Efficient Algorithms for Routing a Net Subject to VLSI Design Rules

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Markus Ahrens

aus

Lahnstein

Bonn, August 2020

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

1. Gutachter:	Herr Professor Dr. Jens Vygen
2. Gutachter:	Herr Professor Dr. Dr. h.c. Bernhard Korte
Tag der Promotion:	20.11.2020
Erscheinungsjahr:	2020

Acknowledgements

First and foremost, I would like to thank my advisor Professor Dr. Jens Vygen for his excellent guidance and for providing insightful feedback on an earlier version of this thesis. Special thanks go to Professor Dr. Dr. h.c. Bernhard Korte and Professor Dr. Jens Vygen for providing perfect working conditions at the Research Institute for Discrete Mathematics at the University of Bonn.

I am thankful to all my present and former colleagues for the friendly and productive working environment, especially Dr. Ulrich Brenner, Dr. Michael Gester, Dorothee Henke, Dr. Anna Hermann, Niko Klewinghaus, Benjamin Klotz, Dr. Dirk Müller, Dr. Philipp Ochsendorf, Stefan Rabenstein, Dr. Rudolf Scheifele, Dr. Jan Schneider, Rasmus Schroeder, and Dr. Jannik Silvanus. I also want to thank Professor Dr. Stephan Held and Professor Dr. Stefan Hougardy for sharing some of their knowledge with me and for many interesting discussions. I would also like to express my deepest thanks to Dr. Anna Hermann, Niko Klewinghaus, and Stefan Rabenstein for proofreading parts of this thesis.

I am grateful for the excellent cooperation with IBM and would like to express my thanks to our business partners Anne Heppner, Niko Klewinghaus, Dr. Christian Roth, and Dr. Gustavo Téllez.

I would like to thank Dorothee Henke and Stefan Rabenstein for their contributions to the Restricted Dijkstra-Steiner algorithm and its implementation. Moreover, I would like to thank Dorothee Henke and Professor Dr. Jens Vygen for the productive joint work on the future costs.

Finally, I would like to thank my family, who supported me throughout my studies.

Contents

Chapter 1

Introduction

Modern integrated circuits, also called chips, can contain billions of transistors and are among the most complicated structures ever designed and manufactured by man. Their design, called VLSI[1] design, is very challenging and is done in several steps. The task in routing is to compute the internal wiring of the chip. On large instances this requires packing millions of node-disjoint Steiner trees into a graph with hundreds of billions of nodes while respecting hundreds of complicated design rules that need to be satisfied to ensure the reliability of the manufacturing process. Routing is solved in two or more stages: Global routing computes an approximate wire layout, solving the global packing problem while ignoring local constraints. This thesis is about detailed routing which computes the actual wiring based on the global routing solution.

We present algorithms for state-of-the-art detailed routing. Our new path search algorithm is more efficient and more flexible than previous algorithms and implementations. It has been used very successfully in the IBM design flow to design complex processor chips. We present experimental results demonstrating that our path search is efficient and can reduce the number of design rule violations drastically. Moreover, we present a new tree search algorithm that generalizes the Dijkstra-Steiner algorithm [Hougardy et al., 2017]. We propose to use this algorithm for finding optimal Steiner trees that are similar to the global routing solution, thereby recovering electrical and timing properties optimized in global routing.

The rest of this thesis is organized as follows:

In Chapter 2 we introduce the detailed routing problem more formally and give an overview of BonnRoute, a state-of-the-art router developed at the University of Bonn in joint work with IBM. BonnRoute is the main routing tool used by IBM for the design of processor chips. The algorithms presented in this thesis were implemented as part of BonnRoute-Detailed, the detailed router of BonnRoute.

Our path search algorithm is presented in Chapter 3 and Chapter 4. Chapter 3 focuses on making the algorithm efficient while being able to handle complicated implicitly given

[1] Very Large Scale Integration

grid graphs. A key component that helps to achieve both objectives is the grid region data structure introduced in Section 3.4. In Section 3.3 we speed up the path search by making it goal-oriented using the well-known concept of future costs. Our future costs are computed by solving a geometric shortest path problem. We prove that this problem can be solved in logarithmic time after a polynomial time preprocessing. Furthermore, we propose an algorithm for solving the geometric shortest path problem in practice that was developed in joint work with Dorothee Henke and Jens Vygen. Our path search is the new algorithmic core of BonnRouteDetailed. Section 3.6 summarizes its advantages compared to the previous implementation based on [Hetzel, 1995, 1998].

In Chapter 4 we consider the problem of respecting design rules. In Section 4.2 we prove that given a two-dimensional grid graph and nodes s, t it is NP-complete to decide whether there is an s-t-path in which each maximal straight subpath has length at least two. Hence, obeying even very simple design rules in the path search is NP-hard. The rest of Chapter 4 describes our framework for making our path search respect many design rules in practice. Its most important component is the multi-labeling introduced in Section 4.5 that allows us to find edge progressions that satisfy certain properties specified by label systems. This allows us to respect design rules in a correct-by-construction manner. Our multi-labeling is more general and more efficient than that of [Nohn, 2012; Gester, 2015; Ahrens et al., 2015]. This allows us to respect more design rules while being less restrictive.

Chapter 6 presents experimental results on real-world IBM instances. The results demonstrate that the speed-up techniques discussed in Chapter 3 are effective and that using the framework for avoiding design rule violations discussed in Chapter 4 reduces the number of violations by a factor of approximately 443.

In Chapter 5 we consider the problem of computing optimal Steiner trees respecting restrictions on the topology and on the location of Steiner points derived from the global routing. To achieve this we introduce the Restricted Dijkstra-Steiner algorithm which generalizes the Dijkstra-Steiner algorithm [Hougardy et al., 2017]. In our application the Restricted Dijkstra-Steiner algorithm achieves near-linear runtime under mild assumptions. This is possible because the restrictions actually make the problem easier. The Restricted Dijkstra-Steiner algorithm was developed and implemented in joint work with Dorothee Henke and Stefan Rabenstein.

Chapter 2

Preliminaries

This chapter introduces the main problems we consider in this thesis and provides background information that can make it easier to understand the rest of this thesis. First, in Section 2.1 we introduce the detailed routing problem and define important terms that are used throughout this thesis. Then, in Section 2.2 we give a high-level overview of BonnRoute, a state-of-the-art router developed at the University of Bonn in joint work with IBM. The overview focuses on its detailed router BonnRouteDetailed.

In the rest of this thesis we use basic notation on graphs and combinatorial optimization without further comments. For an introduction to combinatorial optimization see [Korte and Vygen, 2018]. For more information on routing in general we refer to Part VI of [Alpert et al., 2008] and Chapter 8 of [Lavagno et al., 2016].

2.1 The Detailed Routing Problem

Chips contain *circuits* that can implement simple logical functions like NAND or NOT, but also more complicated logical functions or the storage of data. The connection points of the circuits are called *pins*. There are additional pins, for example for the inputs and outputs of the chip. In detailed routing we are given a set of pins and the information which pins should be connected electrically. A set of pins that should be connected is called a *net*. The task in detailed routing is to compute an optimized wiring that realizes these connections and satisfies a set of complicated *design rules*. The rest of this section introduces notation and provides more information on detailed routing.

The connections can be established using multiple *layers*. Throughout this thesis we use $\mathcal{Z} = \{0, 1, \ldots, z_{max}\}$, where $z_{max} \in \mathbb{N}$, to denote the set of *wiring layers* that are relevant for detailed routing. The interval $(z, z+1)$ between two adjacent wiring layers $z, z+1 \in \mathcal{Z}$ is called a *via layer*. It is used for connecting objects on layers z and $z+1$. In our application we typically have $|\mathcal{Z}| \in \{5, \ldots, 18\}$.

All routing objects are represented by axis-parallel rectangles and cuboids, called shapes.

Definition 2.1 (wire, via, shape). *A* wire *is a two-dimensional rectangle* $[x_1, x_2] \times [y_1, y_2] \times \{z\}$ *with* $x_1, x_2, y_1, y_2 \in \mathbb{Z}$ *and* $z \in \mathcal{Z}$.

A via *is a triple* (v_b, v_m, v_t)*, where* $v_b = r_b \times \{z\}$ *and* $v_t = r_t \times \{z + 1\}$ *are wires and* $v_m = r_m \times (z, z+1)$ *is a three-dimensional axis-parallel cuboid with integral minimum and maximum x- and y-coordinates such that* $r_m \subseteq r_b \cap r_t$. *We call* v_b *and* v_t *the* bottom pad *and* top pad *of the via and* v_m *its* cut shape.

A shape *is a wire or the bottom pad, top pad, or cut shape of a via.*

The wires and vias that make up the wirings are represented by one-dimensional objects and extensions, called wire models and via models.

Definition 2.2 (wire model, via model). *A* wire model *is a rectangle* $[x_1, x_2] \times [y_1, y_2] \times \{0\}$ *with* $x_1, y_1 < 0 < x_2, y_2$ *and* $x_1, x_2, y_1, y_2 \in \mathbb{Z}$.

A via model *is a triple of wire models.*

Definition 2.3 (plain stick, wire stick, via stick, stick). *A* plain stick *is a one-dimensional rectangle* $[x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$ *with* $x_1, x_2, y_1, y_2 \in \mathbb{Z}$, $z_1, z_2 \in \mathcal{Z}$, *and either* $x_1 \neq x_2$, $y_1 \neq y_2$, *or* $z_2 - z_1 = 1$.

A wire stick *is a pair* (s, m^w)*, where* $s = [x_1, x_2] \times [y_1, y_2] \times \{z\}$ *is a plain stick and* m^w *is a wire model. Its wire is the Minkowski sum* $s + m^w$.

A via stick *is a pair* (s, m^v)*, where* $s = \{x\} \times \{y\} \times [z, z+1]$ *is a plain stick and* $m^v = (v_b, v_m, v_t)$ *is a via model. Its via is*

$$(\{x\} \times \{y\} \times \{z\} + v_b, \{x\} \times \{y\} \times (z, z+1) + v_m, \{x\} \times \{y\} \times \{z+1\} + v_t).$$

A stick *is either a wire stick or a via stick.*

See Figure 2.1 for an illustration of shapes and sticks.

There are three kinds of routing objects:

- *Blockages* block an area, making it unusable for other objects. They are represented by shapes.
- *Pins* are represented by connected sets of shapes. They are inputs or outputs of the chip, interfaces across hierarchy, or connection points of circuits. The task in routing is to legally connect certain sets of pins, given by so-called *nets.*
- *Wirings of nets* are represented by sticks.

These terms are formalized in the following definition.

Definition 2.4 (blockage, pin, net, wiring of a net). *A* blockage *is a shape.*

A pin $p = \cup_{i \in \{1,\ldots,k\}} s_i$ *is the union of finitely many shapes* $s_1, \ldots, s_k$ *such that* p *is connected.*

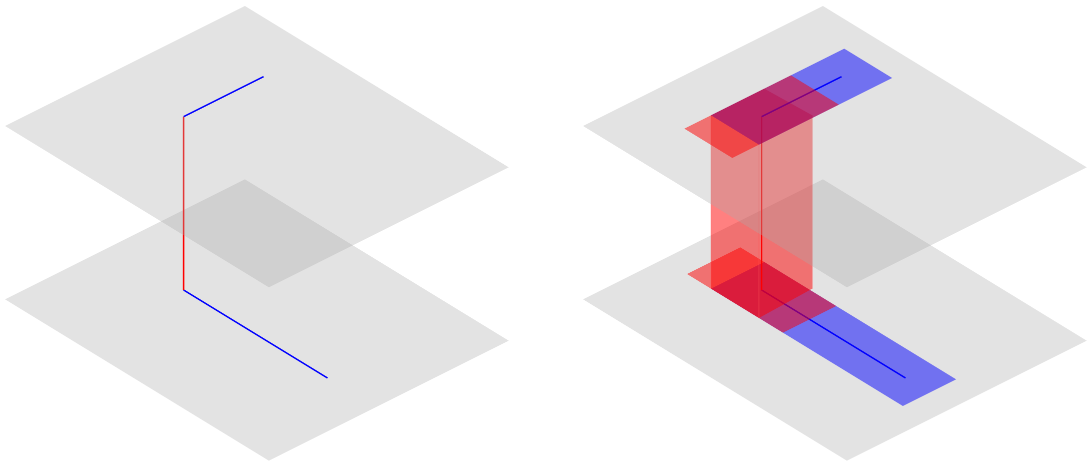


Figure 2.1: Three plain sticks (left, right) and their shapes (right) for appropriate via model and wire models. There are two wire sticks on different wiring layers and a via stick connecting them. Note that the via has different extensions on each layer.

A net *is a triple $N = (P, \mathcal{M}^w, \mathcal{M}^v)$, where P is a set of pins, $\mathcal{M}^w = (M^w_0, \dots, M^w_{|\mathcal{Z}|-1})$ is a set of sets of wire models, specifying* allowed wire models M^w_z *for each wiring layer $z \in \mathcal{Z}$ and $\mathcal{M}^v = (M^v_{0,1}, \dots, M^v_{|\mathcal{Z}|-2,|\mathcal{Z}|-1})$ is a set of sets of via models, specifying* allowed via models $M^v_{z-1,z}$ *for vias on $(z-1, z)$.*

A set of sticks $\mathcal{S} = \mathcal{S}^w \cup \mathcal{S}^v$ consisting of wire sticks $\mathcal{S}^w$ and via sticks $\mathcal{S}^v$ connects N *if*

1. *$\bigcup_{(s,m)\in\mathcal{S}} s$ is connected,*
2. *for each $p \in P$ we have $p \cap \bigcup_{(s,m)\in\mathcal{S}} s \neq \emptyset$,*
3. *for $(s,m) \in \mathcal{S}^w$ we have $m \in M^w_z$, where z is the wiring layer of s, and*
4. *for $(s,m) \in \mathcal{S}^v$ we have $m \in M^v_{z,z+1}$, where z is the minimum layer of s.*

We call such a set of sticks $\mathcal{S}$ a wiring of the net N.

Chips are manufactured layer by layer with a lithographic process, starting from the lowest layer. To allow manufacturing, the shapes on a chip must obey hundreds of *design rules*. Different wiring and via layers can be manufactured with different processes: Lower layers use expensive manufacturing processes that can manufacture very small structures. Higher layers contain larger structures and usually use manufacturing processes that were introduced in older technologies. Every type of layer has its own design rules. We distinguish *diff-net rules* and *same-net rules*:

Diff-net rules specify minimum spacing requirements between shapes of different nets and to blockages. We list simplified versions of the two most important types of rules:

- **Minimum Distance Rules**: The distance of two shapes on the same wiring or via layer must not be smaller than a layer-dependent threshold. See Figure 2.2 for an illustration.

- **Inter Layer Via Minimum Distance Rules**: Two via cut shapes $r_1 \times (z, z+1), r_2 \times (z+1, z+2)$ on adjacent via layers must have distance of at least a layer-dependent threshold value. This rule applies only for some pairs of via layers.

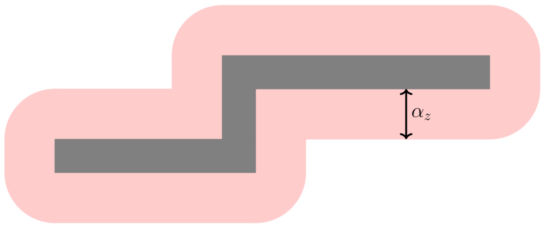


Figure 2.2: An L_2-distance rule of α_z implies that no shape of a different net may intersect the red area.

The rules can specify a minimum L_2-distance, but other distance measures such as L_1-distance, horizontal minimum distance that applies only if there is a vertical overlap, or more complicated distance functions are also possible. Additionally, there are some more complicated diff-net rules, depending on the width of the involved metal, on colors that are assigned to shapes on some layers, or on three or more shapes. We note that in special cases diff-net rules can apply to different parts of the same net.

Same-net rules impose restrictions on the wiring of each individual net. We list simplified versions of some important rules:

- **Minimum Width Rule**: Each shape must have at least a certain layer-dependent minimum width and length.

Most same-net rules do not depend on individual shapes but on maximal sets of connected shapes $\mathcal{S}$ on a layer z and / or their polygonal boundary:

- **Minimum Area Rule**: $\mathcal{S}$ must have an area exceeding a layer-dependent threshold.
- **Via Enclosure Rule**: Let $v_m \times (z, z+1)$ be the cut shape of a via model such that $\mathcal{S} \cap v_m \times \{z\} \neq \emptyset$. Then $v_m \times \{z\} + [-\alpha_z, \alpha_z] \times \{0\} \times \{0\} \subseteq \mathcal{S}$ or $v_m + \{0\} \times [-\alpha_z, \alpha_z] \times \{0\} \subseteq \mathcal{S}$ must hold for some threshold value $\alpha_z > 0$. A similar rule applies for vias on $(z-1, z)$. Typically, via models are designed such that the enclosure rules are satisfied automatically.
- **Minimum Adjacent Edge Length Rule**: Let e_1, e_2 be two adjacent edges of the polygonal boundary of $\mathcal{S}$. Then at least one of the edges must be longer than a threshold value. See Figure 2.3 for an illustration.

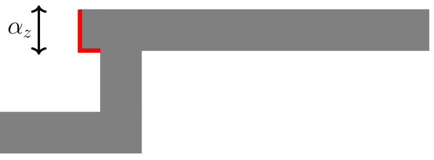


Figure 2.3: The gray shapes violate a minimum adjacent edge length rule of α_z because of the two edges marked in red.

For more information on design rules we refer to [Schulte, 2012] and Part VII of [Alpert et al., 2008].

We can now define the Detailed Routing Problem.

Detailed Routing Problem
Input: A chip area $C = [x_1, x_2] \times [y_1, y_2] \times \mathcal{Z}$, a set of blockages B, a set of nets $\mathcal{N}$, a set of design rules, and an objective function.
Task: Find a wiring for each net $N \in \mathcal{N}$ such that the set consisting of all wirings, all pins, and all blockages satisfies the design rules, each shape is contained in the chip area, and the objective function is optimized.

The objective function could, for example, specify that the chip should be able to run at the highest possible clock frequency or consume as little power as possible. As an additional input a global routing solution may be provided. See Figure 2.4 for an illustration of part of a solution computed by BonnRoute.

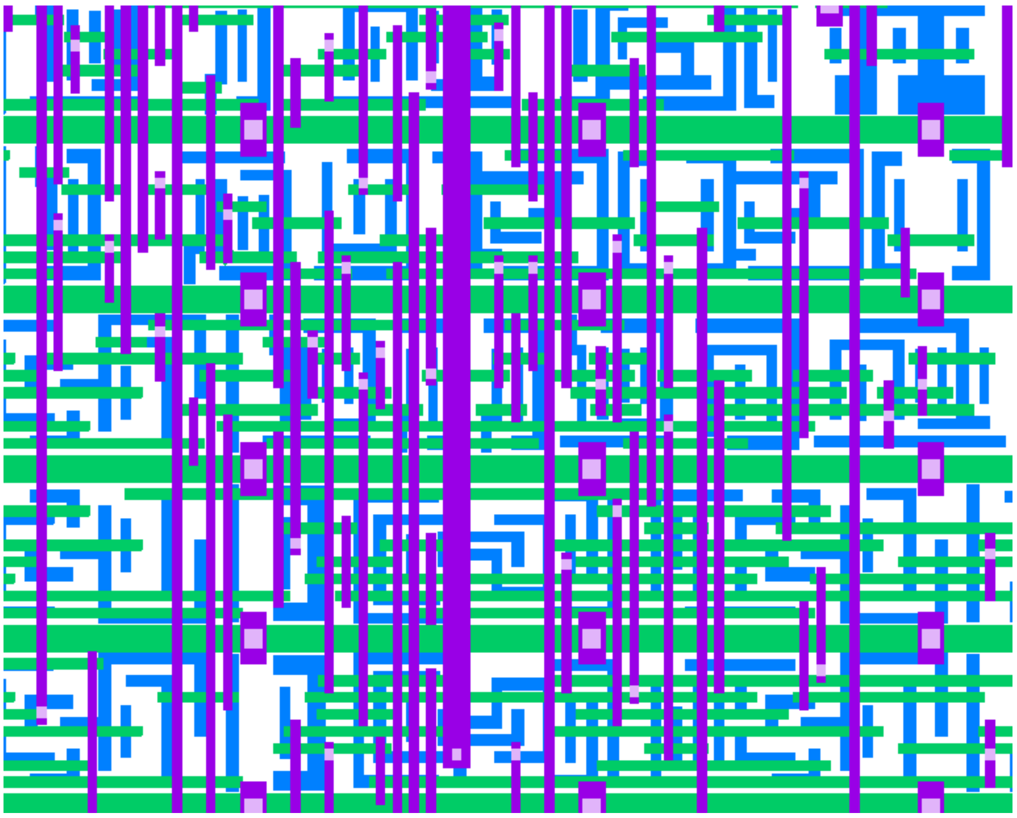


Figure 2.4: A small part of a detailed routing of a 14 nm instance showing only the lowest layers. Approximately $\frac{1}{165000}$ of the chip area is depicted. The lowest wiring layer which consists of pins and blockages is drawn in blue. The second and third lowest wiring layers are depicted in green and dark violet. Vias connecting the third and forth lowest wiring layer are also depicted.

The DETAILED ROUTING PROBLEM is hard to solve in theory and in practice and instances can have millions of nets. To make the problem more accessible many detailed routers make the following simplifications:

First, almost all sticks on each wiring layer run in the same dimension, called the preferred dimension. Before defining the preferred dimension we introduce some notation on dimensions and directions.

Definition 2.5 (dimensions, directions). *By $\leftrightarrow$, $\updownarrow$, and $\swarrow\!\!\!\nearrow$ we refer to the x-, y-, and z-dimension. We use $\rightarrow$, $\uparrow$, and $\nearrow$ to refer to the positive x-, y-, and z-direction and $\leftarrow$, $\downarrow$, and $\swarrow$ to we refer to the negative x-, y-, and z-direction. We define $\mathcal{D} := \{\rightarrow, \leftarrow, \uparrow, \downarrow, \nearrow, \swarrow\}$ as the set of directions. Let $v_1 = (x_1, y_1, z_1), v_2 = (x_2, y_2, z_2) \in \mathbb{R}^3$ be two different points such that the line segment $\overline{v_1 v_2}$ is axis-parallel. We define*

$$\mathrm{Dir}(v_1, v_2) := \begin{cases} \rightarrow & x_1 < x_2 \\ \leftarrow & x_1 > x_2 \\ \uparrow & y_1 < y_2 \\ \downarrow & y_1 > y_2 \\ \nearrow & z_1 < z_2 \\ \swarrow & z_1 > z_2 \end{cases}$$

and

$$\mathrm{Dim}(v_1, v_2) := \begin{cases} \leftrightarrow & x_1 \neq x_2 \\ \updownarrow & y_1 \neq y_2 \\ \swarrow\!\!\!\nearrow & z_1 \neq z_2. \end{cases}$$

Definition 2.6 (preferred dimension, horizontal and vertical layers). *The* preferred dimension *of a wiring layer $z \in \mathcal{Z}$ is defined by*

$$\mathrm{PrefDim}(z) := \begin{cases} \leftrightarrow & \text{if } z \text{ is even} \\ \updownarrow & \text{if } z \text{ is odd.} \end{cases}$$

A wiring layer $z \in \mathcal{Z}$ with $\mathrm{PrefDim}(z) = \leftrightarrow$ *is called a* horizontal layer *and a layer with* $\mathrm{PrefDim}(z) = \updownarrow$ *is called a* vertical layer.

The preferred dimensions alternate. Vertical sticks on a horizontal layer and horizontal sticks on a vertical layer are undesirable, since they could block many sticks running in preferred dimension. Such sticks are called *jogs.*

Second, almost all sticks that run in preferred dimension are placed on so-called tracks.

Definition 2.7 (track). *A* track *is an axis-parallel line segment $[x_1, x_2] \times [y_1, y_2] \times \{z\}$ with $x_1, x_2, y_1, y_2 \in \mathbb{Z}$ and $z \in \mathcal{Z}$ that runs in the preferred dimension of its layer.*

Note that tracks do not need to be uniform across a layer. See Figure 2.5 for an illustration why tracks can help using the routing space efficiently.

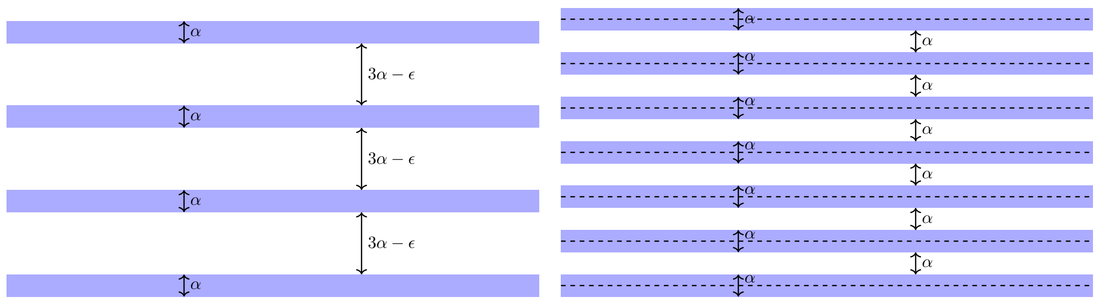


Figure 2.5: The most inefficient (left) and the most efficient (right) packing of wires with width α and spacing requirement α. On the left no additional wires can be added. The right also shows tracks that help packing.

Naturally, wires with different widths or with different spacing requirements need different tracks and optimizing tracks for multiple different widths and spacing requirements is a non-trivial problem. See Section 2.2.2 and [Klewinghaus, 2020] for a description of track optimization in BonnRoute. The tracks define a graph, called the track graph, that represents the routing resources of a chip.

Definition 2.8 (track graph)**.** *Let $T = T_0 \cup \cdots \cup T_{|\mathcal{Z}|-1}$ be a set of tracks such that the set of tracks on layer $z \in \mathcal{Z}$ is T_z. The* track graph *G_T for T is defined as follows:*

$$\begin{aligned} V(G_T) &= \bigcup_{z \in \mathcal{Z}} T_z \cap (P^+(T_{z-1}) \cup P^-(T_{z+1})), \\ E(G_T) &= \{\{v,w\} | \overline{vw} \textit{ is axis-parallel with } \overline{vw} \cap V(G_T) = \{v,w\}\}, \end{aligned}$$

where P^+ and P^- are the functions that map a track to the wiring layer above or below.

In the track graph there is a node wherever tracks on adjacent wiring layers would cross. Edges between nodes exist whenever the line segment connecting the nodes is axis-parallel and does not contain any other nodes. For an example of a track graph see Figure 2.6.

The track graph for all tracks represents the routing resources. Some edges may be unusable, e.g. because of blockages or existing wiring. Moreover, we are often interested in a subset of the routing resources. This can be modeled by routing graphs.

Definition 2.9 (routing graph)**.** *Let G_T be a track graph. A* routing graph *is a subgraph of G_T.*

We now define a simplified version of the Detailed Routing Problem and consider its complexity.

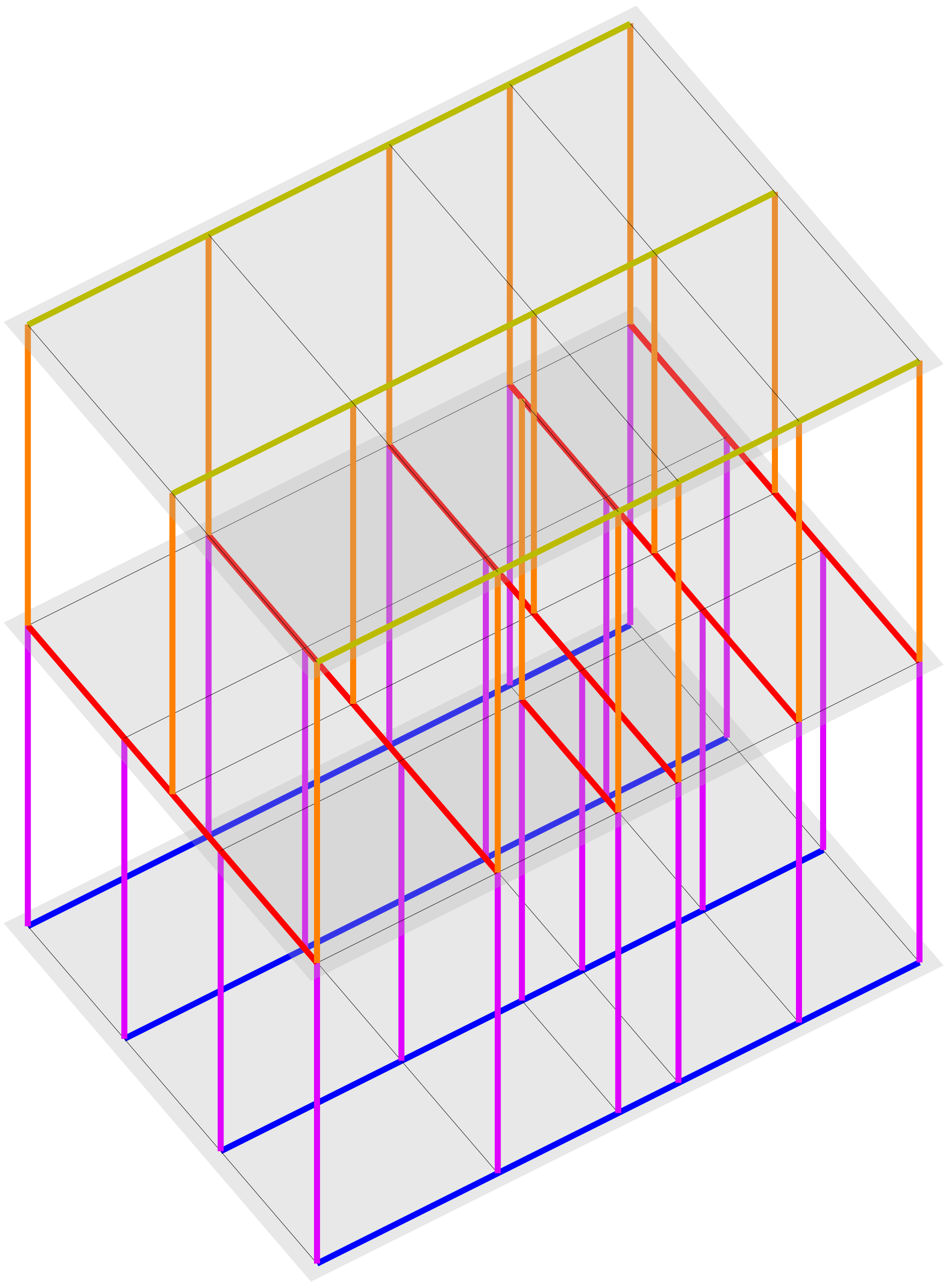


Figure 2.6: Track graph for the set of tracks T_0, T_1, and T_2 as depicted. Nodes exist wherever two or more line segments intersect. Edges in z-dimension are drawn in purple and orange and jog-edges are thin and black.

VERTEX-DISJOINT STEINER TREES PROBLEM IN 3-DIMENSIONAL GRID GRAPHS
Input: A three-dimensional grid graph G and disjoint sets $N_1, \dots, N_k \subseteq V(G)$.
Task: Compute node-disjoint Steiner trees $T_1, \dots, T_k$ in G such that T_i connects N_i for each $i \in \{1, \dots, k\}$ or decide that no such Steiner trees exist.

In the special case of two-dimensional grid graphs and $|N_i| = 2$ for $i \in \{1, \dots, k\}$ the VERTEX-DISJOINT STEINER TREES PROBLEM IN 3-DIMENSIONAL GRID GRAPHS simplifies to the VERTEX-DISJOINT PATHS PROBLEM in two-dimensional grid graphs, which is NP-hard.

Theorem 2.10 ([Kramer and van Leeuwen, 1982])**.** *The* VERTEX-DISJOINT PATHS PROBLEM *is NP-hard even in two-dimensional grid graphs.*

Corollary 2.11. *The* VERTEX-DISJOINT STEINER TREES PROBLEM IN 3-DIMENSIONAL GRID GRAPHS *is NP-hard.*

Practical instances of the DETAILED ROUTING PROBLEM can ask for millions of node-disjoint Steiner trees in a routing graph with hundreds of billions of nodes. Therefore, very efficient algorithms are required. For this reason detailed routers usually route nets one after the other, except for parallelization and conflict resolution. Hence, the main subroutine of most detailed routers is an algorithm for solving a version of the following problem:

NET ROUTING PROBLEM
Input: A routing graph G, an edge cost function $c_E : E(G) \to \mathbb{R}_{\geq 0}$, a node cost function $c_V : V(G) \to \mathbb{R}_{\geq 0}$, a number $k \in \mathbb{N}$, and sets $p_1, \dots, p_k \subseteq V(G)$.
Task: Find a Steiner tree S with $p_i \cap V(S) \neq \emptyset$ for $i \in \{1, \dots, k\}$ such that the cost $c_E(E(S)) + c_V(V(S))$ is minimal or decide that no such Steiner tree exists.

Most detailed routers build Steiner trees out of paths and thus their algorithmic core is a path search that solves the PATH SEARCH PROBLEM, a version of the NET ROUTING PROBLEM with $k = 2$. This is also the case for BonnRouteDetailed. We redesigned the path search of BonnRouteDetailed and reimplemented it from scratch. Chapter 3 and Chapter 4 describe the new path search and its implementation. Chapter 3 focuses on making the algorithm efficient while being able to handle complicated implicitly given grid graphs. This suffices to respect diff-net rules since they can be modeled by deleting edges. Chapter 4 focuses on computing paths that also respect the same-net rules. In Section 4.2 we prove that given a two-dimensional grid graph and nodes s, t it is NP-complete to decide whether there is an s-t-path in which each maximal straight subpath has length at least two. Since minimum segment length constraints are special cases of several same-net rules this shows that finding paths that obey even simple rules is NP-hard. The rest of Chapter 4 describes our framework for making our path search respect many same-net rules in practice. Section 3.6 summarizes the advantages of our path search compared to the previous path search based on [Hetzel, 1995, 1998]. Chapter 6 presents experimental results demonstrating the efficiency of our path search and its ability to respect design rules.

Composing Steiner trees of paths can lead to non-optimal solutions. In Chapter 5 we propose to solve a version of the Net Routing Problem with a single call of a generalized version of the Dijkstra-Steiner algorithm [Hougardy et al., 2017]. The algorithm allows us to find optimal Steiner trees that are similar to the global routing solution in terms of topology and approximate location of Steiner points, thereby recovering electrical and packing properties optimized in global routing. Moreover, the restrictions speed up the algorithm in theory and in practice.

2.2 Overview of BonnRoute

This section gives an overview of BonnRoute, a state-of-the-art router developed at the University of Bonn in joint work with IBM. BonnRoute is part of the BonnTools [Korte et al., 2007] that have been used to design more than one thousand chips over the last three decades. BonnRoute is the main routing tool used by IBM for the design of its processor chips. For a survey of BonnRoute see [Gester et al., 2013]. In BonnRoute the routing problem is solved in two stages: global routing and detailed routing.

This section provides background information on components of BonnRoute that are not discussed in detail in this thesis. Reading it can make it easier to understand the remaining chapters of this thesis, especially Chapter 3 and Chapter 6.

2.2.1 Global Routing with BonnRouteGlobal

BonnRouteGlobal computes an approximate layout of each net, optimizing complex objective functions such as timing, congestion, and wire length. Since optimizing these objectives is very difficult and instances can have millions of nets, global routing simplifies the problem by working on a coarse three-dimensional grid graph, called the global routing graph. Nodes of this graph represent rectangular parts of the chip area, called global routing tiles. The congestion is optimized to prevent that tiles contain too many wires and that there are too many wires on the edges between them. See Figure 2.7 for an illustration.

Since this thesis is about detailed routing we do not describe the algorithms used in BonnRouteGlobal. For details on BonnRouteGlobal we refer to [Müller, 2009; Müller et al., 2011; Held et al., 2018; Scheifele, 2019].

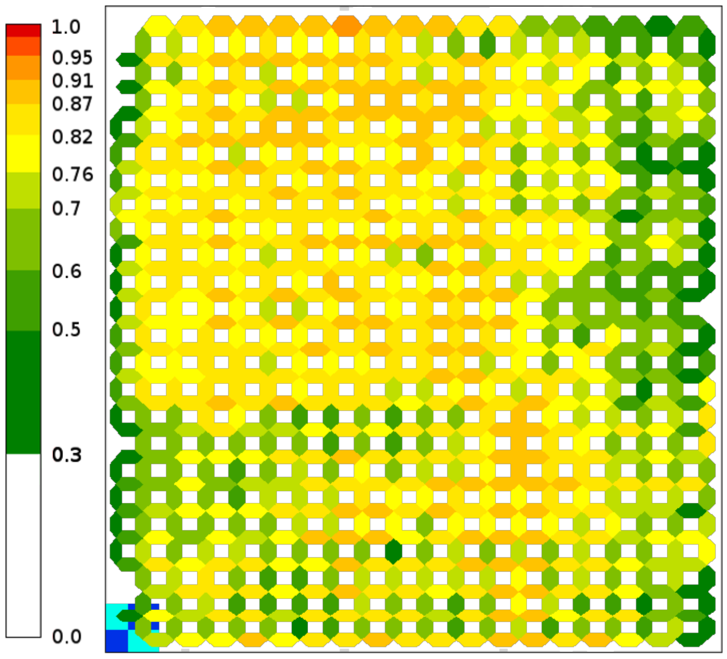


Figure 2.7: Maximum layer-wise edge congestion over all layers. The scale on the left shows which colors correspond to which fraction of used edge capacity. Four global routing tiles are sketched in cyan and blue in the lower left corner.

2.2.2 Detailed Routing with BonnRouteDetailed

BonnRouteDetailed computes the final wiring of a chip. For this reason, the simplifications used in global routing are no longer possible. Thus, millions of nets may have to be routed in a graph with hundreds of billions of nodes and hence very fast algorithms are required.

Algorithm 1 gives an overview of the most important steps of BonnRouteDetailed. First, there are some preparatory steps, such as optimizing the tracks of the given design, initializing data structures, and computing an order ϕ in which the nets should be routed. Then BonnRouteDetailed uses an algorithm based on dynamic programming to ensure that the lowest layers are used as efficiently as possible. Finally, in its main loop, BonnRoute-Detailed traverses all nets roughly in the order specified by ϕ and routes each of them. The following sections briefly describe each of these steps.

We note that the overview omits some important steps and that even the more detailed descriptions in the following sections are simplified. For example, the description omits reading the design rules and the design data. It also omits parallelization: All runtime-critical steps of BonnRouteDetailed are parallelized and [Klewinghaus, 2020] reports a speed-up of 31.1 on large 14 nm instances with 64 threads. See [Klewinghaus, 2013b, 2020] for more details on the parallelization. We refer to [Gester et al., 2013; Ahrens et al., 2015] for a more detailed but slightly outdated description of BonnRouteDetailed.

Algorithm 1 Overview of BonnRouteDetailed

```
1: compute optimized tracks
2: initialize the net data structure, the detailed grid, and the fast grid
3: compute an order $\phi$ of the nets
4: compute short connections and preselect pin access paths
5: while $\exists$ unconnected nets do
6:     pick the first unconnected net $N$ with respect to $\phi$
7:     while $N$ has at least two connected components $C_1, C_2$ do
8:         connect $C_1$ and $C_2$ with the same-net rule aware path search, using ripup and
   reroute if necessary
```

Track Optimization

Using optimized tracks makes it easier to use the available routing space efficiently. Most tracks in BonnRouteDetailed are *global tracks* that span the entire length or width of the chip. See Section 3.5 for some information on non-global tracks. BonnRouteDetailed uses dynamic programming to optimize global tracks. The tracks on different layers are optimized independently. Each wiring layer is partitioned into thin stripes of equal size by so-called power rails or power vias that are part of the power distribution. The tracks are optimized in one of the stripes and then repeated for the entire layer.

The following restrictions are imposed on the tracks for each width and spacing requirement:

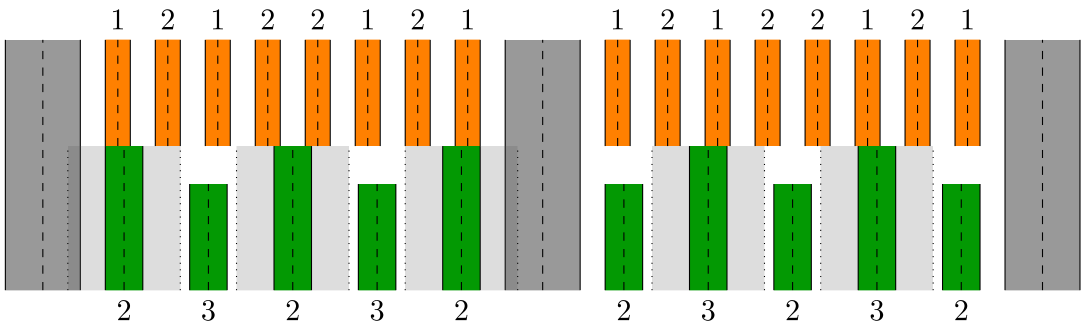
1. Adjacent tracks can be used without violating the diff-net rules.

2. Placing the tracks as in point 1 gives an upper bound on the number of tracks. The number of tracks must be close to this bound.

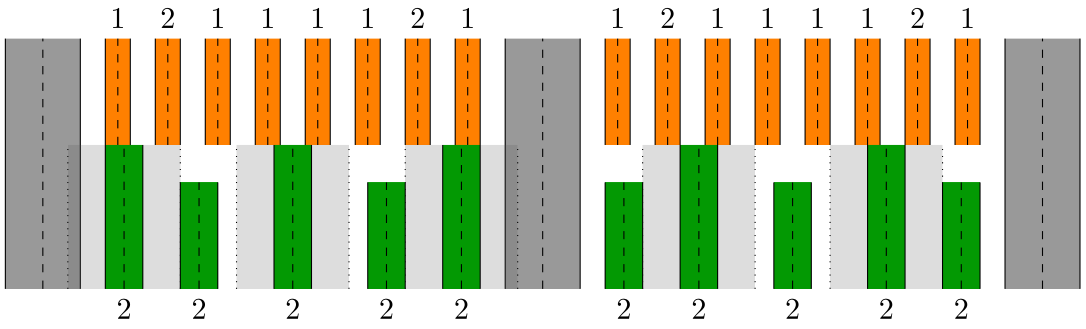
To maximize the available routing resources, the number of tracks should be maximized. On the other hand, placing a wire on one of its tracks should block as few other tracks as possible. Both the average number of blocked tracks and the number of available tracks are scaled with the frequency of the involved widths and spacing requirements. Figure 2.8 illustrates why optimizing tracks is important. It is a condensed version of multiple figures from [Klewinghaus, 2020].

We note that the dynamic programing approach cannot optimize the tracks for too many different widths and spacing requirements at the same time. If there are too many, the algorithm is first called with the most frequent ones. Later calls do not modify the tracks computed in previous calls but consider them in the objective.

See [Klewinghaus, 2020] for a complete description of the track optimization in BonnRoute-Detailed.



(a) Equidistantly spaced tracks for 1.5x wide wires with 1.5x spacing.



(b) Optimized tracks for 1.5x wide wires with 1.5x spacing.

Figure 2.8: Different options for tracks for 1.5x wide wires with 1.5x spacing (green). Power rails are drawn in dark gray. Minimum width wires (1x) are drawn in orange. Their tracks are arranged such that the maximum number of wires fits between the power rails, which defines a unique track pattern in this example. Light gray marks the area where a 1.5x wire blocks other wires and the numbers indicate how many 1x tracks are blocked by each 1.5x track (bottom) and vice versa (top). Using the optimized tracks improves the number of blocked tracks from 2.4 and 1.5 on average to 2 and 1.25 on average.

Important Data Structures

The *net data structure* stores the global routing of each net and, if the net is already partially or fully routed, its detailed routing. Moreover, it stores which parts of the global routing and detailed routing correspond to each other. This data is initialized at the beginning of BonnRouteDetailed and kept up-to-date during routing.

The global routing solution that BonnRouteDetailed gets as input optimizes complex objectives such as global congestion, timing, and wire length. The net data structure is used to attempt to route each net similarly to its global routing solution to benefit from this optimization. See Figure 2.9 for an illustration of a net after BonnRouteGlobal and after BonnRouteDetailed. For more details on the net data structure and how it is used we refer to Section 3.2 and [Klewinghaus, 2013a].

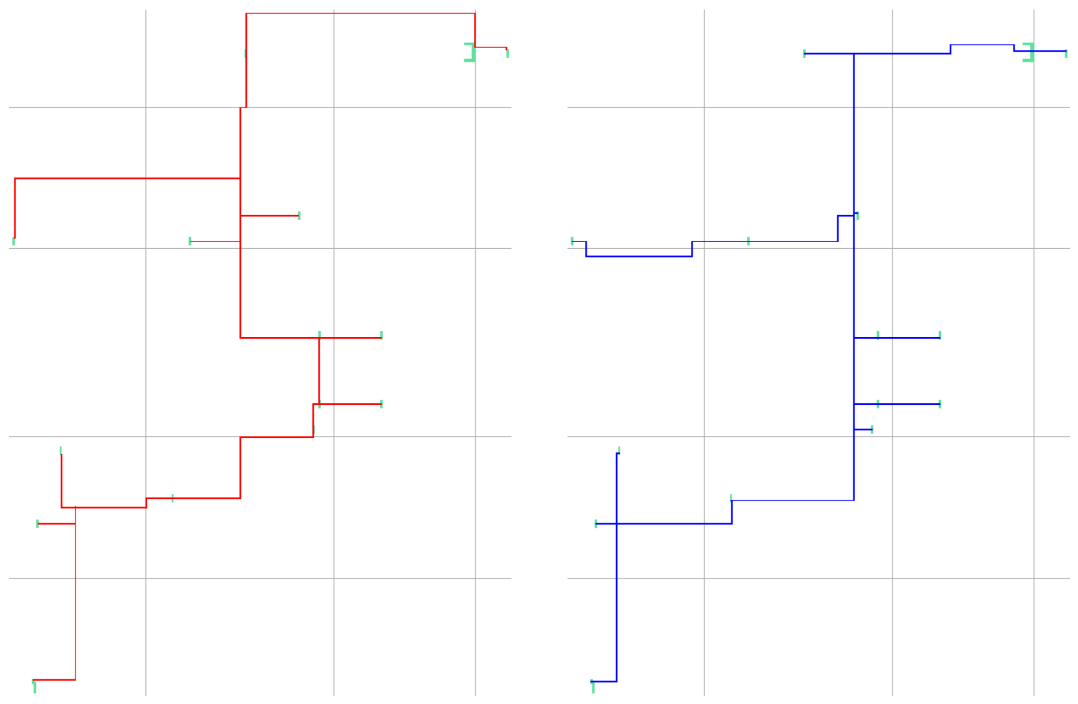


Figure 2.9: A global routing (left) and detailed routing (right) of the same net with 13 pins. The borders of the global routing tiles are drawn as gray lines.

The *detailed grid* stores all routing objects. It supports inserting and removing routing objects and querying all objects intersecting a rectangle. At the beginning of BonnRoute-Detailed the detailed grid is initialized with all routing objects that are given in the input and during BonnRouteDetailed it is always kept up-to-date using the insertion and the removal operation. The detailed grid can be used to check whether new shapes can be added to the existing routing objects without creating conflicts.

However, using the detailed grid for all these checking queries would be too slow. The *fast grid* is a specialized data structure that stores precomputed checking data for a subset of queries that are likely to occur during path searches.

For more details on the detailed grid and the fast grid and how they are used during the path search we refer to Section 3.5. See [Schulte, 2012; Klewinghaus, 2020] for further details on the detailed grid and [Müller, 2009; Klewinghaus, 2020] for more information on the fast grid.

Net Ordering

The order in which nets are routed has a large impact on the quality of the routing result. We describe the two most important criteria, but note that BonnRouteDetailed uses additional criteria.

The most important criterion is wire width and spacing requirement. Some nets require wide wires or wires with larger spacing requirements on some layers to achieve good timing properties. They are routed at the beginning, since routing them later on can be very difficult and may require revising many previously placed wires, especially on congested instances.

The second most important criterion is whether the nets are marked as timing-critical in the input. Generally speaking, nets that are routed early have fewer detours and their detailed routing is more similar to their global routing. Hence, timing-critical nets are routed early.

Computing Short Connections and Pin Access Paths

Typically most pins on a chip are located on the lowest layer, called the *pin layer*, and the second lowest layer, called the *access layer*. We assume that the access layer is partitioned into rows by power rails, which are parts of the power distribution.

There are two important optimization problems on these layers:

The first problem is to compute a short pin access path for each pin on the pin layer while respecting the design rules. The space for these access paths is reserved until the pin is connected to the rest of the net. This makes it unlikely that the pin will become fully blocked, which makes the routing problem much easier. The access paths should optimize certain objectives: For example, there may be a preferred direction for access paths derived from the global routing solution. Another objective is to optimize the spreading of access paths (and their endpoints), which makes it easier to connect to them later on.

The second problem is to connect nearby pins (on the pin layer and the access layer) using only vias and a wire on the access layer. Pins that are connected in this fashion do not need a pin access path. These connections consist of a single wire only and are thus very efficient.

The so-called *circuit row pin access* solves both problems simultaneously within an entire corridor between adjacent power rails, using dynamic programming. Under some mild assumptions, e.g. that the distance between adjacent power rails is fixed, its runtime is $O(n \log(n))$, where n is the number of pins.

See Figure 2.10 for an illustration of a simple instance of this problem and its solution and Figure 2.11 for a solution of a slightly larger instance.

[Ahrens, 2014; Ahrens et al., 2015] show that solving both problems with a single algorithm can lead to large improvements, e.g. a 5% reduction in the number of vias when compared to other approaches that do just pin access. For more details see [Ahrens, 2014; Ahrens et al., 2015]. Both Figures are from [Ahrens, 2014] and this description is a shortened version of the one in [Ahrens et al., 2015].

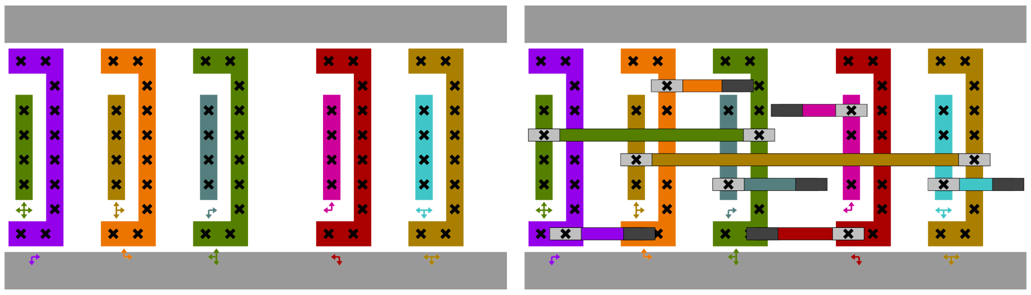


Figure 2.10: A small part of an instance of the generalized pin access problem (left) and its solution (right). The gray shapes are parts of the power rails that separate the instances. Colored shapes are pins and pins of the same color belong to the same net. Arrows indicate the preferred direction for the pin access path. The crosses show positions where we can place a via between pin and access layer to connect to a pin without causing design rule violations.

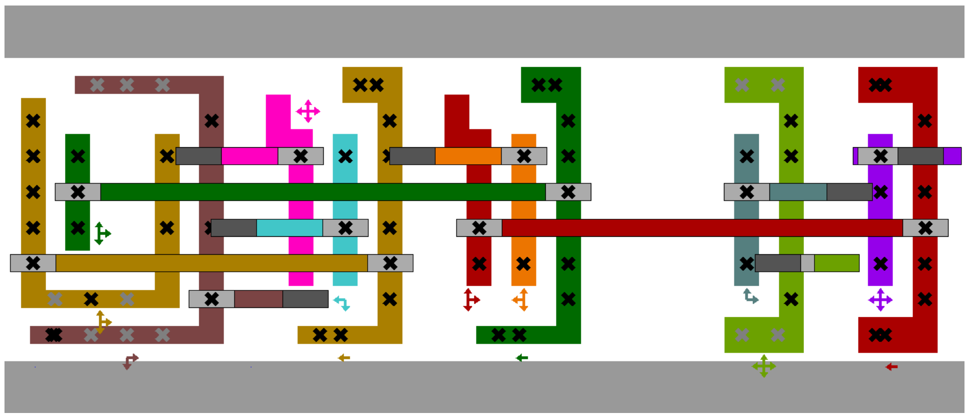


Figure 2.11: A solution to an instance of the generalized pin access problem. The solution highlights the spreading of the access paths. Note that access paths are well-spread in the sense that they are not on neighboring tracks whenever that is possible.

Routing a Net

BonnRouteDetailed routes each net by iteratively connecting two components of the net by a path. This process is repeated until the net is fully routed, i.e. just one component remains. We compute a path connecting two components with an efficient and same-net rule aware path search. The same-net rule aware path search is the main routine of BonnRouteDetailed and the most runtime-critical one: Typically 80-90% of the runtime are spent in the path search. It is described in detail in Chapter 3 and Chapter 4.

Sometimes previously routed connections block all paths between the two components. Then BonnRouteDetailed uses an approach called *ripup and reroute* to revise some of the earlier connections. In this case we start a path search that can use resources that are already blocked by other wires but at high cost. The conflicting parts of the other nets are then ripped out and must be reconnected. The path searches for reconnecting them can trigger additional ripups. This process is repeated until there are no parts that need to be reconnected or until a maximum number of steps is reached. In the latter case none of the modifications is realized and we connect the components with a path that can violate the diff-net rules. For more details on ripup and reroute in BonnRouteDetailed we refer to [Klewinghaus, 2013b].

Composing routings of nets of paths can lead to non-optimal solutions. Chapter 5 discusses the problem of routing an entire net with a single call of a modified version of the Dijkstra-Steiner algorithm. The project is still being developed and the algorithm is not yet used in BonnRouteDetailed in production.

Chapter 3

Efficient Path Search

This chapter and Chapter 4 describe an efficient and flexible path search algorithm that is the new algorithmic core of BonnRouteDetailed. This chapter focuses on making the algorithm efficient while retaining flexibility. Section 3.1 introduces the PATH SEARCH PROBLEM and gives an overview of important speed-up techniques introduced in previous works. Section 3.2 describes how we restrict each path search to a routing area derived from the global routing which speeds up the search and has additional benefits. In Section 3.3 we make the search goal-oriented by using future costs. Our future costs are computed by solving a geometric shortest path problem. We prove that this problem can be solved in $O(\log|T| + \log|\mathcal{Z}|)$ time after a polynomial time preprocessing, where T is the set of targets and $\mathcal{Z}$ is the set of wiring layers. Furthermore, we propose an algorithm for solving this problem in practice that was developed in joint work with Dorothee Henke and Jens Vygen. Section 3.4 introduces the grid region data structure which helps to speed up the path search and allows us to handle complicated implicitly given grid graphs provided by two oracle functions. Section 3.5 provides more information on the oracle functions. Finally, Section 3.6 summarizes the advantages of our implementation compared to the previous implementation based on [Hetzel, 1995, 1998]. Section 6.2 presents experimental results demonstrating the effectiveness of the speed-up techniques presented in this chapter.

3.1 Path Search in Huge Routing Graphs

Definition 3.1 (path search instance)**.** *A* path search instance *is a tuple* (G, c_E, c_V, S, T), *where* G *is a routing graph,* $c_E : E(G) \to \mathbb{R}_{\geq 0}$ *is an edge cost function,* $c_V : V(G) \to \mathbb{R}_{\geq 0}$ *is a node cost function, and* $S, T \subseteq V(G)$ *are sets of source and target nodes.*

Given an edge progression $P = v_1, \ldots, v_k$ *its*

- edge cost *is* $c_E(P) = \sum_{i \in \{1,\ldots,k-1\}} c_E(\{v_i, v_{i+1}\})$ *and its*
- node cost *is* $c_V(P) = \sum_{i \in \{1,\ldots,k\}} c_V(v_i)$.

Note that edges and nodes that are used multiple times in P need to be paid more than once.

PATH SEARCH PROBLEM	
Input:	A path search instance (G, c_E, c_V, S, T).
Task:	Find a shortest S-T-path P, i.e. a path minimizing $c_E(P) + c_V(P)$ or decide that no S-T-path exists.

Note that the PATH SEARCH PROBLEM is a special case of the NET ROUTING PROBLEM with two sets of nodes that need to be connected. A path search instance and its solution are illustrated in Figure 3.1.

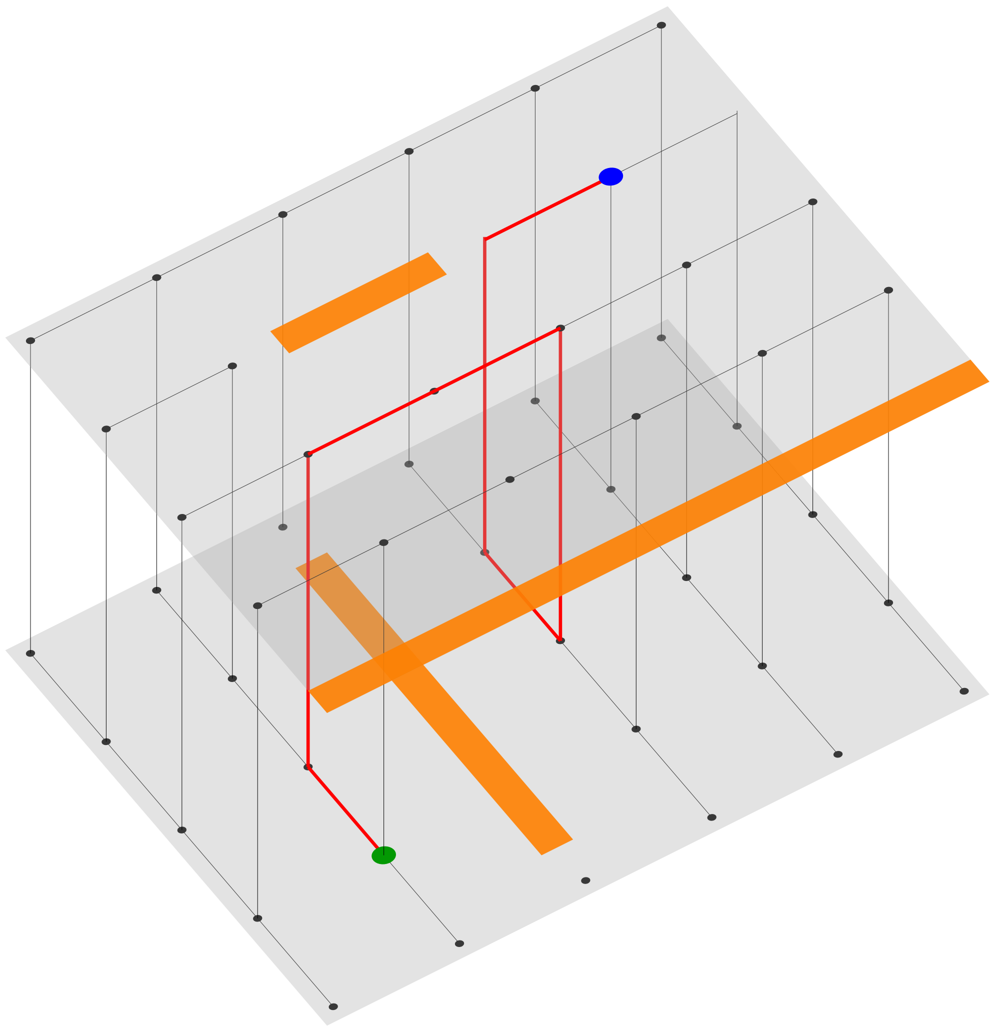


Figure 3.1: Visualization of a toy instance of the PATH SEARCH PROBLEM. The task is to find a shortest path between the source and the target. In this example jogs are forbidden and all edges have cost 1. Moreover, some nodes and edges are missing because of existing wires. A shortest path is also drawn.

Because the node and edge costs are non-negative, we can use Algorithm 2, called Dijkstra's algorithm, to solve the problem. While there are many other algorithms for finding shortest paths, Dijkstra's algorithm is a natural choice for solving this problem in practice (see e.g. [Cherkassky et al., 1996]).

Algorithm 2 Dijkstra's Algorithm ([Dijkstra, 1959])

Input: A graph $G = (V, E)$, edge costs $c_E : E \to \mathbb{R}_{\geq 0}$, node costs $c_V : V \to \mathbb{R}_{\geq 0}$, and sets of source and target nodes $S, T \subseteq V$.
Output: A shortest S-T-path or the information that no such path exists.

```
1: $\forall s \in S$ set $l(s) \leftarrow c_V(s)$, $\forall v \in V \setminus S$ set $l(v) \leftarrow \infty$, $\forall v \in V$ set $p(v) \leftarrow \emptyset$, set $R \leftarrow \emptyset$
2: while $\exists v \in V \setminus R$ s.t. $l(v) \neq \infty$ do
3:     $v \leftarrow \operatorname{argmin}_{w \in V \setminus R} l(w)$
4:     $R \leftarrow R \cup \{v\}$
5:     if $v \in T$ then
6:         return $\Big(s = p(\ldots p(p(v))\ldots), \ldots, p(p(v)), p(v), v\Big)$ with $p(s) = \emptyset$
7:     for all $w \in V \setminus R$ with $\{v, w\} \in E$ do
8:         if $l(w) > l(v) + c_E(\{v, w\}) + c_V(w)$ then
9:             $l(w) \leftarrow l(v) + c_E(\{v, w\}) + c_V(w)$
10:            $p(w) \leftarrow v$
11: return No $S$-$T$ path exists.
```

The algorithm maintains a set of tentative distances l from S to the nodes in the graph, corresponding to the distance of a shortest path to that node that was found so far. A tentative distance of ∞ means that so far no path to that node was found. During the algorithm, the tentative distances decrease and a set of nodes R, for which the tentative distance was proven to be the real distance, is maintained. In each iteration of the main loop, a node in $V \setminus R$ with minimal tentative distance is selected and added to R, i.e. its tentative distance is no longer tentative. Once a target is selected, a shortest S-T-path has been determined and is returned.

We say that a node v is *labeled* by Algorithm 2 if it was assigned a non-infinite distance $l(v) \neq \infty$ and that a node is *labeled permanently* if it is added to R.

Theorem 3.2 ([Dijkstra, 1959])**.** *Dijkstra's algorithms works correctly.*

The tentative distances are organized in a priority queue.

Theorem 3.3 ([Fredman and Tarjan, 1987])**.** *When the priority queue is implemented with a Fibonacci heap, Dijkstra's algorithm runs in $O(|E| + |V| \log|V|)$ time.*

Routing graphs satisfy $|E| \leq 3|V|$ since each node has at most six neighbors, hence the runtime simplifies to $O(|V| \log|V|)$. Instead of using a Fibonacci heap we use a binary heap (see e.g. [Cormen et al., 2009]), which achieves the same theoretical runtime bound on routing graphs.

Using Dijkstra's algorithm in its original form would be too slow: On large chips the path search is called tens of millions of times and the routing graph can have hundreds of billions of nodes. Most of the sections of this chapter discuss speed-up techniques that improve runtime and memory usage in practice:

- The global routing solution specifies an approximate layout of each net. We restrict the path search to a *routing area* derived from the global routing solution. This also has other advantages and is discussed in Section 3.2.
- A goal-oriented search based on *future costs* reduces the number of labels created by Dijkstra's algorithm and is discussed in Section 3.3.
- Working on an *implicitly given routing graph* reduces the memory usage by storing data only for nodes that were labeled. Partitioning the routing area into *grid regions* and precomputing certain data for these regions allows us to speed up the search substantially. This is discussed in Section 3.4.
- Section 3.4 introduces two oracle functions: the track oracle and the checking oracle. Section 3.5 sketches how these data structures are implemented in BonnRouteDetailed.

We now discuss other speed-up techniques that we do not use and explain why they are not considered. See [Wagner and Willhalm, 2007] for a collection of speed-up techniques for Dijkstra's algorithm. For a survey of approaches and algorithms that rely on preprocessing see [Sommer, 2014].

The path search that was previously used in BonnRouteDetailed covers the nodes on each layer $z \in \mathcal{Z}$ by a set of closed intervals $\mathcal{I}_z$ such that:

1. Each interval $I \in \mathcal{I}_z$ is a point or runs in preferred dimension.
2. Each node on the layer z is contained in one of the intervals of $\mathcal{I}_z$.
3. Different intervals $I, I' \in \mathcal{I}_z$ are disjoint, i.e. $I \cap I' = \emptyset$.
4. $v, w \in I \in \mathcal{I}_z$ implies that the straight wire connecting v and w is legal and that the cost of going from v to w is $\|v - w\|$.

The algorithm then labels intervals instead of individual nodes. A label at point p with cost c in an interval I then implies that any point $p' \in I$ can be reached with cost $c + \|p - p'\|$. Each interval maintains a set of non-dominated labels and may be partially or fully permanently labeled. This approach achieves a runtime of $O(\min\{d|\mathcal{I}|\log|\mathcal{I}|, |V|\log|V|\})$, where d is the detour of the shortest path compared to the future cost (or ∞ if no path exists), $\mathcal{I} = \cup_{z\in\mathcal{Z}}\mathcal{I}_z$ is the set of all intervals, and V is the set of all nodes. Thus the algorithm achieves better runtime than Dijkstra's Algorithm if the number of intervals and the detour are small and the same worst case runtime otherwise. For more details on this approach we refer to [Hetzel, 1995, 1998; Peyer et al., 2009; Gester et al., 2013].

Note that Property 4 is very restrictive since it implies that the cost for going one unit in preferred dimension is 1 on every interval irrespective of the layer. It is not clear whether it is possible to relax this assumption without making the interval labeling much less effective. See Figure 3.2 for an illustration.

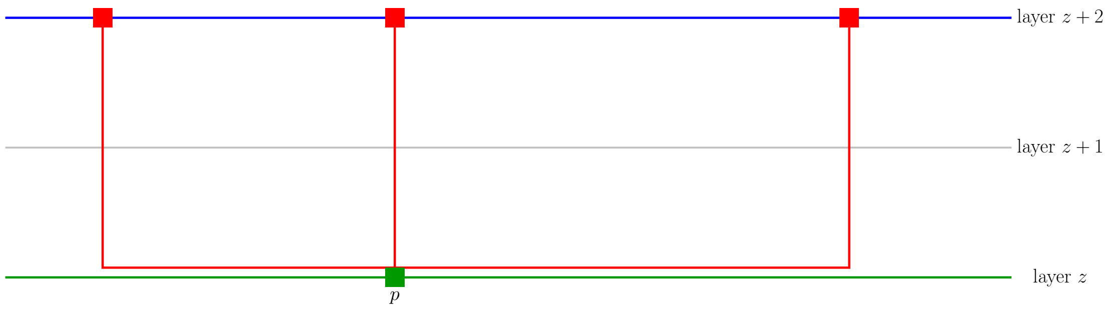


Figure 3.2: Consider an interval I_z on layer z and an interval $I_{z+2} = I_z + \{0\} \times \{0\} \times \{2\}$ two layers above. Assume that the cost for going one unit on I_z is strictly smaller than on I_{z+2}. Further assume that wires are allowed everywhere on these intervals and that vias on $(z, z+1)$ and $(z+1, z+2)$ can be placed everywhere. Then the shortest path from the square label at the bottom to any point $(x, y, z+2) \in I_{z+2}$ consists of a segment from p to (x, y, z) on I_z and two vias. Three such paths are drawn in red. Thus a single label on I_z could lead to an arbitrarily high number of labels on I_{z+2}.

Also note that node costs do not combine well with interval labeling, since nodes with non-zero node cost need their own interval. These restrictions were some of the main reasons why we decided to label individual nodes instead of intervals in the new implementation. See Section 3.6 for a summary of the advantages of our implementation compared to the previous implementation based on [Hetzel, 1995, 1998].

Bidirectional search is a well-known technique for speeding up Dijkstra's algorithm. The idea is to run two searches simultaneously: A forward search from source to target and a backward search from target to source, each maintaining their own distance labels. A shortest path is found when there are nodes that were labeled permanently by both searches and a non-trivial stopping criterion is met. See Figure 3.3 for an illustration of why bidirectional search can reduce the number of labels. See [Nicholson, 1966; Dreyfus, 1969] for more details and e.g. [Goldberg and Werneck, 2005] for a simple and good stopping criterion.

It is unclear whether bidirectional search is worth implementing in our application. First, combining bidirectional search and goal-oriented search requires special care and restricts the possible choices of future cost functions (see e.g. [Ikeda et al., 1994], Section 3-4), reducing the effectiveness of the goal-oriented search. Moreover, there is often a natural direction in which the search is run, since the target component typically consists of a single pin and the source component can consist of multiple pins and paths connecting them. Furthermore, combining bidirectional search with the same-net rule aware search introduced in Chapter 4 would complicate the overall algorithm. For these reasons we do not consider bidirectional search here. Nevertheless it might help to improve runtime.

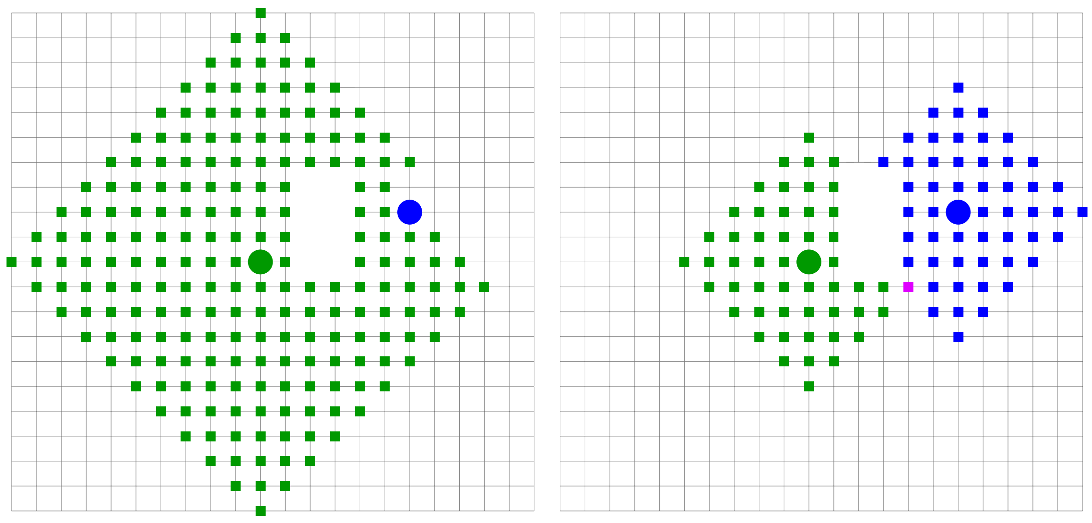


Figure 3.3: The permanent labels (squares) created by a unidirectional search (left) and a version of the bidirectional search (right) from the source (green circle) to the target (blue circle). Permanent labels are colored green if they derive from the source and blue if they derive from the target. The purple point was permanently labeled from both source and target. The unidirectional search requires 205 permanent labels while the bidirectional search needs 104 labels: 51 from the source and 53 from the target.

Another well-known speed-up technique is based on the concept of *vertex reaches* which was introduced in [Gutman, 2004] in the context of road networks. A node can be excluded from a search if its distance to the source plus its distance to the target is larger than its reach. In most cases, nodes on highways have large reach and nodes on small local roads have low reach. The vertex reaches are computed in a preprocessing step. [Goldberg et al., 2006] refine this approach by introducing shortcut edges (v, w) with cost $\text{dist}(v, w)$ between selected pair of nodes. These shortcuts can decrease the reach values of other nodes and thus speed up the algorithm. For example, connecting far apart nodes v and w on the same highway with a shortcut might decrease the reach values of the other nodes on the highway between v and w.

These techniques rely on time-intensive preprocessing which is feasible in the context of road networks. In our application the graph can change for every path search thus such approaches are not feasible. [Bast et al., 2016] surveys algorithms for route planning in road and transportation networks.

3.2 Routing Area

The global routing that BonnRouteDetailed gets as input optimizes complex objectives such as global congestion, timing, and wire length. BonnRouteDetailed attempts to realize each net similar to its global routing to benefit from this optimization. Restricting the path searches to corridors derived from the global routing also improves runtime.

Before starting a path search to connect two components of a net, BonnRouteDetailed computes the part of the global routing that corresponds to this search. The intersected global routing tiles and tiles on neighboring layers form the routing area. This data is computed using the *net data structure* that stores the global routing of each net and, if the net is already partially or fully routed, its detailed routing. Moreover, it stores which parts of the global routing and detailed routing correspond to each other. For more details on this data structure and how it is updated and used during detailed routing we refer to [Klewinghaus, 2013a].

Definition 3.4 (routing area)**.** *A* routing area *is the union of finitely many interior-disjoint axis-parallel rectangles on wiring layers.*

For an illustration see Figure 3.4 and Figure 3.5.

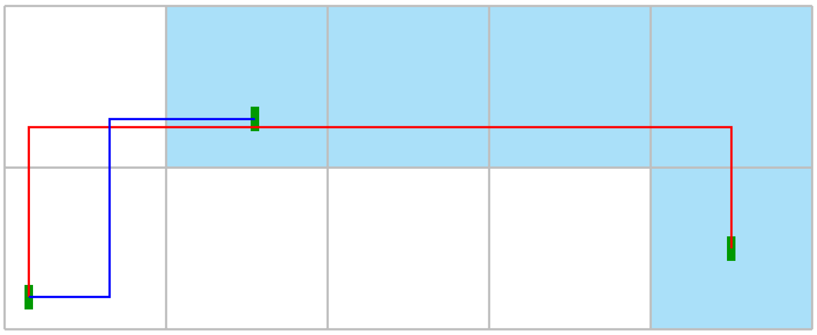


Figure 3.4: The two-dimensional projection of the routing area for connecting the pin on the right to the rest of the net. There are some existing detailed wires of the net. The routing area is derived from the part of the global routing that corresponds to the connection to the right. See Figure 3.5 for a three-dimensional illustration.

The path search is not applied to the routing graph of the entire chip but to its restriction to the routing area. This speeds up the algorithm.

Definition 3.5 (restricting a graph to the routing area)**.** *Let G be a routing graph and let A be a routing area. We define the* restriction G^A of G to A *by*

$$\begin{aligned} V(G^A) &= \{v \in V(G) | v \in A\} \\ E(G^A) &= \{\{(x,y,z),(x',y',z')\} \in E(G) | z = z' \text{ and } \overline{(x,y,z)(x',y',z')} \subseteq A\} \\ &\quad \cup \{\{(x,y,z),(x',y',z')\} \in E(G) | z \neq z' \text{ and } (x,y,z),(x',y',z') \in A\} \end{aligned}$$

We sometimes use G_T^A to refer to the track graph G_T after it is restricted to the routing area A. Note that this is covered by the definition, since the track graph is a routing graph.

In practice the routing area and the way it is computed are slightly more complicated: First, BonnRouteDetailed uses different routing areas for different wire models, for example a

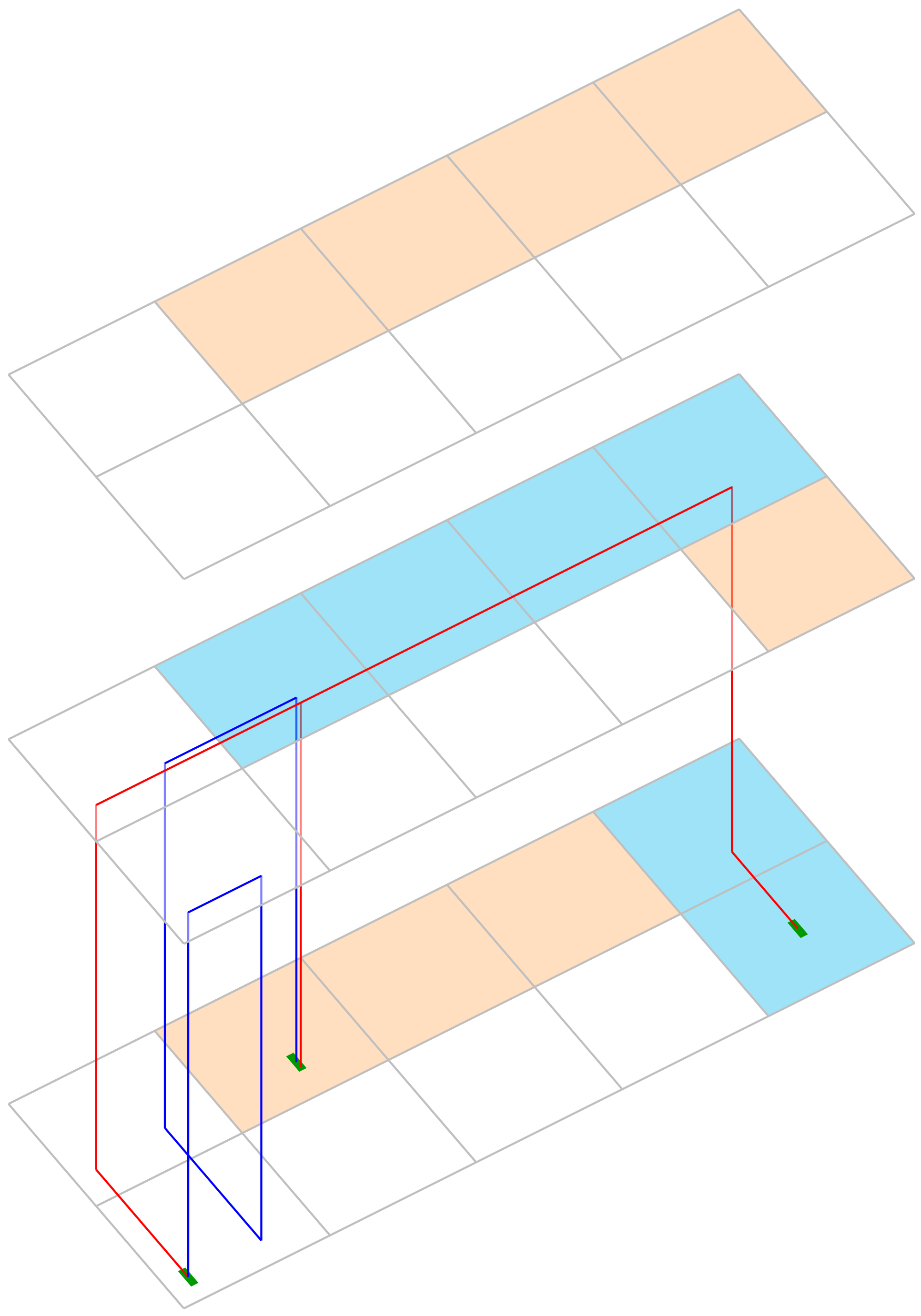


Figure 3.5: The routing area for connecting the pin on the right to the rest of the net. There are some existing detailed wires of the net. We call the set of tiles intersected by the relevant part of the global routing the base routing area. Tiles on neighboring layers are also included to allow for local track changes without jogs. The routing area consists of all colored tiles. The example assumes that the pins are on the lowest usable layer. See Figure 3.4 for a two-dimensional illustration.

small wire model may be allowed only near a sink pin. Second, BonnRouteDetailed cannot always find a solution within the original routing area and then the area is expanded.

3.3 Goal-Oriented Search Using Future Costs

This section describes a speed-up technique for Dijkstra's algorithm that guides the search towards the targets. It is based on the A* search that was proposed in [Hart et al., 1968] in the context of robot motion planning. It was proposed in the context of routing in [Rubin, 1974] and has since been used by different authors, see e.g. [Hetzel, 1998; Gester et al., 2013]. Parts of the work presented in this section were done in joint work with Dorothee Henke and Jens Vygen and are also presented in [Henke, 2016]. For the lemmas and theorems that also appear in [Henke, 2016] both the results and the proofs are very similar, unless indicated otherwise.

In our application it is natural to use so-called future costs, a concept similar to feasible potentials.

Definition 3.6 (future cost function, reduced costs)**.** *Let $G = (V, E)$ be a graph, $c_E : E \to \mathbb{R}_{\geq 0}$ be an edge cost function and $T \subseteq V$ be a set of target nodes. A* future cost function *for G, c_E, and T is a function $f : V \to \mathbb{R}_{\geq 0}$ with the following properties:*

1. *For every $t \in T$ we have $f(t) = 0$.*
2. *For every $\{v, w\} \in E$ we have $c_E(\{v, w\}) - f(v) + f(w) \geq 0$ and $c_E(\{v, w\}) - f(w) + f(v) \geq 0$.*

For a set of edges E we use $\overleftrightarrow{E} := \{(v, w), (w, v) \mid \{v, w\} \in E\}$ to refer to the set that contains both directed versions of each edge. The reduced costs *$c^f_E : \overleftrightarrow{E} \to \mathbb{R}_{\geq 0}$ for a future cost function f are defined by $c^f_E((v, w)) := c_E(\{v, w\}) - f(v) + f(w)$ for all $(v, w) \in \overleftrightarrow{E}$.*

Note that property 2 ensures that the reduced costs are non-negative. The property that gives future cost functions their name is that each v-T-path has cost at least $f(v)$, i.e. we need to pay at least $f(v)$ to get from v to a target. This holds because for any v-T-path $P = (v_1 = v, v_2, \ldots, v_k = t)$ with $t \in T$ we have

$$c_E(P) = \sum_{i \in \{1,\ldots,k-1\}} c_E(\{v_i, v_{i+1}\}) \geq \sum_{i \in \{1,\ldots,k-1\}} f(v_i) - f(v_{i+1}) = f(v_1) - f(t) = f(v).$$

Algorithm 3 is a version of Dijkstra's algorithm with future costs. Differences to Algorithm 2 are marked in red.

We can use the correctness of Dijkstra's algorithm to show that Algorithm 3 works correctly. The following result is well-known.

Corollary 3.7. *Algorithm 3 works correctly. If the future cost function can be evaluated in time $O(F)$, it has runtime $O(|V|(\log|V| + F) + |E|)$.*

Algorithm 3 Dijkstra's Algorithm with Future Costs

Input: A graph $G = (V, E)$, edge costs $c_E : E \to \mathbb{R}_{\geq 0}$, node costs $c_V : V \to \mathbb{R}_{\geq 0}$, sets of source and target nodes $S, T \subseteq V$, and a future cost function $f : V \to \mathbb{R}_{\geq 0}$.

Output: A shortest S-T-path or the information that no such path exists.

```
1:  $\forall s \in S$ set $l(s) \leftarrow c_V(s) + f(s)$, $\forall v \in V \setminus S$ set $l(v) \leftarrow \infty$, $\forall v \in V$ set $p(v) \leftarrow \emptyset$, set
    $R \leftarrow \emptyset$
2:  while $\exists v \in V \setminus R$ s.t. $l(v) \neq \infty$ do
3:      $v \leftarrow \operatorname{argmin}_{w \in V \setminus R} l(w)$
4:      $R \leftarrow R \cup \{v\}$
5:      if $v \in T$ then
6:          return $\big(s = p(\dots p(p(v))\dots), \dots, p(p(v)), p(v), v\big)$ with $p(s) = \emptyset$
7:      for all $w \in V \setminus R$ with $\{v, w\} \in E$ do
8:          if $l(w) > l(v) + c_E^f((v, w)) + c_V(w)$ then
9:              $l(w) \leftarrow l(v) + c_E^f((v, w)) + c_V(w)$
10:             $p(w) \leftarrow v$
11: return No $S$-$T$-path exists.
```

Proof. The algorithm is equivalent to running Dijkstra's algorithm in the directed graph $\overleftrightarrow{G} = (V, \overleftrightarrow{E})$ in which each edge is replaced by two opposing edges with edge costs c_E^f and with node costs c'_V given by

$$c'_V(v) = \begin{cases} c_V(v) & v \notin S \\ c_V(v) + f(v) & v \in S. \end{cases}$$

Thus we get a shortest S-T-path with respect to c'_V and c_E^f. Correctness follows because the cost of an s-T-path $P = (v_1 = s, v_2, \dots, v_k = t)$ with $v_2, \dots, v_k \in V(G) \setminus S$ is

$$\begin{aligned} c_E(P) + c_V(P) &= \sum_{i \in \{1,\dots,k-1\}} c_E(\{v_i, v_{i+1}\}) + c_V(P) \\ &= f(s) - f(t) + \sum_{i \in \{1,\dots,k-1\}} (c_E(\{v_i, v_{i+1}\}) - f(v_i) + f(v_{i+1})) + c_V(P) \\ &= f(s) + \sum_{i \in \{1,\dots,k-1\}} c_E^f((v_i, v_{i+1})) + c_V(P) \\ &= c_E^f(P) + c'_V(P). \end{aligned}$$

After evaluating the future costs of a node we store it. This ensures that the future cost function is called at most once for each node which implies the runtime bound. □

Note that Algorithm 3 with $f \equiv 0$, which is a future cost function, is equivalent to Algorithm 2. To reduce the number of labels it is desirable to have future cost functions that are as large as possible, as indicated by the next lemma.

Lemma 3.8 (similar to [Henke, 2016], Lemma 1.2)**.** *Let (G, c_E, c_V, S, T) be a path search instance. Consider a version of Algorithm 3 that chooses nodes $v \notin T$ before nodes $v \in T$ in line 3. Then for future cost functions $f_1 \geq f_2$ the algorithm with f_1 creates at most as many permanent labels as with f_2.*

Proof. If there is no s-t-path, all nodes in the connected component of s are labeled with both future costs functions.

Otherwise let l^* be the cost of the target that was selected right before the algorithm returns. Note that l^* is the cost of the shortest S-T-path and that l^* is independent of the future cost function. For $i \in \{1, 2\}$ consider the set of non-target nodes $L_i \subseteq V \setminus T$ that are selected in line 3 of Algorithm 3 with future cost function f_i.

We have

$$\begin{aligned} L_i = \{&v \in V \setminus T | \exists s \in S, \text{ an } s\text{-}v\text{-path } P \text{ with} \\ &f_i(s) + c_E^{f_i}(P) + c_V(P) \leq l^* \text{ and } V(P) \cap T = \emptyset\} \\ = \{&v \in V \setminus T | \exists S\text{-}v\text{-path } P \text{ with } f_i(v) + c_E(P) + c_V(P) \leq l^* \text{ and } V(P) \cap T = \emptyset\} \end{aligned}$$

Since $f_1 \geq f_2$ we have $L_1 \subseteq L_2$. This concludes the proof since there are $|L_1| + 1$ and $|L_2| + 1$ permanent labels with f_1 and f_2 respectively. □

Lemma 3.8 does not hold if we remove the condition that nodes in T are chosen as late as possible, but in practice larger future costs typically lead to fewer labels.

For all future costs that we introduce we use the following lemma to show that they are indeed future cost functions.

Lemma 3.9 (similar to [Henke, 2016], Lemma 2.1)**.** *Let G be a graph and $c_E : E(G) \to \mathbb{R}_{\geq 0}$ be an edge cost function. Let G' be a supergraph of G and $c'_E : E(G') \to \mathbb{R}_{\geq 0}$ be an edge cost function on G' with $c'_E(e) \leq c_E(e)$ for each $e \in E(G)$. Then for any $T \subseteq V(G)$ the function $\mathrm{dist}^{G'}_{c'_E}(., T)$ is a future cost function for G, c_E, and T.*

Proof. Clearly, $\mathrm{dist}^{G'}_{c'_E}(t, T) = 0$ for all $t \in T$.

For $\{v, w\} \in E(G)$ we have

$$\begin{aligned} &c_E(\{v, w\}) - \mathrm{dist}^{G'}_{c'_E}(v, T) + \mathrm{dist}^{G'}_{c'_E}(w, T) \\ \geq &c'_E(\{v, w\}) - \mathrm{dist}^{G'}_{c'_E}(v, T) + \mathrm{dist}^{G'}_{c'_E}(w, T) \\ \geq &0 \end{aligned}$$

because of the triangle inequality. Thus $\mathrm{dist}^{G'}_{c'_E}(., T)$ is indeed a future cost function. □

By the lemma, knowing lower bounds for edge costs suffices to compute future costs. One component of the edge costs in BonnRouteDetailed is distance based and we use this part for computing the future costs.

Definition 3.10 (rectilinear edge). *A* rectilinear edge *is a pair (v, w) with $v, w \in \mathbb{Z} \times \mathbb{Z} \times \mathcal{Z}$ and $v \neq w$ such that the line segment $\overline{vw}$ is axis-parallel.*

Definition 3.11 (distance based cost function, compatible). *A* distance based cost function *is a pair $c = (\mathrm{c}^{\mathrm{wire}}, \mathrm{c}^{\mathrm{viadown}})$, where $\mathrm{c}^{\mathrm{wire}} : \mathcal{Z} \times \{\leftrightarrow, \updownarrow\} \to \mathbb{R}_{\geq 0}$ and $\mathrm{c}^{\mathrm{viadown}} : \mathcal{Z} \setminus \{0\} \to \mathbb{R}_{\geq 0}$. The* cost of a rectilinear edge $((v_x, v_y, v_z), (w_x, w_y, w_z))$ *is*

$$\mathrm{cost}^c((v_x, v_y, v_z), (w_x, w_y, w_z)) = \begin{cases} |v_x - w_x| \, \mathrm{c}^{\mathrm{wire}}(v_z, \leftrightarrow) & v_x \neq w_x \\ |v_y - w_y| \, \mathrm{c}^{\mathrm{wire}}(v_z, \updownarrow) & v_y \neq w_y \\ \sum_{z=1+\min\{v_z, w_z\}}^{\max\{v_z, w_z\}} \mathrm{c}^{\mathrm{viadown}}(z) & v_z \neq w_z. \end{cases}$$

Let G be a routing graph and let $c_E : E(G) \to \mathbb{R}_{\geq 0}$ be an edge cost function. We say that c is compatible with G and c_E *if we have $\mathrm{cost}^c(e) \leq c_E(e)$ for each edge $e \in E(G)$.*

We note that in BonnRouteDetailed different calls of the path search may use different distance based cost functions. Moreover, additional costs apply if the edge is unfavorable, e.g. because using it requires using a non-preferred wire or via model, but we do not account for that in the future costs.

Before introducing different future cost functions, we note that future costs that rely on expensive preprocessing of the graph like the landmark based approach of [Goldberg and Harrelson, 2005] do not work well in our application, since the routing graph can change millions of times during detailed routing.

3.3.1 Considering Each Dimension Separately

The first option is to consider each dimension in isolation and derive lower bounds for each of them. The sum of these bounds is used as a future cost function, called *l_1-future cost.* This approach was introduced in [Hetzel, 1995, 1998]. We will consider l_1-future costs in the experiments in Section 6.2.

Lemma 3.12. *Let $c = (\mathrm{c}^{\mathrm{wire}}, \mathrm{c}^{\mathrm{viadown}})$ be a distance based cost function with $\mathrm{c}^{\mathrm{wire}} \geq \alpha$ for some $\alpha \in \mathbb{R}_{\geq 0}$ and let G, c_E be a graph and an edge cost function that are compatible with c. Then for $T \subseteq V(G)$ the function $f : V(G) \to \mathbb{R}_{\geq 0}$ with*

$$f((v_x, v_y, v_z)) = \min_{(t_x, t_y, t_z) \in T} \alpha |v_x - t_x| + \alpha |v_y - t_y| + \sum_{z=1+\min\{v_z, t_z\}}^{\max\{v_z, t_z\}} \mathrm{c}^{\mathrm{viadown}}(z)$$

for all $(v_x, v_y, v_z) \in V(G)$ is a future cost function for G, c_E, and T.

Proof. Let G' be the Hanan grid of $V(G)$ with edge cost function c' defined by $c'(e) = \mathrm{cost}^{(\alpha, \mathrm{c}^{\mathrm{viadown}})}(e)$ for $e \in E(G')$. Then $f(v) = \mathrm{dist}_{c'}^{G'}(v, T)$ for all $v \in V(G)$. The result follows from Lemma 3.9. □

See Figure 3.6 for an illustration of the labels the l_1-future costs can save.

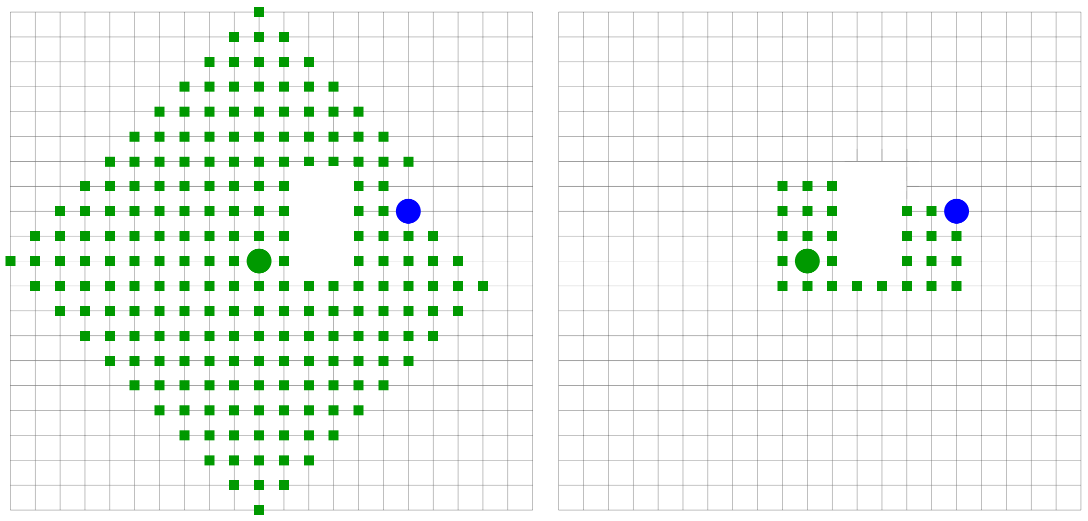


Figure 3.6: The permanent labels (green squares) created by a path search without future costs (left) and a path search with l_1-future costs (right) from the source (green circle) to the target (blue circle). The search without future costs requires 205 permanent labels while the search with l_1-future costs needs only 29.

3.3.2 Considering All Dimensions at Once

The l_1-future cost gives good bounds if $\mathrm{c}^{\mathrm{wire}} \approx \alpha$. In practice BonnRouteDetailed uses different edge costs: First, edges in non-preferred dimension should be more expensive than edges in preferred dimension. Second, each net has assigned layers that should contain most of its wiring. Non-assigned layers get higher edge costs. The future costs presented in this section account for that.

Definition 3.13. *A* rectilinear sequence *is a sequence $P = v_1, \ldots, v_k$, where $v_i \in \mathbb{Z} \times \mathbb{Z} \times \mathcal{Z}$ for $i \in \{1, \ldots, k\}$ and (v_i, v_{i+1}) is a rectilinear edge for $i \in \{1, \ldots, k-1\}$.*

Let $c = (\mathrm{c}^{\mathrm{wire}}, \mathrm{c}^{\mathrm{viadown}})$ be a distance based cost function. The cost *of a rectilinear sequence $P = v_1, \ldots, v_k$ is $\mathrm{cost}^c(P) := \sum_{i=1}^{k-1} \mathrm{cost}^c(v_i, v_{i+1})$.*

RECTILINEAR SHORTEST PATH PROBLEM
Input: 1.) A distance based cost function c and a finite set of target locations $T \subseteq \mathbb{Z} \times \mathbb{Z} \times \mathcal{Z}$.
2.) A point $v \in \mathbb{Z} \times \mathbb{Z} \times \mathcal{Z}$.
Task: Compute the minimum cost of a rectilinear sequence from v to T.

A solution to the RECTILINEAR SHORTEST PATH PROBLEM is an algorithm that builds up a data structure for part 1 of the input and can then answer queries for the cost of a cheapest rectilinear sequence from any given point $v \in \mathbb{Z} \times \mathbb{Z} \times \mathcal{Z}$ to T. Since there can be many such queries low query runtime is typically the most important factor. The problem is interesting in its own right and we present algorithms for solving it that achieve good theoretical and practical running times.

Lemma 3.14. *Let $c = (\mathrm{c}^{\mathrm{wire}}, \mathrm{c}^{\mathrm{viadown}})$ be a distance based cost function, let G, c_E be a graph and an edge cost function that are compatible with c, and let $T \subseteq V(G)$. Then the function $f : V(G) \to \mathbb{R}_{\geq 0}$ with $f(v) = \min\{\mathrm{cost}^c(P) | P$ is a v-T-path$\}$ is a future cost function for G, c_E, and T.*

Proof. Let G' be the Hanan grid of $V(G)$ with edge cost function c' defined by $c'(e) = \mathrm{cost}^{(\mathrm{c}^{\mathrm{wire}}, \mathrm{c}^{\mathrm{viadown}})}(e)$ for $e \in E(G')$. Then $f(v) = \mathrm{dist}_{c'}^{G'}(v, T)$ for all $v \in V(G)$. The result follows from Lemma 3.9. □

For the following structure result we provide a shorter proof than [Henke, 2016].

Lemma 3.15 (similar to [Henke, 2016], Lemma 2.2-2.4)**.** *Let $c = (\mathrm{c}^{\mathrm{wire}}, \mathrm{c}^{\mathrm{viadown}})$ be a distance based cost function. For every $s, t \in \mathbb{Z} \times \mathbb{Z} \times \mathcal{Z}$ there is a rectilinear sequence $P = v_1, \ldots, v_k$ from s to t of minimum cost that has the following properties:*

1. *There is at most one $i \in \{1, \ldots, k-1\}$ such that $\mathrm{Dim}(v_i, v_{i+1}) = \leftrightarrow$ and at most one $i \in \{1, \ldots, k-1\}$ with $\mathrm{Dim}(v_i, v_{i+1}) = \updownarrow$.*
2. *No two consecutive rectilinear edges have the same dimension, i.e. for all $i \in \{1, \ldots, k-2\}$ we have $\mathrm{Dim}(v_i, v_{i+1}) \neq \mathrm{Dim}(v_{i+1}, v_{i+2})$.*

Proof. Let $P = v_1, \ldots, v_k$ be a rectilinear sequence from s to t of minimum cost that minimizes k.

If property 2 is not satisfied there is an index $i \in \{1, \ldots, k-2\}$ such that $\mathrm{Dim}(v_i, v_{i+1}) = \mathrm{Dim}(v_{i+1}, v_{i+2})$. Then $v_1, \ldots, v_i, v_{i+2}, \ldots, v_k$ has fewer edges and the same or lower cost than P, a contradiction to the choice of P.

It remains to show that property 1 holds. Assume for the sake of contradiction that there are $i < j \in \{1, \ldots, k\}$ with $\mathrm{Dim}(v_i, v_{i+1}) = \mathrm{Dim}(v_j, v_{j+1}) \neq \swarrow\!\!\!\!\nearrow$. Consider the layers z_i of v_i and z_j of v_j.

If $\mathrm{c}^{\mathrm{wire}}(z_i, \mathrm{Dim}(v_i, v_{i+1})) \leq \mathrm{c}^{\mathrm{wire}}(z_j, \mathrm{Dim}(v_j, v_{j+1}))$ we define a new sequence by

$$P' = v_1, \ldots, v_i, v_{i+1} + (v_{j+1} - v_j), \ldots, v_j + (v_{j+1} - v_j), v_{j+2}, \ldots, v_k$$

where the rectilinear edge (v_j, v_{j+1}) is merged into the one on layer z_i. Otherwise we define P' by

$$P' = v_1, \ldots, v_i, v_{i+2} - (v_{i+1} - v_i), \ldots, v_j - (v_{i+1} - v_i), v_{j+1}, \ldots, v_k$$

where the rectilinear edge (v_i, v_{i+1}) is merged into the one on layer z_j. This ensures that the cost of the combined edge is at most the cost of the two edges. The cost of the other segments does not change. Thus P' has fewer edges and the same or lower cost than P. This is a contradiction to the choice of P. □

Note that Lemma 3.15 implies that there is a shortest rectilinear sequence consisting of at most 5 rectilinear edges. Moreover, the choice of the via segments is obvious once the layers of the horizontal and vertical segments are determined. This allows us to solve the Rectilinear Shortest Path Problem with a single target in $O(|\mathcal{Z}|^2)$ query time without doing any preprocessing by enumerating all choices of layers of the horizontal and vertical edges. Dorothee Henke showed that this can be improved to $O(|\mathcal{Z}|)$ query time.

Theorem 3.16 (corollary of [Henke, 2016], Satz 2.1)**.** *Even without preprocessing, the* Rectilinear Shortest Path Problem *with* $|T| = 1$ *can be solved in* $O(|\mathcal{Z}|)$ *query time.*

This is the best runtime bound we can hope for without preprocessing. The algorithm uses dynamic programming to compute the future costs and is too slow to be used in practice [Henke, 2016]. Hence, we will not consider it in our experiments in Section 6.2.

Doing a preprocessing step at the beginning allows us to achieve good practical and theoretical runtime. We compute the data for query locations on different layers independently. Let v_z be the layer we are currently considering. The idea is that for fixed target layer t_z and minimum and maximum layer z_{min}, z_{max} used by the rectilinear sequence, the cost of a shortest path is given by the following simple function, where Δ_x and Δ_y are the horizontal and vertical distances between query location and target:

$$\begin{aligned}
\text{cost}^{t_z,v_z}_{z_{min},z_{max}}(\Delta_x,\Delta_y) := \Delta_x & \min_{z\in\{z_{min},\dots,z_{max}\}} \text{c}^{\text{wire}}(z,\leftrightarrow) \\
&+ \Delta_y \min_{z\in\{z_{min},\dots,z_{max}\}} \text{c}^{\text{wire}}(z,\updownarrow) \\
&+ \sum_{z=\min\{v_z,t_z\}+1}^{\max\{v_z,t_z\}} \text{c}^{\text{viadown}}(z) \\
&+ 2\sum_{z=z_{min}+1}^{\min\{v_z,t_z\}} \text{c}^{\text{viadown}}(z) \\
&+ 2\sum_{z=\max\{v_z,t_z\}+1}^{z_{max}} \text{c}^{\text{viadown}}(z) \\
&= \Delta_x c^{t_z,v_z,\leftrightarrow}_{z_{min},z_{max}} + \Delta_y c^{t_z,v_z,\updownarrow}_{z_{min},z_{max}} + c^{t_z,v_z,\nearrow}_{z_{min},z_{max}},
\end{aligned} \tag{3.1}$$

where $c^{t_z,v_z,\leftrightarrow}_{z_{min},z_{max}}$, $c^{t_z,v_z,\updownarrow}_{z_{min},z_{max}}$, and $c^{t_z,v_z,\nearrow}_{z_{min},z_{max}}$ are appropriate constants. The cost of a shortest rectilinear path to t is then given by

$$\text{cost}^{t_z,v_z}(\Delta_x,\Delta_y) = \min_{z_{min}\in\{0,\dots,\min\{t_z,v_z\}\}} \min_{z_{max}\in\{\max\{t_z,v_z\},\dots,|\mathcal{Z}|-1\}} \text{cost}^{t_z,v_z}_{z_{min},z_{max}}(\Delta_x,\Delta_y).$$

We now show how to achieve a query time of $O(\log|\mathcal{Z}| + \log|T|)$ with polynomial preprocessing time. This was developed independently and not done in joint work with Dorothee Henke and Jens Vygen. Before describing the algorithm we note that we believe that it is not a good choice for computing future costs during BonnRouteDetailed, despite its good theoretical query time. Hence, we will not consider it for the experiments in Section 6.2. For more information see page 40.

The first ingredient is an algorithm for solving the planar point location problem, a well-studied problem in computational geometry. In the following, we regard any connected, closed part of a line as a line segment and any bounded or non-bounded region whose boundary consists of a finite number of line segments as a polygon. See Figure 3.7 for an illustration.

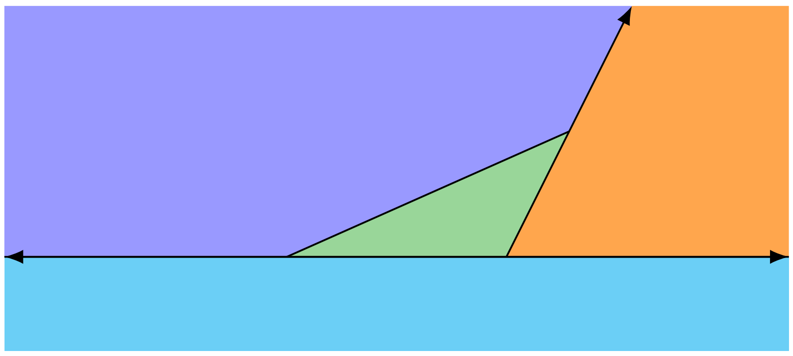


Figure 3.7: Three line segments dividing the plane into four polygons (colored blue, green, orange, and cyan). An arrow means that the line segment is going to infinity. Similarly, a polygon intersecting the border of the image means that it goes to infinity.

Point Location Problem	
Input:	1. A set of line segments L that intersect only at their endpoints, subdividing $\mathbb{R}^2 \setminus L$ into a set of open polygons $\mathcal{P}$. 2. A point $p \in \mathbb{R}^2$.
Task:	Compute a polygon $P \in \mathcal{P}$ such that $p \in \bar{P}$, i.e. p is in the closure of P.

See Figure 3.8 for an illustration. A solution to the Point Location Problem is an algorithm that builds up a data structure for part 1 of the input and can then answer queries for the polygon containing any point $p \in \mathbb{R}^2$. [Dobkin and Lipton, 1976] describe a simple and efficient algorithm for solving the problem.

Theorem 3.17 ([Dobkin and Lipton, 1976])**.** *Let L be a set of line segments as in part 1 of the input of the* Point Location Problem. *There is an algorithm that preprocesses L in time polynomial in $|L|$ and can then answer queries for the polygon containing any point $p \in \mathbb{R}^2$ in $O(\log|L|)$ time.*

Their solution uses a slab decomposition, which introduces vertical lines at each endpoint of a line. For an illustration see Figure 3.9.

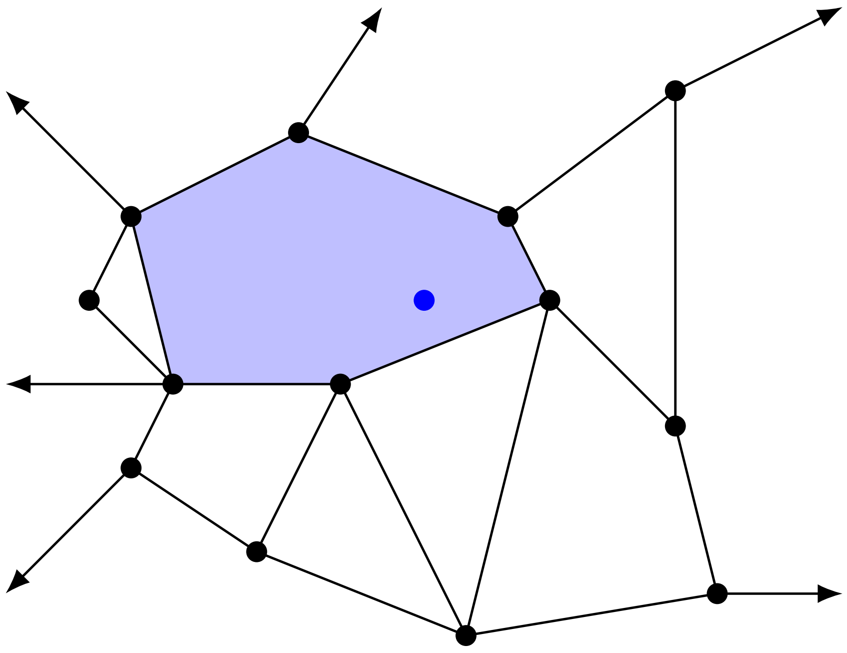


Figure 3.8: An instance of the point location problem. Arrows denote line segments going to infinity. Part 1 of the instance is drawn in black. A query location and the polygon containing it are drawn in blue.

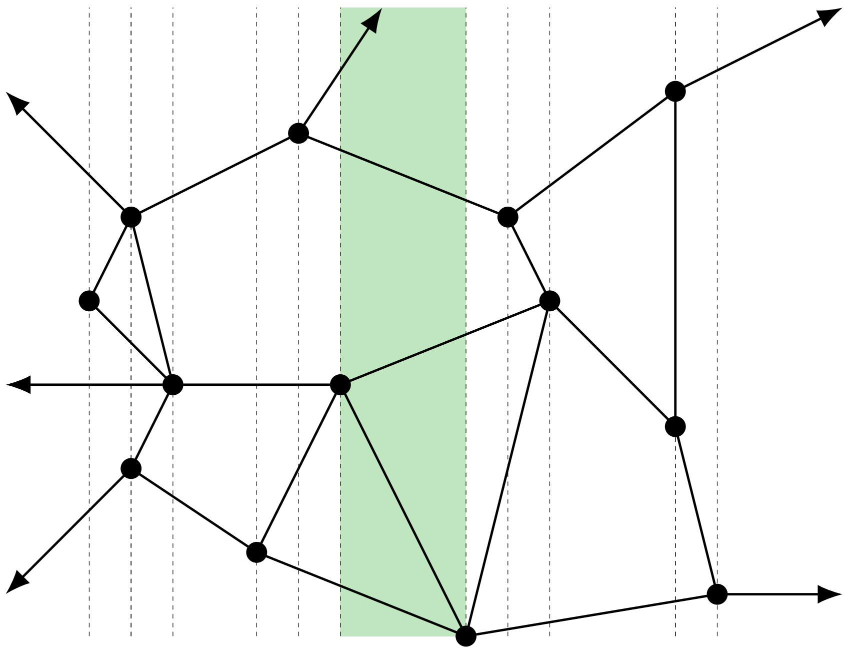


Figure 3.9: A slab decomposition of the instance from Figure 3.8 is indicated by dashed black lines. A single slab is drawn in green.

In each slab the order of the line segments is determined since line segments intersect only at their endpoints. The polygons and their separating line segments are stored in each slab. Answering a query requires two binary searches: first in x-dimension and then in y-dimension.

Note that storing the data for all slabs requires $O(|L|^2)$ space. Later, algorithms with the same asymptotic query time, requiring only $O(|L|)$ space or having better preprocessing time were proposed in [Lipton and Tarjan, 1980; Kirkpatrick, 1983; Edelsbrunner et al., 1986; Sarnak and Tarjan, 1986].

Recall from equation (3.1) on page 35 that the minimum cost of a rectilinear sequence is a minimum of functions of the type $(x, y) \mapsto a + |x - p^{\leftrightarrow}|s^{\leftrightarrow} + |y - p^{\updownarrow}|s^{\updownarrow}$. The second ingredient for achieving a query time of $O(\log|\mathcal{Z}| + \log|T|)$ is a lemma that allows us to cover the *boundary* of the set of points where two functions of this type are equal with a small number of lines. Figure 3.10 provides four examples of such sets.

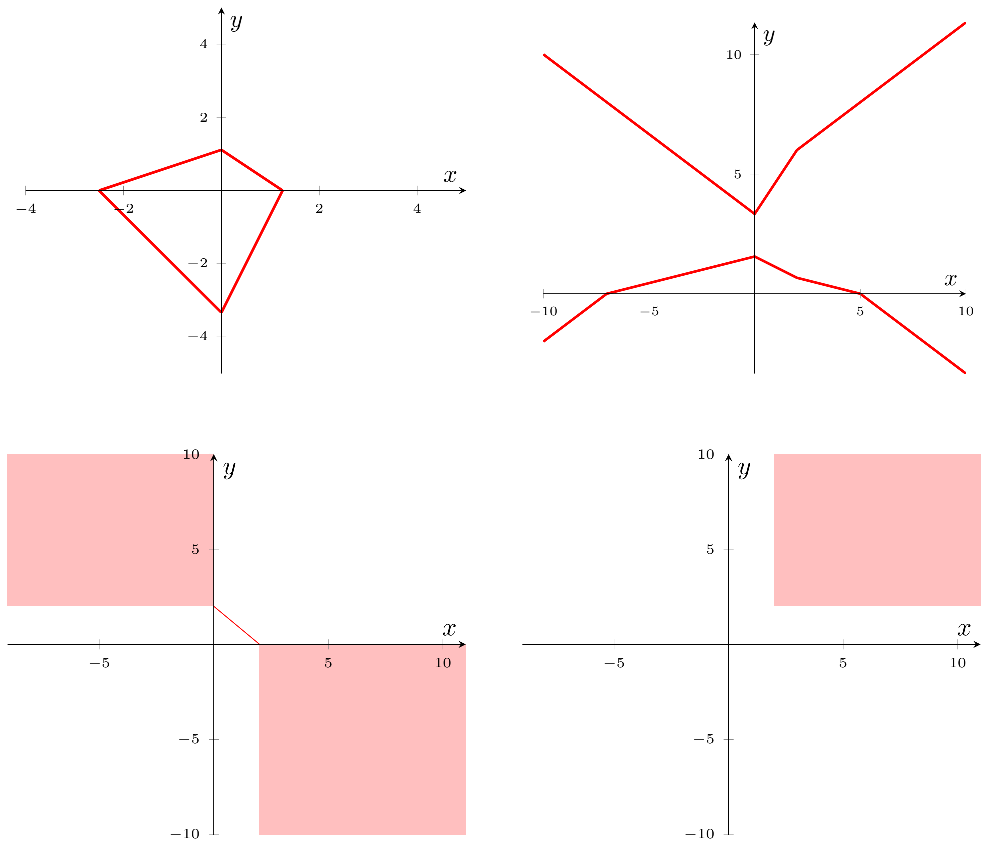


Figure 3.10: Illustration of the set of points $\{(x, y) \in \mathbb{R}^2 | f_1(x, y) = f_2(x, y)\}$ for $f_1(x, y) = |x|+|y|$ and different f_2. In the upper row we have $f_2(x, y) = \frac{1}{3}|x-2|+\frac{1}{2}|y-2|$ on the left and $f_2(x, y) = \frac{1}{3}|x-2|+2|y-2|$ on the right. In the lower row we have $f_2(x, y) = |x-2|+|y-2|$ on the left and $f_2(x, y) = 4 + |x - 2| + |y - 2|$ on the right. This row showcases that there are non-trivial examples of two-dimensional areas with $f_1 = f_2$. Line segments and areas intersecting the border of the image continue to infinity.

Lemma 3.18. *Let $f_1, f_2 : \mathbb{R}^2 \to \mathbb{R}$ be two functions with*

$$f_i(x,y) = a_i + |x - p_i^{\leftrightarrow}|s_i^{\leftrightarrow} + |y - p_i^{\updownarrow}|s_i^{\updownarrow}$$

for some $a_i, s_i^{\leftrightarrow}, s_i^{\updownarrow}, p_i^{\leftrightarrow}, p_i^{\updownarrow} \in \mathbb{R}$ and $i \in \{1,2\}$. Then there is a set L consisting of at most 13 lines such that the boundary of the roots of $f_1 - f_2$ is contained in L, i.e. $\partial\{(x,y) \in \mathbb{R}^2 | f_1(x,y) = f_2(x,y)\} \subseteq L$. Moreover such a set L can be computed in time $O(1)$.

Proof. We denote the set of roots of $f_1 - f_2$ by $N = \{(x,y) \in \mathbb{R}^2 | f_1(x,y) = f_2(x,y)\}$ and define $L_b = p_1^{\leftrightarrow} \times \mathbb{R} \cup p_2^{\leftrightarrow} \times \mathbb{R} \cup p_1^{\updownarrow} \times \mathbb{R} \cup p_2^{\updownarrow} \times \mathbb{R}$ which consists of four lines. Then $\mathbb{R}^2 \setminus L_b$ is partitioned into up to nine non-empty, open and connected polygons $O_1, \dots, O_k$. For $j \in \{1, \dots, k\}$ we define $N_j = \{(x,y) \in O_j | f_1 - f_2 = 0\}$. Note that $\partial N \subseteq \cup_{j=1}^{k} \partial N_j \cup L_b$. Since $k \leq 9$ it suffices to show that ∂N_j is contained in L_b or in a single line and that these lines can be computed in $O(1)$ time. Since $f_1 - f_2$ is affine in O_j there are three cases:

1. $N_j = O_j$
2. $N_j = \emptyset$
3. N_j is a line segment. Note that N_j cannot be a single point, since O_j is open.

In case 1 and case 2 we have $\partial N_j \subseteq L_b$. In the third case, we add the line containing N_j to L.

The lines in L_b and the line segments for the polygons can be computed in $O(1)$ time. □

Theorem 3.19. *There is a data structure that requires polynomial preprocessing time and can then solve the* Rectilinear Shortest Path Problem *for each query location in $O(\log|\mathcal{Z}| + \log|T|)$ time, where $|T|$ is the number of targets and $|\mathcal{Z}|$ is the number of layers.*

Proof. We compute independent data structures for each layer of the query location. Let v_z be the layer we are currently considering.

The cost of a shortest path from a query location (v_x, v_y, v_z) to T is then given by

$$\begin{aligned} \text{cost}^{v_z}(v_x, v_y) = & \min_{(t_x,t_y,t_z)\in T} \; \min_{z_{min}\in\{0,\dots,\min\{v_z,t_z\}\}} \; \min_{z_{max}\in\{\max\{v_z,t_z\},\dots,|\mathcal{Z}|-1\}} \\ & \left(c^{t_z,v_z,\nearrow}_{z_{min},z_{max}} + |v_x - t_x| c^{t_z,v_z,\leftrightarrow}_{z_{min},z_{max}} + |v_y - t_y| c^{t_z,v_z,\updownarrow}_{z_{min},z_{max}} \right), \end{aligned}$$

where the $c^{t_z,v_z,dim}_{z_{min},z_{max}}$ depend on distance based cost function and are computed as in equation (3.1).

Hence, the cost is the minimum of $O(|T||\mathcal{Z}|^2)$ functions $f_1, \dots, f_k$. Applying Lemma 3.18 to every pair of functions yields a set of lines L' with $|L'| = O(|T|^2|\mathcal{Z}|^4)$ such that for every $i, j \in \{1, \dots, k\}$ we have $\partial\{(x,y)|f_i(x,y) = f_j(x,y)\} \subseteq L'$. Splitting the lines at each intersection yields a set of line segments L with $|L| = O(|T|^4|\mathcal{Z}|^8)$.

Note that $\mathbb{R}^2 \setminus L$ is partitioned into $O(|T|^4|\mathcal{Z}|^8)$ non-empty, open, and connected polygons $R_1, \dots, R_l$. We claim that for every $m \in \{1, \dots, l\}$ and for every $i, j \in \{1, \dots, k\}$ we have either

1. $f_i - f_j < 0$ in R_m,
2. $f_i - f_j = 0$ in R_m, or
3. $f_i - f_j > 0$ in R_m.

Indeed, if $x, y \in R_m$ then there is a path P in R_m that connects x and y since R_m is connected. The statement follows from $\partial\{(x,y)|f_i(x,y) = f_j(x,y)\} \cap P = \emptyset$ and the continuity of $f_i - f_j$ on P. This allows us to find a function that attains the minimum in R_m by picking a point $p \in R_m$ arbitrarily and determining any minimizer of $f_i(p)$. Note that this minimizer also attains the minimum on the boundary $\partial(R_m)$.

All this can be done in polynomial time.

We then initialize a data structure for the point location problem with L in polynomial time. This allows us to find the polygon containing a query location in $O(\log|L|) = O(\log(|T|^4|\mathcal{Z}|^8)) = O(\log|\mathcal{Z}| + \log|T|)$ time. We then evaluate the minimizing function to obtain the length of a shortest path to the query location. □

We note that while the algorithm from Theorem 3.19 has very good theoretical query runtime, we believe that is not a good choice for computing future costs during BonnRouteDetailed. In the proof the plane is partitioned into $O(|T|^4|\mathcal{Z}|^8)$ polygons. Even determining these polygons is computationally expensive and requires some care, because vertices of the polygons are not necessarily on integral coordinates even if all input coordinates are integral. After the polygons are determined they are used to initialize a data structure for the point location problem, which will be computationally expensive if the number of polygons is large. Moreover, this preprocessing must be done from scratch for every instance of the Path Search Problem.

We now present a simple preprocessing, introduced in Section 2.4 of [Henke, 2016], that has a worst-case query time of $O(|\mathcal{Z}|^2)$ in case of a single target $t \in T$. Recall that we can achieve a faster query time of $O(|\mathcal{Z}|)$ without any preprocessing with an impractical approach by Theorem 3.16. Nevertheless, the simple preprocessing performs very well in practice and is used by default in BonnRouteDetailed. It is considered in the experiments in Section 6.2. The idea is to compute and store the coefficients of equation (3.1) on page 35 using Algorithm 4.

Lemma 3.20. *Algorithm 4 returns $(F_{z_1,z_2})_{z_1,z_2 \in \mathcal{Z}, z_1 \le z_2}$ such that for $t = (t_x, t_y, t_z), v = (v_x, v_y, v_z) \in \mathbb{R} \times \mathbb{R} \times \mathcal{Z}$ the cost of a shortest rectilinear sequence from v to t is*

$$\min_{f \in F_{\min\{v_z,t_z\},\max\{v_z,t_z\}}} f(|t_x - v_x|, |t_y - v_y|). \tag{3.2}$$

The algorithm runs in $O(|\mathcal{Z}|^5)$ time.

Proof. Without the pruning in line 6 and 7 the correctness follows from Lemma 3.15 on page 34. The pruning in case $z_{min} < z_1$, $c^{\leftrightarrow} \neq d(z_{min}, \leftrightarrow)$, and $c^{\updownarrow} \neq d(z_{min}, \updownarrow)$ is valid because in this case we could save the two vias to layer z_{min} which cannot increase the cost. Note that the pruning works even if it is applied to multiple consecutive layers. The same argument applies to the pruning in line 7.

Algorithm 4 Simple Preprocessing

Input: A distance based cost function $c = (\mathrm{c}^{\mathrm{wire}}, \mathrm{c}^{\mathrm{viadown}})$.

```
1: for $z_1, z_2 \in \mathcal{Z}$ with $z_1 \leq z_2$ do
2:     $F_{z_1,z_2} \leftarrow \emptyset$
3:     for $z_{min} \in \{0, \ldots, z_1\}, z_{max} \in \{z_2, \ldots, |\mathcal{Z}| - 1\}$ do
4:         $c^{\leftrightarrow} \leftarrow \min\{\mathrm{c}^{\mathrm{wire}}(z, \leftrightarrow) | z \in \{z_{min}, \ldots, z_{max}\}\}$
5:         $c^{\updownarrow} \leftarrow \min\{\mathrm{c}^{\mathrm{wire}}(z, \updownarrow) | z \in \{z_{min}, \ldots, z_{max}\}\}$
6:         if $z_{min} = z_1$ or $c^{\leftrightarrow} = \mathrm{c}^{\mathrm{wire}}(z_{min}, \leftrightarrow)$ or $c^{\updownarrow} = \mathrm{c}^{\mathrm{wire}}(z_{min}, \updownarrow)$ then
7:             if $z_{max} = z_2$ or $c^{\leftrightarrow} = \mathrm{c}^{\mathrm{wire}}(z_{max}, \leftrightarrow)$ or $c^{\updownarrow} = \mathrm{c}^{\mathrm{wire}}(z_{max}, \updownarrow)$ then
8:                 $F_{z_1,z_2} \leftarrow F_{z_1,z_2} \cup \mathrm{cost}^{z_1,z_2}_{z_{min},z_{max}}(.,.)$    ▷ cost is defined as in equation 3.1
9: return $(F_{z_1,z_2})_{z_1,z_2 \in \mathcal{Z}, z_1 \leq z_2}$
```

The runtime bound is obvious. □

The runtime of Algorithm 4 can be improved to $O(|\mathcal{Z}|^4)$ by incrementally updating the via costs and the minimum costs in horizontal and vertical dimension. Given the output of the algorithm future costs can be computed by evaluating the minimum as in equation (3.2). The number of terms can be quadratic in the number of layers, but in our application and for our choice of cost functions the average number of terms is less than five. This is why this approach is very fast in practice despite its slow worst-case runtime.

Section 6.2 provides experimental results that show the effects of different future cost functions.

3.3.3 Possible Improvements

The future costs discussed in the previous sections can be improved by considering detours that are necessary because of the routing area. An approach that works for certain routing areas composed of few rectangles is described in Section 2.6 of [Henke, 2016]. The approach works with a three-dimensional Hanan grid that is derived from some of the corners of the routing area and requires precomputing the distances between the nodes of the Hanan grid.

Recall that the distance based cost function used by BonnRouteDetailed has fixed cost per unit for each pair of layer and dimension. However, it might make sense to let this cost depend on the location in the routing area for various reasons:

- In timing-critical nets, lower layers should be expensive near the electrical source and cheaper near the electrical sinks. The reason is that lower layers have thinner wires with higher electrical resistance that have a large impact on timing near the electrical source. Near the sinks the additional resistance is less important and electrical capacitance is the most important factor.

- Congested parts of the routing area should be used as efficiently as possible. Making them more expensive than other parts could help to achieve this.
- Similarly, parts of the routing area that were added only for local track changes, as shown in Figure 3.5 on page 28, should be more expensive, since global routing did not account for any usage of the net on these layers. That might lead to detailed routings that are more similar to the global routing solution.

If such area-dependent base cost functions were used in BonnRouteDetailed it would make sense to consider them in the future costs. This could probably be incorporated into a Hanan grid based approach.

An alternative approach is the rectangle-labeling approach of [Peyer et al., 2009]. The algorithm is a generalization of Dijkstra's algorithm that propagates cost functions on rectangles instead of individual nodes (or intervals). Such an approach could handle both routing areas and area-dependent costs in a very natural way. Moreover, it could probably be adapted to provide future costs, as they are required for the efficient Steiner tree search described in Chapter 5. While the approach required too much preprocessing time to be used in BonnRouteDetailed in the past, we believe that it is probably possible to make a similar approach practical.

3.4 Working on Implicitly Given Routing Graphs

In BonnRouteDetailed the routing graph used in the path search is described by a routing area A and two oracle functions: the track oracle and the checking oracle. The track oracle provides a track graph G_T, which is restricted to G_T^A by the routing area. The checking oracle provides the information which edges of G_T^A are usable in the context of the current path search. The following pages introduce the oracle functions and explain how they are used in the path search.

Definition 3.21 (track oracle). *A track oracle $\mathcal{T}$ for a set of tracks T is a function $\mathcal{T}$: $\{[x_1, x_2] \times [y_1, y_2] \times \{z\} | x_1, x_2, y_1, y_2 \in \mathbb{Z}, z \in \mathcal{Z}\} \to 2^T$ that returns the tracks intersecting a rectangle r, i.e. $\mathcal{T}(r) = \{t \in T | t \cap r \neq \emptyset\}$.*

Section 3.5 sketches how the track oracle is implemented in BonnRouteDetailed. Using the track oracle efficiently requires preprocessing. At the beginning of the path search we partition the bounding box of the routing area into regions with uniform track structure.

Definition 3.22 (grid region partition, region). *Let $A = \cup_{i \in \{1,\ldots,n\}} a_i$ be a routing area with minimum and maximum layer z_{min}, z_{max} and let T be a set of tracks. A* candidate grid region partition *of (A, T) is a pair of ordered sets of integers $((x_1, \ldots, x_k), (y_1, \ldots, y_l))$ with $x_1 < \cdots < x_k, y_1 < \cdots < y_l$. A* grid region *or* region *of a candidate grid region partition is a rectangle $[x_i, x_{i+1} - 1] \times [y_j, y_{j+1} - 1] \times \{z\}$ for some $i \in \{1, \ldots, k\}$, $j \in \{1, \ldots, l\}$, and $z \in \{z_{min}, \ldots, z_{max}\}$. We call a candidate grid region partition a* grid region partition *if it has the following properties:*

1. *The regions partition the integer points of the bounding box of the routing area* BoundingBox(A)*, i.e.* BoundingBox$(A) \cap \mathbb{Z}^3 = \{x_1, \ldots, x_k - 1\} \times \{y_1, \ldots, y_l - 1\} \times \{z_{min}, \ldots, z_{max}\}$.
2. *Regions do not cross or strictly contain rectangles of the routing area, i.e. for every region r and every $a \in \{a_1, \ldots, a_n\}$ we have either $a \cap r = \emptyset$ or $a \cap r = r$.*
3. *Tracks do not stop in the middle of regions contained in the routing area, i.e. for every region $r = I_x \times I_y \times \{z\}$, and every track $t \in T$ we have either $t \cap r \cap A = \emptyset$ or*

$$t \cap r = \begin{cases} I_x \times \{c\} \times \{z\}, & \textit{if } z \textit{ is horizontal} \\ \{c\} \times I_y \times \{z\}, & \textit{if } z \textit{ is vertical} \end{cases}$$

   for some $c \in \mathbb{Z}$.

Note that adding coordinates to a grid region partition gives us another grid region partition if the minimum and maximum coordinates of the sets remain unchanged.

Property 1 ensures that every integral point of the routing area is contained in exactly one grid region. In practice, the routing area provides more information than in the simplified version described here. For example, different rectangles for the routing area may allow different wire and via models. Property 2 ensures that this data is uniform within each region. Property 3 enables efficient indexing of the nodes within each region.

We compute a grid region partition with Algorithm 5. A grid region partition computed by the algorithm is illustrated in Figure 3.11.

Algorithm 5 ComputeGridRegionPartition

Input: A routing area $A = \cup_{i=1}^{n} a_i$ and a set of tracks T given by a track oracle $\mathcal{T}$.
Output: A grid region partition of (A, T).

```
 1: $X \leftarrow \emptyset$
 2: $Y \leftarrow \emptyset$
 3: for $a = [x_1^a, x_2^a] \times [y_1^a, y_2^a] \times \{z\} \in \{a_1, \ldots, a_n\}$ do
 4:     for $t \in \mathcal{T}(a)$ do                                ▷ call track oracle
 5:         $[x_1^t, x_2^t] \times [y_1^t, y_2^t] \times \{z\} \leftarrow t \cap a$
 6:         if PrefDim$(z) = \leftrightarrow$ then
 7:             $X \leftarrow X \dot{\cup} \{x_1^t, x_2^t\}$
 8:         else
 9:             $Y \leftarrow Y \dot{\cup} \{y_1^t, y_2^t\}$
10:     $X \leftarrow X \dot{\cup} \{x_1^a, x_2^a\}$
11:     $Y \leftarrow Y \dot{\cup} \{y_1^a, y_2^a\}$
12: sort $X$ and $Y$ and remove duplicates to obtain $(x_1, \ldots, x_k)$ and $(y_1, \ldots, y_l)$
13: return $((x_1, \ldots, x_{k-1}, x_k + 1), (y_1, \ldots, y_{l-1}, y_l + 1))$
```

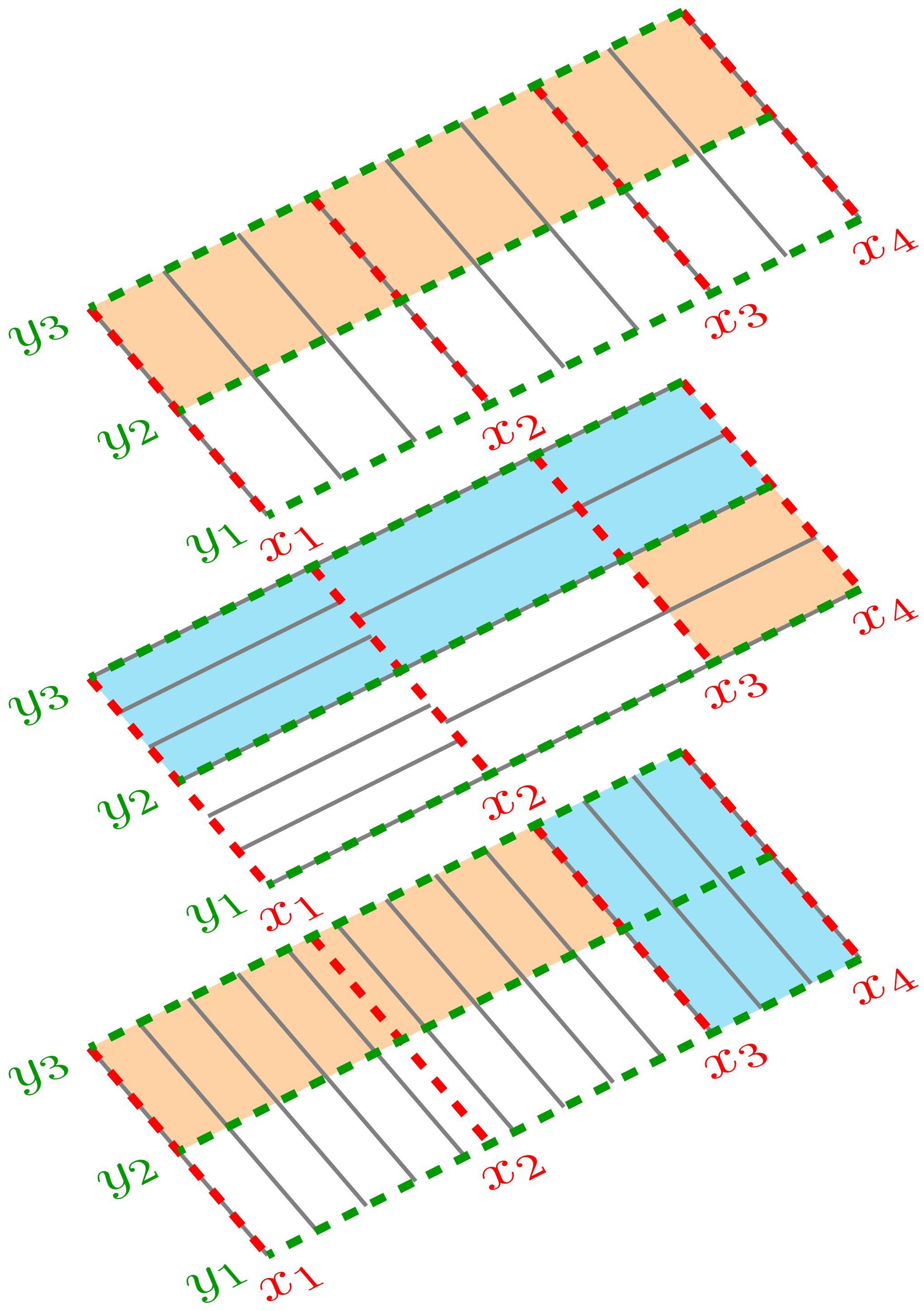


Figure 3.11: The grid region partition $((x_1, x_2, x_3, x_4), (y_1, y_2, y_3))$ computed by Algorithm 5 for the routing area from Figure 3.5, consisting of the cyan and orange rectangles, and the tracks drawn in gray. Note that the coordinate x_2 is inserted because of the change in track structure on the middle layer.

Lemma 3.23. *Algorithm 5 computes a grid region partition of (A, T) in time $O((|A| + |T|)\log(|A| + |T|) + \beta)$, where $O(\beta)$ is the runtime of the track oracle when it is called for each rectangle of the routing area.*

Proof. The result is a candidate grid region partition because the coordinates are sorted and duplicates are removed in line 12.

Property 1:
In suffices to show that $\text{BoundingBox}(A) = [x_1, x_k - 1] \times [y_1, y_l - 1] \times \{z_{min}, \ldots, z_{max}\}$. After line 10 we have $\text{BoundingBox}(A) = [x_1, x_k] \times [y_1, y_l] \times \{z_{min}, \ldots, z_{max}\}$ because of line 10 and 11 and because line 5 ensures that the tracks have no effect on the minimum and maximum coordinates.

Property 2:
Let $a \in \{a_1, \ldots, a_n\}$ and let $r = [x, x'] \times [y, y'] \times \{z\}$ be a rectangle where $x, x' \in X$ are consecutive coordinates of X, $y, y' \in Y$ are consecutive coordinates of Y, and $a \cap r \neq \emptyset$. Since minimum and maximum x- / y-coordinates of a are contained in X / Y because of line 10 and 11 we have $r \subseteq a$. Note that r is not necessarily a region, but that every region is contained in an r of this type and that this implies that property 2 holds.

Property 3:
Let $r = I_x \times I_y \times \{z\}$ be a region on a layer $z \in \mathcal{Z}$. If $A \cap r = \emptyset$ we are done. Otherwise let $a \in \{a_1, \ldots, a_n\}$ s.t. $r \subseteq a$ and let t be a track such that $t \cap r \neq \emptyset$. Then line 7 and 9 imply that $t \cap r$ runs over the whole length or width of the region, i.e.

$$t \cap r = \begin{cases} I_x \times \{c\} \times \{z\}, & \text{if } z \text{ is horizontal} \\ \{c\} \times I_y \times \{z\}, & \text{if } z \text{ is vertical,} \end{cases}$$

where c is the coordinate in non-preferred dimension on layer z of t.

The runtime bound holds because X and Y each contain at most $|A| + |T|$ numbers. □

Note that any other grid region partition (X', Y') must be a refinement of the result (X, Y) of Algorithm 5, i.e. $X \subseteq X', Y \subseteq Y'$.

During the path search we store node identifiers which store the region of a node and local indices.

Definition 3.24 (node identifier)**.** *Let G_T be a track graph, let (X, Y) be a grid region partition, and let r be a region. Further let $\{x_1, \ldots, x_k\} = \{x | \exists y, z : (x, y, z) \in V(G_T) \cap r\}$ and $\{y_1, \ldots, y_l\} = \{y | \exists x, z : (x, y, z) \in V(G_T) \cap r\}$ with $x_1 < \cdots < x_k$ and $y_1 < \cdots < y_l$. For a node $(x_i, y_j, z) \in V(G_T)$ we define its* node identifier *by*

$$\text{NodeId}((x_i, y_j, z)) := \begin{cases} (r, i, j), & \text{if } z \text{ is horizontal} \\ (r, j, i), & \text{if } z \text{ is vertical.} \end{cases}$$

The location *of a node identifier (r,p,o) of a node in region r on layer z is*

$$\mathrm{Location}((r,p,o)) := \begin{cases} (x_p, y_o, z), & \textit{if } z \textit{ is horizontal} \\ (x_o, y_p, z), & \textit{if } z \textit{ is vertical.} \end{cases}$$

We now describe the data structures we use for organizing grid regions. After it is initialized at the beginning of the path search it enables efficient queries for node identifiers, neighbors, and locations of nodes.

Lemma 3.25 (grid region data structure)**.** *Let A be a routing area, let T be a set of tracks given by a track oracle, let G_T^A be the associated track graph when restricted to A, and let (X,Y) be a grid region partition of (A,T). There is a data structure that can be initialized in $O(\beta + T_R \log(T_R) + |R|\log|R|)$ time, where $O(\beta)$ is the runtime of the track oracle when it is called for each rectangle of A, R is the set of grid regions that are contained in the routing area, and $T_R = \sum_{r\in R} |\{t \in T | t \cap r \neq \emptyset\}|$. After the initialization the data structure allows the following queries:*

1. *Given a node $v \in V(G_T^A)$ in the routing area compute its node identifier* NodeId(v) *in $O(\log|R| + \log T_R)$ time.*
2. *Given a node identifier id compute its location* Location(id) *in $O(1)$ time.*
3. *Given a node identifier id and a direction $d \in \mathcal{D}$ compute the node identifier of the next neighbor of* Location(id) *in direction $d \in \mathcal{D}$ in G_T^A or decide that no such node exists. This operation can be performed in $O(1)$ time. We refer to the id of this neighbor by* NeighborId(id,d) *and set it to an invalid identifier if no such neighbor exists.*

Proof. We store the grid regions that are contained in the routing area in an array of balanced search trees, called GridRegion. The regions on layer $z \in \mathcal{Z}$ are stored in GridRegion$[z]$ and the key of a region $[x_1, x_2 - 1] \times [y_1, y_2 - 1] \times \{z\}$ is its pair of indices in (X,Y), i.e. $(|\{x \in X | x \leq x_1\}|, |\{y \in Y | y \leq y_1\}|)$. To initialize GridRegion we traverse each rectangle of the routing area, compute the indices of the intersecting regions and add them to the corresponding search tree. This initialization takes $O(|R|(\log|X| + \log|Y|)) = O(|R|\log|R|)$ time, since each region is initialized at most once. Whenever we refer to a specific region $r \in R$ in the future we mean the instance stored in GridRegion.

Each grid region $r \in R$ on layer z stores the following data:

- Sorted arrays PrefCoor and OrthoCoor storing the coordinates of nodes in the region r, i.e. for each node identifier (r,p,o) we have

  $$\mathrm{Location}((r,p,o)) = \begin{cases} (\mathrm{PrefCoor}[p], \mathrm{OrthoCoor}[o], z) & \text{if } z \text{ is horizontal} \\ (\mathrm{OrthoCoor}[o], \mathrm{PrefCoor}[p], z) & \text{if } z \text{ is vertical.} \end{cases}$$

  This allows us to process queries for the location of a node identifier, as in 2., in $O(1)$ time.

- An array NextRegion of references to grid regions. For $d \in \mathcal{D}$ the entry NextRegion$[d]$ is a reference to its neighboring region in direction d, if such a neighbor exists in R. Otherwise we set NextRegion$[d] = \emptyset$. There is one exception: If the neighbor r' is in R but $V(G_T^A) \cap r' = \emptyset$ we let NextRegion$[d]$ point to the neighbor of r' in direction d. This process is iterated until the next region contains a node or there is no such neighbor in R.

- Two-dimensional arrays PrefCoorMap and OrthoCoorMap allowing efficient computation of indices of neighbors in the next region in a direction. More precisely, the following holds for each node identifier (r, p, o):

  - For $d \in \{\leftarrow, \rightarrow, \downarrow, \uparrow\}$ the neighbor in G_T^A in direction d has node identifier $(\text{NextRegion}[d], \text{PrefCoorMap}[d][p], \text{OrthoCoorMap}[d][o])$.

  - Similarly, for $d \in \{\swarrow, \nearrow\}$ the node identifier of the neighbor in G_T in direction d is $(\text{NextRegion}[d], \text{OrthoCoorMap}[d][o], \text{PrefCoorMap}[d][p])$.

  If there is no such neighbor then PrefCoor$[d][p] =$ OrthoCoor$[d][o] = -1$. This data allows us to process queries for neighbors, as in 3., in $O(1)$ time.

To initialize this data we first initialize NextRegion with the neighbors in R or $\emptyset$ if there is no initialized neighbor. This requires $O(|R| \log|R|)$ time. Then we initialize the PrefCoor array for all regions by traversing the rectangles in the routing area, querying their tracks using the track oracle, and adding them to all regions they intersect. This takes $O(\beta + T_R \log|R|)$ time. This allows us to initialize OrthoCoor for each region in R by taking the union of the PrefCoor arrays of NextRegion$[\nearrow]$ and NextRegion$[\swarrow]$. Doing this for all regions requires $O(T_R + |R|)$ time. This yields the information whether regions are empty. We discard empty regions and update NextRegion accordingly in $O(|R|)$ time. Finally, we sort PrefCoor and OrthoCoor and initialize PrefCoorMap and OrthoCoorMap. It is straightforward to do this in $O(T_R \log T_R)$ time.

The runtime for initialization of GridRegion and the data for all regions in R is

$$\begin{aligned} & O(|R| \log|R| + |R| \log|R| + \beta + T_R \log|R| + T_R \log T_R) \\ = \; & O(\beta + T_R \log T_R + |R| \log|R|), \end{aligned}$$

as required.

To determine the node identifier of a node $(x, y, z) \in V(G_T^A)$, as in 1., we do binary search for x in X and for y in Y to determine the indices of the grid region containing it. Querying GridRegion$[z]$ allows us to find the region r containing v in $O(\log|X| + \log|Y|) = O(\log|R|)$ time. We use binary search in the sets of coordinates stored in r to determine the remaining part of the node identifier. This can be done in $O(\log T_R)$ time. □

During the path search additional data is stored with the node identifiers. Before describing how this data is stored, we introduce the checking oracle. It specifies which edges of G_T^A are usable in the context of the current path search.

Definition 3.26 (checking oracle). *A* checking oracle *for a routing graph G is a pair $\psi = (\psi_{\text{leg}}, \psi_{\text{int}})$, where $\psi_{\text{leg}} : V(G) \to 2^{\{\to,\uparrow,\nearrow\}}$ and $\psi_{\text{int}} : V(G) \to \{\overline{vw} | v, w \in V(G)\}$ are functions with the following properties:*

- $\psi_{\text{leg}}(v) = \{d \in \{\to, \uparrow, \nearrow\} |$ *there is a neighbor $w \in \Gamma(v)$ in direction d of v*$\}$
- *$\psi_{\text{int}}(v)$ is a point or a one-dimensional rectangle that runs in preferred dimension of its layer.*
- *We have $v \in \psi_{\text{int}}(v)$ and $\psi_{\text{leg}}(v) = \psi_{\text{leg}}(w)$ for all $w \in \psi_{\text{int}}(v) \cap V(G)$.*

The oracle ψ_{leg} provides the information which edges in positive dimension are legal. The purpose of ψ_{int} is to reduce the number of oracle calls: A node $w \in \psi_{\text{int}}(v)$ can use the same checking data as v. This is crucial because the checking oracle can be called billions of times and is very runtime-critical.

The description here is simplified. In BonnRouteDetailed the checking oracle gets a set of wire and via models as input and returns which of them can be used for the relevant edges. It also returns whether using the edge with a model requires ripping out parts of wirings of other nets. Edges that require using an undesirable wire or via model or ripup get a higher cost. To simplify the presentation we assume that the edge costs already account for this and that the checking data is binary. Section 3.5 sketches how the checking oracle is implemented in BonnRouteDetailed.

During the search, additional data, such as checking data and labels, needs to be stored for some node identifiers. For this purpose we store the following data at each grid region r:

- A one-dimensional array GridNode_r, storing all data on initialized nodes within r.
- A two-dimensional array StorageLocation_r such that for a node identifier (r, p, o) we have either $\text{StorageLocation}_r[o][p] = -1$, if the node was not initialized or the data for the identifier is $\text{GridNode}_r(\text{StorageLocation}_r[o][p])$.

This allows us to access the data $\text{Data}(r, p, o) := \text{GridNode}_r[\text{StorageLocation}_r[o][p]]$ associated with an initialized node identifier (r, p, o) in $O(1)$ time.

Moreover, each grid region r stores the checking data of the checking queries done in that region in a member CheckingData_r. Note that checking data can be reused for other nodes of the same region if they are in the interval returned by ψ_{int}. In the simplified version described here, the checking data could also be reused across grid regions. In practice this is not always possible, because some data, e.g. the set of allowed wire and via models and the set of usable tracks, is passed to the checking oracle. Within a region this data is constant, but it can vary across regions.

During the path search we call Algorithm 6 to ensure that all required data for a node is initialized. If the node is not initialized yet, the algorithm initializes the checking data and indicates that the node is not a target and that it does not have any label yet. Checking data is reused if possible. We note that the initialization is simplified. In practice we also initialize data such as future costs and node costs and the checking data is much more complicated.

Algorithm 6 EnsureInitialized(id, ψ)

Input: A node identifier $id = (r, p, o)$ and a checking oracle $\psi = (\psi_{\text{leg}}, \psi_{\text{int}})$.
Output: Ensure that basic data on id is initialized.

```
1:  if $\text{StorageLocation}_r[o][p] = -1$ then
2:      $v \leftarrow \text{Location}(id)$
3:      if $\exists(leg, I) \in \text{CheckingData}_r[o]$ with $v \in I$ then
4:          data.$legality \leftarrow leg$
5:      else
6:          data.$legality \leftarrow \psi_{\text{leg}}(v)$
7:          add $(\psi_{\text{leg}}(v), \psi_{\text{int}}(v))$ to $\text{CheckingData}_r[o]$
8:      data.$label \leftarrow \emptyset$
9:      data.$target \leftarrow$ $false$
10:     $\text{StorageLocation}_r[o][p] \leftarrow \text{GridNode}_r$.push(data)    ▷ returns index of new element
```

Lemma 3.27. *Algorithm 6 initializes the checking data correctly. Calling* EnsureInitialized *n times for k different nodes requires $O(n+k(C+\log T_R))$ time, where T_R is the maximum number of tracks intersecting a grid region and $O(C)$ is the runtime of the checking oracle.*

Proof. The runtime of all steps except for the queries and updates of $\text{CheckingData}_r[o]$ in line 3 and line 7 is $O(n + kC)$.

It remains to show that the queries in line 3 and the updates in line 7 can be done in $O(k \log T_R)$ time if the checking data is readily available in line 7. We store the checking data for each non-preferred coordinate in a balanced search tree B. The key is the minimum coordinate of the stored valid interval. We ensure that all stored intervals are disjoint and that each valid interval contains a grid node. This makes sure that the number of intervals is at most $2T_R$. When a new interval is added in line 7 we merge it with all intervals it intersects and remove the intersected intervals from B. Note that in this case the legality data of all intersecting intervals must be identical. Inserting an interval that triggers the removal of l intervals takes $O((l + 1) \log T_R)$ time. Since the number of intervals that can be inserted and removed is k the total runtime is $O(k \log T_R)$. □

In practice there is only a small number of checking calls for each track in a grid region. Thus we use an array instead of a balanced search tree and simply traverse all elements in line 3.

Algorithm 7 is a version of Dijkstra's algorithm with future costs that runs on an implicitly given routing graph G^A. At the beginning, the algorithm computes a grid region partition and initializes the grid region data structure from Lemma 3.25. Then it initializes all source and target nodes using EnsureInitialized and creates a label for each source node. These labels are stored in a priority queue Q. Each iteration of the main loop labels a node

$v \in V(G_T^A)$ permanently and removes its label from Q. The grid region data structure and the checking oracle are used to compute the node identifiers of all neighbors of v (and we ensure that they are initialized using EnsureInitialized). If a neighbor v' has no label or the path through v is shorter than the best path found so far, its label and Q are updated accordingly. Once a target becomes permanently labeled a shortest S-T-path has been determined and can be found by backtracking. If no target is labeled permanently during the algorithm there is no S-T-path.

Under some mild assumptions the algorithm has the same theoretical runtime as Algorithm 3 plus the runtime required by the track and the checking oracle. One of its benefits is that it initializes a node only if it is a source or target or if it is adjacent to a permanently labeled node in the track graph G_T^A. This can reduce runtime and memory usage, since most path searches with good future costs label only a small fraction of the nodes. Moreover, the grid region data structure allows us to handle complex, non-uniform track structures.

Lemma 3.28. *Algorithm 7 works correctly and can be implemented to run in time*

$$O(\beta + n(\log n + C + F)),$$

where $n = |V(G^A)|$ is the number of nodes, $O(\beta)$ is the runtime of the track oracle when called for each rectangle of the routing area, $O(C)$ is the runtime of the checking oracle, and $O(F)$ is the runtime of the future cost oracle. This runtime holds under the assumption that the number of rectangles of the routing area, the number of grid regions in the routing area, and $T_R = \sum_{r\in R} |\{t \in T | t \cap r \neq \emptyset\}|$ are all in $O(n)$.

Proof. Line 1 and 2 compute the grid region partition and initialize the grid region data structure. By Lemma 3.23 and Lemma 3.25 this can be done in $O(\beta + n \log n)$ time. Node identifiers for locations, locations of node identifiers, and neighbors of nodes are computed $O(n)$ times which requires $O(n \log n)$ time by Lemma 3.25. EnsureInitialized is called $O(n)$ times for $O(n)$ different nodes. By Lemma 3.27 this requires $O(n(C + \log n))$ time in total. The algorithm performs $O(n)$ insert and decrease key operations on Q, which takes $O(n \log n)$ time, if Q is implemented with a binary heap. Computing the future costs requires $O(nF)$ time. Finally, the backtracking in line 17 can be done in $O(n)$ time, since we store the predecessor ids and getting the id of the predecessor, its label, and its location takes $O(1)$ time.

Correctness follows from Corollary 3.7, the correctness of Dijkstra's algorithm with future costs. □

Note that storing the node identifier of the predecessor in each label is not necessary. The predecessor of a label can be found by enumerating all neighbors and checking whether the label could have been derived from their label. This is straightforward, but can become very complicated and error-prone when the algorithm is extended to avoid design rule violations as in Chapter 4. We therefore store the node identifier of the predecessor in each label.

Algorithm 7 Simplified Path Search Algorithm

Input: A routing graph G^A given by a checking oracle ψ, a track oracle $\mathcal{T}$, and a routing area A. Edge costs $c_E : E(G^A) \to \mathbb{R}_{\geq 0}$, node costs $c_V : V(G^A) \to \mathbb{R}_{\geq 0}$, a future cost function f, and sets of source and target nodes $S, T \subseteq V(G^A)$.
Output: A shortest S-T-path or the information that no such path exists.

```
 1: $(X, Y) \leftarrow$ ComputeGridRegionPartition$(A, \mathcal{T})$
 2: initialize the grid region data structure of Lemma 3.25 with $A$, $T$, and $(X, Y)$
 3: $Q \leftarrow$ empty priority queue
 4: for $v \in S \cup T$ do
 5:     $id \leftarrow$ NodeId$(v)$
 6:     EnsureInitialized$(id, \psi)$
 7:     if $v \in S$ then
 8:         Data$(id).label \leftarrow (id, \emptyset, c_V(v) + f(v))$
 9:         $Q.insert($Data$(id).label)$
10:     if $v \in T$ then
11:         Data$(id).target \leftarrow$ $true$
12: while $Q$ is not empty do
13:     $(id, pred, c) \leftarrow Q.top$
14:     $Q.pop$
15:     $v \leftarrow$ Location$(id)$
16:     if Data$(id).target$ then
17:         return                ▷ backtracking from $Data(id).label$ yields a shortest path
18:     for $d \in \mathcal{D}$ do
19:         $id' \leftarrow$ NeighborId$(id, d)$
20:         if $id'$ is valid then
21:             EnsureInitialized$(id', \psi)$
22:             $v' \leftarrow$ Location$(id')$
23:             if Data$(id).legality$ and Data$(id').legality$ indicate that $\{v, v'\}$ is legal then
24:                 $c' \leftarrow c + c^f_E(v, w) + c_V(v')$
25:                 if Data$(id').label = \emptyset$ then
26:                     Data$(id').label \leftarrow (id', id, c')$
27:                     $Q.insert($Data$(id').label)$
28:                 else if cost of Data$(id').label > c'$ then
29:                     Data$(id').label \leftarrow (id', id, c')$
30:                     adjust $Q$ for the decreased key of Data$(id').label$
31: return No $S$-$T$-path exists.
```

Our implementation in BonnRouteDetailed uses additional data structures and speed-up techniques to reduce memory usage and to speed up Algorithm 7.

We list some examples:

- We initialize certain data of a region r when the first node in the region is initialized, for example the array StorageLocation$_r$. Having very large regions makes this optimization less effective. Therefore, if the gap between two adjacent x-coordinates x_i and x_{i+1} is larger than a cutoff value d^*, we introduce $\lfloor \frac{x_{i+1}-x_i-1}{d^*} \rfloor$ additional coordinates spread uniformly between x_i and x_{i+1}. We do the same for the y-coordinates. Since this does not change the minimum or maximum coordinates we get another grid region partition.

- Labels are stored in a data structure called *label manager* that can perform the following operations in $O(1)$ amortized time:

  - CreateNonPermanentLabel(*label*) stores the label and returns its index.

  - ConvertToPermanentLabel(i) compresses the non-permanent label associated with index i into a permanent label, which stores only the data needed for backtracking, i.e the node identifier of the current node and that of the predecessor. The operation returns the index of the new permanent label and invalidates index i. The index ranges used for permanent and non-permanent nodes do not overlap.

  - GetNonPermanentLabel(i) returns a reference to the label associated with index i. The index must have been returned by CreateNonPermanentLabel more often than it was invalidated by ConvertToPermanentLabel.

  - GetPermanentLabel(i) returns a reference to the label associated with index i. The index must have been returned by a call of ConvertToPermanentLabel.

  - IsPermanentLabel(i) returns whether an index i is associated with a permanent label or with a non-permanent label.

  - InvalidLabelIndex returns an invalid index that is never associated with any permanent or non-permanent label.

  When the data for a node identifier *id* is initialized, its index is set to the index returned by InvalidLabelIndex. Once the first label l on *id* is created, we call CreateNonPermanentLabel(l) and store the returned index i. After *id* is permanently labeled, i.e. it is selected in the main loop and the label has been propagated to the neighbors, we call ConvertToPermanentLabel(i) and store the returned index. Implementing the label manager is straightforward.

  Compressing permanent labels saves memory, since it suffices to store the predecessor and id in the permanent label, i.e. storing the cost c is no longer necessary. In practice we need to store additional data for non-permanent labels that can also be discarded. This is very useful since on average the maximum number of labels that are unprocessed at any point in time is approximately 5-25% of the number of permanent labels at the end of the search. Storing a label index also has the advantage that we can check whether a node is already permanently labeled in $O(1)$ time. Once a node is permanently labeled no new labels for it have to be considered.

- Typically, most tracks are not usable with the wire models that are relevant in the current path search. We ignore these tracks and adapt checking oracle and track oracle accordingly. This can reduce the number of nodes and edges in the graph considerably. We note that this changes the locations where jogs can be placed and thus alters the instance slightly.

3.5 Track Oracle and Checking Oracle

This section provides more details on the track oracle and the checking oracle and their implementation in BonnRouteDetailed. The oracles have been introduced in Section 3.4 (see Definition 3.21 on page 42 and Definition 3.26 on page 48).

In BonnRouteDetailed most tracks are *global tracks* that span the entire length or width of the chip. These tracks are stored using so-called track patterns.

A *track pattern* is a triple $t = (z, s, (d_1, \dots, d_k))$, where $z \in \mathcal{Z}$ is a layer, $s \in \mathbb{Z}$ is an integer, and $d_1, \dots, d_k \in \mathbb{N}_{>0}$ are positive integers. Its set of track coordinates is

$$T_t := \left\{ s + n \left(\sum_{i=1}^{k} d_i \right) + \sum_{i=1}^{j} d_i | n \in \mathbb{Z} \text{ and } j \in \{0, \dots, k-1\} \right\}.$$

If z is horizontal there is a track spanning the entire width of the chip at each y-coordinate in T_t that intersects the chip area. Similarly, we have vertical tracks spanning the entire height of the chips if z is vertical.

Each layer stores the track patterns that are usable for each wire model. Note that the tracks intersecting a rectangle can be computed efficiently using the regular structure of the tracks. When a path search is started we collect the track patterns of all wire models that are usable in the search and use them as part of the track oracle.

There are additional *non-global tracks* that may be needed in the context of the current search. These tracks occur in the context of two special routing scenarios in which parts of the global wiring already satisfy the design rules. First, BonnRouteDetailed may be called after the locations of the global wires have been optimized by a *track assignment* which tries to avoid design rule violations while optimizing other objectives. Second, BonnRouteDetailed may be called on a fully detailed routed design that was altered slightly, e.g. by moving a small subset of the pins. This scenario is called engineering change orders (ECO) routing. In both scenarios BonnRouteDetailed should keep most of this legal global wiring, even if it is not placed on the global tracks. When routing a net we add additional tracks for the legal global wires that are relevant for the current path search and consider them in the track oracle. For more information on these routing scenarios and how BonnRouteDetailed handles them we refer to [Rabenstein, 2019]. In the future

BonnRouteDetailed could also add local tracks to make it easier to access pins that cannot be accessed legally using the global tracks.

The checking oracle is implemented by two data structures: the *detailed grid* and the *fast grid*.

The detailed grid stores all routing objects, i.e. blockages, wirings of nets, and pins. It supports three operations:

1. insert a shape or stick
2. remove a shape or stick
3. query all shapes and sticks intersecting a rectangle

At the beginning of BonnRouteDetailed the detailed grid is initialized with all routing objects that are given in the input and during BonnRouteDetailed it is always kept up-to-date using operations 1 and 2.

The detailed grid partitions each layer z into rectangular *subgrids*. Each subgrid is subdivided into stripes that span the entire subgrid in preferred dimension but are narrow in non-preferred dimension. Shapes and sticks are stored in each stripe they intersect. See Figure 3.12 for an illustration. For more details on the detailed grid see [Schulte, 2012; Klewinghaus, 2020].

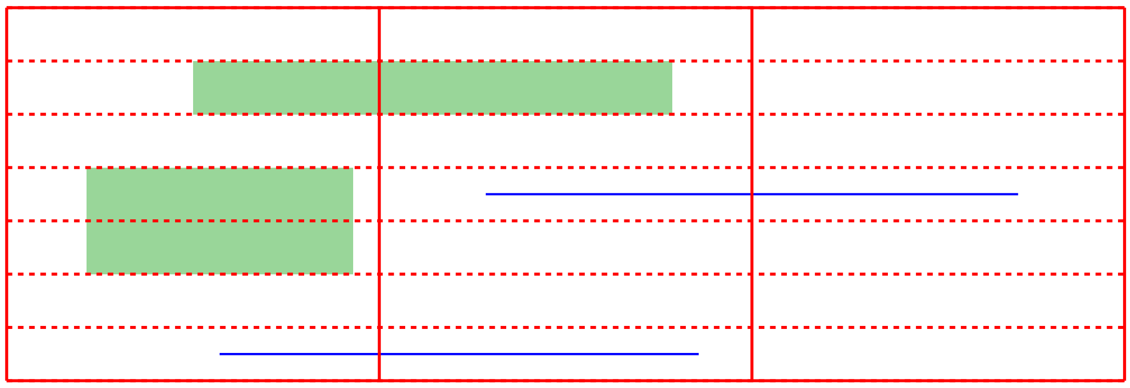


Figure 3.12: Three subgrids, whose borders are drawn as red lines on a horizontal layer. Each subgrid is partitioned into stripes. The borders of the stripes are drawn as dashed and non-dashed red lines. Each shape and each stick is stored in all stripes that it intersects. We note that the shapes associated with a stick are not stored in the detailed grid. Typically, subgrids and stripes are much larger compared to the shapes than in this example.

The detailed grid can be used to check whether a non-via edge e can be used with a certain wire model m by the following operations:

- Compute the shape $[x_1, x_2] \times [y_1, y_2] \times \{z\} = e + m$ resulting from using e with m.
- Compute an *influence area* $R_I = [x_1 - d_z, x_2 + d_z] \times [y_1 - d_z, y_2 + d_z] \times \{z\}$ such that no shape or stick not intersecting R_I can have a conflict with $e + m$.

- Query all relevant shapes and sticks intersecting R_I and check them against the shape.

The last step ignores shapes that should be ignored for the current path search, e.g. pins or wires that may be accessed or are tentatively removed in the current ripup and reroute sequence. Similarly, it may check additional shapes that were tentatively added in ripup and reroute. Moreover, additional constraints may be checked, e.g. arising from the chip area or from protections (see Section 4.6 for more information on protections).

Via edges can be checked by checking all three associated shapes with a similar procedure. Checking the via cut shape may require an influence area spanning multiple via layers.

Processing all checking queries using the detailed grid would be too slow. For this reason BonnRouteDetailed uses a fast grid that stores precomputed checking data and valid intervals for all edges of the track graph for the most important wire and via models. Just like the detailed grid the fast grid is organized by subgrids. Within each subgrid and on each track, adjacent nodes whose edges in all three positive dimensions are identical (for all relevant wire models and via models) are merged into intervals. The precomputed values for these intervals are organized in a binary search tree. Just like the detailed grid, the fast grid is always kept up-to-date.

For each path search an auxiliary data structure based on a quadtree is built. It provides the information whether the fast grid can answer a query in the context of the current search. For example, a query near a pin that the current search may need to access cannot be answered since that pin might block edges that would otherwise be legal. If the fast grid can be used, the interval tree is queried and the corresponding interval is returned (or part of the interval). Otherwise, the query is forwarded to the detailed grid.

For more details on the fast grid we refer to [Müller, 2009; Klewinghaus, 2020].

In practice the checking oracle and its output are slightly more complicated: An edge can be

- unconditionally usable,
- usable only if conflicts to modifiable / ignorable wiring are allowed, or
- unconditionally unusable.

Moreover, this data is provided for each wire and via model that is relevant for the current path search. This data is used to determine whether the edge can be used and to determine its cost.

3.6 Replacing the Algorithmic Core of BonnRouteDetailed

This section compares the new path search of BonnRouteDetailed presented in Chapter 3 and Chapter 4 of this thesis to the old path search which is based on [Hetzel, 1995, 1998] and labels intervals instead of individual nodes. See page 24 for a brief description of interval labeling and [Hetzel, 1995, 1998; Humpola, 2009; Nohn, 2012; Gester, 2015; Ahrens et al., 2015] for more details on the old path search. Because the old path search was deleted several years ago, a consistent comparison is no longer possible, but we can still highlight differences and discuss key statistics from the time when the old path search was replaced. We note that while the old path search does not compare favorably to the new path search it was used very successfully in BonnRouteDetailed and the IBM design flow for more than two decades. It was tailored to older technologies for which the interval labeling was much more effective and its disadvantages were much less important.

A key advantage of the new path search is that it supports more general cost functions. The old path search requires that for every layer the cost for going one unit in preferred dimension is one, since this must hold in every interval. In the new path search the cost can depend on the layer. This is useful for example for avoiding long segments on layers that are undesirable in the context of the current path search. Moreover, the new path search supports node costs. In the old path search node costs were not supported since nodes with non-zero node cost need their own interval. Instead, it supported *interval costs* that need to be paid when entering an interval. Interval costs are less useful since the length of intervals can vary and we may want to apply costs at individual locations rather than for intervals. To partially mitigate this disadvantage, the old path search has to split intervals into smaller intervals in certain cases, which makes it less efficient.

Replacing the old path search led to drastic improvements in routing quality, possibly due to improved cost modeling: The number of vias reduced by 5%, the wire length by 1.15%, and the number of scenic nets with detour of at least 25% compared to a Steiner tree estimate and length at least 25 µm was reduced by roughly one third.

Furthermore, the new path search is much simpler. Labeling intervals instead of nodes complicates the underlying algorithm of the old path search. The multi-labeling (see Section 4.5), which can respect many same-net rules in a correct-by-construction manner, does not combine well with interval-based labeling: intervals had to be split further during the path search and the overall algorithm and its implementation became very complicated. Moreover, the implementation of the new path search is much simpler and easier to extend. Most notably, the new path search was extended to allow both path searches and Steiner tree searches, as discussed in Chapter 5. This would not have been possible with the old path search.

As mentioned above, the old path search speeds up the search by labeling intervals instead of individual nodes. Nevertheless, the new path search had approximately the same runtime at the time when the old path search was replaced. Standard path searches became slightly slower but multi-label path searches became faster. Back then the grid region data structure from Section 3.4, which reduces the runtime of the new path search by approximately one fourth, was not yet implemented.

The grid region data structure also allows the new path search to handle complex regional track structures and to modify the routing graph locally, e.g. to simplify and improve pin access. This would have required major changes with the old path search.

Chapter 4

Same-Net Rule Aware Path Search

This chapter continues the description of an efficient and flexible path search algorithm that is the new algorithmic core of BonnRouteDetailed. This chapter focuses on the problem of computing paths that respect same-net rules. Section 4.1 introduces the problem, mentions related work, and gives an overview of our framework for avoiding same-net violations. Section 4.2 shows that given a two-dimensional grid graph and nodes s, t it is NP-complete to decide whether there is an s-t-path in which every maximal straight subpath has length at least two. Hence, obeying even very simple rules is NP-hard. Nevertheless, BonnRouteDetailed is very good at respecting same-net rules in practice: Section 4.3, Section 4.4, Section 4.5, and Section 4.6 present the components of the same-net rule aware path search framework and Section 4.7 describes how these components work together in BonnRouteDetailed. Its most important component is the multi-labeling that allows us to find edge progressions that satisfy same-net rules in a correct-by-construction manner. Our multi-labeling is more general and more efficient than that of [Nohn, 2012; Gester, 2015; Ahrens et al., 2015], which allows us to respect more same-net rules while being less restrictive. Section 6.3 presents experimental results demonstrating that using our framework can reduce the number of design rule violations by a factor of approximately 443.

4.1 Obeying Same-Net Rules

In modern technologies, connecting components of nets by shortest paths in the routing graph frequently leads to same-net errors, for example the one illustrated in Figure 4.1. The violations illustrated in the figure occur very frequently especially if jogs are forbidden on some wiring layers. Moreover, they often cannot or should not be fixed locally, since moving the vias further apart requires changing the tracks of one of the wires and keeping the same tracks requires at least two additional vias, which is undesirable. Thus, using a post-processing to avoid violations is insufficient and we need to consider same-net rules during the path search. Section 4.2 shows that finding paths respecting same-net rules is NP-hard, even for very simple rules. Thus, we cannot hope for a polynomial time algorithm guaranteeing that the same-net rules are satisfied.

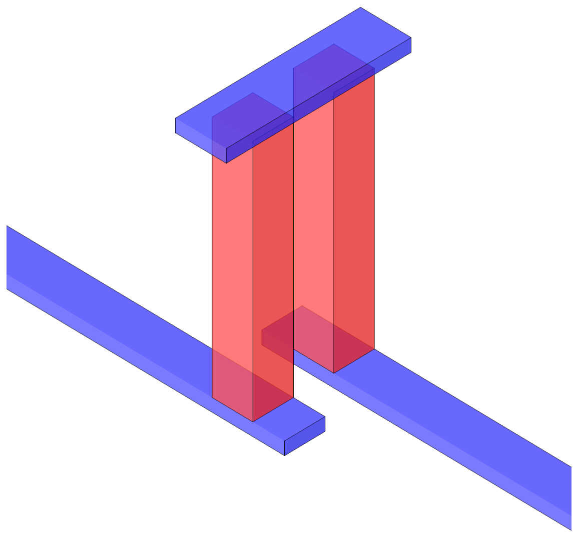


Figure 4.1: Vias of the same net need to satisfy a set of minimum distance rules similar to the diff-net rules for vias. These rules are usually violated if vias are used to switch to the neighboring track, as in the figure.

Nevertheless, we need to solve the problem in practice. We propose to use our *same-net rule aware path search* framework, consisting of four main components:

- The *same-net checking* can identify same-net errors and return their precise location and error type. It is sketched in Section 4.3.
- Every path computed by the path search is handed to a *post-processing* routine, which attempts to resolve same-net errors and other undesirable configurations. Section 4.4 provides some additional details. The post-processing of BonnRouteDetailed is an extended version of the one described in [Sterin, 2015].
- The *multi-labeling* is a powerful framework that allows us to find edge progressions that satisfy certain properties, specified by *label systems*. This allows us to satisfy same-net rules in a correct-by-construction manner. The framework and the label systems used in BonnRouteDetailed are described in Section 4.5. The multi-labeling we present is an extended and improved version of the one described in [Nohn, 2012; Ahrens et al., 2015; Gester, 2015].
- To avoid errors at the start and end of paths BonnRouteDetailed uses *protections*, as sketched in Section 4.6.

Finally, Section 4.7 describes how these components work together in the same-net rule aware path search framework. We present a *framework* for respecting same-net rules and do not focus too much on individual rules, giving only very simple examples. There are two main reasons: First, since BonnRouteDetailed is used in industry it must respect dozens of different error-types and many of them need to be considered in all or most of the four components. Presenting all details is not feasible. Second, some of the rules are confidential. Section 6.3 provides experimental results and demonstrates that our framework is effective at avoiding dozens of error-types on real-world instances.

In academia there are a few papers that consider same-net rule aware detailed routing but most are limited to one or two simple rules.

An algorithm for computing shortest rectilinear edge progressions with minimum segment lengths among rectilinear obstacles is proposed in [Maßberg and Nieberg, 2013]. It is used in the pin access framework of [Nieberg, 2011] to compute legal pin access paths. [Chang et al., 2013] propose an algorithm, called *MANA*, that uses dynamic programming to find optimal paths that satisfy the minimum area rules and an end-of-line spacing constraint. In contrast to [Maßberg and Nieberg, 2013] they allow paths consisting of two vias and a short pref-wire, if the resulting metal component can be extended legally. A similar approach is used by *Dr. CU* [Chen et al., 2019] to respect minimum area constraints. *Dr. CU 2.0* [Li et al., 2019] can respect a few additional rules. *DRAPS* [Gonçalves et al., 2019] proposes a path search algorithm that can respect minimum area constraints and same-net via spacing constraints and can handle multiple different via models on each via layer.

A different approach is used by *RegularRoute* [Zhang and Chu, 2011] and *TritonRoute* [Kahng et al., 2018]. The idea of both approaches is to find a simple layout for each net, decreasing the probability of same-net errors. In their main routing step they route layer by layer from bottom to top. Each layer is partitioned into so-called *panels*, thin stripes that have small width in non-preferred dimension and run across the entire layer. Each panel is routed by computing candidate wires that form the nodes of a conflict graph. There is an edge between two candidate wires if they are candidates realizing the same connection or if they conflict with each other. RegularRoute solves the resulting instance of the Maximum Weight Independent Set Problem heuristically, while TritionRoute solves it optimally using an ILP-solver. Moreover, TritonRoute considers some simple design rules when constructing the candidate wires.

Some papers discuss how to handle constraints imposed by special manufacturing techniques. Triple Patterning is discussed e.g. in [Lin et al., 2012; Ma et al., 2012]. Self-aligned double patterning is discussed e.g. in [Mirsaeedi et al., 2011; Gao and Pan, 2012; Xu et al., 2015, 2016]. We note that while we also consider multiple patterning, which generalizes triple patterning, we do not consider self-aligned double patterning.

4.2 Respecting Same-Net Rules is NP-Hard

This section shows that finding paths that obey same-net rules is NP-hard, even for very simple rules. We consider the following problem:

Path with Minimum Segment Lengths Problem

Input: A two-dimensional grid graph G, source and target nodes $s, t \in V(G)$, and a number $\alpha > 0$.

Task: Decide whether an s-t-path $v_1, P_1, v_2, \dots, v_k, P_k, v_{k+1}$ exists, such that for each $i \in \{1, \dots, k\}$ the path P_i is straight and $|E(P_i)| \geq \alpha$.

This problem arises naturally when considering simple design rules and basic constraints. A minimum segment length threshold can be derived from multiple same-net rules, e.g. the *Minimum Edge Length* rule that specifies that the length of each edge of the polygonal boundary must be no smaller than a threshold value. Forbidding cycles is also natural, since the set of sticks connecting a net should form a Steiner tree.

We now prove that the problem is NP-complete.

Theorem 4.1. Path with Minimum Segment Lengths Problem *is NP-complete, even for $\alpha = 2$.*

Proof. Membership in NP is obvious.

We describe a polynomial reduction from a restricted version of the Planar 3Sat Problem [Lichtenstein, 1982], the Planar Cycle 3Sat Problem (see [Kratochvíl et al., 1991], Section 4). An instance is a triple $(X, Z, \mathcal{C})$, where X is the set of variables, Z is a set of clauses over X, $\mathcal{C}$ is a directed cycle through the clauses, and the graph $G = (X \cup Z, \{(x,z) \in X \times Z | x \in z \vee \neg x \in z\} \cup E(\mathcal{C}))$ is planar. We assume that each variable occurs in at most three clauses and exactly once negated. The problem remains NP-complete under these assumptions (see Lemma 2.1 of [Fellows et al., 1995]).

Step 1: Embedding the graph into a two-dimensional grid:

We construct a modified graph G' with node degrees at most three by splitting clause-nodes which can have degree up to five. In order to make certain that the graph remains planar, we compute a planar embedding and ensure that the embedding can be modified locally.

For each clause $z \in Z$, we perform the following local replacement: Let $x_1, \ldots, x_k \in \Gamma(z) \cap X$ and $z_1, z_2 \in \Gamma(z) \setminus X$ be the neighbors of z. Without loss of generality assume that there are faces F_1 and F_2 such that x_1 and z_1 are on the border of F_1 and x_k and z_2 are on the border of F_2. We remove the node z and all adjacent edges and replace them by new clause-nodes $z^{x_1}, \ldots, z^{x_k}$ and edges $\{z^{x_1}, x_1\}, \ldots, \{z^{x_k}, x_k\}$, $\{z_1, z^{x_1}\}$, $\{z^{x_1}, z^{x_2}\}, \ldots, \{z^{x_{k-1}}, z^{x_k}\}$, $\{z^{x_k}, z_2\}$. The choice of x_1, z_1, x_k, and z_2 ensures that the graph remains planar and the blue and orange edges form a cycle through all clause-nodes. See Figure 4.2 for an illustration.

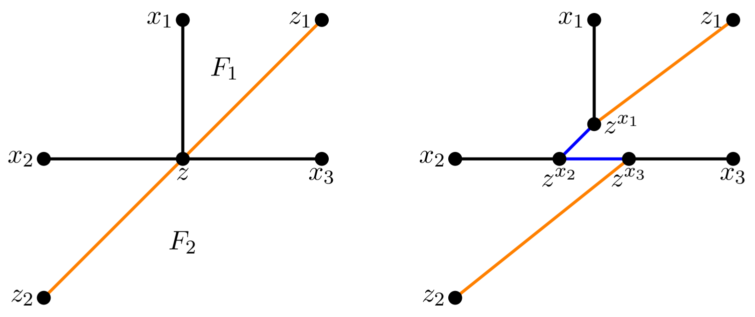


Figure 4.2: The configuration on the left is replaced by the one on the right. Note that the graph remains planar.

In the modified graph G' each node has degree at most three, the graph is planar, and it admits a cycle $\mathcal{C}'$ through the clause-nodes. By [Liu et al., 1998] an embedding of G' into an $(|V(G')| + |E(G')| + 1) \times (|V(G')| + |E(G')| + 1)$ grid can be found in linear time, such that edges are non-crossing paths. See Figure 4.3 for an illustration.

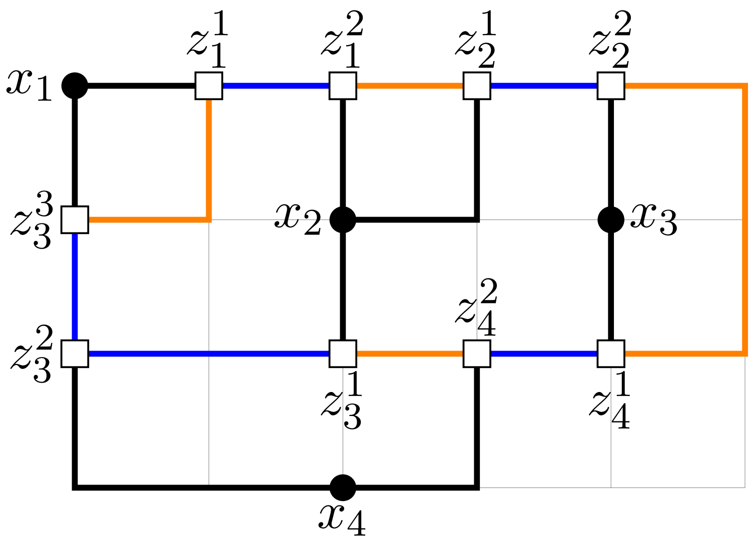


Figure 4.3: Embedding of an example instance with clauses $z_1 = \{x_1, \neg x_2\}$, $z_2 = \{x_2, x_3\}$, $z_3 = \{\neg x_1, x_2, x_4\}$, and $z_4 = \{\neg x_3, \neg x_4\}$ (black edges) and a cycle through the clause-nodes (blue and orange edges). Only the thick edges are in the graph.

Step 2: Modifying the instance:

We will modify the graph by replacing variable nodes and clause nodes by gadgets. The path with minimum segment lengths of 2 will need to follow a path through all clause-gadgets that is derived from $\mathcal{C}'$. The instance will be designed such that for each clause the path needs to make a detour from one of its clause-gadgets to the corresponding variable-gadget and back again, before advancing to the first gadget of the next clause.

Therefore, we will double the paths between clause-gadgets and variable-gadgets. The variable-gadgets will ensure that a variable cannot be used both negated and non-negated simultaneously. We will also need two alternative paths between the gadgets belonging to the same clause. Entering or leaving a clause-gadget through one of the paths indicates that we already traversed a variable-gadget for the current clause, i.e. that it is already satisfied. The other path has the opposite meaning. We now describe the modifications in more detail.

The gadgets require additional space. Therefore, we add 17 additional grid lines between adjacent ones and before the first and after the last one. Moreover, we clear the area with l_∞-distance at most 7 to any clause- or variable-node to fill it with gadgets later on. Note that all resulting paths have minimum segment lengths of at least 4. See Figure 4.4 for an illustration.

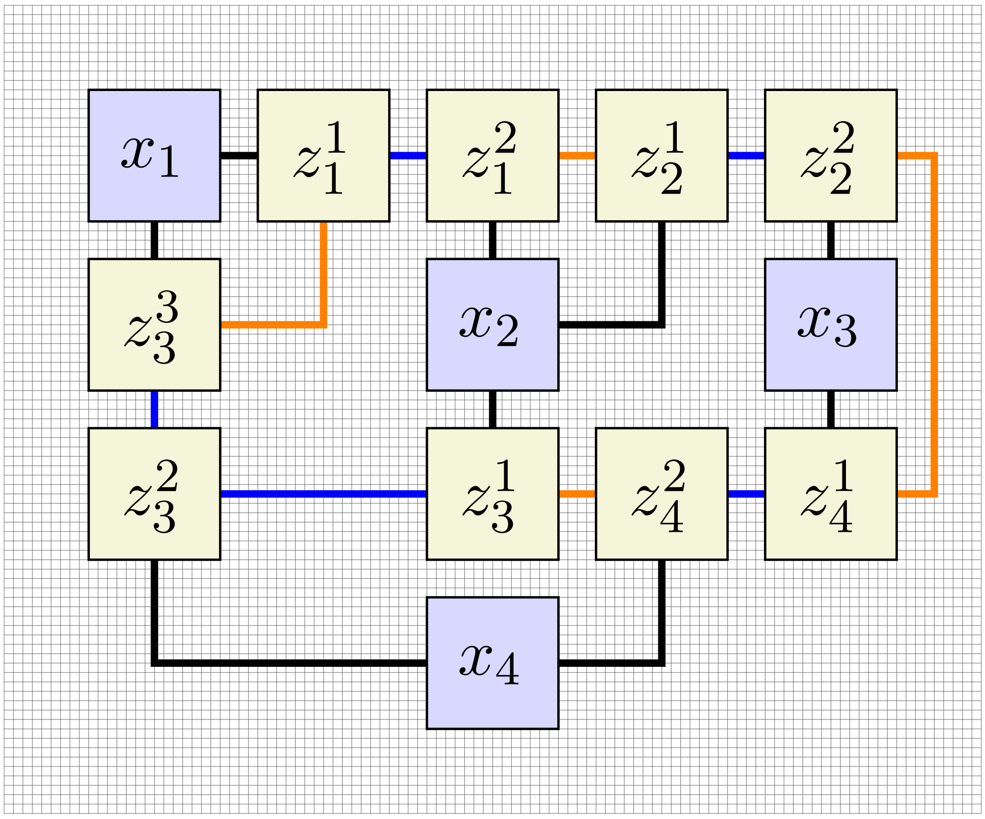


Figure 4.4: Instance after adding grid lines and clearing space for gadgets. Only the thick edges are in the graph.

Then we double the paths between clause-gadgets and variable-gadgets and the paths between clause-gadgets belonging to the same clause. We do this in such a way that the two paths run parallel to the original path with an l_∞-distance of 1 and start and end at the border of the areas we cleared for gadgets. We also modify the paths between clause-gadgets from different clauses such that they start one unit before the original start node in clockwise order and end one unit after the original end in clockwise order. We make all these modifications in such a way that all paths keep minimum segments lengths of at least 2, no two paths intersect each other, and no path intersects the area we cleared for the gadgets, except in their start- and endpoints. For the sake of brevity, we omit a detailed description. Figure 4.5 provides a rough sketch how these modifications can be done. See Figure 4.6 for an illustration of the resulting instance.

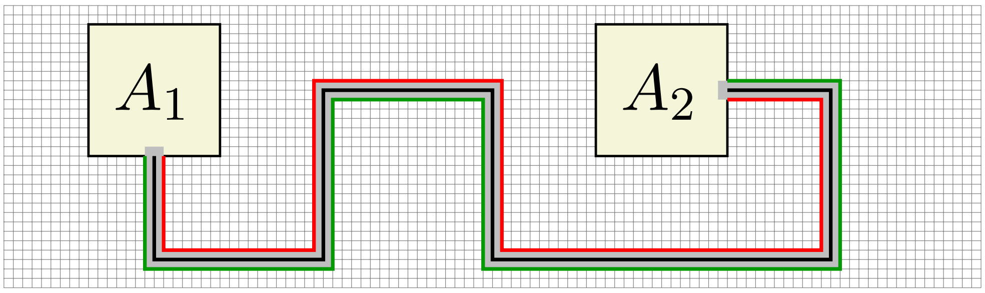


Figure 4.5: Illustration of the construction of the parallel paths. The original black path P is replaced by the red and green paths. The l_∞-ball $B^1_{l_\infty}(P)$ with radius 1 around P, is drawn in gray. The set of interior points of the new paths is $\partial(B^1_{l_\infty}(P)) \setminus (A_1 \cup A_2)$, where A_1 and A_2 are the areas of the gadgets. This yields exactly two paths because the segments have length ≥ 3 and non-intersecting segments have distance ≥ 3. Note that this construction reduces the length of segments by at most two and thus the paths have minimum segment lengths of at least two.

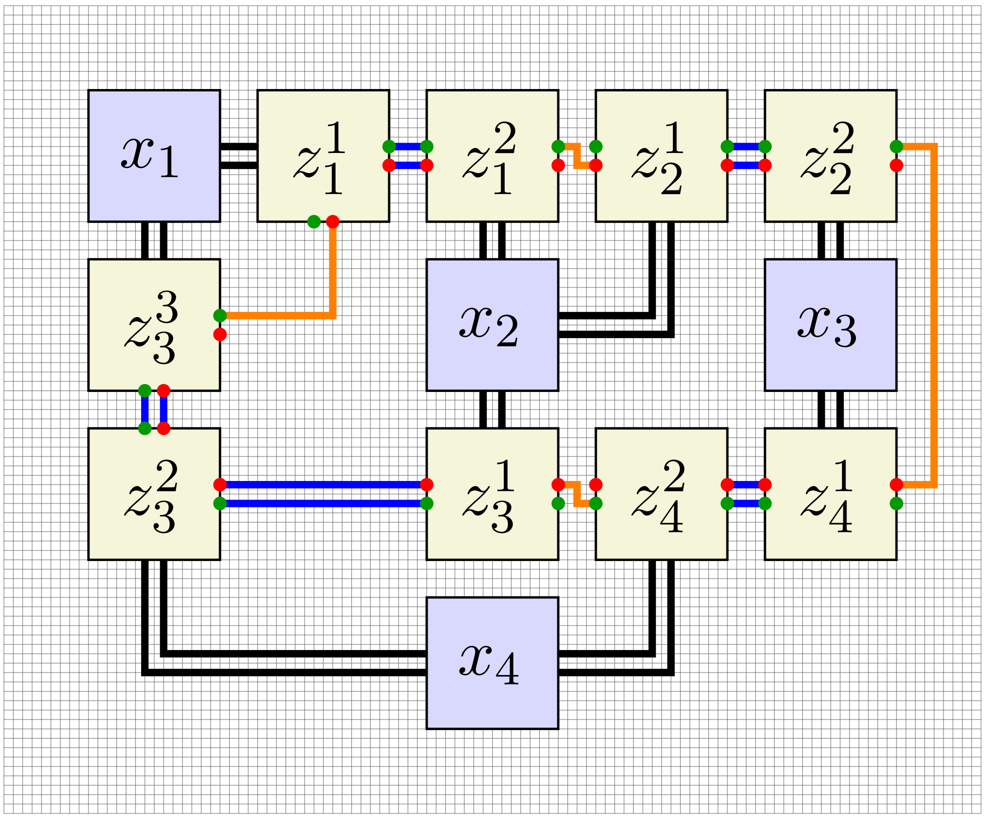


Figure 4.6: Instance after adding grid lines, clearing space for gadgets, and modifying the paths between the gadgets. Entering or leaving a clause-gadget with a green node indicates that we already traversed a variable-gadget for the current clause, i.e. that it is satisfied already. Red nodes have the opposite meaning. Only the thick edges are in the graph.

Step 2.1: Clause-gadgets:

A version of one of the gadgets in Figure 4.7 is used for each clause-node:

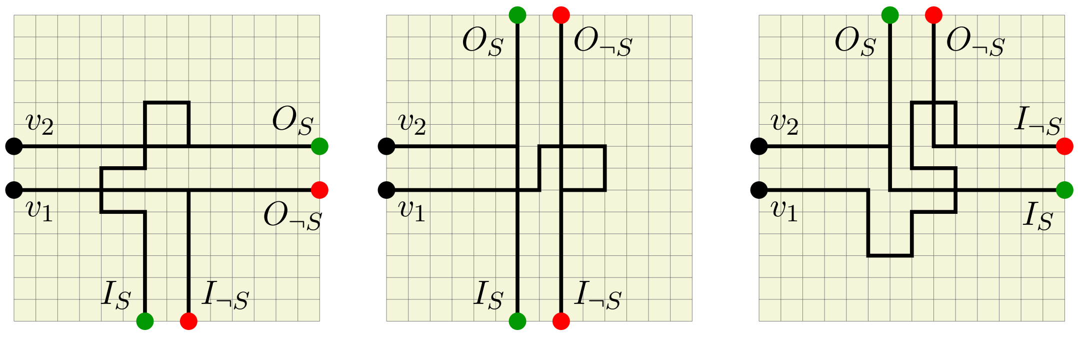


Figure 4.7: Clause-gadgets. The "inputs" of the gadget are I_S, indicating that the clause is satisfied already and $I_{\neg S}$, indicating that the clause is not satisfied yet. The "outputs" O_S and $O_{\neg S}$ have analogous meaning. The nodes v_1 and v_2 will be connected to a variable-gadget.

Each of the gadgets has the following properties if we require paths with minimum segment lengths of 2:

1. There is an I_S-O_S-path and an $I_{\neg S}$-$O_{\neg S}$-path.
2. There is an $I_{\neg S}$-O_S-path, if and only if we can pass through the variable-gadget to get from v_1 to v_2.
3. There is no I_S-$O_{\neg S}$-path.
4. I_S comes directly after $I_{\neg S}$ in clockwise order and O_S comes directly before $O_{\neg S}$.

We position the gadget such that I_S and $I_{\neg S}$ lead to the previous clause-gadget in $\mathcal{C}'$, O_S and $O_{\neg S}$ to the next clause-gadget in $\mathcal{C}'$, and v_1 and v_2 are connected through the variable-gadget. A gadget for each situation can be derived from the ones in Figure 4.7 by rotation and the following relabeling operation:

- swap the labels I_S with $O_{\neg S}$ and $I_{\neg S}$ with O_S
- recolor the nodes from red to green and vice versa.

Note that the properties 1-4 are preserved when rotating or relabeling.

Step 2.2: Variable-gadgets:

A version of the gadget in Figure 4.8 is used for each variable:

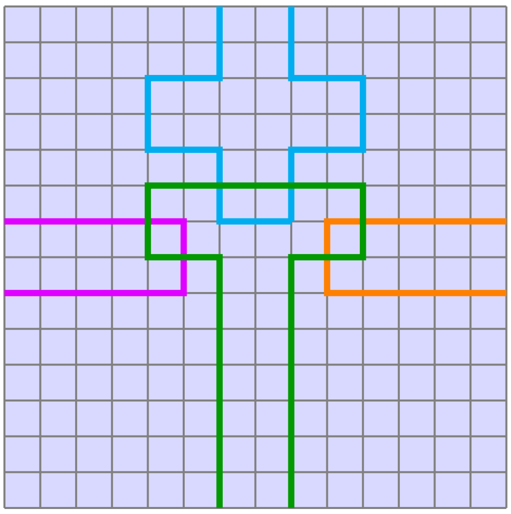


Figure 4.8: Variable-gadget. The gadget is oriented such that the green path points to the clause using the variable in its negated form. Recall that we assumed exactly one of the three appearances of each literal to be negated.

With minimum segment lengths of 2 any path through that gadget must use all edges of one color but no edges of any other color. The gadget may be traversed more than once. There are the following options:

1. using arbitrary many of the purple, cyan, and orange paths or
2. using the green path but no other path.

Therefore, we rotate the gadget such that the unique negated use corresponds to the part containing the green path.

Step 2.3: Source and target:

We arbitrarily select an edge $(z_t, z_s) \in E(\mathcal{C})$ of the cycle $\mathcal{C}$ through the clauses. This yields a first clause z_s and a last clause z_t. We put the source s at $I_{\neg S}$ of the first clause-gadget corresponding to z_s and the target t at O_S of the last one corresponding to z_t. Moreover, we delete the direct s-t-path. See Figure 4.9 for an illustration of the final instance.

Step 3: Final analysis:

We claim that the resulting graph admits an s-t-path with minimum segment lengths of 2 if and only if Z is satisfiable. Note that any such path must traverse all clause-gadgets. Moreover, for any clause $z \in Z$ the first clause-gadget corresponding to z is entered through a red node and the last clause-gadget corresponding to z is left through a green node. That means that we need to switch from a red node to a green one within

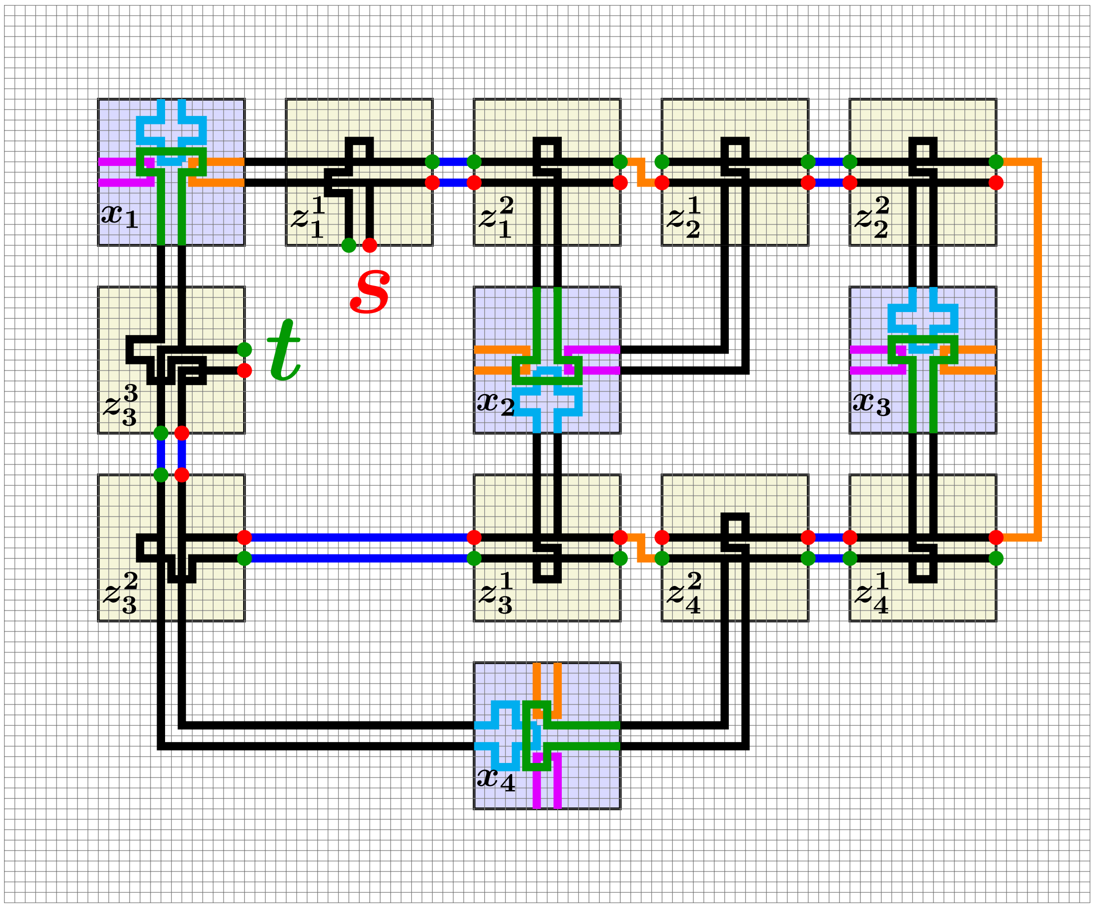


Figure 4.9: Instance of the Path with Minimum Segment Lengths Problem for the Planar Cycle 3Sat Problem-instance with clauses $z_1 = \{x_1, \neg x_2\}$, $z_2 = \{x_2, x_3\}$, $z_3 = \{\neg x_1, x_2, x_4\}$ and $z_4 = \{\neg x_3, \neg x_4\}$. Only the thick edges are in the graph.

one of the corresponding clause-gadgets. This requires visiting the corresponding variable-gadget. The variable-gadgets ensure that a variable cannot be used in both its negated and its non-negated form within the same solution.

Let P be such an s-t-path. We define a truth assignment by setting a variable to *false* if P contains the green path of the corresponding variable-gadget. Otherwise it is set to *true*. By the properties of the gadgets and the construction of the paths between the gadgets, each clause contains a literal that is *true*.

Conversely, any satisfying truth assignment defines a set of literals that is set to *true*. We pick one *true* literal for each clause arbitrarily and use the corresponding variable-gadget to get from $I_{\neg S}$ to O_S for the corresponding clause-gadget. The path can be completed by appropriate I_S-O_S-paths, $I_{\neg S}$-$O_{\neg S}$-paths, and paths connecting the clause-gadgets of different clauses. $\square$

We note that the Path with Minimum Segment Lengths Problem is NP-complete because we are interested in a *path* with minimum segment lengths. The problem of finding an edge-progression with minimum segment lengths can be solved in $O(|V(G)|\log|V(G)|)$ time. This is an easy corollary of the algorithm for finding rectilinear edge progressions with minimum segment length among rectilinear obstacles of [Maßberg and Nieberg, 2013].

Corollary 4.2 ([Maßberg and Nieberg, 2013])**.** *If we are interested in an edge progression instead of a path in the* Path with Minimum Segment Lengths Problem*, the problem can be solved in* $O(|V(G)|\log|V(G)|)$ *time.*

We note without proof that Corollary 4.2 also follows from Theorem 4.8 on page 74.

4.3 Same-Net Checking

BonnRouteDetailed has an internal *same-net checking* that can check connected sets of shapes and sticks for same-net errors and undesirable configurations, e.g. layer fuses, introduced in Section 4.5 on page 80. It can, for example, check entire nets, paths, or individual metal components. The same-net checking currently supports 18 different error classes. In addition to statistics on violations it returns a set of *error-annotations*, each consisting of the type of the violation and an error-rectangle, specifying the precise location.

In its initialization, the same-net checking computes the layer-wise connected metal components and the vias connecting them. Moreover, it computes their polygonal boundary. This takes $O(n\log n + p)$ time, where n is the number of sticks and shapes and p is the number of edges in the boundary [Güting, 1984]. Once this data is computed most constraints can be checked by simple algorithms in near-linear time. For an example see Figure 4.10.

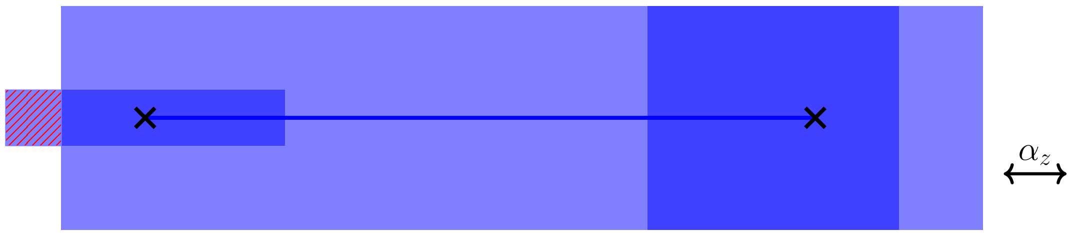


Figure 4.10: The minimum adjacent edge length rule, introduced in Section 2 on page 6 can be checked by considering each pair of adjacent edges of the polygonal boundary. There is a violation if there is a pair $\{e, f\}$ such that both edges have length less than the threshold value α_z. Its error-rectangle is the bounding box of e and f. Since the polygonal boundary is available checking this rule requires $O(p)$ time. In this example, there are two violations involving the three short edges on the left. Both have the same error-rectangle, that is hatched in red.

The same-net checking is an important part of BonnRouteDetailed and is used heavily in the post-processing presented in Section 4.4, to control the multi-labeling as described in Section 4.5.2, and to evaluate different solutions in the same-net rule aware path search framework presented in Section 4.7.

The current same-net checking in BonnRouteDetailed was created in joint work with many former and present members of the BonnRouteDetailed team, most notably Michael Gester, Niko Klewinghaus, Stefan Rabenstein, Andrei Sterin, and Mirko Speth.

4.4 Post-Processing

The post-processing modifies paths found by the path search locally to reduce the number of same-net errors. It uses the error-annotations returned by the same-net checking to determine error types and precise error-locations and to check whether modified paths are better. Currently, the post-processing of BonnRouteDetailed consists of twelve fixing routines.

All of these routines make only very local changes to the path. Let P be a path consisting of a set of sticks $\mathcal{S}$. All but two routines do not modify the geometry of the sticks and resolve violations only by modifying their corresponding shapes. This is done by changing their wire and via models or by using so-called *endstyles* that allow us to enlarge the shapes associated with a stick. Many routines use the same-net checking to evaluate the original solution and various modified solutions, taking the modification that minimizes a weighted error function. See Figure 4.11 for an illustration.

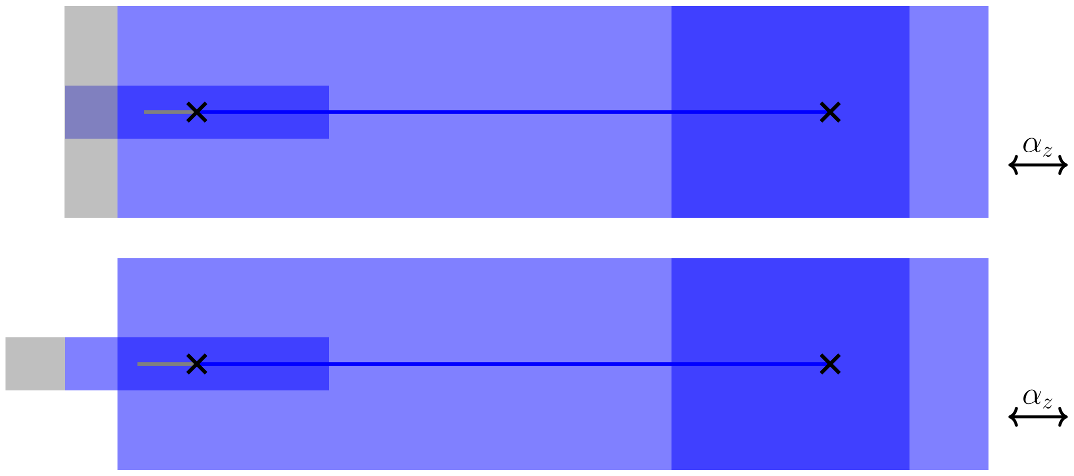


Figure 4.11: The minimum adjacent edge length violation from Figure 4.10 can be resolved by extending the wire stick by the gray endstyle, as indicated on the top. Alternatively, we can expand the pad of the via by using an endstyle, as indicated on the bottom. Depending on other same-net rules none, one, or both of the new shapes may cause other errors. Moreover, extending the metal area can lead to diff-net rule violations.

The post-processing of BonnRouteDetailed was developed in joint work with Andrei Sterin and Niko Klewinghaus. For more details we refer to [Sterin, 2015].

4.5 Multi-Labeling

Multi-labeling uses *label systems* to impose additional constraints on edge progressions and paths. It was introduced in [Nohn, 2012] and also presented in [Ahrens et al., 2015; Gester, 2015]. We present a generalized and more efficient version. Moreover, in Section 4.5.1, we present the label systems currently used by BonnRouteDetailed, which are much more efficient at avoiding same-net errors. While the previous works use up to eight different label systems we have hundreds of label systems that are built on demand out of a few building blocks. Some definitions and results in this section are similar to definitions and results in [Ahrens et al., 2015; Gester, 2015].

Definition 4.3 (label system). *A* label system *is a pair (L, ξ), where L is a finite set of* label types *and $\xi : L \times \mathcal{Z} \times \mathcal{D} \to 2^{L \times \mathbb{N}_{>0} \times \mathbb{N}_{\geq 0}}$ is a* label transition function *that satisfies the following property. For all $l_1, l_2 \in L, z \in \mathcal{Z}$, and $dir \in \{\rightarrow, \leftarrow, \uparrow, \downarrow\}$*

1. *$(l_2, d, c_\xi) \in \xi(l_1, z, dir)$ implies $(l_2, 1, 0) \in \xi(l_2, z, dir)$.*

The transition function ξ provides all transitions from a label type $l_1 \in L$ in a given direction $dir \in \mathcal{D}$ and for a given layer $z \in \mathcal{Z}$. A transition $(l_2, d, c_\xi) \in \xi(l_1, z, dir)$ means that when starting with label type l_1 on layer z we can switch to label type l_2 by going at least d units in direction dir. On top of the edge and node costs we need to pay a transition cost of c_ξ. The following definition formalizes which kinds of paths and edge progressions are valid.

Definition 4.4 (multi-label progression, multi-label path). *Let G be a routing graph, let c_V, c_E be node and edge cost functions, and let $\mathcal{L} = (L, \xi)$ be a label system. A* multi-label s-t-progression *for G and $\mathcal{L}$ is a pair (P, ϕ), where P is an edge progression $v_1, P_1, v_2, \ldots, v_k, P_k, v_{k+1}$ such that*

- *$v_1 = s$ and $v_{k+1} = t$,*

for each $i \in \{1, \ldots, k\}$

- *P_i is a straight v_i-v_{i+1}-path*
- *$\mathrm{Dir}(v_i, v_{i+1}) \in \{\swarrow, \nearrow\}$ implies $|E(P_i)| = 1$*

and $\phi : \{v_1, \ldots, v_{k+1}\} \to L$ is a function assigning a label type to each node, such that for every $i \in \{1, \ldots, k\}$ there is $(\phi(v_{i+1}), d, c_\xi) \in \xi(\phi(v_i), z, \mathrm{Dir}(v_i, v_{i+1}))$ with $\|v_{i+1} - v_i\| \geq d$.

The cost of (P, ϕ) is given by

$$\begin{aligned} c_{\mathcal{L}}(P) &:= c_E(P) + c_V(P) \\ &\quad + \sum_{i \in \{1,\ldots,k\}} \min\{c_\xi | (\phi(v_{i+1}), d, c_\xi) \in \xi(\phi(v_i), z, \mathrm{Dir}(v_i, v_{i+1})) \text{ and } \|v_{i+1} - v_i\| \geq d\}, \end{aligned}$$

where $c_E(P)$ and $c_V(P)$ denote the edge and node cost of the edge progression P and may pay multiple times for edges or nodes if they are used more than once.

If P is a path we call (P, ϕ) a multi-label s-t-path.

An edge progression or path that can be extended to a multi-label path or multi-label progression is called a valid progression *or* valid path.

In a label system (L, ξ) the transition function ξ provides the transitions for a given label type, layer, and direction. That allows us to have multiple transitions with different required distance and different transition costs between the same label types, on the same layer, and in the same direction. This can be very useful when designing label systems that should avoid same-net errors if possible, but still find a path with errors if there is no legal path. Moreover, it allows us to retrieve all transitions that are relevant for a pair of node and label type efficiently by querying ξ with the appropriate values.

We note that [Nohn, 2012; Ahrens et al., 2015; Gester, 2015] store the transitions in a 4-dimensional matrix, indexed by the layer, both involved label types, and the direction. Each entry stores the minimum required distance value and the transition cost. This does not allow different transitions with different required distances and transition costs. We also note that their definition of label systems requires a property similar to property 1 of Definition 4.3 for all directions, including positive and negative z-direction. In that sense our multi-labeling is more general.

MULTI-LABEL PATH SEARCH PROBLEM

Input: A path search instance (G, c_E, c_V, S, T) and a label system $\mathcal{L}$.

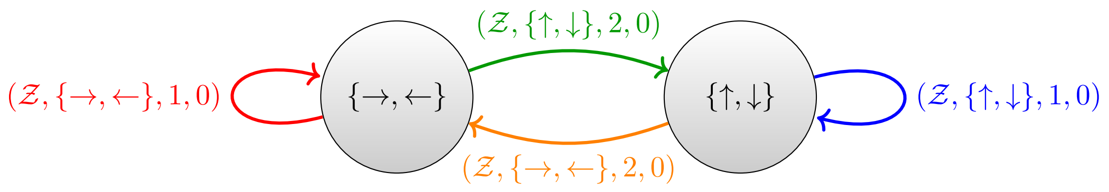
Task: Find a cost-minimal multi-label S-T-path with respect to $\mathcal{L}$ in G or decide that no such path exists.

See Figure 4.12 and Figure 4.13 for an illustration. We call graphs as in Figure 4.12 *transition graphs.* We use them extensively in Section 4.5.1 to define label systems. Sometimes we omit the last element c_ξ of an edge-label (Z, D, d, c_ξ), indicating that the transition cost is zero.

We note that the label system does not permit any control over the label types used at start and end. Section 4.5.1 provides a more general framework for controlling what happens at source and target nodes in the section on the SourceRestriction and TargetRestriction extension.

[Gester, 2015; Ahrens et al., 2015] showed that a version of the MULTI-LABEL PATH SEARCH PROBLEM is NP-hard. The proof uses label systems with $\theta(|V(G)|)$ label types and 2 layers. Most label systems used by BonnRouteDetailed have few label types and the number of label types does not depend on the number of nodes in the graph. In practice, the average number across all path searches is approximately 2. Therefore, we show the following stronger result:

Theorem 4.5. *The* MULTI-LABEL PATH SEARCH PROBLEM *is NP-hard even in two-dimensional grid graphs and with two label types.*



$(\mathcal{Z}, \{\rightarrow, \leftarrow\}, 1, 0)$
$\{\rightarrow, \leftarrow\}$
$(\mathcal{Z}, \{\uparrow, \downarrow\}, 2, 0)$
$(\mathcal{Z}, \{\rightarrow, \leftarrow\}, 2, 0)$
$\{\uparrow, \downarrow\}$
$(\mathcal{Z}, \{\uparrow, \downarrow\}, 1, 0)$

Figure 4.12: Transition graph of the label system $\mathcal{L}_{\geq 2} = (L, \xi)$ with $L = \{\{\rightarrow, \leftarrow\}, \{\uparrow, \downarrow\}\}$. An edge from label type l_1 to label type l_2 labeled as (Z, D, d, c_ξ) means $(l_2, d, c_\xi) \in \xi(l_1, z, dir)$ for all $z \in Z$ and $dir \in D$. In this example the transition function ξ is defined by $\xi(\{\rightarrow, \leftarrow\}, z, \rightarrow) = \xi(\{\rightarrow, \leftarrow\}, z, \leftarrow) = \{(\{\rightarrow, \leftarrow\}, 1, 0)\}$, $\xi(\{\uparrow, \downarrow\}, z, \uparrow) = \xi(\{\uparrow, \downarrow\}, z, \downarrow) = \{(\{\uparrow, \downarrow\}, 1, 0)\}$, $\xi(\{\rightarrow, \leftarrow\}, z, \uparrow) = \xi(\{\rightarrow, \leftarrow\}, z, \downarrow) = \{(\{\uparrow, \downarrow\}, 2, 0)\}$, $\xi(\{\uparrow, \downarrow\}, z, \rightarrow) = \xi(\{\uparrow, \downarrow\}, z, \leftarrow) = \{(\{\rightarrow, \leftarrow\}, 2, 0)\}$, and $\xi(l, z, \nearrow) = \xi(l, z, \swarrow) = \emptyset$ for every $z \in \mathcal{Z}$ and $l \in L$. The label system does not allow any vias.

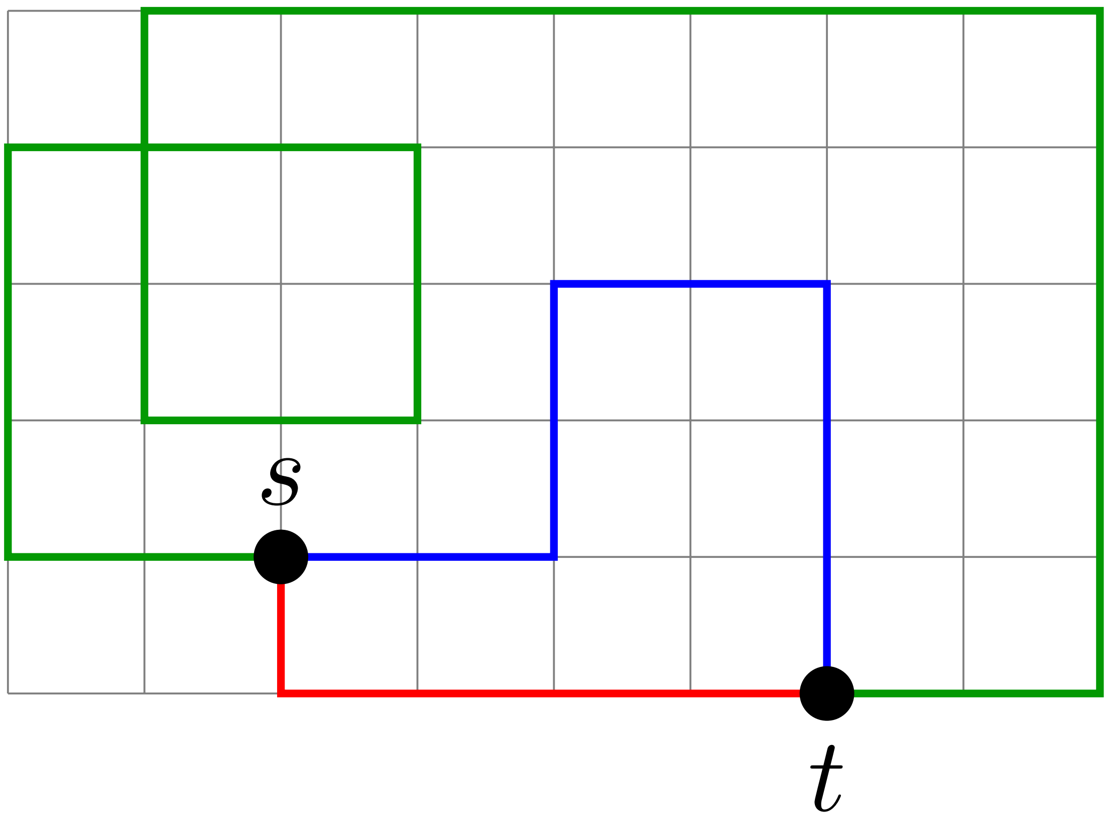


Figure 4.13: Example of multi-label progressions and paths for the label system $\mathcal{L}_{\geq 2}$ from Figure 4.12. The red path is a shortest valid s-t-path with respect to $\mathcal{L}_{\geq 2}$. The blue path is a shortest valid s-t-path with respect to $\mathcal{L}_{\geq 2}$ if we may only start with label type $\{\rightarrow, \leftarrow\}$. The green edges form a valid s-t-progression but not a path.

Proof. We show that PATH WITH MINIMUM SEGMENT LENGTHS PROBLEM with $\alpha = 2$ polynomially transforms to MULTI-LABEL PATH SEARCH PROBLEM in two-dimensional grid graphs and with two label types.

Let G, s, and t be an instance of the PATH WITH MINIMUM SEGMENT LENGTHS PROBLEM with $\alpha = 2$. We define four graphs $G^{\rightarrow}$, $G^{\leftarrow}$, $G^{\uparrow}$, and $G^{\downarrow}$ by

$$\begin{aligned} V(G^{dir}) &= V(G) \\ E(G^{dir}) &= E(G) \setminus \{\{s,v\} \in E(G) | \operatorname{Dir}(s,v) \neq dir \\ &\qquad\qquad \setminus \{\{s^{dir}, v\} \in E(G) | \{s, s^{dir}\} \in E(G), \operatorname{Dir}(s, s^{dir}) = dir, \\ &\qquad\qquad\quad dir \notin \{\operatorname{Dir}(v, s^{dir}), \operatorname{Dir}(s^{dir}, v)\}\} \end{aligned}$$

in which the edges to the neighbors of s that run in a different direction are removed and edges that run orthogonal to dir are removed from the neighbor of s in direction dir (if they exist). See Figure 4.14 for an illustration.

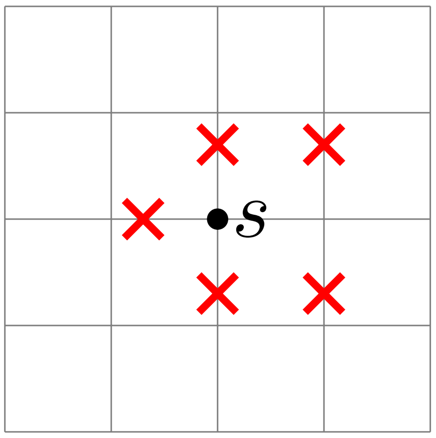


Figure 4.14: The edges marked with a cross are removed in $G^{\rightarrow}$ if they are in the grid graph G. This forces any path from s in $G^{\rightarrow}$ to start with two edges running in direction $\rightarrow$. The graphs G^{dir} for $dir \in \{\leftarrow, \uparrow, \downarrow\}$ have an analogous property.

First assume that t is not a neighbor of s. We claim that the instance of the PATH WITH MINIMUM SEGMENT LENGTHS PROBLEM is solvable if and only if $G^{\rightarrow}$, $G^{\leftarrow}$, $G^{\uparrow}$, or $G^{\downarrow}$ admit a multi-label s-t-path with respect to $\mathcal{L}_{\geq 2}$, where $\mathcal{L}_{\geq 2}$ is the label system from Figure 4.12.

Let $dir \in \{\rightarrow, \leftarrow, \uparrow, \downarrow\}$ and let (P, ϕ) be a multi-label s-t-path in G^{dir} that minimizes the number of straight paths k in $P = v_1, P_1, v_2, \dots, v_k, P_k, v_{k+1}$. Then for $i \in \{2, \dots, k\}$ we have $\phi(v_i) \neq \phi(v_{i+1})$, implying $|E(P_i)| \geq 2$. The construction of G^{dir} and $\{s,t\} \notin E(G)$ imply $|E(P_1)| \geq 2$. Thus P is a solution of the instance of the PATH WITH MINIMUM SEGMENT LENGTHS PROBLEM.

Conversely, given an s-t-path $P = v_1, P_1, v_2, \dots, v_k, P_k, v_{k+1}$, where for $i \in \{1, \dots, k\}$ the path P_i is straight and $|E(P_i)| \geq 2$ we define:

$$\phi(v_1) = \begin{cases} \{\leftarrow, \rightarrow\}, & \mathrm{Dir}(v_1, v_2) \in \{\uparrow, \downarrow\} \\ \{\uparrow, \downarrow\}, & \mathrm{Dir}(v_1, v_2) \in \{\leftarrow, \rightarrow\} \end{cases}$$
$$\phi(v_{i+1}) = \begin{cases} \{\leftarrow, \rightarrow\}, & \mathrm{Dir}(v_i, v_{i+1}) \in \{\leftarrow, \rightarrow\} \\ \{\uparrow, \downarrow\}, & \mathrm{Dir}(v_i, v_{i+1}) \in \{\uparrow, \downarrow\}. \end{cases}$$

(P, ϕ) is a multi-label s-t-path with respect to $\mathcal{L}$ in $G^{\mathrm{Dir}(v_1,v_2)}$, because going to label type $\{\leftarrow, \rightarrow\}$ and $\{\uparrow, \downarrow\}$ is always allowed in $\mathcal{L}_{\geq 2}$ when going at least 2 units in one of the contained directions.

In the case that t is a neighbor of s the argument is the same, except that we have to ignore the graph G^{dir} for $dir = \mathrm{Dir}(s, t)$. □

Since the Multi-Label Path Search Problem is NP-hard we consider the Multi-Label Progression Search Problem in which we search for a multi-label progression instead of a multi-label path.

Multi-Label Progression Search Problem
Input: A path search instance (G, c_E, c_V, S, T) and a label system $\mathcal{L}$.
Task: Find a cost-minimal multi-label S-T-progression with respect to $\mathcal{L}$ in G or decide that no such progression exists.

This problem can be solved in polynomial time by running Dijkstra's algorithm on a modified instance. In practice, edges and nodes that are traversed more than once often lead to problems. Therefore, in BonnRouteDetailed we define label systems that make it likely that the multi-label progression is a multi-label path. See Section 4.5.1 for details on the label systems used by BonnRouteDetailed.

Definition 4.6 (next node). *Given a routing graph G we define* $\mathrm{NextNode} : V(G) \times \mathcal{D} \times \mathbb{N}_{\geq 0} \rightarrow V(G) \cup \{\emptyset\}$ *by*

$$\mathrm{NextNode}(v, dir, d) := \underset{\substack{w \in V(G) \\ \|v-w\| \geq d \\ \mathrm{Dir}(v,w) = dir}}{\mathrm{argmin}} \|v - w\|,$$

for all $v \in V(G)$, $dir \in \mathcal{D}$, and $d \in \mathbb{N}_{\geq 0}$.

Definition 4.7 (multi label instance, multi-label graph). *Let (G, c_E, c_V, S, T) be a path search instance and let (L, ξ) be a label system. The corresponding* multi-label instance *$(G^*, c^*_E, c^*_V, S^*, T^*)$ is defined in the following way:*

- *$V(G^*) := V(G) \times L$, $S^* = S \times L$, $T^* = T \times L$*
- *$c^*_V((v, l)) := c_V(v)$ for $(v, l) \in V(G^*)$*

- *For every $(v, l_v) \in V \times L$, $dir \in \mathcal{D}$, and $(l_w, d, c_\xi) \in \xi(l_v, z(v), dir)$ with $w =$ $\text{NextNode}(v, dir, d) \neq \emptyset$ we introduce an edge $((v, l_v), (w, l_w))$ if there is a dir-path P between v and w in G. The cost of the edge is*

  $$c^*_E((v, l_v), (w, l_w)) \coloneqq c_\xi + c_E(P) + c_V(P) - c_V(v) - c_V(w).$$

  The collection of all edges of this type forms $E(G^*)$.

The graph G^* is a directed multi-graph, called multi-label graph.

Note that the multi-label graph can contain parallel edges. See Figure 4.15 for an illustration of a multi-label graph.

The Multi-Label Progression Search Problem can be solved by using Dijkstra's algorithm to search for a shortest S^*-T^*-path in the multi-label graph. The following theorem generalizes and strengthens Satz 3.11 of [Nohn, 2012] and Theorem 5.23 of [Gester, 2015] in the case that no intervals are used.

Theorem 4.8. *The* Multi-Label Progression Search Problem *can be solved in $O(|V(G)||L|(\log|V(G)| + \log|L|) + d_{max}|V(G)|\xi_{max})$ time, where*

$$\xi_{max} = \max_{z \in \mathcal{Z}} \sum_{l_v \in L} \sum_{dir \in \mathcal{D}} |\xi(l_v, z, dir)|$$

is the maximum number of transitions on a layer and d_{max} is the number of edges in the longest straight path that needs to be considered in the construction of the multi-label graph.

Proof. Let (P, ϕ) be a multi-label S-T-progression, where $P = v_1, P_1, v_2, \ldots, v_k, P_k, v_{k+1}$ and for $i \in \{1, \ldots, k\}$ the path $P_i = v_i^1, \ldots, v_i^{k_i}$ is straight. For $i \in \{1, \ldots, k\}$ let $(\phi(v_{i+1}), d_i, c_\xi^i) \in \xi(\phi(v_i), z, dir)$ be a minimizer of the transition cost for the distance $\|v_{i+1} - v_i\|$. Moreover, for $i \in \{1, \ldots, k\}$ let $j_i^* \in \{1, \ldots, k_i\}$ be the indices with

$$v_i^{j_i^*} = \text{NextNode}(v_i, \text{Dir}(v_i, v_{i+1}), d_i).$$

We define the paths

$$P_i^* \coloneqq (v_i^1, \phi(v_i)), (v_i^{j_1^*}, \phi(v_{i+1})), (v_i^{j_1^*+1}, \phi(v_{i+1})), \ldots, (v_i^{k_i}, \phi(v_{i+1}))$$

and their concatenation $P^* \coloneqq P_1^*, \ldots, P_k^*$. Note that $((v_i^1, \phi(v_i)), (v_i^{j_i^*}, \phi(v_{i+1}))) \in E(G^*)$ and that its cost is $c_\xi^i + \sum_{j=2}^{i} c_V(v_i^j) + c_E(\{v_i^{j-1}, v_i^j\})$. Note that the other edges are also in G^* because of property 1 of Definition 4.3. Thus P^* is a path in G^* and its cost is $c_{\mathcal{L}}(P)$.

Conversely, let $P^* = ((v_1, l_1), \ldots, (v_k, l_k))$ be a shortest path in G^*. For $i \in \{1, \ldots, k\}$ we define P_i to be the straight v_i-v_{i+1}-path and set $P = v_1, P_1, v_2, \ldots, v_k, P_k, v_{k+1}$. Moreover, we define ϕ by $\phi(v_i) = l_i$ for $i \in \{1, \ldots, k+1\}$. Then (P, ϕ) is a multi-label progression which has the same cost as P^*.

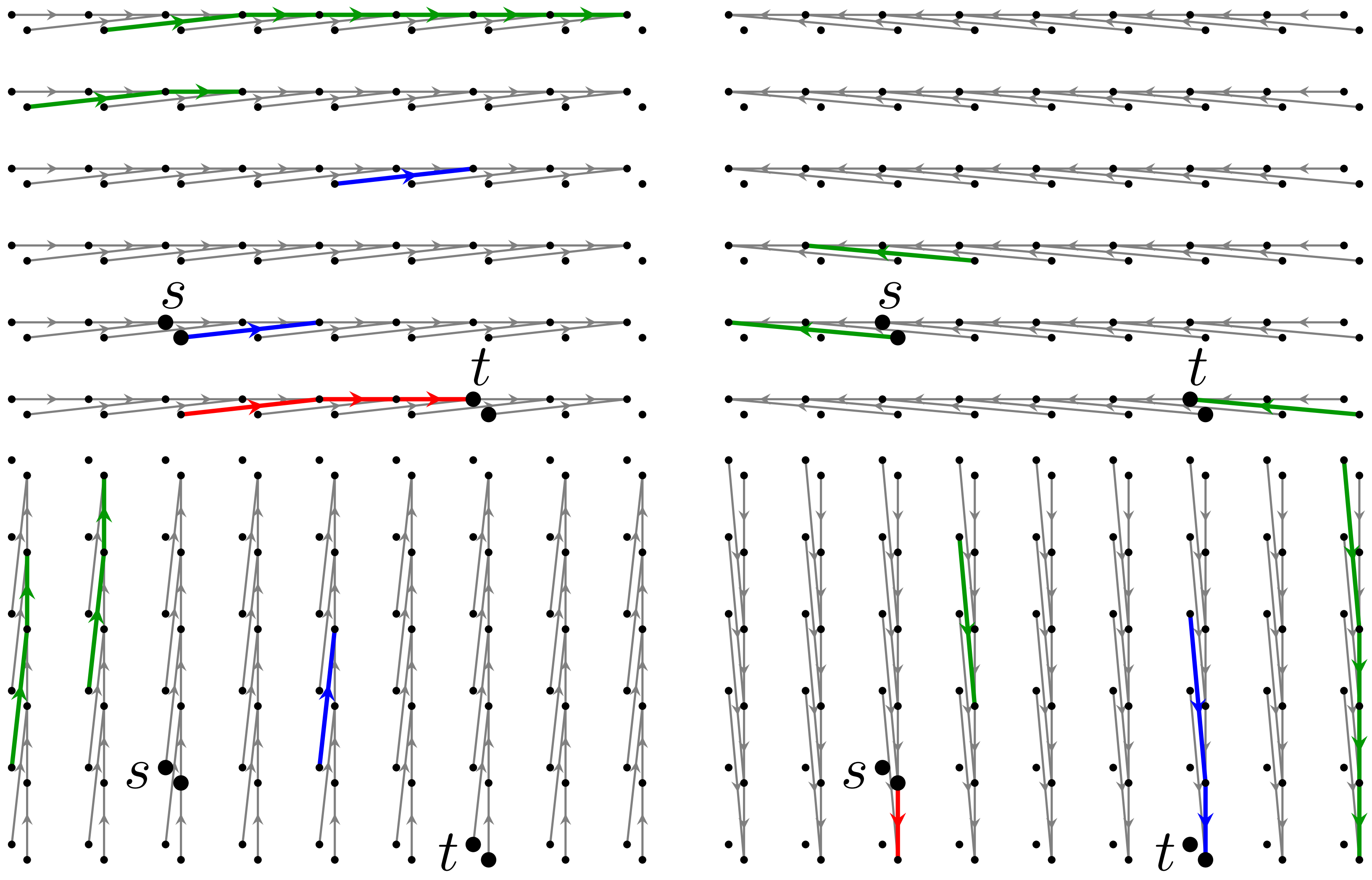


Figure 4.15: The multi-label graph when applying the label system $\mathcal{L}_{\geq 2}$ from Figure 4.12 to the graph from Figure 4.13. The edges are drawn in four graphs. The multi-label graph consists of their union. The edges corresponding to the three edge progressions from Figure 4.13 are drawn in the same colors as in that figure.

To construct G^* we traverse every layer $z \in \mathcal{Z}$, every node $(v, l_v) \in V \times L$ with v on z, every direction $dir \in \mathcal{D}$, and every transition $(l_w, d, c_\xi) \in \xi(l_v, z, dir)$. There are

$$\sum_{z \in \mathcal{Z}} n_z \sum_{l_v \in L} \sum_{dir \in \mathcal{D}} \xi(l_v, z, dir) \leq |V(G)| \xi_{max}$$

such combinations, where n_z is the number of nodes on layer z. For every such combination we check whether the straight path from v to NextNode(v, dir, d) is a path in G and add it to $E(G^*)$ if it is. Computing the candidate neighbor, checking whether there is a straight path in G, and computing the edge costs requires $O(d_{max})$ time. Thus, the runtime for the construction of G^* is $O\left(d_{max}|V(G)|\xi_{max}\right)$.

The multi-label graph G^* has $|V(G)||L|$ nodes and at most $|V(G)|\xi_{max}$ edges. Thus, computing the shortest path from S^* to T^* in G^* with Dijkstra's algorithm takes

$$O(|V(G)||L|(\log|V(G)| + \log|L|) + |V(G)|\xi_{max})$$

time, implying the claimed runtime bound. □

We note that the runtime of the multi-labeling presented in [Nohn, 2012; Ahrens et al., 2015; Gester, 2015] depends quadratically on the number of label types, instead of linearly on the number of transitions. The reason is the more efficient representation of the label transition function discussed on page 70. Another difference to the descriptions in [Nohn, 2012; Gester, 2015] is that their search labels intervals, which complicates the description.

Recall from Section 3.4 that the path search in BonnRouteDetailed works on an implicitly given routing graph. In the implementation in BonnRouteDetailed we construct edges of the multi-label graph as needed, but do not store it explicitly.

We note that property 1 of Definition 4.3 is crucial. Without it, subdividing edges in the routing graph could make paths in the multi-label graph illegal and even disconnect source and target.

4.5.1 Label Systems used by BonnRouteDetailed

BonnRouteDetailed uses hundreds of different label systems with up to 32 label types. Each is constructed out of a *base label system* and a set of *label system extensions* that specify how the base label system is modified. We note that the earlier version presented in [Nohn, 2012; Ahrens et al., 2015; Gester, 2015] uses eight different label systems and that their label systems are much simpler and less efficient at avoiding design rule violations. One reason why we can use more complex label systems is that the runtime of our version of multi-labeling scales better than the one presented in [Nohn, 2012; Ahrens et al., 2015; Gester, 2015], as discussed in the previous section.

The trivial base label system StandardRouting and the label system extension Colored-Routing presented here were previously presented in these works. All other base label systems and label system extensions are new, except for the SourceRestriction and TargetRestriction extensions, which have been completely revised.

We now present the three base label systems and the four label system extensions used by BonnRouteDetailed.

StandardRouting Base Label System

StandardRouting is the trivial label system whose multi-label graph is the original routing graph. It is illustrated in Figure 4.16.

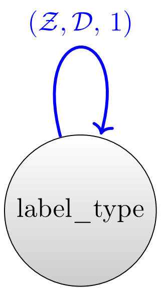


Figure 4.16: Transition graph of the StandardRouting label system.

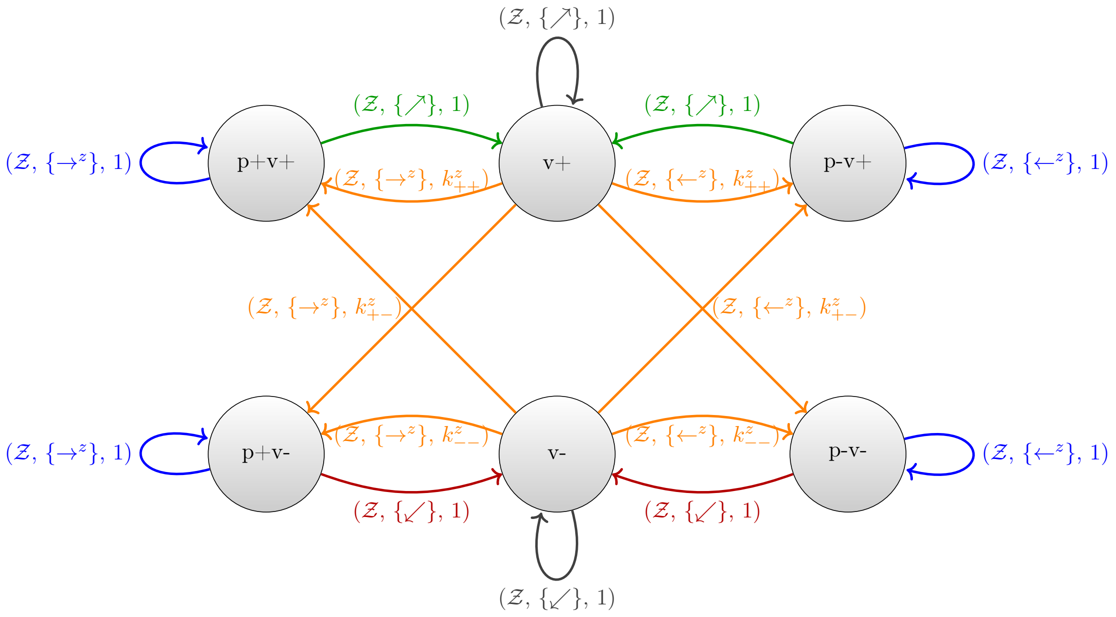
AvoidSpecificMistakes and AvoidSpecificMistakes+ Base Label Systems

The AvoidSpecificMistakes label system is designed to forbid only very few non-legal paths, while ensuring that some design rules are satisfied. Figure 4.17 illustrates the AvoidSpecificMistakes label system if jogs are forbidden. We omit a description in case that jogs are allowed for the sake of brevity.

For each subpath e_v^1, P, e_v^2 consisting of two vias e_v^1 and e_v^2 and a straight path P running in preferred dimension it imposes a minimum length constraint on P. The minimum length depends on the layer and the other layers of both vias. In practice it can also depend on the direction of P. In the AvoidAllMistakes label system of [Nohn, 2012; Ahrens et al., 2015; Gester, 2015] the minimum length depends on the layer only. The increased flexibility in AvoidSpecificMistakes allows us to create less restrictive label systems that can avoid the same same-net errors.

Note that the label system is designed to avoid cycles with two edges. Indeed, after going in a direction we always reach a label type that does not admit a transition in the opposite direction. This property is key to reduce the number of cycles in practice.

AvoidSpecificMistakes offers parameters that can be used to make it more restrictive, thereby invalidating paths with same-net errors but also some legal paths. These restrictions can be applied in a very fine-grained manner by layer and by individual error types. We summarize all restrictive versions of AvoidSpecificMistakes under the name AvoidSpecificMistakes+. In AvoidSpecificMistakes+ the required minimum distance may



p+v+
v+
p-v+
p+v-
v-
p-v-
$(\mathcal{Z}, \{\nearrow\}, 1)$
$(\mathcal{Z}, \{\rightarrow^z\}, 1)$
$(\mathcal{Z}, \{\leftarrow^z\}, 1)$
$(\mathcal{Z}, \{\rightarrow^z\}, k^z_{++})$
$(\mathcal{Z}, \{\leftarrow^z\}, k^z_{++})$
$(\mathcal{Z}, \{\rightarrow^z\}, k^z_{+-})$
$(\mathcal{Z}, \{\leftarrow^z\}, k^z_{+-})$
$(\mathcal{Z}, \{\rightarrow^z\}, k^z_{--})$
$(\mathcal{Z}, \{\leftarrow^z\}, k^z_{--})$
$(\mathcal{Z}, \{\swarrow\}, 1)$

Figure 4.17: Transition graph of the AvoidSpecificMistakes base label system if jogs are forbidden on all wiring layers. Here we use superscript z to indicate that the values depend on the layer. We use $\rightarrow^z$ and $\leftarrow^z$ to refer to the positive and negative preferred dimension on layer z. Note that the required minimum distances $k^z_{--}, k^z_{+-}, k^z_{++} > 0$ are layer-dependent. The loops at $v-$ and $v+$ may be missing for some layers if stacked vias are forbidden.

be larger on some layers or transitions may be missing or more expensive, depending on the specific restrictions.

ColoredRouting Extension

Pitch splitting is a technique for increasing feature density that is used to manufacture some wiring layers. Metal on these layers is manufactured by $k \in \mathbb{N}$ exposures and etchings. We assign a color $c \in \{1, \ldots, k\}$ to each shape indicating the step in which it is manufactured. Typically, shapes that touch or intersect need the same color. Distance requirements between shapes of the same color are larger than distance requirements for shapes of different colors.

In BonnRouteDetailed we assign a color to each track and usually color sticks by the color of their track. However, in some cases we need to deviate from the track coloring, for example to access pins that are not colored according to the track coloring. Then we use the ColoredRouting extension. Given a label system (L, ξ) and the number of colors $k_1, \ldots, k_{|\mathcal{Z}|-1} \in \mathbb{N}$ on each wiring layer, the ColoredRouting extension yields a modified label system $(L \times \{1, \ldots, \max_{z \in \mathcal{Z}} k_z\}, \xi')$ with the transition function:

$$\xi'((l_v,c),z,dir) = \begin{cases} \{((l_w,c),d,c_\xi) | (l_w,d,c_\xi) \in \xi(l_v,z,dir)\} & dir \notin \{\swarrow,\nearrow\}, c \le k_z \\ \{((l_w,c'),d,c_\xi) | (l_w,d,c_\xi) \in \xi(l_v,z,dir), c' \in \{1,\dots,k_{z_{dir}}\}\} & dir \in \{\swarrow,\nearrow\}, c \le k_z \end{cases}$$

for all $l_v, \in L$, $z \in \mathcal{Z}$, $c \in \{1,\dots,k_z\}$, and $dir \in \mathcal{D}$ and where

$$z_{dir} := \begin{cases} z-1 & dir = \swarrow \\ z+1 & dir = \nearrow \\ z & \text{otherwise} \end{cases}$$

is the layer reached when going from z in direction dir. See Figure 4.18 for an illustration.

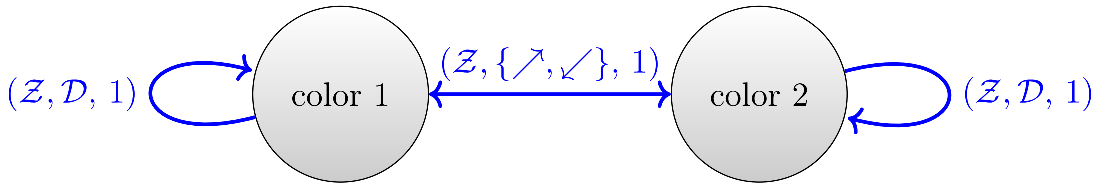

$(\mathcal{Z}, \mathcal{D}, 1)$
color 1
$(\mathcal{Z}, \{\nearrow, \swarrow\}, 1)$
color 2
$(\mathcal{Z}, \mathcal{D}, 1)$

Figure 4.18: Transition graph of the label system resulting from applying the ColoredRouting extension to the StandardRouting base label system if all wiring layers have two colors.

With ColoredRouting the checking data of a node depends on the label type, i.e. a node can be usable with one color but not usable with another. In BonnRouteDetailed we penalize labels that would lead to wiring that deviates from the track coloring with additional label-type- and location-dependent node costs. This helps to improve packing.

The ColoredRouting label system was also proposed in [Nohn, 2012; Ahrens et al., 2015; Gester, 2015].

SourceRestriction and TargetRestriction Extension

In the implementation in BonnRouteDetailed label systems can specify a source transition function ξ_S that is used at source nodes instead of ξ. Moreover, each transition in ξ and ξ_S has two additional flags: valid_to_reach_target and valid_to_reach_non_target, indicating whether the transition can be used to go to a target or non-target node. Furthermore, the label types that may be used may depend on the source $s \in S$ or the target $t \in T$. These mechanisms allow us to impose additional constraints on how source and target nodes can be accessed. In the base label systems we have $\xi_S = \xi$ and both flags are set to *true* for each transition.

SourceRestriction modifies ξ_S to avoid same-net errors near the source. To the same end, TargetRestriction may set the valid_to_reach_target flag of existing transitions to *false* and introduce additional transitions with valid_to_reach_non_target set to *false* and valid_to_reach_target set to *true* to avoid same-net errors near the target. Moreover, the SourceRestriction or TargetRestriction may restrict which label system may be used at which source or target. We note that BonnRouteDetailed additionally uses protections that will be introduced in Section 4.6 to avoid violations near source and target.

LayerFuse Extension

Each net N has a minimum assigned layer $z_a \in \mathcal{Z}$ and most of its wiring should be on z_a or above. Let T be a Steiner tree connecting N. Then each source-sink-path P of T should have the following property:

- Let P_a be the forest resulting from P by deleting all edges that do not intersect $\mathbb{R} \times \mathbb{R} \times [z_a, |\mathcal{Z}| - 1]$. Then P_a is a path or empty.

We call each source-sink-path that does not have this property a *layer fuse*. BonnRoute-Detailed avoids layer fuses, since they often lead to bad timing behavior. For an illustration see Figure 4.19.

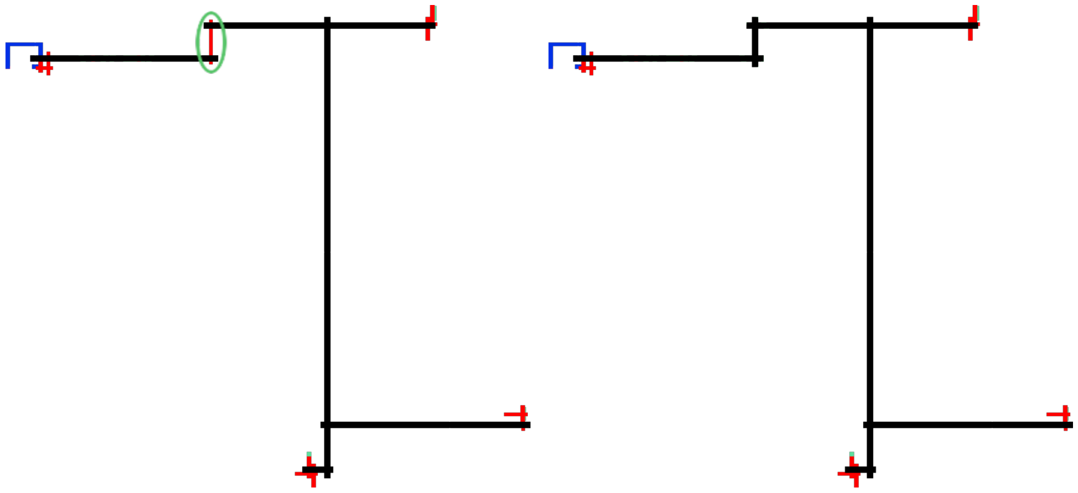


Figure 4.19: Two wirings of the same net, consisting of a source pin in the upper left and three sink pins. Shapes on the assigned layers are black and parts below the assigned layers drawn in red. On the left, each source-sink-path is a layer fuse, caused by the highlighted shape. On the right, this shape and the connecting vias are moved to an assigned layer and there is no layer fuse.

BonnRouteDetailed uses three mechanisms to avoid layer fuses. First, for path searches whose path should be part of the trunk we restrict the set of source locations in the path search to nodes on or above the assigned layers. This was implemented by Niko Klewinghaus. Second, the edge costs on layers below z_a are increased. Third, we use the LayerFuse extension. Given a label system (L, ξ) and a minimum assigned layer $z_a \in \mathcal{Z}$ it defines a modified label system $(L \times \{A, \overline{A}\}, \xi')$ with the following transition function:

$$\xi'((l_v, A), z, dir) = \begin{cases} \{((l_w, A), d, c_\xi) | (l_w, d, c_\xi) \in \xi(l_v, z, dir)\} & z \neq z_a \text{ or } dir \neq \swarrow \\ \{((l_w, \overline{A}), d, c_\xi) | (l_w, d, c_\xi) \in \xi(l_v, z, dir)\} & \text{otherwise} \end{cases}$$

$$\xi'((l_v, \overline{A}), z, dir) = \begin{cases} \emptyset & z \geq z_a \text{ or } z_{dir} \geq z_a \\ \{((l_w, \overline{A}), d, c_\xi) | (l_w, d, c_\xi) \in \xi(l_v, z, dir)\} & \text{otherwise} \end{cases}$$

for all $l_v, \in L$, $z \in \mathcal{Z}$, and $dir \in \mathcal{D}$ and where z_{dir} is the layer reached when going from z in direction dir (as defined on page 79). See Figure 4.20 for an illustration.

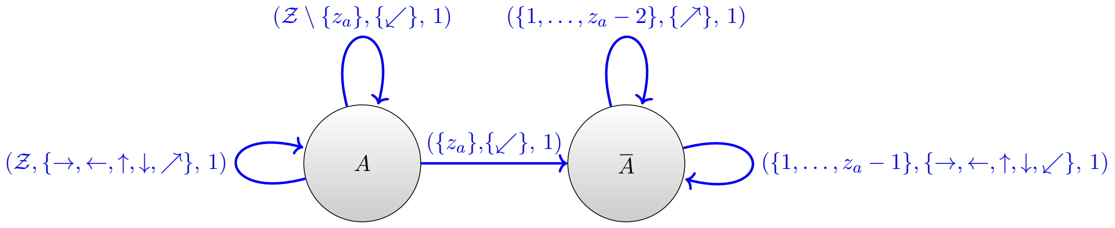

$$\begin{tikzcd}[column sep=huge] A \arrow[loop above, "(\mathcal{Z} \setminus \{z_a\}, \{\swarrow\}, 1)"] \arrow[loop left, "(\mathcal{Z}, \{\rightarrow, \leftarrow, \uparrow, \downarrow, \nearrow\}, 1)"] \arrow[r, "(\{z_a\}, \{\swarrow\}, 1)"] & \overline{A} \arrow[loop above, "(\{1, \ldots, z_a - 2\}, \{\nearrow\}, 1)"] \arrow[loop right, "(\{1, \ldots, z_a - 1\}, \{\rightarrow, \leftarrow, \uparrow, \downarrow, \swarrow\}, 1)"] \end{tikzcd}$$

Figure 4.20: Transition graph of the label system resulting from applying the LayerFuse extension to the StandardRouting base label system.

Label types in $L \times \{\overline{A}\}$ indicate that we were already on an assigned layer $z \geq z_a$ and are now below z_a. Thus, going up to an assigned layer is forbidden. Conversely, label types in $L \times \{A\}$ mean that the path is either on an assigned layer $z \geq z_a$ or that it is on a layer $z < z_a$ and can still go up to assigned layers. Going from layer z_a to $z_a - 1$ requires switching to a label type in $L \times \{\overline{A}\}$. Using the LayerFuse extension can avoid all layer fuses if the usable label types are initialized appropriately for each source. Mechanisms 1 and 2 improve the quality of the wiring, e.g. by reducing the amount of parallel wiring.

4.5.2 Selecting a Label System

Using the most restrictive base label system combined with all label system extensions in every path search would lead to high runtime, because of the high complexity of the resulting label system. Moreover, this might lead to unnecessary detours, since some of the restrictions can forbid legal paths. Therefore, the same-net rule aware path search starts with the StandardRouting label system and determines additional label systems that should be used with Algorithm 8, if necessary.

The algorithm is very straightforward. It gets a set of error-annotations that were computed by the same-net checking. First, in line 1-4, it computes the least restrictive base label system that can resolve as many errors in E as possible. This is StandardRouting if there are no violations that can be resolved by AvoidSpecificMistakes+, AvoidSpecificMistakes if there are violations that can be resolved by AvoidSpecificMistakes but no violations that require AvoidSpecificMistakes+, and a version of AvoidSpecificMistakes+ otherwise. The result is $\mathcal{L}$. Then it considers the four label system extensions: ColoredRouting, LayerFuse,

Algorithm 8 GetLeastRestrictiveFixingLabelSystem(E)

Input: A set of error-annotations E.
Output: A label system.

```
1: $\mathcal{L} \leftarrow$ StandardRouting
2: if there are errors in $E$ that can be avoided by AvoidSpecificMistakes+ then
3:     $\mathcal{L} \leftarrow$ AvoidSpecificMistakes
4:     apply restrictions to $\mathcal{L}$ to avoid errors in $E$ that can be resolved by AvoidSpecific-
       Mistakes+ but not by AvoidSpecificMistakes
5: if $E$ contains coloring errors then
6:     apply the ColoredRouting extension to $\mathcal{L}$
7: if $E$ contains layer fuses then
8:     apply the LayerFuse extension to $\mathcal{L}$
9: if $E$ contains errors near the source then
10:    apply the SourceRestriction extension to $\mathcal{L}$
11: if $E$ contains errors near the target then
12:    apply the TargetRestriction extension to $\mathcal{L}$
13: return $\mathcal{L}$
```

SourceRestriction, and TargetRestriction. If there is a violation that may be resolved by the extension, i.e. a coloring problem, a layer fuse, an error near the source, or an error near the target, the corresponding extension is applied to $\mathcal{L}$. None or all of the extensions may be applied. Finally, the label system $\mathcal{L}$ is returned.

4.6 Protections

Many same-net errors occur near the ends of paths and BonnRouteDetailed uses *protections* to avoid them. Each protection is a pair (r, m), consisting of a rectangle on a wiring layer and a wire model or a cuboid on a via layer and a via model. Every edge that intersects r becomes illegal for wire or via model m. For each path search instance, BonnRouteDetailed computes protections avoiding different kinds of same-net errors near the source and target.

For example, a bump-out rule could specify: If an edge between two convex vertices of a rectangular metal polygon has length less than α then at least one of its adjacent edges must have length at least β. Figure 4.21 shows a protection computed to avoid bump-out violations.

We note that BonnRouteDetailed has an additional mechanism for avoiding violations near the ends of paths. If the source or target is a pin that cannot be accessed legally using the routing graph, we compute a set of *access paths* that connect to the pin without causing same-net errors. The endpoints of these paths are used as source and target locations in the path search. After the search the path in the routing graph is combined with the

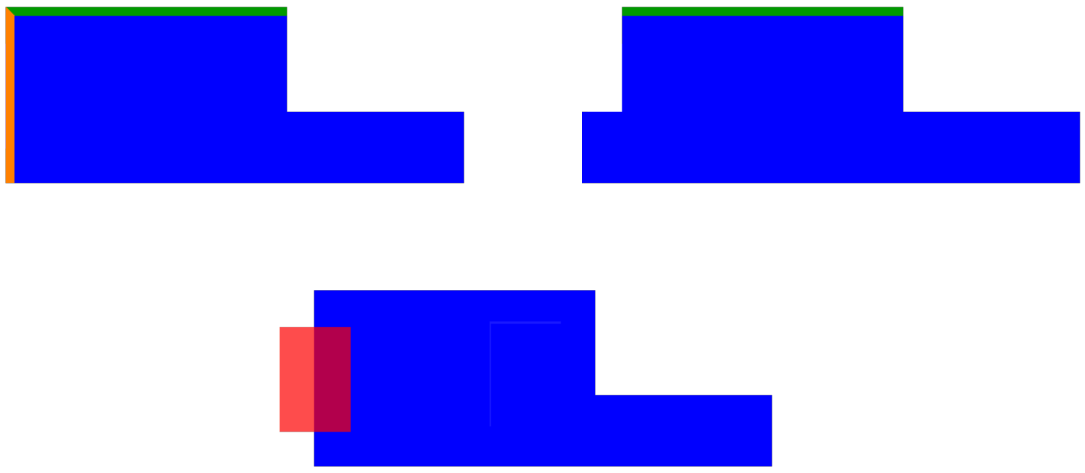


Figure 4.21: The length of the green edge is less than α and the length of the orange edge is β. There is no violation in the top-left, but in the top-right the additional metal leads to a violation. The bottom shows a protection to avoid bump-out violations for a specific wire model when accessing the metal component from the left.

access paths. This mechanism works well for resolving same-net errors but it has one weakness: Both the access paths and the path in the routing graph may respect the design rules, but there may be violations after they are combined, especially if combining the paths leads to cycles. For this reason and for the sake of simplicity we plan to get rid of access paths in BonnRouteDetailed and modify the routing graph locally to allow legal pin access by subdividing edges or adding additional grid lines. Note that this is supported by our path search (see Section 3.4). Then the protections and the SourceRestriction and TargetRestriction extensions can be used to access the pins legally.

4.7 Same-Net Rule Aware Path Search in BonnRouteDetailed

This section presents the same-net rule aware path search framework used in BonnRoute-Detailed. It uses the components described in Section 4.3, Section 4.4, Section 4.5, and Section 4.6 and is sketched in Algorithm 9. It is a modified and extended version of the DRC-aware path search framework presented in [Ahrens et al., 2015] (see Figure 13 on page 11) and [Gester, 2015] (see Algorithm 7 on page 91).

At the beginning, the algorithm computes protections, thereby removing edges near source and target from G. Then it sets the label system $\mathcal{L}$ to be used in the first path search to StandardRouting. E_{all} is the set of error-annotations of all paths computed by the algorithm. Initially, it is set to $\emptyset$. To indicate that no path was found so far, P_{best} and E_{best} are initialized with $\emptyset$. Then, the algorithm initializes some data D that is common for all path searches. This includes the grid region partition and the grid region data structure from Lemma 3.25 on page 46.

Algorithm 9 Simplified Same-Net Rule Aware Path Search

Input: A path search instance (G, c_E, c_V, S, T).
Output: An S-T-path or the information that no such path exists.

```
1: compute protections    ▷ the protections may remove edges from $G$
2: $E_{all} \leftarrow \emptyset$, $E_{best} \leftarrow \emptyset$, $P_{best} \leftarrow \emptyset$
3: $\mathcal{L} \leftarrow StandardRouting$
4: initialize data structure $D$ for path search    ▷ e.g. grid region data structure
5: while ($true$) do
6:     prepare $D$ for $\mathcal{L}$    ▷ delete labels, keep checking data if it can be reused
7:     $P \leftarrow$ shortest edge progression in the multi-label instance for $\mathcal{L}$ and $(G, c_E, c_V, S, T)$
8:     if $P = \emptyset$ then
9:         break
10:    remove cycles and loops from $P$
11:    $P \leftarrow$ PostProcessPath($P$)
12:    $E \leftarrow$ ComputeErrorAnnotations($P$)
13:    $E_{all} \leftarrow E_{all} \cup E$
14:    if $P_{best} = \emptyset$ or WeightedErrorSum($E$) $<$ WeightedErrorSum($E_{best}$) then
15:        $P_{best} \leftarrow P$
16:        $E_{best} \leftarrow E$
17:    $\mathcal{L}_{next} \leftarrow$ GetLeastRestrictiveFixingLabelSystem($E_{all}$)
18:    if $\mathcal{L}_{next} = \mathcal{L}$ then
19:        break
20:    $\mathcal{L} \leftarrow \mathcal{L}_{next}$
21: if $P_{best} = \emptyset$ then
22:    return No $S$-$T$-path exists.
23: return $P$
```

Each iteration of the main loop performs a multi-label search for a different label system $\mathcal{L}$. If there is no valid progression we break out of the while-loop and return the best path found so far. Otherwise, cycles are removed from the path P and it is post-processed. Moreover, we use the same-net checking to compute the error-annotations E of P. If P is the best path found so far we update P_{best} and E_{best} accordingly. We then compute the label system $\mathcal{L}_{next}$ to be used in the next iteration of the main loop using GetLeastRestrictiveFixingLabelSystem(E_{all}), introduced in Section 4.5.2. The function returns the least-restrictive available label system that can fix all errors in E_{all} (that the label systems of BonnRouteDetailed can resolve). If the returned label system $\mathcal{L}_{next}$ is identical to $\mathcal{L}$, we break out of the while-loop and return the best path found so far. Otherwise, we set $\mathcal{L}$ to $\mathcal{L}_{next}$ and continue with the next iteration of the main loop.

Line 4 and line 6 are important optimizations. Instead of recomputing data such as the grid region data structure, the future costs, the checking data, etc. from scratch for every search we reuse as much data as possible. The grid region data structure can always be reused, just like the future cost of nodes for which it was initialized. If the previous path search did not use the ColoredRouting and the next path search uses it, the checking data cannot be reused, because using the ColoredRouting extension requires color-dependent checking. In all other cases the checking data can be reused. We note that we delete all labels computed by the previous search and reinitialize the source labels in every iteration of the main loop.

Section 6.3 provides experimental results on the same-net rule aware path search and its components for avoiding violations: the post-processing, the multi-labeling, and the protections.

Chapter 5

Steiner Tree Search

This chapter considers the problem of computing optimal Steiner trees respecting restrictions on the topology and on the location of Steiner points derived from the global routing. First, in Section 5.1 we give an overview of known results on the STEINER TREE PROBLEM. Then, in Section 5.2 we introduce the RESTRICTED STEINER TREE PROBLEM and present the Restricted Dijkstra-Steiner algorithm, which generalizes the Dijkstra-Steiner algorithm [Hougardy et al., 2017]. Finally, in Section 5.3 we show how this algorithm can be used to compute optimal Steiner trees that are similar to the global routing in terms of topology and approximate location of Steiner points. This allows us to find shorter and more efficient detailed routings and to recover electrical and timing properties optimized in global routing. We show that in our application the algorithm achieves near-linear runtime under mild assumptions. This is possible because the restrictions actually make the problem easier. The Restricted Dijkstra-Steiner algorithm was developed and implemented in joint work with Dorothee Henke and Stefan Rabenstein.

5.1 The Steiner Tree Problem

In the STEINER TREE PROBLEM we are given a graph G, a set of terminals T, and an edge cost function $c_E : E(G) \to \mathbb{R}_{\geq 0}$. The task is to compute a Steiner tree S in G such that $T \subseteq V(S)$ and $c_E(E(S))$ is minimum. The problem is NP-hard, even for $c_E \equiv 1$ [Karp, 1972].

It is NP-hard to approximate the problem within a factor of $\frac{96}{95}$ [Chlebikova and Chlebík, 2008]. It remains NP-hard even in two-dimensional grid graphs [Garey and Johnson, 1977]. The best known approximation algorithm is LP-based and uses iterative randomized rounding to achieve an approximation ratio of 1.39 [Byrka et al., 2013].

An exact algorithm with runtime $O(3^k n + 2^k n^2 + n(\log n + m))$ was proposed in [Dreyfus and Wagner, 1971], where k is the number of terminals, n is the number of nodes and m the number of edges. [Erickson et al., 1987] showed that this algorithm can be implemented to achieve runtime $O(3^k n + 2^k(n \log n + m))$. The resulting algorithm is called Dreyfus-

Wagner algorithm. While faster algorithms exist for large values of k (see [Vygen, 2011]) this is the fastest known exact algorithm for the STEINER TREE PROBLEM for small values of k. The Dijkstra-Steiner algorithm [Hougardy et al., 2017] is a goal-oriented version of the Dreyfus-Wagner algorithm that achieves the same worse-case runtime but can perform significantly better in practice.

5.2 The Restricted Dijkstra-Steiner Algorithm

This section introduces the RESTRICTED STEINER TREE PROBLEM and presents the Restricted Dijkstra-Steiner algorithm. The Restricted Dijkstra-Steiner algorithm was developed and implemented in joint work with Dorothee Henke and Stefan Rabenstein. The algorithm is also presented in [Rabenstein, 2019]. Our description of the algorithm, some definitions, Lemma 5.5 and its proof, and Theorem 5.6 appear in [Rabenstein, 2019] in a similar form. The other results are new.

The Dijkstra-Steiner algorithm [Hougardy et al., 2017] uses a dynamic programming approach similar to the one in Dijkstra's algorithm. It selects a root terminal $t_0 \in T$ and labels from the terminals in $T \setminus \{t_0\}$ towards the root. The algorithm labels elements of $V(G) \times 2^{T\setminus\{t_0\}}$, and a label of $(v, I) \in V(G) \times 2^{T\setminus\{t_0\}}$ corresponds to a Steiner tree connecting $I \cup \{v\}$ with a certain cost. We consider a restricted version of the STEINER TREE PROBLEM and the restrictions we impose correspond to allowing only a subset of elements of $V(G) \times 2^{T\setminus\{t_0\}}$ to be labeled.

Definition 5.1 (set of allowed labels, allowed label)**.** *Let G be a graph, let $T \subseteq V(G)$ be a set of terminals, and let $t_0 \in T$ be a root terminal. A set $\mathcal{A} \subseteq V(G) \times 2^{T\setminus\{t_0\}}$ with $(t_0, T \setminus \{t_0\}) \in \mathcal{A}$ and $(t, \{t\}) \in \mathcal{A}$ for $t \in T \setminus \{t_0\}$ is called a* set of allowed labels *for G, T, and t_0. An element of $\mathcal{A}$ is called an* allowed label*.*

Each allowed label (v, I) corresponds to a subproblem. We now define (v, I)-trees that are possible solutions to the subproblem, their cost, and the cost $\mathrm{smt}(v, I)$ of an optimal solution.

Definition 5.2 ((v, I)-tree, $\mathrm{smt}(v, I)$)**.** *Let G be a graph, let $T \subseteq V(G)$ be a set of terminals, let $t_0 \in T$ be a root terminal, and let $\mathcal{A} \subseteq V(G) \times 2^{T\setminus\{t_0\}}$ be a set of allowed labels. An arborescence S with $V(S) \subseteq \mathcal{A}$ is called a* (v, I)-tree *for G, T, t_0, and $\mathcal{A}$ if (v, I) is the root of S and for every $(w, J) \in V(S)$ one of the following holds:*

1. *$\Gamma^+_S(w, J) = \emptyset$, $w \in T \setminus \{t_0\}$, and $J = \{w\}$*
2. *$\Gamma^+_S(w, J) = \{(w', J)\}$ where $\{w, w'\} \in E(G)$*
3. *$\Gamma^+_S(w, J) = \{(w, J_1), (w, J_2)\}$ where $J_1 \cup J_2 = J$ and $J_1 \cap J_2 = \emptyset$*

Given an edge cost function $c_E : E(G) \to \mathbb{R}_{\geq 0}$ the cost of S is defined by

$$c_E(S) := \sum_{((w,J),(w',J)) \in E(S)} c_E(\{w, w'\}).$$

We denote the minimum cost of a (v, I)-tree by $\operatorname{smt}(v, I)$. *If there is no (v, I)-tree we set* $\operatorname{smt}(v, I)$ *to ∞.*

Note that every (v, I)-tree contains a leaf $(t, \{t\})$ for every $t \in I$.

We can now define the Restricted Steiner Tree Problem.

Restricted Steiner Tree Problem

Input: A graph $G = (V, E)$, edge costs $c_E : E \to \mathbb{R}_{\geq 0}$, a set of terminals $T \subseteq V$, a root terminal $t_0 \in T$, and a set of allowed labels $\mathcal{A} \subseteq V \times 2^{T \setminus \{t_0\}}$.

Task: Find a $(t_0, T \setminus \{t_0\})$-tree with minimum cost or decide that no $(t_0, T \setminus \{t_0\})$-tree exists.

For $\mathcal{A} = V \times 2^{T \setminus \{t_0\}}$ the problem is equivalent to the Steiner Tree Problem. Hence it is NP-hard. See Figure 5.1 for an example with $\mathcal{A} \neq V \times 2^{T \setminus \{t_0\}}$.

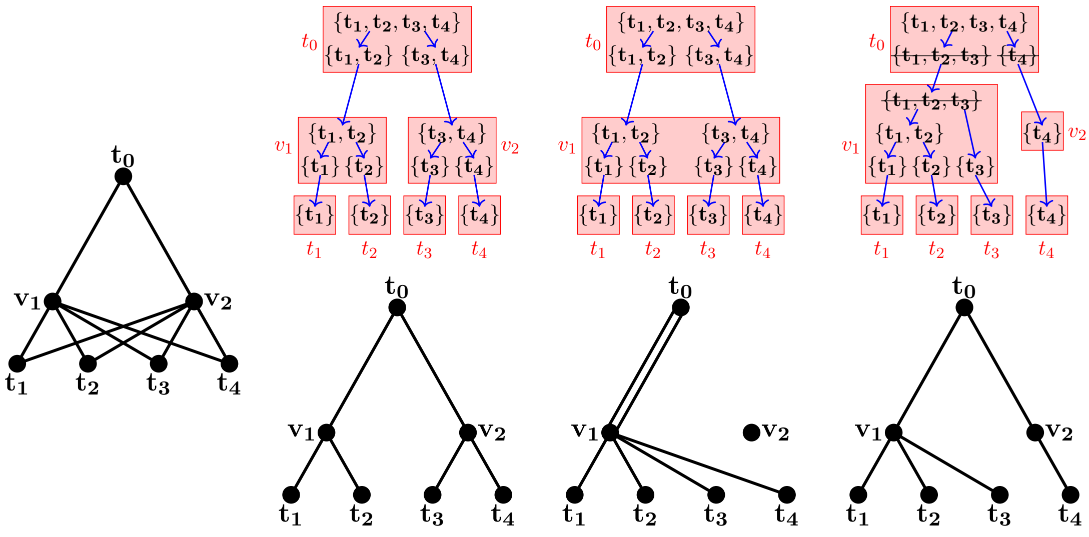


Figure 5.1: The left shows an instance with terminals $\{t_0, t_1, t_2, t_3, t_4\}$, root terminal t_0, the graph as drawn, and $\mathcal{A} = \{(t_i, \{t_i\}) | i \in \{1, 2, 3, 4\}\} \cup \{(v_i, I) | i \in \{1, 2\}, I \subseteq T \setminus \{t_0\}, |I| \in \{1, 2\}\} \cup \{(t_0, I) | I \subseteq T \setminus \{t_0\}, |I| \in \{2, 4\}\}$. The middle shows two $(t_0, \{t_1, t_2, t_3, t_4\})$-trees. A terminal set I in a box labeled w refers to the allowed label (w, I). Up to renaming elements of $\{v_1, v_2\}$ and $\{t_1, t_2, t_3, t_4\}$ these are the only solutions. The arborescence on the right is not a solution because it uses $(v_1, \{t_1, t_2, t_3\}), (t_0, \{t_1, t_2, t_3\}), (t_0, \{t_4\}) \notin \mathcal{A}$ which are not allowed labels and are struck through.

For $\mathcal{A} \neq V \times 2^{T\setminus\{t_0\}}$ a solution S can use an edge multiple times with different terminal sets, i.e. there can be an edge $\{v, w\} \in E(G)$ and terminal sets $I, J \subseteq T \setminus \{t_0\}$ with $I \neq J$ such that $((v, I), (w, I)), ((v, J), (w, J)) \in E(S)$. Longer cycles in $\{\{v, w\} | \exists I : ((v, I), (w, I)) \in E(S)\}$ are also possible.

In the problem statement there is a special root terminal that the set of allowed labels depends on. We now show that in a sense the choice of root terminal does not matter.

Lemma 5.3. *Let $(G, c_E, T, t_0, \mathcal{A})$ be an instance of the* Restricted Steiner Tree Problem. *Then for every $t'_0 \in T \setminus \{t_0\}$ there is $\mathcal{A}'$ with $|\mathcal{A}'| = |\mathcal{A}|$ such that any $(t_0, T \setminus \{t_0\})$-tree S with respect to $\mathcal{A}$ can be converted into a $(t'_0, T \setminus \{t'_0\})$-tree S' with respect to $\mathcal{A}'$ and vice versa. Moreover, the trees have the same number of edges and the same cost and the conversion takes $O(|E(S)|)$ time.*

Proof. Let $t'_0 \in T \setminus \{t_0\}$. We define a bijection $F : 2^{T\setminus\{t_0\}} \to 2^{T\setminus\{t'_0\}}$ by

$$F(I) = \begin{cases} I & t'_0 \notin I \\ T \setminus I & t'_0 \in I \end{cases}$$

and set $\mathcal{A}' = \{(v, F(I)) | (v, I) \in \mathcal{A}\}$. Let S be a $(t_0, T \setminus \{t_0\})$-tree with respect to $\mathcal{A}$.

We now describe how to construct a $(t'_0, T \setminus \{t'_0\})$-tree S' with respect to $\mathcal{A}'$ with the desired properties. We set $V(S') = \{(v, F(I)) | (v, I) \in V(S)\} \subseteq \mathcal{A}'$ and traverse S starting at $(t'_0, \{t'_0\})$. We will add outgoing edges (if any) of a node $(v, F(I))$ only in the iteration in which (v, I) is considered.

We maintain a set of nodes $N \subseteq V(S)$ that still need to be traversed and initialize it with $(t'_0, \{t'_0\})$. In each iteration we select an element (v, I) from N and remove it from N. When that is no longer possible because N is empty we are done. We distinguish three cases:

Case 1: (v, I) has a neighbor (w, I) that was not added to N yet.
We add an edge $((v, F(I)), (w, F(I)))$ to $E(S')$ and add (w, I) to N. Hence property 2 of Definition 5.2 is satisfied.

Case 2: (v, I) has a neighbor (v, J) that was not added to N yet.
Because of property 3 of Definition 5.2 either (v, I) or (v, J) has another neighbor (v, J') such that $K_1 \cup K_2 = K_3$ and $K_1 \cap K_2 = \emptyset$ for some choice of $K_1, K_2, K_3 \in \{I, J, J'\}$. We add edges $((v, F(I)), (v, F(J)))$ and $((v, F(I)), (v, F(J')))$ to $E(S')$ and add (v, J) and (v, J') to N. See Figure 5.2 for an illustration. Observe that we have $F(I) = F(J) \cup F(J')$ and $F(J) \cap F(J') = \emptyset$ which implies that property 3 of Definition 5.2 is satisfied.

Case 3: (v, I) has no neighbor in S that was not added to N yet.
We do not add any edges. Because of the way we traverse S we have $(v, I) = (t_0, T \setminus \{t_0\})$ or $(v, I) = (t, \{t\})$ for some $t \in T \setminus \{t_0, t'_0\}$. Hence, $(v, F(I)) = (t, \{t\})$ for some $t \in T \setminus \{t'_0\}$ and property 1 of Definition 5.2 is satisfied.

Note that S' is connected and rooted at $(t'_0, F(\{t'_0\})) = (t'_0, T \setminus \{t'_0\})$ and hence a $(t'_0, T \setminus \{t'_0\})$-tree with respect to $\mathcal{A}'$. Clearly $|E(S)| = |E(S')|$, $c_E(S) = c_E(S')$, and the construction can be done in $O(|E(S)|)$ time. $\square$

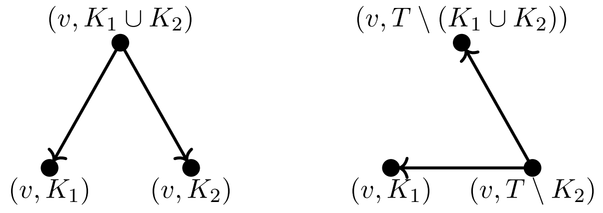


Figure 5.2: If the configuration on the left is in S we get the same configuration in S' if $t'_0 \notin K_1 \cup K_2$. Note that in this case the first drawn node that is traversed is $(v, K_1 \cup K_2)$. If $t'_0 \in K_2$ we get the configuration on the right in S'. Note that in this case the first drawn node that is traversed is (v, K_2).

Before presenting the Restricted Dijkstra-Steiner algorithm, we introduce future cost functions, which will make the search goal-oriented, as in the Dijkstra-Steiner algorithm.

Definition 5.4 (future cost function). *Let $(G, c_E, T, t_0, \mathcal{A})$ be an instance of the* Restricted Steiner Tree Problem. *A* future cost function *is a function $f : \mathcal{A} \to \mathbb{R}_{\geq 0}$ with the following properties:*

1. *$f(t_0, T \setminus \{t_0\}) = 0$*
2. *$f(v, I) \leq f(w, I) + c_E(\{v, w\})$ for all $\{v, w\} \in E(G)$ and all $I \subseteq T \setminus \{t_0\}$ with $(v, I), (w, I) \in \mathcal{A}$*
3. *$f(v, I) \leq f(v, I \cup J) + smt(v, J)$ for all $v \in V(G)$ and $I, J \subseteq T \setminus \{t_0\}$ with $I \cap J = \emptyset$ and $(v, I), (v, J), (v, I \cup J) \in \mathcal{A}$*

Next we prove that the name *future cost* function is justified in the sense that extending a (v, I)-tree to a $(t_0, T \setminus \{t_0\})$-tree without modifying the sub-arborescence rooted at (v, I) requires cost at least $f(v, I)$. This follows immediately from the next lemma and $f(t_0, T \setminus \{t_0\}) = 0$. The lemma and its proof are similar to Lemma 20 of [Rabenstein, 2019] and its proof.

Lemma 5.5. *Let $(G, c_E, T, t_0, \mathcal{A})$ be an instance of the* Restricted Steiner Tree Problem, *let f be a future cost function, let $(v, I) \in \mathcal{A}$ be an allowed label, let S_v be a (v, I)-tree, let (w, J) be a child of (v, I) in S_v, and let S_w be the subtree rooted at (w, J). Then*

$$c_E(S_w) + f(w, J) \leq c_E(S_v) + f(v, I).$$

Proof. We may assume that the claim holds for all sub-arborescences rooted at successors of (v, I), by induction. We distinguish two cases. If (w, J) is the only child of (v, I) in S_v, then $I = J$ and property 2 of Definition 5.4 yields

$$f(w, J) \leq f(v, I) + c_E(\{v, w\}) = f(v, I) + c_E(S_v) - c_E(S_w).$$

Otherwise we have $v = w$ and (v, I) has exactly two children (w, J) and $(w, I \setminus J)$. Let S' be the sub-arborescence rooted at $(w, I \setminus J)$. Then by property 3 of Definition 5.4 we get

$$f(w, J) \leq f(v, I) + \mathrm{smt}(v, I \setminus J) \leq f(v, I) + c_E(S') = f(v, I) + c_E(S_v) - c_E(S_w). \quad \square$$

Note that $f \equiv 0$ is a future cost function. As in Dijkstra's algorithm, future costs that are as large as possible are desirable (see Lemma 3.8 on page 31). Note that future cost functions for a modified instance with $\mathcal{A}' = V(G) \times 2^{T \setminus \{t_0\}}$ are also future cost functions for the original instance. Hence we can use future cost functions for the Dijkstra-Steiner algorithm, for example half the cost of a 1-tree (v being the node connected to the tree with two edges) or half the cost of a TSP-tour. For more information we refer to Section 2 of [Hougardy et al., 2017]. See page 98 for a discussion of future cost functions that might be suited for our application.

Algorithm 10 presents the Restricted Dijkstra-Steiner algorithm. The algorithm is very similar to the Dijkstra-Steiner algorithm. It maintains a set of tentative costs l for each allowed label $(v, I) \in \mathcal{A}$, corresponding to the cost of a cheapest (v, I)-tree that was found so far. A tentative cost of ∞ means that so far no such tree was found. During the algorithm, the tentative costs decrease and a set of allowed labels R for which the tentative cost was proven to be the real cost is maintained. In each iteration of the main loop, an allowed label $(v, I) \in \mathcal{A} \setminus R$ with minimum tentative cost plus future cost is selected and added to R, i.e. its tentative cost is no longer tentative. Then the backtracking data $b(v, I)$, that is always updated along with $l(v, I)$, defines a (v, I)-tree with cost $\mathrm{smt}(v, I)$. Costs for other allowed labels can be derived from $l(v, I)$ by extending the tree by an edge or by merging it with a tree for another permanent label $(v, J) \in R$ with $I \cap J = \emptyset$. Once $(t_0, T \setminus \{t_0\})$ is selected, a minimum cost $(t_0, T \setminus \{t_0\})$-tree has been determined and is returned. If $(t_0, T \setminus \{t_0\})$ is never selected, there is no solution.

The following result was achieved in joint work with Stefan Rabenstein.

Theorem 5.6. *The Restricted Dijkstra-Steiner algorithm works correctly.*

We omit the correctness proof because it is very similar to that of the Dijkstra-Steiner algorithm. A detailed proof can be found in [Rabenstein, 2019].

Next, we analyze the runtime of the Restricted Dijkstra Steiner algorithm. The runtime depends on the size of the set of allowed edges and allowed merges that are introduced in the following definition.

Definition 5.7 (set of allowed edges, set of allowed merges)**.** *Let G be a graph, let T be a set of terminals, let $t_0 \in T$ be a root terminal, and let $\mathcal{A}$ be a set of allowed labels. The* set of allowed edges $E_{\mathcal{A}}$ *and the* set of allowed merges $M_{\mathcal{A}}$ *are defined by*

$$E_{\mathcal{A}} := \{\{(v, I), (w, I)\} | \{v, w\} \in E(G) \text{ and } (v, I), (w, I) \in \mathcal{A}\} \text{ and}$$
$$M_{\mathcal{A}} := \{\{(v, I), (v, J)\} | I \cap J = \emptyset \text{ and } (v, I), (v, J), (v, I \cup J) \in \mathcal{A}\}.$$

Algorithm 10 Restricted Dijkstra-Steiner Algorithm

Input: A graph $G = (V, E)$, edge costs $c_E : E \to \mathbb{R}_{\geq 0}$, a set of terminals $T \subseteq V$, a root terminal $t_0 \in T$, a set of allowed labels $\mathcal{A} \subseteq V \times 2^{T \setminus \{t_0\}}$, and a future cost function $f : \mathcal{A} \to \mathbb{R}_{\geq 0}$.

Output: A $(t_0, T \setminus \{t_0\})$-tree with minimum cost or the information that no such tree exists.

```
1: $\forall (v, I) \in \mathcal{A}$ set $l(v, I) \leftarrow \infty$ and $b(v, I) \leftarrow \emptyset$, $\forall t \in T \setminus \{t_0\}$ set $l(t, \{t\}) \leftarrow 0$, set $R \leftarrow \emptyset$
2: while $\exists (v, I) \in \mathcal{A} \setminus R$ s.t. $l(v, I) \neq \infty$ do
3:     $(v, I) \leftarrow \text{argmin}_{(w,J) \in \mathcal{A} \setminus R}\, l(w, J) + f(w, J)$
4:     $R \leftarrow R \cup \{(v, I)\}$
5:     if $v = t_0$ and $I = T \setminus \{t_0\}$ then
6:         return the arborescence rooted at $(t_0, T \setminus \{t_0\})$ defined by $b$
7:     for all $w \in V$ with $\{v, w\} \in E$ and $(w, I) \in \mathcal{A} \setminus R$ do
8:         if $l(w, I) > l(v, I) + c_E(\{v, w\})$ then
9:             $l(w, I) \leftarrow l(v, I) + c_E(\{v, w\})$
10:            $b(w, I) \leftarrow \{(v, I)\}$
11:    for all $(v, J) \in R$ with $I \cap J = \emptyset$ and $(v, I \cup J) \in \mathcal{A} \setminus R$ do
12:        if $l(v, I \cup J) > l(v, I) + l(v, J)$ then
13:            $l(v, I \cup J) \leftarrow l(v, I) + l(v, J)$
14:            $b(v, I \cup J) \leftarrow \{(v, I), (v, J)\}$
15: return No $(t_0, T \setminus \{t_0\})$-tree exists.
```

Theorem 5.8. *The Restricted Dijkstra-Steiner algorithm can be implemented to run in*

$$O(|\mathcal{A}|(\log|\mathcal{A}| + F + \tau) + |E_{\mathcal{A}}| + |M_{\mathcal{A}}|)$$

time, if

- *the future cost of each allowed label can be evaluated in $O(F)$ time and*
- *τ is a chosen such that for each allowed label $(v, I) \in \mathcal{A}$ the set of allowed edges and merges $M = \{(w, J) | \{(v, I), (w, J)\} \in E_{\mathcal{A}} \cup M_{\mathcal{A}}\}$ that contain the label can be computed in $O(\tau + |M|)$ time.*

Proof. By caching the future cost values we can ensure that the future cost function is evaluated at most once for every element of $\mathcal{A}$. Hence, the runtime for all evaluations is bounded by $O(|\mathcal{A}|F)$.

We use a Fibonacci heap [Fredman and Tarjan, 1987] to store the labels in $\{(v, I) \in \mathcal{A} \setminus R | l(v, I) \neq \infty\}$. Since every allowed label is selected at most once in line 3 we have at most $|\mathcal{A}|$ extract-min operations, which require $O(|\mathcal{A}| \log|\mathcal{A}|)$ time. The number of insert and decrease-key operations in line 9 and line 13 is at most $|E_{\mathcal{A}}|$ and $|M_{\mathcal{A}}|$ and each operation requires amortized constant time. Hence the runtime of all heap operations is $O(|\mathcal{A}| \log|\mathcal{A}| + |E_{\mathcal{A}}| + |M_{\mathcal{A}}|)$.

Since every allowed label is selected at most once in line 3, determining the sets of labels that need to be considered in line 7 and line 11 requires $O(|\mathcal{A}|\tau + |E_{\mathcal{A}}| + |M_{\mathcal{A}}|)$ time in total.

The runtime for backtracking and for all other operations is dominated by the runtime for the heap operations. Hence we get the desired runtime bound. □

We note that the Restricted Dijkstra-Steiner algorithm can easily be extended to allow non-negative node costs and merge costs in addition to edge costs. Moreover, each of these types of costs can be generalized to depend on the involved terminal set or terminal sets.

A different strategy for selecting the next allowed label (v, I) in line 3 of Algorithm 10 was developed by Stefan Rabenstein. Instead of future cost functions it uses so-called *certificates of validity* and *guaranteed errors* to reduce the number of iterations of the main loop. The approach can reduce the number of iterations but the selection of the next allowed label becomes more difficult. For more information we refer to [Rabenstein, 2019].

5.3 Global Routing Aware Steiner Tree Search

In this section we discuss how the Restricted Dijkstra-Steiner algorithm can be used in our application. We will restrict the search to solutions that are similar to the global routing of the net in terms of topology and approximate location of Steiner points. Then using the Restricted Dijkstra-Steiner algorithm has two main advantages: First, optimizing an entire net can yield a shorter and more efficient solution. Second, this allows computing solutions that are similar to the global routing, which is beneficial since global routing optimizes objectives such as timing (see [Held et al., 2018]) that are not considered by BonnRouteDetailed directly.

To simplify the presentation we consider the case that each pin of the net has a single location where it should be accessed. Extending the approach to the general case is straightforward. Hence, we identify each pin with a terminal $t \in \mathbb{Z} \times \mathbb{Z} \times \mathcal{Z}$. We first define what we mean by a global routing for a set of terminals T with root terminal t_0.

Definition 5.9 (global routing). *Let $T \subset \mathbb{Z} \times \mathbb{Z} \times \mathcal{Z}$ be a set of terminals with root terminal $t_0 \in T$. A* global routing *S_{global} for T and t_0 is an arborescence with root t_0 and set of leaves $T \setminus \{t_0\}$ such that $V(S_{global}) \subset \mathbb{Z} \times \mathbb{Z} \times \mathcal{Z}$ and either $x \neq x'$, $y \neq y'$, or $|z - z'| = 1$ for each edge $((x, y, z), (x', y', z')) \in E(S_{global})$.*

Recall from Definition 3.4 on page 27 that a routing area $A = \cup_{i \in \{1,\dots,l\}} a_i$ is the union of finitely many axis-parallel rectangles with disjoint interior. The rectangles a_i are usually the global routing tiles that intersect the global routing. We need the routing graph, the global routing, and the routing area to be compatible in the sense of the following definition.

Definition 5.10 (compatible). *Let S_{global} be a global routing, let G be a routing graph, and let $A = \cup_{i\in\{1,\dots,l\}} a_i$ be a routing area. S_{global}, G, and $A = \cup_{i\in\{1,\dots,l\}} a_i$ are* compatible *if*

- $V(S_{global}) \subset A$,
- *there is a node $v \in V(S_{global}) \cap a_i$ for every $i \in \{1,\dots,l\}$,*
- $\overline{vw} \cap \partial(a_i) \in \{\emptyset, v, w\}$ *for all $\{v,w\} \in E(S_{global})$ and $i \in \{1,\dots,l\}$, and*
- *for every $v \in V(G)$ there is exactly one $i \in \{1,\dots,l\}$ such that $v \in a_i$.*

We now define which terminal sets may be used in which parts of the routing area.

Definition 5.11 (allowed terminal sets, induced set of allowed labels). *Let T be a set of terminals with root terminal $t_0 \in T$, let S_{global} be a global routing for T and t_0, let G be a routing graph such that $T \subseteq V(G)$, and let $A = \cup_{i\in\{1,\dots,l\}} a_i$ be a routing area such that S_{global}, G, and $A = \cup_{i\in\{1,\dots,l\}} a_i$ are compatible.*

For each $i \in \{1,\dots,l\}$ the set of allowed terminal sets $\mathcal{I}_{a_i}$ *is defined by*

$$\mathcal{I}_{a_i} := \{ \cup_{i\in\{1,\dots,m\}} J_i | m \in \mathbb{N} \text{ and } \forall i \in \{1,\dots,m\} \text{ we have } J_i \in \mathcal{I}'_{a_i}\},$$

where

$$\begin{aligned}\mathcal{I}'_{a_i} := \{I \subseteq T \setminus \{t_0\} | \exists w \in (T \cap a_i) \cup (V(S_{global}) \cap \partial(a_i)) \text{ s.t.}\\ I \text{ is the set of leaves of the sub-arborescence of } S_{global} \text{ rooted at } w\}.\end{aligned}$$

We define the set of allowed labels $\mathcal{A}_{\mathcal{I}}$ for T, t_0, S_{global}, G, and A by

$$\mathcal{A}_{\mathcal{I}} := \bigcup_{i\in\{1,\dots,l\}} (V(G) \cap a_i) \times \mathcal{I}_{a_i}.$$

The idea is to define the set of allowed labels in such a way that we allow trees that are similar to the global routing but can deviate from it locally to reduce the cost of the tree. Within each tile we allow changing the topology and the location of Steiner points. See Figure 5.3 for an illustration.

We note that our definition allows shortcutting for example if a global routing contains a U-shape that intersects a 2×2-subgrid of global routing tiles. Moreover, if the global routing crosses a tile boundary with two parallel paths and merges them afterwards it allows merging the corresponding terminal sets in both tiles. If desired, this could be avoided by subdividing edges that connect nodes of adjacent tiles. Then, on the new nodes only the terminal sets which are used for crossing the corresponding boundary by the global routing would be allowed.

It is also possible to use slightly different allowed terminal sets for example to avoid solutions with unfavorable timing properties or to improve runtime. For example, we might remove $\{t_2, t_3\}$ and $\{t_2, t_4\}$ from the set of allowed terminals sets in the tile containing t_3

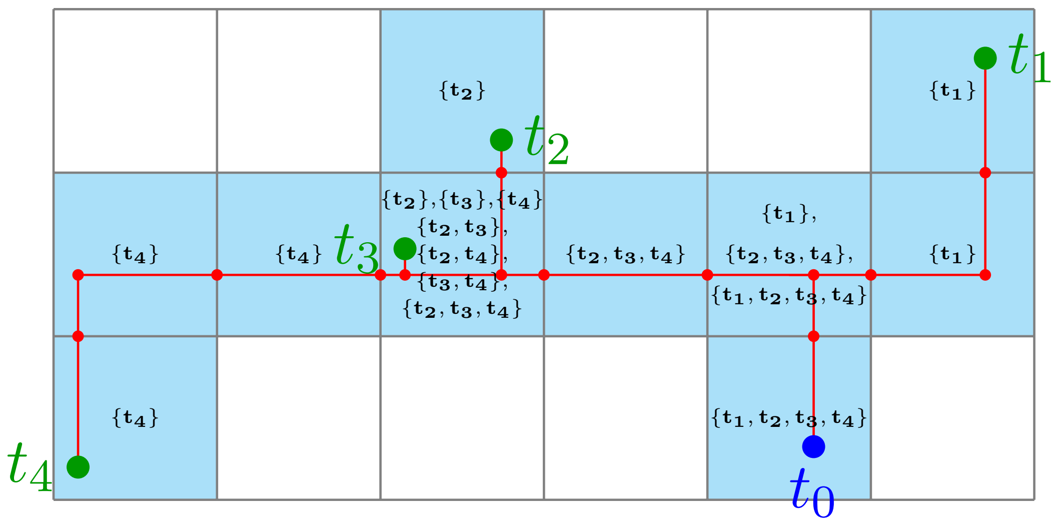

$\{t_2\}$
$\{t_1\}$
t_1
t_2
$\{t_4\}$
$\{t_4\}$
t_3
$\{t_2\},\{t_3\},\{t_4\}$, $\{t_2,t_3\}$, $\{t_2,t_4\}$, $\{t_3,t_4\}$, $\{t_2,t_3,t_4\}$
$\{t_2,t_3,t_4\}$
$\{t_1\}$, $\{t_2,t_3,t_4\}$, $\{t_1,t_2,t_3,t_4\}$
$\{t_1\}$
$\{t_4\}$
$\{t_1,t_2,t_3,t_4\}$
t_4
t_0

Figure 5.3: Illustration of the sets of allowed terminal sets for the drawn global routing and routing area for a net with five terminals. To make the figure clearer, the directed edges of the global routing are drawn as if they were undirected. The routing area is partitioned into 10 rectangles, corresponding to global routing tiles, each of which is labeled with the elements of its allowed terminal sets. In the tile containing t_3 we have $\mathcal{I}'_{a_i} = \{\{t_2\},\{t_3\},\{t_4\},\{t_2,t_3,t_4\}\}$. In reality, the routing area is three-dimensional.

in Figure 5.3. In practice, the routing area is also extended to neighboring layers to allow local track changes, as discussed in Section 3.2, and appropriate sets of allowed terminal sets would be defined there.

We can now define the Global Routing Aware Steiner Tree Problem.

Global Routing Aware Steiner Tree Problem

Input: A set of terminals T with root terminal $t_0 \in T$, a global routing S_{global} for T and t_0, a routing graph G such that $T \subseteq V(G)$, an edge cost function $c_E : E(G) \to \mathbb{R}_{\geq 0}$, and a routing area $A = \cup_{i\in\{1,\ldots,l\}} a_i$, such that S_{global}, G, and $A = \cup_{i\in\{1,\ldots,l\}} a_i$ are compatible.

Task: Find a $(t_0, T \setminus \{t_0\})$-tree with respect to $\mathcal{A}_\mathcal{I}$ with minimum cost or decide that no $(t_0, T \setminus \{t_0\})$-tree exists. Here $\mathcal{A}_\mathcal{I}$ denotes the set of allowed labels for T, t_0, S_{global}, G, and A.

The Global Routing Aware Steiner Tree Problem is NP-hard since it contains the Rectilinear Steiner Tree Problem, which is NP-hard [Garey and Johnson, 1977]. However, in our application the cardinality $|\mathcal{I}_{a_i}|$ of each set of allowed terminal sets is bounded, which allows us to solve the problem efficiently.

Definition 5.12 (η-bounded). *We call an instance $(T, t_0, S_{global}, A = \cup_{i\in\{1,\ldots,l\}} a_i, G, c_E)$ of the Global Routing Aware Steiner Tree Problem η-bounded, if $|\mathcal{I}_{a_i}| \leq \eta$ for every $i \in \{1,\ldots,l\}$.*

Lemma 5.13. *Let $\eta \in \mathbb{N}$. Then η-bounded instances $(T, t_0, S_{global}, A = \cup_{i\in\{1,\dots,l\}} a_i, G, c_E)$ of the* Global Routing Aware Steiner Tree Problem *can be solved in time*

$$O(|V(G)|\log|V(G)| + |E(G)| + \alpha(|V(G)| + |V(S_{global})|)),$$

where α is chosen such that for any given point $v \in \mathbb{Z}\times\mathbb{Z}\times\mathcal{Z}$ the set of indices $\{i \in \{1,\dots,l\}|v \in \overline{a_i}\}$ can be computed in $O(\alpha)$ time.

Proof. By traversing S_{global} in post-order the sets

$$\begin{aligned}\mathcal{I}'_{a_i} = \{I \subseteq T \setminus \{t_0\}|\exists w \in (T \cap a_i) \cup (V(S_{global}) \cap \partial(a_i)) \text{ s.t.}\\ I \text{ is the set of leaves of the sub-arborescence of } S_{global} \text{ rooted at } w\}\end{aligned}$$

for all $i \in \{1,\dots,l\}$ can be computed in $O(\alpha|V(S_{global})|)$ time. Since $|\mathcal{I}'_{a_i}| \le \eta$, the sets

$$\mathcal{I}_{a_i} = \{\cup_{i\in\{1,\dots,m\}} J_i | m \in \mathbb{N} \text{ and } \forall i \in \{1,\dots,m\} \text{ we have } J_i \in \mathcal{I}'_{a_i}\}$$

for all $i \in \{1,\dots,l\}$ can also be computed in $O(\alpha|V(S_{global})|)$ time.

In η-bounded instances we have at most η allowed labels for each node $v \in V(G)$, i.e. the set $\{I|(v,I) \in \mathcal{A}\}$ has cardinality at most η. This implies the bound $|E_{\mathcal{A}}| \le \eta|E(G)|$ on the size of the set of allowed edges and $|M_{\mathcal{A}}| \le \eta^2|V(G)|$ on the size of the set of allowed merges. Hence we can compute $E_{\mathcal{A}}$ and $M_{\mathcal{A}}$ and organize them in a graph $(\mathcal{A}_{\mathcal{I}}, E_{\mathcal{A}} \cup M_{\mathcal{A}})$ in $O(\alpha|V(G)|)$ time.

By Theorem 5.8 with $f \equiv 0$ we can solve the resulting instance of the Restricted Steiner Tree Problem in time

$$\begin{aligned}&O(|\mathcal{A}|(\log|\mathcal{A}| + \tau) + |E_{\mathcal{A}}| + |M_{\mathcal{A}}|)\\ =\,&O(|V(G)|(\log|V(G)|) + \tau) + |E(G)|)\\ =\,&O(|V(G)|(\log|V(G)|) + |E(G)|).\end{aligned}$$

The last bound follows because we can use the graph $(\mathcal{A}_{\mathcal{I}}, E_{\mathcal{A}} \cup M_{\mathcal{A}})$ to compute $M = \{(w,J)|\{(v,I),(w,J)\} \in E_{\mathcal{A}} \cup M_{\mathcal{A}}\}$ in $O(|M|)$ time for every $(v,I) \in \mathcal{A}$.

Hence, we get the desired runtime bound. □

In BonnRouteGlobal the size of the global routing tiles is constant and does not depend on the size of the instance and hence each tile contains only a constant number of nodes in $V(S_{global})$. Therefore our instances are η-bounded for some large constant $\eta \in \mathbb{N}$. Moreover, the tiles whose closure contains a point can be computed with two binary searches in arrays containing the boundary locations of the tiles in x- and y-dimension. If we assume that these arrays are not too large compared to the routing graph the runtime of the algorithm is near-linear in theory.

In practice more than 95% of the nets lead to 20-bounded instances, but there is also a small fraction of instances that is not 10000-bounded. For such instances we can limit the runtime by removing elements from the sets of allowed terminal sets.

To speed up the implementation we do not compute the sets of allowed edges $E_{\mathcal{A}}$ and allowed merges $M_{\mathcal{A}}$ before the tree search. Instead, we compute the set of allowed terminal sets in each rectangle of the routing area and use that information to determine which edges or terminal sets to consider in line 7 and line 11 of the algorithm.

Moreover, we use non-zero future costs to solve the Global Routing Aware Steiner Tree Problem faster in practice. In the Dijkstra-Steiner algorithm good future costs are much more important than in Dijkstra's algorithm. With their best future costs [Hougardy et al., 2017] report a reduction of the number of iterations of the main loop by a factor of at least 500 on several instances. Hence, they use future cost functions that are comparatively expensive to compute and even solve instances of the NP-hard Traveling Salesman Problem to derive good future costs.

In our application good future cost are not as important as in the case $\mathcal{A} = V \times 2^{T \setminus \{t_0\}}$ and the future costs proposed by [Hougardy et al., 2017] would be too expensive to compute. We need future cost functions that can be evaluated very efficiently because the graphs in our application are much larger and we need to connect millions of nets.

One option is to use *bounding box future costs* that extend the future costs for the path search that consider all dimensions at once. For more information on the future costs for the path search see Section 3.3.2. This generalizes an approach discussed briefly on page 23 of [Hougardy et al., 2017].

We will need some notation from Section 3.3. Recall that one component of the edge costs in BonnRouteDetailed is given by a *distance based cost function*, a term introduced in Definition 3.11 on page 32. For a distance based cost function $c = (\mathrm{c}^{\mathrm{wire}}, \mathrm{c}^{\mathrm{viadown}})$ the function $\mathrm{c}^{\mathrm{wire}}$ specifies a cost per length for a given layer and dimension and $\mathrm{c}^{\mathrm{viadown}}$ specifies the cost for vias which depends only on the involved via layer. It is *compatible* with a routing graph G and an edge cost function c_E if we have $\mathrm{cost}^c(e) \leq c_E(e)$ for each edge $e \in E(G)$, where $\mathrm{cost}^c(e)$ denotes the cost of e with respect to c. For $[x_1, x_2] \times [y_1, y_2] \times [z_1, z_2] \subset \mathbb{R}^3$ nonempty with $z_1, z_2 \in \mathcal{Z}$ we use $\mathrm{cost}^c([x_1, x_2] \times [y_1, y_2] \times [z_1, z_2])$ to refer to the cost of a cheapest (x_1, y_1, z_1)-(x_2, y_2, z_2)-path with respect to c.

Lemma 5.14. *Let $(T, t_0, S_{global}, A, G, c_E)$ be an instance of the* Global Routing Aware Steiner Tree Problem *with induced set of allowed labels $\mathcal{A}_{\mathcal{I}}$ and let c be a distance based cost function that is compatible with G and c_E. Then the function $f_{bb} : \mathcal{A}_{\mathcal{I}} \to \mathbb{R}_{\geq 0}$ defined by*

$$f_{bb}(v, I) := \mathrm{cost}^c(\mathrm{BoundingBox}(\{v\} \cup (T \setminus I)))$$

is a future cost function.

Proof. We need to show that f_{bb} satisfies the three properties from Definition 5.4.

Property 1:
We have $f_{bb}(t_0, T \setminus \{t_0\}) = \mathrm{cost}^c(\mathrm{BoundingBox}(\{t_0\} \cup (T \setminus (T \setminus \{t_0\})))) = \mathrm{cost}^c(\{t_0\}) = 0$, as required.

We will use the following simple observation to prove the other properties: Let B be an axis-parallel cuboid and let $\{v, w\} \in E(G)$ be an edge with $w \in B$. Then for $B' = \text{BoundingBox}(B \cup \{v\})$ we have $\text{cost}^c(B') \leq \text{cost}^c(B) + \text{cost}^c(\overline{vw})$. Since c is compatible with G and c_E we get $\text{cost}^c(B') \leq \text{cost}^c(B) + c_E(\{v, w\})$.

Property 2:
Let $(v, I), (w, I) \in \mathcal{A}_\mathcal{I}$ with $\{v, w\} \in E(G)$. Then using the observation for $\{v, w\}$ and $\text{BoundingBox}(\{w\} \cup (T \setminus I))$ yields

$$\begin{aligned} f_{bb}(v, I) &\leq \text{cost}^c(\text{BoundingBox}(\{v, w\} \cup (T \setminus I))) \\ &\leq \text{cost}^c(\text{BoundingBox}(\{w\} \cup (T \setminus I))) + c_E(\{v, w\}) \\ &= f_{bb}(w, I) + c_E(\{v, w\}), \end{aligned}$$

as desired.

Property 3:
Let $(v, I), (v, J) \in \mathcal{A}_\mathcal{I}$ such that $(v, I \cup J) \in \mathcal{A}_\mathcal{I}$ and $I \cap J = \emptyset$. Let S be a cheapest (v, J)-tree and let

$$\begin{aligned} V_S &= \{w | \exists K \text{ s.t. } (w, K) \in V(S)\} \text{ and} \\ E_S &= \{\{w, w'\} | \exists K \text{ s.t. } ((w, K), (w', K)) \in E(S)\} \end{aligned}$$

be set of nodes and edges of G that occur in S. By using the observation iteratively we get

$$\begin{aligned} f_{bb}(v, I) &\leq \text{cost}^c(\text{BoundingBox}(\{v\} \cup (T \setminus (I \cup J)) \cup V_S\}) \\ &\leq \text{cost}^c(\text{BoundingBox}(\{v\} \cup (T \setminus (I \cup J))) + \sum_{e \in E_S} c_E(e) \\ &\leq f_{bb}(v, I \cup J) + \text{smt}(v, J), \end{aligned}$$

as desired. □

To enable efficient queries of f_{bb} we can compute the bounding box of each terminal set in $\{T \setminus I | \exists v \text{ s.t. } (v, I) \in \mathcal{A}_\mathcal{I}\}$ before the tree search. Then evaluating $f_{bb}(v, I)$ requires accessing the bounding box for $T \setminus I$, extending it to include v, and querying the future cost function from the path search. All these operations can be performed very efficiently. In practice the computation is slightly more complicated because every pin can have multiple locations where it can be accessed.

A strategy for obtaining better future costs is to generalize the rectangle-labeling approach of [Peyer et al., 2009]. Their algorithm is a generalization of Dijkstra's algorithm that propagates cost functions on rectangles instead of individual nodes. We believe that it is possible to generalize the Restricted Dijkstra-Steiner algorithm to label rectangles instead of individual nodes and that this approach can be used to efficiently compute good future costs. This is an interesting area for future research.

We note that the components of the same-net rule aware path search framework introduced in Chapter 4 can be extended in a natural way to handle Steiner trees searches. The only non-trivial modification is extending the multi-labeling to avoid violations at Steiner points.

Finally, Figure 5.4 shows an example of a net that was routed by BonnRouteDetailed with path searches and with a single tree search.

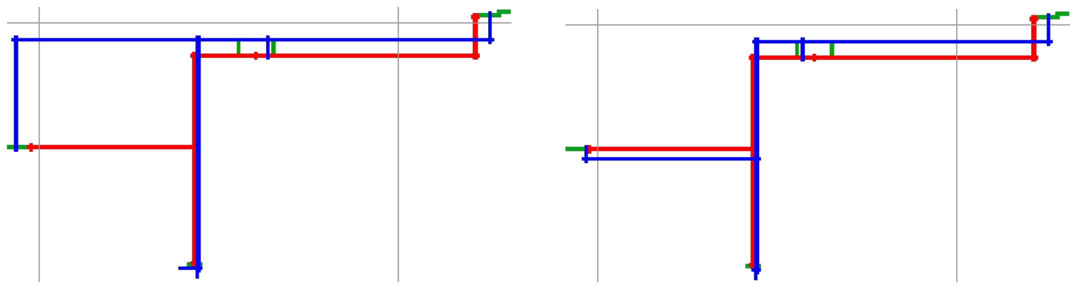


Figure 5.4: A global routing and two different detailed routings of the same net. The detailed routing on the left was computed with path searches and the one on the right with a single tree search. Note that the tree search leads to a shorter detailed routing that is more similar to the global routing.

Chapter 6

Experimental Results

In this chapter we analyze how the path search presented in Chapter 3 and Chapter 4 performs on real-world IBM instances. First, in Section 6.1 we describe our testbed and other parts of the setup that are identical for all runs. Then, in Section 6.2, we provide experimental results demonstrating the effectiveness of the speed-up techniques presented in Chapter 3. For example, making the search goal-oriented as discussed in Section 3.3 reduces the runtime of the path search by approximately two-thirds. Finally, Section 6.3 provides experimental results on the same-net rule aware path search framework presented in Chapter 4 and on its components. Using all components reduces the number of violations by a factor of approximately 443.

6.1 Testbed and Setup

The algorithms examined in this chapter were implemented as part of BonnRouteDetailed, a state-of-the-art detailed router developed at the University of Bonn in joint work with IBM. BonnRouteDetailed is the main detailed routing tool used by IBM for the design of its processor chips. For more details on BonnRouteDetailed we refer to Section 2.2 and [Gester et al., 2013; Ahrens et al., 2015].

Table 6.1 gives an overview of our testbed. It consists of 62 real-world instances varying in size between 149 and 1.78 million nets. Each instance belongs to one of four groups: For both the 7 nm and the 14 nm technology there is a set of Random Logic Macros (RLMs), comparatively small instances, and a set of blocks, which are approximately one order of magnitude larger on average. Moreover, the blocks have roughly twice as many layers on average.

Some wiring, e.g. for clock nets is already present on the instances. On our testbed, the input wiring has almost 17 thousand design rule violations in total (input DRVs), among those roughly 14 thousand on the 14 nm blocks. Since BonnRouteDetailed does not try to clean up violations in the input, most of them will still be present after BonnRouteDetailed.

Chip Set	**# Instances**	**Average # Layers**	**# Nets**	**Image Size [mm^2]**	**Input DRVs**
14 nm RLMs	28	6.0	1954118	2.34	261
14 nm blocks	4	13.5	4236762	4.89	13971
7 nm RLMs	21	8.9	2016798	0.92	1447
7 nm blocks	9	15.0	7612200	4.29	1204
Sum	**62**		**15819878**		**16883**

Table 6.1: Testbed consisting of 62 real-world 14 nm and 7 nm instances. The statistics are summed up over all instances in the group of instances, except for the average number of layers.

This is important in Section 6.3, which uses the number of design rule violations after BonnRouteDetailed as a metric.

All runs were done on the same AMD EPYC 7601 machine with 64 CPUs and 512 GB main memory using 64 threads.

6.2 Efficient Path Search

We begin by analyzing the effect of the future cost functions presented in Section 3.3 which speed up the search by guiding it towards the targets. In practice, we do an additional preprocessing step and cluster nearby targets on the same layer, yielding a bounding box for each cluster. The future costs are initialized with these bounding boxes instead of the original targets. This can degrade the quality of the future costs but speeds up the future cost computation and makes it more robust against instances with many targets. In practice the effects cancel each other out: The clustering saves approximately 1% runtime without any negative side-effects. It is used in all runs, except for the run with no future costs.

Table 6.2 compares results with different future cost functions. Compared to using no future costs, using the l_1-future costs reduces the runtime of BonnRouteDetailed by 28.0%. The number of labels and the runtime of the path search are reduced by 54.4% and 50.3%. Using the shortest path future costs leads to a slightly larger improvement of 34.0% BonnRouteDetailed runtime, 65.7% number of labels, and 62.2% path search runtime. Since the shortest path future costs perform best, they are used in all other experiments.

In ripup searches, using l_1- or shortest path future costs leads to a much smaller improvement, e.g. 33.9% in the path search runtime with shortest path future costs, compared to 62.2% gain in all searches. This is expected since ripup searches use large penalty costs for wires with conflicts and these costs are not accounted for in our future cost functions.

We note that the runtime for the preprocessing of the shortest path future costs (Algorithm 4 on page 41) is less than one hour on the entire testbed, which is less than 1% of the path search runtime.

	BRD	**all searches**		**ripup searches**	
Chip Set	**time [h:m]**	**Labels $\times 10^6$**	**time [h:m]**	**Labels $\times 10^6$**	**time [h:m]**
14nm RLMs	4:31	243384	137:10	14881	29:22
	3:25	97819	61:34	10608	21:42
	-24.4%	-59.8%	-55.1%	-28.7%	-26.1%
	3:07	82766	53:46	10074	21:09
	-31.0%	-66.0%	-60.8%	-32.3%	-28.0%
14nm blocks	9:43	537151	347:43	37885	63:22
	7:16	249557	181:59	28623	49:43
	-25.2%	-53.5%	-47.7%	-24.4%	-21.5%
	6:34	177728	132:09	24735	44:42
	-32.3%	-66.9%	-62.0%	-34.7%	-29.5%
7nm RLMs	3:22	140056	105:13	9211	17:06
	2:19	53122	41:28	6075	11:06
	-31.2%	-62.1%	-60.6%	-34.0%	-35.1%
	2:20	40943	32:14	5577	10:11
	-30.8%	-70.8%	-69.8%	-39.5%	-40.4%
7nm blocks	19:10	802722	700:00	69017	141:22
	13:30	385727	355:58	47866	97:57
	-29.6%	-51.9%	-49.1%	-30.6%	-30.7%
	12:14	289082	269:59	43399	89:56
	-36.2%	-64.0%	-61.4%	-37.1%	-36.4%
Sum	**36:46**	**1723313**	**1290:06**	**130994**	**251:13**
	26:29	**786225**	**641:00**	**93171**	**180:29**
	-28.0%	**-54.4%**	**-50.3%**	**-28.9%**	**-28.2%**
	24:15	**590519**	**488:07**	**83784**	**165:58**
	-34.0%	**-65.7%**	**-62.2%**	**-36.0%**	**-33.9%**

Table 6.2: Comparison of BonnRouteDetailed with no future costs (black row), l_1-future costs (blue row) introduced on page 32, and shortest path future costs (orange row) using Algorithm 4 on page 41 as preprocessing. The table shows *BRD time*, i.e. the wall time of BonnRouteDetailed, the number of labels, and the runtime for all searches and for ripup searches. Here *ripup searches* denotes all searches in which existing routing objects are not seen as blockages and *all searches* denotes both ripup and non-ripup searches. The runtime of the path searches is summed up over all 64 threads.

We now examine the effects of the grid region data structure presented in Section 3.4. The grid region data structure allows BonnRouteDetailed to handle complex regional track structures in the path search and to modify the routing graph locally, e.g. to simplify and improve pin access. While this advantage is not fully used yet, the data structure also performs some preprocessing steps that speed up queries during the path search. Table 6.3 shows that this reduces the runtime of BonnRouteDetailed by 10.3% and the runtime of the path search by 24.2%. The bulk of the speed-up is due to the precomputation of the track graph at the beginning of the path search when using the grid region data structure which allows us to use handles to represent nodes in the path search and enables very efficient queries as shown in Lemma 3.25 on page 46 and Lemma 3.28 on page 50. Without the grid region data structure, we need to store node locations during the path search. Thus, queries for the location of neighbors and for the data associated with node-locations are necessary (e.g. for accessing the label), which slows down the search. We note that in ripup searches a larger percentage of the runtime is spent by the checking oracle, which is not affected, and thus the runtime improvement is smaller with 10.9% compared to 24.2% in all searches. Since using the grid region data structure leads to better results it is used in all other experiments.

	BRD	**all searches**	**ripup searches**
Chip Set	**time [h:m]**	**time [h:m]**	**time [h:m]**
14nm RLMs	3:30	69:30	22:47
	3:07	53:46	21:09
	-10.9%	-22.6%	-7.2%
14nm blocks	7:42	185:44	51:29
	6:34	132:09	44:42
	-14.6%	-28.9%	-13.2%
7nm RLMs	2:24	41:52	11:38
	2:20	32:14	10:11
	-3.0%	-23.0%	-12.4%
7nm blocks	13:26	346:40	100:26
	12:14	269:59	89:56
	-9.0%	-22.1%	-10.5%
Sum	**27:02**	**643:46**	**186:20**
	24:15	**488:07**	**165:58**
	-10.3%	**-24.2%**	**-10.9%**

Table 6.3: Comparison of BonnRouteDetailed without the grid region data structure and with the grid region data structure. The metrics are explained in Table 6.2.

Finally, we note (without providing a table) that returning intervals in the checking oracle, i.e. not checking each point individually but sharing checking data if possible, is an important optimization: Disabling it increases the runtime of BonnRouteDetailed by approximately 20%.

6.3 Same-Net Rule Aware Path Search

A layout can be manufactured only if it respects the design rules, i.e. there are no design rule violations. In production an industrial router is used to clean up the violations remaining after BonnRouteDetailed. This step is very disruptive and often degrades timing properties of the net with the violation, but also of bystander nets that have to be modified to resolve the violation. Violations remaining after the clean up must be fixed manually. For this reason it is crucial to minimize the number of violations after BonnRouteDetailed.

Table 6.4 shows that using the same-net rule aware path search (see Algorithm 9 on page 84) reduces the number of design rule violations by 99.77%, compared to a version of BonnRouteDetailed that uses shortest paths. This corresponds to a reduction in the number of design rule violations by a factor of ≈ 443 from 2180 violations per one thousand nets to 4.9 violations for every one thousand nets. If we subtract the roughly 17 thousand input errors mentioned in Table 6.1 the violations even reduce by a factor of ≈ 565. The version of BonnRouteDetailed used for these experiments is optimized for 7 nm. On the 7 nm RLMs and 7 nm blocks, the total number of design rule violations is even lower with 2.3 and 2.7 errors per thousand nets. The number of layer fuses is reduced by 85.3%.

Chip Set	BRD time [h:m]	# Vias $\times 10^3$	Wire Length (m)	Scenics 25	Layer Fuses	# DRVs
14nm RLMs	2:53	15815	34.1423	13750	2588	3677984
	3:07	15854	34.2431	14743	872	9686
	+8.37%	+0.24%	+0.30%	+7.22%	-66.31%	-99.74%
14nm blocks	5:41	36009	87.7320	19675	18905	9549941
	6:34	36036	88.0097	22865	4787	43252
	+15.66%	+0.08%	+0.32%	+16.21%	-74.68%	-99.55%
7nm RLMs	1:51	19766	20.1593	11167	15569	3561478
	2:20	19831	20.2504	12025	1161	4729
	+25.65%	+0.33%	+0.45%	+7.68%	-92.54%	-99.87%
7nm blocks	9:27	83766	117.2131	75561	176256	17704337
	12:14	84007	117.8299	85137	24570	20210
	+29.44%	+0.29%	+0.53%	+12.67%	-86.06%	-99.89%
Sum	**19:52**	**155356**	**259.2467**	**120153**	**213318**	**34493740**
	24:15	**155727**	**260.3331**	**134770**	**31390**	**77877**
	+22.09%	**+0.24%**	**+0.42%**	**+12.17%**	**-85.28%**	**-99.77%**

Table 6.4: BonnRouteDetailed when using shortest paths (black row) and when using the same-net rule aware path search (blue row). The table shows *BRD time*, i.e. the wall time of BonnRouteDetailed, *# vias*, i.e. interconnects between different wiring layers, the *wire length*, i.e. the length of the wiring in meters, and *# DRVs*, i.e. the number of design rule violations. Furthermore it shows the number of scenic nets with detour of at least 25% compared to a Steiner tree estimate and length at least 25 µm and the number of layer fuses, a configuration that often leads to bad timing behavior that is defined on page 80.

[Gester, 2015] and [Ahrens et al., 2015] report approximately one order of magnitude more violations with roughly 53.5 and 58.4 violations per thousand nets. This indicates that BonnRouteDetailed got much better at avoiding design rule violations.

Moreover, the same-net rule aware path search has reasonably little overhead, increasing the runtime by 22.1%, the number of vias by 0.24%, and the wire length by 0.42%. "Scenics 25", the number of nets with 25% or more detour compared to a Steiner tree estimate and length at least 25 µm increase by 12.2%. Since avoiding design rule violations and unfavorable configurations such as layer fuses may require detours, an increase in the number of vias, the wire length, and the number of scenic nets is expected.

The enormous improvement in the number of design rule violations with relatively little side effects demonstrates the effectiveness of the same-net rule aware path search framework.

The remaining part of this section analyzes the contribution of its components: the post-processing, the multi-labeling, and the protections.

6.3.1 Post-Processing

The post-processing from Section 4.4 is called after each path search and tries to resolve violations by modifying the path locally. Table 6.5 shows that the post-processing reduces the number of design rule violations by 89.5% with 16.1% runtime penalty and a small increase in the number of vias, the wire length, and the number of scenic nets. Most of the violations remaining after post-processing cannot be fixed locally and require modifying the structure of the path. In that sense, the post-processing resolves the easiest-to-fix violations.

Chip Set	**BRD time [h:m]**	**# Vias** $\times 10^3$	**Wire Length (m)**	**Scenics 25**	**Layer Fuses**	**# DRVs**
14nm RLMs	2:53	15815	34.1423	13750	2588	3677984
	2:58	15828	34.1640	13847	2478	701852
	+2.87%	+0.08%	+0.06%	+0.71%	-4.25%	-80.92%
14nm blocks	5:41	36009	87.7320	19675	18905	9549941
	6:58	36022	87.7950	20434	18938	1507334
	+22.61%	+0.04%	+0.07%	+3.86%	+0.17%	-84.22%
7nm RLMs	1:51	19766	20.1593	11167	15569	3561478
	2:09	19790	20.1909	11450	15666	227366
	+16.27%	+0.12%	+0.16%	+2.53%	+0.62%	-93.62%
7nm blocks	9:27	83766	117.2131	75561	176256	17704337
	10:58	83888	117.4539	77532	176717	1188227
	+16.13%	+0.15%	+0.21%	+2.61%	+0.26%	-93.29%
Sum	**19:52**	**155356**	**259.2467**	**120153**	**213318**	**34493740**
	23:03	**155528**	**259.6039**	**123263**	**213799**	**3624779**
	+16.07%	**+0.11%**	**+0.14%**	**+2.59%**	**+0.23%**	**-89.49%**

Table 6.5: BonnRouteDetailed when using shortest paths (black row) and when using shortest paths with post-processing (blue row). The metrics are the same as in Table 6.4.

6.3.2 Multi-Labeling

The multi-labeling from Section 4.5 modifies Dijkstra's algorithm [Dijkstra, 1959] to avoid design rule violations. Table 6.6 shows that using the default multi-labeling reduces the number of violations by 95.7%. This improvement is on top of the improvement of the post-processing which already resolved 89.5% of the violations. Moreover, the number of layer fuses reduces by 85.5%. Runtime increases by only 4.2%, there are roughly the same number of vias, 0.26% increased wire length, and 8.5% more scenic nets. Except for the increase in the number of scenic nets, the increases are negligible. This shows that the default multi-labeling resolves approximately 22 out of 23 violations remaining after post-processing with very few negative side-effects.

Chip Set	**BRD time [h:m]**	**# Vias $\times 10^3$**	**Wire Length (m)**	**Scenics 25**	**Layer Fuses**	**# DRVs**
14nm RLMs	2:58	15828	34.1640	13847	2478	701852
	3:03	15833	34.2338	14650	696	30882
	+3.16%	+0.03%	+0.20%	+5.80%	-71.91%	-95.60%
	3:33	15840	34.2410	14594	687	22550
	+19.71%	+0.07%	+0.23%	+5.39%	-72.28%	-96.79%
14nm blocks	6:58	36022	87.7950	20434	18938	1507334
	6:33	36024	88.0042	22695	4818	61785
	-5.90%	+0.00%	+0.24%	+11.06%	-74.56%	-95.90%
	7:20	36022	88.0256	22687	4452	44681
	+5.30%	+0.00%	+0.26%	+11.03%	-76.49%	-97.04%
7nm RLMs	2:09	19790	20.1909	11450	15666	227366
	2:14	19792	20.2401	11966	1081	10529
	+3.46%	+0.01%	+0.24%	+4.51%	-93.10%	-95.37%
	2:31	19888	20.2819	12168	1193	9962
	+16.62%	+0.49%	+0.45%	+6.27%	-92.38%	-95.62%
7nm blocks	10:58	83888	117.4539	77532	176717	1188227
	12:11	83873	117.7888	84465	24508	51473
	+11.07%	-0.02%	+0.29%	+8.94%	-86.13%	-95.67%
	13:44	84228	117.9477	86238	23640	46088
	+25.17%	+0.41%	+0.42%	+11.23%	-86.62%	-96.12%
Sum	**23:03**	**155528**	**259.6039**	**123263**	**213799**	**3624779**
	24:01	**155523**	**260.2670**	**133776**	**31103**	**154669**
	+4.22%	**-0.00%**	**+0.26%**	**+8.53%**	**-85.45%**	**-95.73%**
	27:07	**155978**	**260.4961**	**135687**	**29972**	**123281**
	+17.67%	**+0.29%**	**+0.34%**	**+10.08%**	**-85.98%**	**-96.60%**

Table 6.6: The first row shows the results of BonnRouteDetailed when using shortest paths with post-processing. In the blue row the multi-labeling is enabled as in Algorithm 9 on page 84, i.e. we first search for a shortest path and repeat the search with increasingly more restrictive label systems if violations remain. We refer to this version as default multi-labeling. In the orange row the setup is the same as in the blue row, except that the StandardRouting base label system is not available and that the framework starts with the AvoidSpecificMistakes label system, i.e. multi-labeling is enabled in every path search. The metrics are the same as in Table 6.4.

Recall that the checking data computed for one path search may be reused in a path search with a different label system, as explained in Algorithm 9 on page 84. Without this optimization the runtime of BonnRouteDetailed with default multi-labeling would be approximately 10% higher.

Using multi-labeling in every search decreases the number of design rule violations even further by 96.6% instead of 95.7% and the layer fuses by 86.0% instead of 85.5%. On the other hand, it increases the number of vias by 0.29% instead of 0%, the wire length by 0.34% instead of 0.26%, and the number of scenic nets by 10.1% instead of 8.5%. Moreover, the runtime increases by 17.7% instead of 4.2%.

Comparing the results of both runs with multi-labeling yields some interesting insights:

First, observe that using multi-labeling in every path search reduces the number of design rule violations by approximately 20%, compared to the default multi-labeling. This indicates that the same-net checking, which is used to control the multi-labeling, is reasonably effective, but that it might be possible to reduce the number of design rule violations with the default multi-labeling by up to 20 additional percent by improving the same-net checking.

Second, note that using AvoidSpecificMistakes or a more restrictive label system in every path search leads to a very small increase in the number of vias, the wire length, and the number of scenic nets. This shows that AvoidSpecificMistakes works as designed in the sense that it has very little pessimism. Using the least restrictive version of the AvoidAllMistakes label system of [Gester, 2015; Ahrens et al., 2015] in every search, which we replaced by AvoidSpecificMistakes, lead to an increase in the number of vias by 2-3%, compared to 0.29% with the AvoidSpecificMistakes label system, demonstrating that AvoidSpecificMistakes has much less pessimism.

6.3.3 Protections

The protections from Section 4.6 impose restrictions to avoid design rule violations at the start and end of paths. Table 6.7 shows that using the protections reduces the number of design rule violations by 49.6% at the cost of some additional layer fuses on the RLMs and 0.13% additional vias. All other differences are so small that they might be caused by random fluctuations. The improvement in the number of design rule violations is on top of the 89.5% gained by the post-processing and the 95.7% gained by the multi-labeling.

Chip Set	**BRD time [h:m]**	**# Vias $\times 10^3$**	**Wire Length (m)**	**Scenics 25**	**Layer Fuses**	**# DRVs**
14nm RLMs	3:03	15833	34.2338	14650	696	30882
	3:07	15854	34.2431	14743	872	9686
	+2.13%	+0.13%	+0.03%	+0.63%	+25.29%	-68.64%
14nm blocks	6:33	36024	88.0042	22695	4818	61785
	6:34	36036	88.0097	22865	4787	43252
	+0.25%	+0.04%	+0.01%	+0.75%	-0.64%	-30.00%
7nm RLMs	2:14	19792	20.2401	11966	1081	10529
	2:20	19831	20.2504	12025	1161	4729
	+4.45%	+0.19%	+0.05%	+0.49%	+7.40%	-55.09%
7nm blocks	12:11	83873	117.7888	84465	24508	51473
	12:14	84007	117.8299	85137	24570	20210
	+0.35%	+0.16%	+0.03%	+0.80%	+0.25%	-60.74%
Sum	**24:01**	**155523**	**260.2670**	**133776**	**31103**	**154669**
	24:15	**155727**	**260.3331**	**134770**	**31390**	**77877**
	+0.93%	**+0.13%**	**+0.03%**	**+0.74%**	**+0.92%**	**-49.65%**

Table 6.7: The first row shows the results of BonnRouteDetailed with protections disabled. In the blue row the protections are enabled. The metrics are the same as in Table 6.4.

Summary

In this thesis we consider detailed routing, an important step in the design of integrated circuits. On large instances detailed routing requires packing millions of node-disjoint Steiner trees into a graph with hundreds of billions of nodes, while respecting hundreds of complicated design rules. The Steiner trees are usually composed of paths.

One of our main contributions is an efficient and flexible path search algorithm. Our path search is the algorithmic core of BonnRouteDetailed, a state-of-the-art detailed router developed at the University of Bonn in joint work with IBM. It is being used very successfully in the IBM design flow to design complex processor chips.

We show how to make the path search efficient while retaining flexibility. A key component is our grid region data structure that speeds up the search and allows us to handle complicated implicitly given grid graphs. Moreover, we make the search goal-oriented using the well-known concept of future costs. Our future costs are computed by solving a geometric shortest path problem. We prove that the problem can be solved in logarithmic time after a polynomial time preprocessing. Furthermore, we propose an algorithm for solving this problem efficiently in practice, leading to a reduction of path search runtime by approximately two-thirds in our experiments.

Moreover, we consider the problem of respecting design rules in the path search. Obeying even simple rules is NP-hard. This follows from one of our main theoretical results that given a two-dimensional grid graph and nodes s, t it is NP-complete to decide whether there is an s-t-path in which each maximal straight subpath has length at least two. Nevertheless, our path search is very good at respecting design rules in practice. Using our framework, consisting of same-net checking, post-processing, multi-labeling, and protections, reduces the number of violations by a factor of approximately 443 in our experiments. Its most important component is the multi-labeling that allows us to find edge progressions that satisfy certain properties specified by label systems. This allows us to respect design rules in a correct-by-construction manner. Our multi-labeling is more general and more efficient than previous ones, which allows us to respect more rules while being less restrictive.

We compare our path search to the previous implementation in BonnRouteDetailed based on [Hetzel, 1995, 1998]: Ours supports more general cost functions, leads to vastly superior detailed routings, and is less complicated and much easier to extend.

Composing Steiner trees of paths can lead to non-optimal solutions. Hence, we extend our path search to compute optimal Steiner trees respecting restrictions on the topology and on the location of Steiner points derived from the global routing. To achieve this we introduce the Restricted Dijkstra-Steiner algorithm, which generalizes the Dijkstra-Steiner algorithm. In our application the Restricted Dijkstra-Steiner algorithm achieves near-linear runtime under mild assumptions. This is possible because the restrictions actually make the problem easier.

Due to the better algorithmic core that we describe in this thesis, our new BonnRoute computes excellent routing solutions fast and is being used by IBM for the design of all its processor chips. Moreover, we lay the foundation for future enhancements.

Bibliography

Ahrens, M. (2014). Pin access in VLSI-routing. Master's thesis, University of Bonn. (Cited on page 17)

Ahrens, M., Gester, M., Klewinghaus, N., Müller, D., Peyer, S., Schulte, C., and Téllez, G. (2015). Detailed routing algorithms for advanced technology nodes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(4):563–576. (Cited on pages 2, 13, 17, 56, 57, 58, 69, 70, 76, 77, 79, 83, 101, 105, and 108)

Alpert, C. J., Mehta, D. P., and Sapatnekar, S. S. (2008). *Handbook of algorithms for physical design automation*. CRC Press. (Cited on pages 3 and 7)

Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., and Werneck, R. F. (2016). Route planning in transportation networks. In *Algorithm Engineering*, pages 19–80. Springer. (Cited on page 26)

Byrka, J., Grandoni, F., Rothvoß, T., and Sanità, L. (2013). Steiner tree approximation via iterative randomized rounding. *Journal of the ACM (JACM)*, 60(1):1–33. (Cited on page 87)

Chang, F.-Y., Tsay, R.-S., Mak, W.-K., and Chen, S.-H. (2013). 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. (Cited on page 59)

Chen, G., Pui, C.-W., Li, H., and Young, E. F. (2019). Dr. CU: Detailed routing by sparse grid graph and minimum-area-captured path search. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. (Cited on page 59)

Cherkassky, B. V., Goldberg, A. V., and Radzik, T. (1996). Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174. (Cited on page 23)

Chlebikova, J. and Chlebík, M. (2008). The Steiner tree problem on graphs: Inapproximability results. *Theoretical Computer Science*, 406(3):207–214. (Cited on page 87)

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press. (Cited on page 24)

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271. (Cited on pages 23 and 106)

Dobkin, D. and Lipton, R. J. (1976). Multidimensional searching problems. *SIAM Journal on Computing*, 5(2):181–186. (Cited on page 36)

Dreyfus, S. E. (1969). An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412. (Cited on page 25)

Dreyfus, S. E. and Wagner, R. A. (1971). The Steiner problem in graphs. *Networks*, 1(3):195–207. (Cited on page 87)

Edelsbrunner, H., Guibas, L. J., and Stolfi, J. (1986). Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340. (Cited on page 38)

Erickson, R. E., Monma, C. L., and Veinott Jr, A. F. (1987). Send-and-split method for minimum-concave-cost network flows. *Mathematics of Operations Research*, 12(4):634–664. (Cited on page 87)

Fellows, M. R., Kratochvíl, J., Middendorf, M., and Pfeiffer, F. (1995). The complexity of induced minors and related problems. *Algorithmica*, 13(3):266–282. (Cited on page 60)

Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615. (Cited on pages 23 and 93)

Gao, J.-R. and Pan, D. Z. (2012). Flexible self-aligned double patterning aware detailed routing with prescribed layout planning. In *Proceedings of the ACM International Symposium on Physical Design*, pages 25–32. (Cited on page 59)

Garey, M. R. and Johnson, D. S. (1977). The rectilinear Steiner tree problem is NP-complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834. (Cited on pages 87 and 96)

Gester, M. (2015). *VLSI Routing for Advanced Technology.* PhD thesis, University of Bonn. (Cited on pages 2, 56, 57, 58, 69, 70, 74, 76, 77, 79, 83, 105, and 108)

Gester, M., Müller, D., Nieberg, T., Panten, C., Schulte, C., and Vygen, J. (2013). BonnRoute: Algorithms and data structures for fast and good VLSI routing. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(2):32. (Cited on pages 12, 13, 24, 29, and 101)

Goldberg, A. V. and Harrelson, C. (2005). Computing the shortest path: A search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165. Society for Industrial and Applied Mathematics. (Cited on page 32)

Goldberg, A. V., Kaplan, H., and Werneck, R. F. (2006). Reach for A*: Efficient point-to-point shortest path algorithms. In *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 129–143. SIAM. (Cited on page 26)

Goldberg, A. V. and Werneck, R. F. F. (2005). Computing point-to-point shortest paths from external memory. In *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments (ALENEX 2005)*, pages 26–40. (Cited on page 25)

Gonçalves, S. M., Rosa, L. S., and Marques, F. S. (2019). DRAPS: A design rule aware path search algorithm for detailed routing. *IEEE Transactions on Circuits and Systems II: Express Briefs*. (Cited on page 59)

Güting, R. H. (1984). An optimal contour algorithm for iso-oriented rectangles. *Journal of Algorithms*, 5(3):303–326. (Cited on page 67)

Gutman, R. J. (2004). Reach-based routing: A new approach to shortest path algorithms optimized for road networks. *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 4:100–111. (Cited on page 26)

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107. (Cited on page 29)

Held, S., Müller, D., Rotter, D., Scheifele, R., Traub, V., and Vygen, J. (2018). Global routing with timing constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(2):406–419. (Cited on pages 12 and 94)

Henke, D. (2016). Pfadsuche im Detailed Routing. Bachelor's thesis, University of Bonn. (Cited on pages 29, 31, 34, 35, 40, and 41)

Hetzel, A. (1995). *Verdrahtung im VLSI-Design: Spezielle Teilprobleme und ein sequentielles Lösungsverfahren*. PhD thesis, University of Bonn. (Cited on pages 2, 11, 21, 24, 25, 32, 55, 56, and 109)

Hetzel, A. (1998). A sequential detailed router for huge grid graphs. In *Design, Automation and Test in Europe, 1998., Proceedings*, pages 332–338. IEEE. (Cited on pages 2, 11, 21, 24, 25, 29, 32, 55, 56, and 109)

Hougardy, S., Silvanus, J., and Vygen, J. (2017). Dijkstra meets Steiner: a fast exact goal-oriented Steiner tree algorithm. *Mathematical Programming Computation*, 9(2):135–202. (Cited on pages 1, 2, 12, 87, 88, 92, and 98)

Humpola, J. (2009). Schneller Algorithmus für kürzeste Wege in irregulären Gittergraphen. Diplomarbeit, University of Bonn. (Cited on page 56)

Ikeda, T., Hsu, M.-Y., Imai, H., Nishimura, S., Shimoura, H., Hashimoto, T., Tenmoku, K., and Mitoh, K. (1994). A fast algorithm for finding better routes by AI search techniques. In *Proceedings of VNIS'94-1994 Vehicle Navigation and Information Systems Conference*, pages 291–296. IEEE. (Cited on page 25)

Kahng, A. B., Wang, L., and Xu, B. (2018). TritonRoute: an initial detailed router for advanced VLSI technologies. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE. (Cited on page 59)

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer. (Cited on page 87)

Kirkpatrick, D. (1983). Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35. (Cited on page 38)

Klewinghaus, N. (2013a). Effiziente Detailverdrahtung (efficient detailed routing). Diplomarbeit, University of Bonn. (Cited on pages 15 and 27)

Klewinghaus, N. (2013b). Fast parallelisation for detailed routing in VLSI design. Diplomarbeit, University of Bonn. (Cited on pages 13 and 19)

Klewinghaus, N. (2020). Efficient Detailed Routing. Draft of PhD thesis, University of Bonn. (Cited on pages 9, 13, 14, 16, 54, and 55)

Korte, B., Rautenbach, D., and Vygen, J. (2007). Bonntools: Mathematical innovation for layout and timing closure of systems on a chip. *Proceedings of the IEEE*, 95(3):555–572. (Cited on page 12)

Korte, B. and Vygen, J. (2018). *Combinatorial Optimization: Theory and Algorithms. Sixth edition.* Springer. (Cited on page 3)

Kramer, M. R. and van Leeuwen, J. (1982). *Wire-routing is NP-complete.* Department of Computer Science, University of Utrecht. (Cited on page 11)

Kratochvíl, J., Lubiw, A., and Nešetřil, J. (1991). Noncrossing subgraphs in topological layouts. *SIAM Journal on Discrete Mathematics*, 4(2):223–244. (Cited on page 60)

Lavagno, L., Markov, I. L., Martin, G., and Scheffer, L. K. (2016). *Electronic design automation for IC implementation, circuit design, and process technology.* CRC Press. (Cited on page 3)

Li, H., Chen, G., Jiang, B., Chen, J., and Young, E. F. (2019). Dr. CU 2.0: A scalable detailed routing framework with correct-by-construction design rule satisfaction. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7. IEEE. (Cited on page 59)

Lichtenstein, D. (1982). Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343. (Cited on page 60)

Lin, Y.-H., Yu, B., Pan, D. Z., and Li, Y.-L. (2012). TRIAD: A triple patterning lithography aware detailed router. In *Proceedings of the International Conference on Computer-Aided Design*, pages 123–129. (Cited on page 59)

Lipton, R. J. and Tarjan, R. E. (1980). Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627. (Cited on page 38)

Liu, Y., Morgana, A., and Simeone, B. (1998). A linear algorithm for 2-bend embeddings of planar graphs in the two-dimensional grid. *Discrete Applied Mathematics*, 81(1-3):69–91. (Cited on page 61)

Ma, Q., Zhang, H., and Wong, M. D. (2012). 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. (Cited on page 59)

Maßberg, J. and Nieberg, T. (2013). Rectilinear paths with minimum segment lengths. *Discrete Applied Mathematics*, 161(12):1769–1775. (Cited on pages 59 and 67)

Mirsaeedi, M., Torres, J. A., and Anis, M. (2011). Self-aligned double-patterning (SADP) friendly detailed routing. In *Design for Manufacturability through Design-Process Integration V*, volume 7974, pages 79740O–1–79740O–9. International Society for Optics and Photonics. (Cited on page 59)

Müller, D. (2009). *Fast Resource Sharing in VLSI routing.* PhD thesis, University of Bonn. (Cited on pages 12, 16, and 55)

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

Nicholson, T. A. J. (1966). Finding the shortest route between two points in a network. *The Computer Journal*, 9(3):275–280. (Cited on page 25)

Nieberg, T. (2011). Gridless pin access in detailed routing. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 170–175. IEEE. (Cited on page 59)

Nohn, F. (2012). Detailed Routing im VLSI-Design unter Berücksichtigung von Multiple-Patterning. Diplomarbeit, University of Bonn. (Cited on pages 2, 56, 57, 58, 69, 70, 74, 76, 77, and 79)

Peyer, S., Rautenbach, D., and Vygen, J. (2009). A generalization of Dijkstra's shortest path algorithm with applications to VLSI routing. *Journal of Discrete Algorithms*, 7(4):377–390. (Cited on pages 24, 42, and 99)

Rabenstein, S. (2019). Cheapest detailed routes with restrictions and reservations. Master's thesis, University of Bonn. (Cited on pages 53, 88, 91, 92, and 94)

Rubin, F. (1974). The Lee path connection algorithm. *IEEE Transactions on Computers*, 100(9):907–914. (Cited on page 29)

Sarnak, N. and Tarjan, R. E. (1986). Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679. (Cited on page 38)

Scheifele, R. (2019). *Timing-Constrained Global Routing with RC-Aware Steiner Trees and Routing Based Optimization.* PhD thesis, University of Bonn. (Cited on page 12)

Schulte, C. (2012). *Design Rules in VLSI Routing.* PhD thesis, University of Bonn. (Cited on pages 7, 16, and 54)

Sommer, C. (2014). Shortest-path queries in static networks. *ACM Computing Surveys (CSUR)*, 46(4):1–31. (Cited on page 24)

Sterin, A. (2015). Postprocessing von Pfaden im Detailed Routing. Bachelor's thesis, University of Bonn. (Cited on pages 58 and 68)

Vygen, J. (2011). Faster algorithm for optimum Steiner trees. *Information Processing Letters*, 111(21-22):1075–1079. (Cited on page 88)

Wagner, D. and Willhalm, T. (2007). Speed-up techniques for shortest-path computations. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 23–36. Springer. (Cited on page 24)

Xu, X., Cline, B., Yeric, G., Yu, B., and Pan, D. Z. (2015). Self-aligned double patterning aware pin access and standard cell layout co-optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(5):699–712. (Cited on page 59)

Xu, X., Yu, B., Gao, J.-R., Hsu, C.-L., and Pan, D. Z. (2016). PARR: Pin-access planning and regular routing for self-aligned double patterning. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 21(3):42. (Cited on page 59)

Zhang, Y. and Chu, C. (2011). RegularRoute: An efficient detailed router with regular routing patterns. In *Proceedings of the 2011 International Symposium on Physical Design*, pages 45–52. (Cited on page 59)