4,749	0.55	328,124	233	37	1	1,973	1,972	1	0	6	107	97	350	19
S	3:16	8,162	0.62	386,436	74	4	0	1,155	1,125	30	0	0	48	159	73	51
T	2:42	5,069	0.47	322,694	151	4	0	529	517	12	0	0	42	71	88	10
U	6:48	6,927	1.04	595,456	50	0	0	1,748	1,501	247	0	0	61	296	115	65
V	8:28	7,869	1.28	763,921	135	4	0	2,503	2,487	16	0	0	194	138	518	10
W	4:04	6,188	1.20	712,617	526	118	4	1,124	1,099	25	0	0	43	284	44	25
X	10:34	12,860	3.29	1,511,988	250	1	0	2,854	2,850	4	0	31	267	363	227	60
Y	33:24	22,529	6.09	2,889,711	1,249	18	0	40,706	40,478	228	2,293	61	13,765	1,632	4,985	798
Z	30:43	22,978	6.08	3,910,696	779	51	6	27,577	26,999	578	104	22	580	1,314	16,398	195
$\sum$	2:07:07	149,574	23.17	12,852,558	4,316	402	12	90,078	88,084	1,994	2,397	183	16,116	4,998	24,649	1,347

Table 5.8: BonnRoute run using 4STAGE.

Bibliography

D. Achlioptas and C. Moore. Almost all graphs with average degree 4 are 3-colorable. *Journal of Computer and System Sciences*, 67(2):441–471, 2003. 69

H.-K. Ahn, S. W. Bae, C. Knauer, M. Lee, C.-S. Shin, and A. Vigneron. Realistic roofs over a rectilinear polygon. *Computational Geometry: Theory and Applications*, 46(9):1042–1055, 2013. 33

M. Ahrens. Ein Färbungsalgorithmus für Chipverdrahtung. B.Sc. Thesis, University of Bonn, Germany, 2012. 68

M. Ahrens. Pin access in VLSI-routing. M.Sc. Thesis, University of Bonn, Germany, 2014. 13, 14, 68, 70

M. Ahrens, M. Gester, N. Klewinghaus, D. Müller, S. Peyer, C. Schulte, and G. Tellez. Detailed routing algorithms for advanced technology nodes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2015. To appear. 13, 70, 71

O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner. A novel type of skeleton for polygons. *Journal of Universal Computer Science*, 1(12):752–761, 1995. 33

O. Aichholzer, H. Cheng, S. L. Devadoss, T. Hackl, S. Huber, B. Li, and A. Risteski. What makes a tree a straight skeleton? In *CCCG*, pages 253–258, 2012. 33

V. Alekseev, V. Lozin, D. Malyshev, and M. Milanič. The maximum independent set problem in planar graphs. *Mathematical Foundations of Computer Science 2008*, pages 96–107, 2008. 67

C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, editors. *Handbook of Algorithms for Physical Design Automation.* Auerbach Publications, 2008. 1

K. Appel, W. Haken, et al. Every planar map is four colorable. Part I: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 1977a. 61

K. Appel, W. Haken, J. Koch, et al. Every planar map is four colorable. Part II: Reducibility. *Illinois Journal of Mathematics*, 21(3):491–567, 1977b. 61

J. Augustine, S. Roy, S. Das, A. Maheshwari, S. Nandy, and S. Sarvattomananda. Recognizing the largest empty circle and axis-parallel rectangle in a desired location. Technical report, 2010. 28

J. Augustine, S. Das, A. Maheshwari, S. C. Nandy, S. Roy, and S. Sarvattomananda. Localized geometric query problems. *Computational Geometry*, 46(3):340–357, 2013. 28

F. Aurenhammer, R. Klein, and D.-T. Lee. *Voronoi Diagrams and Delaunay Triangulations*. World Scientific, 2013. 30, 42, 45

A. Bagheri and M. Razzazi. Drawing free trees inside simple polygons using polygon skeleton. *Computing and Informatics*, 23(3):239–254, 2012. 33

M. Baïou and F. Barahona. Maximum weighted induced bipartite subgraphs and acyclic subgraphs of planar cubic graphs. In *Integer Programming and Combinatorial Optimization*, pages 88–101. Springer, 2014. 63

B. S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *Journal of the ACM (JACM)*, 41(1):153–180, 1994. 66

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 2002 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '02, pages 59–66, 2002. 12, 72

R. Beigel and D. Eppstein. 3-coloring in time $O(1.3289^n)$. *Journal of Algorithms*, 54(2):168–204, 2005. 61

M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008. 49

P. Berman and B. DasGupta. Approximating the rectilinear polygon cover problems. In *Proceedings of the 4th Canadian Conference on Computational Geometry*, pages 229–235, 1992. 18

P. Berman and T. Fujito. On approximation properties of the independent set problem for degree 3 graphs. *Algorithms and Data Structures*, pages 449–460, 1995. 66

H. Blum et al. A transformation for extracting new descriptors of shape. *Models for the perception of speech and visual form*, 19(5):362–380, 1967. 33

H. L. Bodlaender. A tourist guide through treewidth. *Acta cybernetica*, 11(1-2): 1–21, 1994. 68

J.-D. Boissonnat, M. Yvinec, and H. Brönnimann. *Algorithmic geometry*, volume 5. Cambridge university press, 1998. 24

J.-D. Boissonnat, J. Czyzowicz, O. Devillers, and M. Yvinec. Circular separability of polygons. *Algorithmica*, 30(1):67–82, 2001. 28

N. Boltyanskii and N. Yaglom. *Convex figures.* Holt, Rinehart and Winston, 1961. 48

O. V. Borodin. Colorings of plane graphs: A survey. *Discrete Mathematics*, 313(4): 517–539, 2013. 69

O. V. Borodin and A. N. Glebov. Planar graphs with neither 5-cycles nor close 3-cycles are 3-colorable. *Journal of Graph Theory*, 66(1):1–31, 2011. 69

O. V. Borodin, A. N. Glebov, A. Raspaud, and M. R. Salavatipour. Planar graphs without cycles of length from 4 to 7 are 3-colorable. *Journal of Combinatorial Theory, Series B*, 93(2):303–311, 2005. 69

O. V. Borodin, A. N. Glebov, M. Montassier, and A. Raspaud. Planar graphs without 5-and 7-cycles and without adjacent triangles are 3-colorable. *Journal of Combinatorial Theory, Series B*, 99(4):668–673, 2009. 69

T. M. Chan. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *6th Canadian Conference on Computational Geometry*, pages 263–268, 1994. 49

C. Chang and J. Cong. Pseudopin assignment with crosstalk noise control. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20 (5):598–611, 2001. 12

F.-Y. Chang, R.-S. Tsay, W.-K. Mak, and S.-H. Chen. MANA: A shortest path maze algorithm under separation and minimum length nanometer rules. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32 (10):1557–1568, 2013. 78

B. Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6(1):485–524, 1991. 18, 24

H. Chen, C.-K. Cheng, A. Kahng, I. Mandoiu, Q. Wang, and B. Yao. The Y-architecture for on-chip interconnect: Analysis and methodology. In *Proc. DAC*, pages 13–19, 2003. 7

S. W. Cheng and R. Janardan. Efficient maintenance of the union of intervals on a line, with applications. *Journal of Algorithms*, 12(1):57–74, 1991. 48

N. Chiba, T. Nishizeki, and N. Saito. An approximation algorithm for the maximum independent set problem on planar graphs. *SIAM Journal on Computing*, 11(4): 663–675, 1982. 66

F. Chin, J. Snoeyink, and C. A. Wang. Finding the medial axis of a simple polygon in linear time. *Discrete & Computational Geometry*, 21(3):405–420, 1999. 30, 41, 42

M. Cho, Y. Ban, and D. Z. Pan. Double patterning technology friendly detailed routing. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 506–511. IEEE, 2008. 70

H.-A. Choi, K. Nakajima, and C. S. Rim. Graph bipartization and via minimization. *SIAM Journal on Discrete Mathematics*, 2(1):38–47, 1989. 62, 63

A. Coja-Oghlan and D. Vilenchik. Chasing the k-colorability threshold. In *54th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 380–389. IEEE, 2013. 69

P. Coussy and A. Morawiec. *High-level synthesis*. Springer, 2010. 1

J. C. Culberson and R. A. Reckhow. Covering polygons is hard. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 601–611. IEEE, 1988. 18

D. P. Dailey. Uniqueness of colorability and colorability of planar 4-regular graphs are NP-complete. *Discrete Mathematics*, 30(3):289–293, 1980. 61

E. de Klerk, D. V. Pasechnik, and J. P. Warners. On approximate graph colouring and max-k-cut algorithms based on the θ-function. *Journal of Combinatorial Optimization*, 8(3):267–294, 2004. 64

S. Devadas, A. Ghosh, and K. Keutzer. *Logic synthesis*. McGraw-Hill, Inc., 1994. 1

E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 86

R. Duda and P. Hart. Pattern classification and scene analysis, 1973. 33

A. Dumitrescu and M. Jiang. Maximal empty boxes amidst random points. In A. Gupta, K. Jansen, J. Rolim, and R. Servedio, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, volume 7408 of *Lecture Notes in Computer Science*, pages 529–540. Springer Berlin Heidelberg, 2012. 28

H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *Discrete & Computational Geometry*, 1(1):25–44, 1986. 46

P. Erdős and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci*, 5:17–61, 1960. 69

S.-Y. Fang, Y.-W. Chang, and W.-Y. Chen. A novel layout decomposition algorithm for triple patterning lithography. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1185–1190. ACM, 2012. 67

L. Ferrari, P. Sankar, and J. Sklansky. Minimal rectangular partitions of digitized blobs. *Computer vision, graphics, and image processing*, 28(1):58–71, 1984. 19

J. Finders, M. Dusa, and S. Hsu. Double patterning lithography: The bridge between low k1 ArF and EUV. *Microlithography World*, February 2008. 55

S. Fiorini, N. Hardy, B. Reed, and A. Vetta. Planar graph bipartization in linear time. *Discrete Applied Mathematics*, 156(7):1175–1180, 2008. 62

A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics (TOG)*, 3(2):153–174, 1984. 24, 49

D. S. Franzblau. Performance guarantees on a sweep-line heuristic for covering rectilinear polygons with rectangles. *SIAM Journal on Discrete Mathematics*, 2 (3):307–321, 1989. 18

G. N. Frederickson. On linear-time algorithms for five-coloring planar graphs. *Information Processing Letters*, 19(5):219–224, 1984. 61

R. Fritsch and G. Fritsch. *Der Vierfarbensatz*. BI-Wissenschaftsverlag, 1994. 61

J.-R. Gao and D. Z. Pan. Flexible self-aligned double patterning aware detailed routing with prescribed layout planning. In *Proceedings of the 2012 ACM International Symposium on Physical Design*, pages 25–32. ACM, 2012. 70

X. Gao and L. Macchiarulo. Enhancing double-patterning detailed routing with lazy coloring and within-path conflict avoidance. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1279–1284, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association. ISBN 978-3-9810801-6-2. 70

M. R. Garey and D. S. Johnson. The rectilinearSteiner tree problem is NP-complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977. 66

M. Gester. Voronoi-Diagramme von Achtecken in der Maximum-Metrik. *Diploma Thesis, University of Bonn*, 2009. 47, 48, 86

M. Gester, D. Müller, T. Nieberg, C. Panten, C. Schulte, and J. Vygen. BonnRoute: Algorithms and data structures for fast and good VLSI routing. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(2):32:1–32:24, 2013. Preliminary version in the Proceedings of the 49th Annual Design Automation Conference, pages 459–464. 3, 71, 86, 87, 89, 101, 120

M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, 42(6):1115–1145, 1995. 64

M. X. Goemans and D. P. Williamson. Primal-dual approximation algorithms for feedback problems in planar graphs. *Combinatorica*, 18(1):37–59, 1998. 63

H. Grötzsch. Ein Dreifarbensatz für dreikreisfreie Netze auf der Kugel. *Wiss. Z. Martin Luther Univ. Halle-Wittenberg, Math. Naturwiss Reihe*, 8:109–120, 1959. 69

B. Grünbaum et al. Grötzsch's theorem on 3-colorings. *The Michigan Mathematical Journal*, 10(3):303–310, 1963. 69

L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Transactions on Graphics (TOG)*, 4(2):74–123, 1985. 41

H. N. Gürsoy and N. M. Patrikalakis. An automatic coarse and fine surface mesh generation scheme based on medial axis transform: Part I algorithms. *Engineering with computers*, 8(3):121–137, 1992. 33

F. Hadlock. Finding a maximum cut of a planar graph in polynomial time. *SIAM Journal on Computing*, 4(3):221–225, 1975. 64

M. M. Halldórsson. Approximations of weighted independent set and hereditary subset problems. In *Computing and Combinatorics*, pages 261–270. Springer Berlin Heidelberg, 1999. 67

M. M. Halldórsson and J. Radhakrishnan. Greed is good: Approximating independent sets in sparse and bounded-degree graphs. *Algorithmica*, 18(1):145–163, 1997. 66

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*, 2:100–107, 1968. 86

S. Held, B. Korte, D. Rautenbach, and J. Vygen. Combinatorial optimization in VLSI design. In V. Chvátal, editor, *Combinatorial Optimization: Methods and Applications*, pages 33–96. IOS Press, Amsterdam, 2011. 1

S. Held, W. Cook, and E. C. Sewell. Maximum-weight stable sets and safe lower bounds for graph coloring. *Mathematical Programming Computation*, 4(4):363–381, 2012. 67

A. Hetzel. A sequential detailed router for huge grid graphs. In *Proc. DATE*, pages 332–339, 1998. 14, 86, 87, 88, 94

R. B. Hitchcock. Cellular wiring and the cellular modeling technique. In *Proceedings of the 6th annual Design Automation Conference*, DAC '69, pages 25–41, 1969. 12

T.-Y. Ho, C.-F. Chang, Y.-W. Cheang, and S.-J. Chen. Multilevel full-chip routing for the X-based architecture. In *Proc. DAC*, pages 597–602, 2005. 7

J. E. Hopcroft and R. M. Karp. An $n^{\frac{5}{2}}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973. 19

J. E. Hopcroft and R. Tarjan. Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568, 1974. 61

F. Hüffner. Algorithm engineering for optimal graph bipartization. In *Experimental and Efficient Algorithms*, pages 240–252. Springer, 2005. 62

J. Humpola. Schneller Algorithmus für kürzeste Wege in irregulären Gittergraphen. *Diploma Thesis, University of Bonn*, 2009. 87, 88

F. K. Hwang. An $O(n \log n)$ algorithm for rectilinear minimal spanning trees. *Journal of the ACM*, 26(2):177–182, 1979. 27

H. Imai and T. Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *Journal of algorithms*, 4(4): 310–323, 1983. 60

A. Itai, C. H. Papadimitriou, and J. L. Szwarcfiter. Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686, 1982. 80

R. C. Jaeger. *Lithography. Introduction to microelectronic fabrication.* Prentice Hall, 2002. 2

A. Kahng, S. Vaya, and A. Zelikovsky. New graph bipartizations for double-exposure, bright field alternating phase-shift mask layout. In *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, pages 133–138. IEEE, 2001. 63

A. B. Kahng, C.-H. Park, X. Xu, and H. Yao. Layout decomposition for double patterning lithography. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 465–472. IEEE Press, 2008. 67

A. B. Kahng, C.-H. Park, X. Xu, and H. Yao. Layout decomposition approaches for double patterning lithography. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(6):939–952, 2010. 67

V. Kann, S. Khanna, J. Lagergren, and A. Panconesi. On the hardness of approximating max k-cut and its dual. *Chicago Journal of Theoretical Computer Science*, 2, 1997. 64

H. Kaplan and M. Sharir. Finding the maximal empty disk containing a query point. In *Proceedings of the 2012 Symposium on Computational Geometry*, pages 287–292. ACM, 2012. 28

H. Kaplan, S. Mozes, Y. Nussbaum, and M. Sharir. Submatrix maximum queries in monge matrices and monge partial matrices, and their applications. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 338–355. SIAM, 2012. 28

A. C. Kaporis, L. M. Kirousis, and Y. C. Stamatiou. A note on the non-colorability threshold of a random graph. *Journal of Combinatorics*, 7(1), 2001. 69

J. M. Keil. Polygon decomposition. *Handbook of Computational Geometry*, 2:491–518, 2000. 17, 18

S. Khanna, N. Linial, and S. Safra. On the hardness of approximating the chromatic number. *Combinatorica*, 20(3):393–415, 2000. 68

S. Khot. On the power of unique 2-prover 1-round games. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 767–775. ACM, 2002. 64

S. Khot, G. Kindler, E. Mossel, and R. O'Donnell. Optimal inapproximability results for MAX-CUT and other 2-variable CSPs. *SIAM Journal on Computing*, 37(1): 319–357, 2007. 64

S.-M. Kim, S.-Y. Koo, J.-S. Choi, Y.-S. Hwang, J.-W. Park, E.-K. Kang, C.-M. Lim, S.-C. Moon, and J.-W. Kim. Issues and challenges of double patterning lithography in DRAM. In *Proc. SPIE Conf. on Optical Microlithography*, 2006. 55

D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983. 25, 42, 45, 86

N. Klewinghaus. Fast parallelisation for detailed routing in VLSI Design. *Diploma Thesis, University of Bonn*, 2013. 14

D. König. Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre. *Mathematische Annalen*, 77(4):453–465, 1916. 61

B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer Publishing Company, Incorporated, 2012. 6, 59, 61, 66

B. Korte, D. Rautenbach, and J. Vygen. BonnTools: Mathematical innovation for layout and timing closure of systems on a chip. *Proceedings of the IEEE*, 95(3): 555–572, 2007. 3

S. Kratsch and M. Wahlström. Compression via matroids: a randomized polynomial kernel for odd cycle transversal. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 94–103. SIAM, 2012. 62

V. A. Kumar and H. Ramesh. Covering rectilinear polygons with axis-parallel rectangles. *SIAM Journal on Computing*, 32(6):1509–1541, 2003. 18

D.-T. Lee and C. K. Wong. Voronoi diagrams in $L_1(L_\infty)$ metrics with 2-dimensional storage applications. *SIAM Journal on Computing*, 9(1):200–211, 1980. 27

D. Leven and M. Sharir. Planning a purely translational motion for a convex object in two-dimensional space using generalized Voronoi diagrams. *Discrete & Computational Geometry*, 2(1):9–31, 1987. 45

L. Liebmann, L. Pileggi, J. Hibbeler, V. Rovner, T. Jhaveri, and G. Northrop. Simplify to survive: prescriptive layouts ensure profitable scaling to 32nm and beyond. In *SPIE Advanced Lithography*. International Society for Optics and Photonics, 2009. 70

L. W. Liebmann. Layout impact of resolution enhancement techniques: impediment or opportunity? In *Proceedings of the 2003 international symposium on Physical design*, pages 110–117. ACM, 2003. 2

F. Liers and G. Pardella. Partitioning planar graphs: a fast combinatorial approach for max-cut. *Computational Optimization and Applications*, 51(1):323–344, 2012. 64

Y.-H. Lin and Y.-L. Li. Double patterning lithography aware gridless detailed routing with innovative conflict graph. In *Proceedings of the 47th Annual Design Automation Conference*, pages 398–403. ACM, 2010. 70

Y.-H. Lin, B. Yu, D. Pan, and Y.-L. Li. Triad: A triple patterning lithography aware detailed router. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 123–129, Nov 2012. 70

A. Lingas. The power of non-rectilinear holes. In *Automata, Languages and Programming*, pages 369–383. Springer, 1982. 19

W. Liou, J. Tan, and R. Lee. Minimum partitioning simple rectilinear polygons in $O(n \log \log n)$-time. In *Proceedings of the fifth annual symposium on Computational geometry*, pages 344–353. ACM, 1989. 18, 19

W. Lipski, E. Lodi, F. Luccio, C. Mugnai, and L. Pagli. On two dimensional data organization II. *Fundamenta Informaticae*, 2(3):245–260, 1979. 19

Q. Ma, H. Zhang, and M. D. Wong. Triple patterning aware routing and its comparison with double patterning aware routing in 14nm technology. In *Proceedings of the 49th Annual Design Automation Conference*, pages 591–596. ACM, 2012. 70

M. Maenhoudt, J. Versluijs, H. Struyf, J. Van Olmen, and M. Van Hove. Double patterning scheme for sub-0.25 k1 single damascene structures at NA $= 0.75, \lambda =$ 193 nm. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 5754, pages 1508–1518, 2005. 55

J. Maßberg. *Facility location and clock tree synthesis.* PhD thesis, Dissertation, Universität Bonn, 2009. 46, 47

J. Maßberg and T. Nieberg. Rectilinear paths with minimum segment lengths. *Discrete Applied Mathematics*, 161(12):1769–1775, 2013. 78, 85

D. Müller. *Fast resource sharing in VLSI routing.* PhD thesis, University of Bonn, 2009. 11, 12, 72, 93

D. Müller, K. Radke, and J. Vygen. Faster min–max resource sharing in theory and practice. *Mathematical Programming Computation*, 3(1):1–35, 2011. 12, 14

D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7(2):217–236, 1978. 41

F. Nohn. Detailed Routing im VLSI-Design unter Berücksichtigung von Multiple-Patterning. *Diploma Thesis, University of Bonn*, 2012. 68, 78, 87

T. Ohtsuki. Minimum dissection of rectilinear regions. In *Proc. 1982 IEEE Symp. on Circuits and Systems, Rome*, pages 1210–1213, 1982. 19

A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial tessellations: concepts and applications of Voronoi diagrams*, volume 501. John Wiley & Sons, 2009. 30

E. Papadopoulou and D.-T. Lee. The L_∞ Voronoi diagram of segments and VLSI applications. *International Journal of Computational Geometry & Applications*, 11(05):503–528, 2001. 28, 30, 41, 86

G. Peschka. Kotirte Ebenen und deren Anwendung. *Verlag Buschak & Irrgang, Brünn*, 1877. 33

S. Peyer. *Shortest Paths and Steiner Trees in VLSI Routing.* PhD thesis, University of Bonn, 2007. 13

S. Peyer, D. Rautenbach, and J. Vygen. A generalization of Dijkstra's shortest path algorithm with applications to VLSI routing. *Journal of Discrete Algorithms*, 7: 377–390, 2009. 87, 88

B. Reed, K. Smith, and A. Vetta. Finding odd cycle transversals. *Operations Research Letters*, 32(4):299–301, 2004. 62

N. Robertson and P. D. Seymour. Graph minors. I. excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, 1983. 68

N. Robertson, D. Sanders, P. Seymour, and R. Thomas. The four-colour theorem. *Journal of Combinatorial Theory, Series B*, 70(1):2–44, 1997. 61

A. Rosenfeld. Axial representations of shape. *Computer Vision, Graphics, and Image Processing*, 33(2):156–173, 1986. 33

F. Rubin. The Lee path connection algorithm. *IEEE Transactions on Computers*, 100(9):907–914, 1974. 86

J. S. Salowe. Rip-up and reroute. In C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, editors, *Handbook of Algorithms for Physical Design Automation.* Auerbach Publications, 2008. 14

J. Schneider. *Transistor-Level Layout of Integrated Circuits.* PhD thesis, University of Bonn, 2014. 7

C. Schulte. *Design Rules in VLSI Routing.* PhD thesis, University of Bonn, 2012. 15, 20, 21, 74, 93

R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry*, 1(1):51–64, 1991. 49

R. Seidel. The nature and meaning of perturbations in geometric computing. *Discrete & Computational Geometry*, 19(1):1–17, 1998. 29

M. I. Shamos and D. Hoey. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science*, pages 151–162, 1975. 27

M. Sharir and P. K. Agarwal. *Davenport-Schinzel sequences and their geometric applications.* Cambridge university press, 1995. 44, 45, 46

V. Soltan and A. Gorpinevich. Minimum dissection of a rectilinear polygon with arbitrary holes into rectangles. *Discrete & Computational Geometry*, 9(1):57–79, 1993. 19

M. Tanase and R. C. Veltkamp. Polygon decomposition based on the straight line skeleton. In *Proceedings of the nineteenth annual symposium on Computational geometry*, pages 58–67. ACM, 2003. 33

S. Teig. The X architecture: Not your father's diagonal wiring. In *Proc. International Workshop on System-Level Interconnect Prediction*, pages 33–37, 2002. 7

C. Thomassen. Grötzsch's 3-color theorem and its counterparts for the torus and the projective plane. *Journal of Combinatorial Theory Series B*, 62(2):268–279, 1994. 69

H. Tian, H. Zhang, Q. Ma, Z. Xiao, and M. D. Wong. A polynomial time triple patterning algorithm for cell based row-structure layout. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pages 57–64. IEEE, 2012. 68

P. J. Vermeer. Two-dimensional MAT to boundary conversion. In *Proceedings on the second ACM Symposium on Solid Modeling and Applications*, pages 493–494. ACM, 1993. 33

W.-f. Wang and M. Chen. Planar graphs without 4, 6, 8-cycles are 3-colorable. *Science in China Series A: Mathematics*, 50(11):1552–1562, 2007. 69

B. Yu and D. Z. Pan. Layout decomposition for quadruple patterning lithography and beyond. *arXiv preprint arXiv:1404.0321*, 2014. 67

B. Yu, K. Yuan, B. Zhang, D. Ding, and D. Z. Pan. Layout decomposition for triple patterning lithography. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pages 1–8. IEEE, 2011. 67

K. Yuan, K. Lu, and D. Z. Pan. Double patterning lithography friendly detailed routing with redundant via consideration. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pages 63–66. IEEE, 2009. 70

G. M. Ziegler. *Lectures on polytopes*, volume 152. Springer, 1995. 48

Summary

Routing is a major step in *VLSI design*, the design process of complex integrated circuits (commonly known as *chips*). The basic task in routing is to connect predetermined locations on a chip (*pins*) with *wires* which serve as electrical connections. One main challenge in routing for advanced chip technology is the increasing complexity of *design rules* which reflect manufacturing requirements. In this thesis we investigate various aspects of this challenge.

In Chapter 4 we consider polygon decomposition problems in the context of VLSI design. We introduce two different width notions for polygons which are important for width-dependent design rules in VLSI routing, and we present efficient algorithms for computing *width-preserving* decompositions of rectilinear polygons into rectangles. Such decompositions are used in routing to allow for fast design rule checking. A main contribution of this thesis is an $O(n)$ time algorithm for computing a decomposition of a simple rectilinear polygon with n vertices into $O(n)$ rectangles, preseverving *two-dimensional width.* For rectilinear polygons with holes the runtime increases to $O(n \log n)$. Here the two-dimensional width at a point of the polygon is defined as the edge length of the largest square that contains the point and is contained in the polygon. In order to obtain these results we establish a connection between such decompositions and L_∞ *Voronoi diagrams.* In particular, we associate each Voronoi edge with a corresponding *edge rectangle.* We prove that there exists a width-preserving decomposition such that each of its rectangles can be obtained by performing only four set operations on five edge rectangles. Finally, we show how to preprocess the Voronoi diagram in linear time such that the involved edge rectangles can be queried in constant time, implying the desired results. In Section 4.3 we consider polygon sets which originate from disjoint polygons which are all expanded by a fixed *zonogon* of constant complexity via Minkowski sum. We describe a simple and efficient $O(n \log n)$ algorithm for decomposing such polygon sets into $O(n)$ interior-disjoint trapezoids. This algorithm has applications in *clock network design*, another important part of VLSI design.

In Chapter 5 we consider implications of *multiple patterning* and other advanced design rules for VLSI routing. The main contribution in this context is the detailed description of a routing approach which is able to manage such advanced design rules. As a main algorithmic concept we use *multi-label shortest paths* where certain path properties (which model design rules) can be enforced by defining *labels* assigned to path vertices and allowing only certain *label transitions.* We prove that computing multi-label shortest paths is NP-hard for three-dimensional grid graphs,

the typical setting in routing. Furthermore, we describe a polynomial time algorithm by relaxing the property that vertices may not be visited more than once for a path. Possible cycles are simply removed afterwards if necessary. In practice, this approach is preferable to standard shortest path algorithms since design rules can be considered earlier and more accurately. This is confirmed by our experimental results.

The multiple patterning approach described in Chapter 5 has been implemented in *BonnRoute* (Gester et al. [2013]), a routing tool developed at the Research Institute for Discrete Mathematics, University of Bonn, in cooperation with IBM. We present results confirming that a flow combining BonnRoute and an external cleanup step produces far superior results compared to an industry standard router. In particular, our proposed flow runs more than twice as fast, reduces the *via count* by more than 20%, the *wiring length* by more than 10%, and the number of remaining design rule errors by more than 60%. These results obtained by applying our multiple patterning approach to real-world chip instances provided by IBM are another main contribution of this thesis. We note that IBM uses our proposed combined BonnRoute flow as the default tool for *signal routing*.